The Journal of Systems and Software 140 (2018) 111–125

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

The relation between developers' communication and fix-Inducing changes: An empirical study

Mario Luca Bernardi[a,*], Gerardo Canfora[b], Giuseppe A. Di Lucca[b], Massimiliano Di Penta[b], Damiano Distante[c]

[a] Giustino Fortunato University, Benevento, Italy
[b] University of Sannio, Benevento, Italy
[c] Unitelma Sapienza University, Rome, Italy

ARTICLE INFO

Article history:
Received 23 June 2016
Revised 14 February 2018
Accepted 28 February 2018
Available online 2 March 2018

Keywords:
Bug management
Developers' communication
Social network analysis
Empirical study

ABSTRACT

Background Many open source and industrial projects involve several developers spread around the world and working in different timezones. Such developers usually communicate through mailing lists, issue tracking systems or chats. Lack of adequate communication can create misunderstanding and could possibly cause the introduction of bugs.

Aim This paper aims at investigating the relation between the bug inducing and fixing phenomenon and the lack of written communication between committers in open source projects.

Method We performed an empirical study that involved four open source projects, namely Apache httpd, GNU GCC, Mozilla Firefox, and Xorg Xserver. For each project change history data, issue tracker comments, mailing list messages, and chat logs were analyzed in order to answer four research questions about the relation between the *social importance* and communication level of committers and their proneness to induce bug fixes.

Results and implications Results indicate that the majority of bugs are fixed by committers who did not induce them, a smaller but substantial percentage of bugs is fixed by committers that induced them, and very few bugs are fixed by committers that were not directly involved in previous changes on the same files of the fix. More importantly, committers inducing fixes tend to have a lower level of communication between each other than that of other committers. This last finding suggests that increasing the level of communication between fix-inducing committers could reduce the number of fixes induced in a software project.

1. Introduction

Open source development, as well as the development of many industrial software projects, entail cooperative work, and thus interaction among hundreds of developers spread across the world. In such a scenario, the developers' interaction through traditional communication means—face-to-face, phone, or chat—may result problematic, not only because of the distance and of different timezones, but also due to some social aspects: (i) developers might not know very well each other, and (ii) it is desirable to keep track of the communication.

For such reasons, systems that handle discussion on software maintenance activities are of great relevance to support the communication between developers, as well as between developers and other project contributors, e.g., users who submit bug reports or other requests for changes. Recent work by Panichella et al. (2014a,b) shows that, very often, developers tend to use issue trackers and mailing lists to explain each other how software artifacts work. In other words, such communication means become sources for retrieving software documentation. Clearly, there are noticeable exceptions to this consideration, especially in industrial projects where, very often, the rationale of a change is not recorded in any mailing list or issue tracker, as the communication takes place mostly face-to-face or by phone (Aranda and Venolia, 2009).

* Corresponding author.
E-mail addresses: m.bernardi@unifortunato.it, m.bernardi@unifortunato.eu, mlbernar@unisannio.it (M.L. Bernardi), canfora@unisannio.it (G. Canfora), dilucca@unisannio.it (G.A. Di Lucca), dipenta@unisannio.it (M. Di Penta), damiano.distante@unitelma.it (D. Distante).

https://doi.org/10.1016/j.jss.2018.02.065
0164-1212/

The lack of an adequate communication can likely cause misunderstanding among developers who could introduce defects in the system, either because they do not properly comprehend a software artifact, or because they do not coordinate with other developers who are doing other changes on the same/related artifacts.

Previous work on analyzing the intensity of communication in periods of time close to fix-inducing commits indicates that bugs tend to be introduced when there are communication peaks (Abreu and Premraj, 2009). In other words, if on the one hand the common wisdom seems to suggest that the lack of communication could induce faults, on the other hand, Abreu and Premraj (2009) found that faults seem to be induced when the communication is particularly intense, which could be an indicator of critical issues difficult to be solved. Last, but not least, the way issues are fixed in software projects may depend on the ownership of artifacts to be changed: a recent study by Bird et al. (2011) indicates a negative correlation between ownership and fault proneness, i.e., if a developer is not a file owner—and therefore has not enough knowledge about the source code, he/she could be more prone to introduce a defect when changing it. To the best of our knowledge, nobody has investigated yet the level of communication between developers involved in changes that revealed to be fix-inducing. Our conjecture is that the lack of (or a limited) communication between these developers could have increased their likelihood of introducing defects.

This paper investigates the relationship between developer communication through issue trackers and mailing lists and their proneness to induce bug fixes. Specifically, we use the SZZ algorithm[1] developed by Sliwerski et al. (2005) and then improved by Kim et al. (2006) to identify changes that induced a fix.[2]

Then, we merge this information with a description of the committers communication network recovered from issue trackers and mailing lists, and we analyze (i) the technical activity of developers, characterized through their file ownership, and (ii) their social activity, characterized through social network analysis measures computed over issue tracker/mailing list communication. Finally, we compare social and technical characteristics of committers who likely induced a bug fix, as opposed to those of other committers.

Specifically, our study aims at answering the following research questions:

- **RQ1:** *What is the social importance of committers that worked on the files of a fix, if compared to those that did not?*
- **RQ2:** *What is the level of communication of committers that induced fixes, if compared with other committers?*
- **RQ3:** *To what extent are file owners/top committers involved in inducing bug fixes?*
- **RQ4:** *Who are the bug-fixing committers? Are they the same developers who induced the bug fix, or, instead, other committers?*

We carried out a study on data from four open source projects, namely Apache httpd, GNU GCC, Mozilla Firefox, and Xorg Xserver. Results indicate that:

- fix-inducing committers are people having a high social importance—measured in terms of *degree* and *betweenness* (Scott, 2000) in the communication network—compared to other committers that did not modify the files of the fix; moreover.
- the communication among fix-inducing committers is scarce if compared with that of committers who modified the same files of the fix without inducing the fix. This is in agreement with Abreu and Premraj (2009): peaks in communication among key developers do not correlate with faults induction whereas peaks in communication, *at community or project level*, among all developers (including those that have lower social importance) is symptom of higher levels of fix inducing changes.
- most of the fixes tend to be performed by the same people who induced them or by people who modified the files in the past without inducing a fix; very few fixes are performed by people who did not recently modify the affected files; moreover, most of the fixes are induced and performed by files owners/top committers.

This paper is an extension of our previous work presented in Bernardi et al. (2012). Compared to that work, the present paper:

- analyzes the communication between contributors and their social role in the project by considering, for all the considered systems, in addition to bug reports available in issue trackers, also messages posted in mailing lists and in IRC chats;
- extends the study to a wider period and to three new systems (i.e. Apache httpd, GNU GCC, and Xorg Xserver) involving a total of four systems (including Mozilla Firefox) of different sizes and for which all three different written communication channels (bug tracking system, mailing lists and IRC chats) were available;
- adds a new research question (RQ3) that studies to what extent file owners and top committers are involved in fix-inducing changes. The role of file owners is also taken into account in the new analyses of RQ2;
- dedicates a whole section to a qualitative analysis discussing the most interesting social networks identified in the study;
- provides more details on the whole extended study and a deeper discussion of results and threats to validity.

Paper structure. Section 2 discusses related work. Section 3 describes the process followed to extract data needed for our study. Section 4 defines the study and the analyzed variables. Results and practical implications are discussed in Section 5, while Section 6 provides some qualitative evidence to better corroborate the quantitative results. Then, Section 7 discusses the study threats to validity. Finally, Section 8 concludes the paper and outlines directions for future work.

2. Related work

The past and recent software engineering literature reports several works studying developers' communication and "sociality" and the relationships between bug inducing and bug fixing committers in open source software. For example, Cataldo et al. related the communication activity with software quality (Cataldo et al., 2006), and developers' productivity (Cataldo et al., 2008).

To the best of our knowledge, the closest work to ours has been carried out by Abreu and Premraj (2009). 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. (2008)). 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 when there is a peak of communication, either among core developers or other developers as well. Though there are similarities in

[1] To identify bug-introducing changes, the SZZ algorithm tracks down the origins of the source code deleted or modified by the bug fix using the built-in annotate feature of Software Configuration Management (SCM) systems. The annotate feature computes, for each line in the source code, the most recent revision in which the line was changed, and the developer who made the change.

[2] We use the expression "changes that induced a (bug) fix" (or, equivalently, "fix-inducing changes") instead of "bug-introducing changes" because the SZZ algorithm determines, based on the annotate/blame feature of versioning systems, changes that induced a fix. It cannot tell whether a change "introduced" a bug.

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's focus (Abreu and Premraj, 2009) is on *when* peaks of communication and fix-inducing changes occur, while our work specifically investigates the communication level of *who* was likely responsible of fix-inducing changes, and compares it with the communication of other committers. While the high social importance of fix-inducing committers reflects, in some sense, what Abreu and Premraj found about high peaks of communication during periods when bugs were introduced, we found that, however, fix-inducing committers had a low communication level among themselves.

Various authors have studied developers' collaboration by analyzing communication on mailing lists. Among them (Bird et al., 2006; Bacchelli et al., 2012; Guzzi et al., 2013; Wagstrom et al., 2005),Bird et al. (2006) found a high correlation between social network status metrics and source code development activities. They also analyzed the relationship between communications structure and the resulting code modularity finding that subcommunities identified using communication information are related to code collaboration behavior (Bird et al., 2008a). Moreover, mailing list communication often crosses the boundaries of single projects as highlighted by Canfora et al. (2011) on the collaboration between OpenBSD and FreeBSD developers. The heterogeneity of email content and discussion was investigated by Bacchelli et al. (2012) and by Guzzi et al. (2013). Bacchelli et al. (2012) presented a technique that classifies email lines into five categories (text, junk, code, patch, and stack trace) and evaluated such approach on a (statistically) significant amount of emails gathered from mailing lists of four unrelated open source systems. Guzzi et al. (2013) quantitatively and qualitatively analyzed a sample of 506 email threads from the development mailing list of Apache Lucene. Their study shows that developers participate in less than 75% of the threads, and that in only about 35% of the threads source code details are discussed.

Panichella et al. (2014a) investigated how developers' collaboration links vary and complement each other when they are identified through data from different communication channels. In particular, they analyzed developers' communication using three information sources (mailing lists, issue trackers, and IRC chat logs) for seven open source projects and results indicated that the overlap of communication links between the various sources is relatively low, and that the application of networks obtained from different sources may lead to different results. Panichella et al. (2014b) also used data from mailing lists and issue trackers to analyze the evolution of developer teams, and related this evolution to the files they change. Results of their work show that emerging team splits imply working on more cohesive groups of files and emerging team merges imply working on groups of files that are cohesive from structural perspective.

Zhou and Mockus (2011) investigated, on three industrial and three open source projects, how the sociality level of a project in a particular moment influences the likelihood for newcomers to become long-term contributors. Their results indicate that such an "upgrade" often occurs when the project sociality is low, because senior developers have more time to train the newcomers. We share with them the conclusion that developers' sociality over issue trackers and mailing lists play an important role in a software project. However, while Zhou and Mockus related such a phenomenon to the extent to which newcomers join a project, we relate sociality and communication with bug-inducing changes.

Singh (2010) analyzed over 4000 projects from SourceForge to understand how the relations among developers influence development activities. His results show that "small-world communities"—i.e., projects where there are small clusters of developers discussing with each other — are a factor for the project success. We share with this work the importance of developers' discussion during development and the fact that such a discussion, possibly, occurs in small clusters related to specific components or to specific issues to be handled.

In software development contexts, socio-technical congruence among different systems has been analyzed in the work of Bird et al. (2008b), who adopted 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 (2008) showed how social networks could be used to determine roles in a community of developers within a single project. In our case, we categorize committers with respect to the role played on each specific bug-fixing, i.e., whether they induced the bug, they did other changes, or (one of them) fixed the bug.

Canfora et al. (2011) analyzed the role of committers involved in cross-system bug fixing between FreeBSD and OpenBSD, finding that those committers exhibit higher brokerage, degree, and betweenness metrics than others. Similarly to this work, the paper by Canfora et al. highlights the role of developers having a high sociality in open source projects. However, while Canfora et al. related developers' sociality to their cross-system bug fixing activity, in this work we relate sociality with fix-inducing changes.

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 (2010) or in the work of Bird et al. (2009b). In particular, Bettenburg and Hassan found that social metrics have the same explanatory power of structural metrics for what concerns bug prediction, and that they can complement structural metrics to build better prediction models. Similarly, Bird et al. (2009b) combined dependency information between components with social metrics extracted from developers' communication to predict fault-prone components. While the aim of our work is not to build a fault-prediction model, we relate high sociality—on the one hand—and low communication—on the other hand—with fix-inducing changes.

Other than communication, another very important factor that can be related 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. (2011) investigated 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. This work—and **RQ3** in particular—confirms Bird et al. findings, this time on open source projects, indicating that a minority of bug fixes is induced by file owners, and the same happens for what concerns bug fixing.

Izquierdo-Cortazar et al. (2011) investigated whether—in Free/Libre/Open Source Software (FLOSS)—the contributors fixing a bug are the same inducing it, according to the SZZ algorithm (Sliwerski et al., 2005; Kim et al., 2008). Their study—carried out on some Mozilla components (Thunderbird, SeaMonkey, and Sunbird)—indicates that only a small percentage of bugs (3.5%) is fixed by people who likely induced them. In general, results of our study (**RQ4** in particular) agrees with the findings in Izquierdo-Cortazar et al. (2011), i.e., the majority of bugs is not fixed by inducers, although (i) we found higher percentages (ranging between 15% and 44%) than Izquierdo-Cortazar et al. (2011), and (ii) results for Mozilla are different, likely because of the different period of analysis (they only analyzed the period between 2008 and 2010, whereas we analyzed the period 1998–2017), and the different set of considered components.

3. Analyzing and relating fix inducing changes and developers' communication

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

3.1. Step 1: Identification of bug-fixing commits

First, we download the versioning system log using git[3] repositories of the considered systems and extract, for each commit, the files changed, the commit ID, the commit timestamp, the committer ID, and the commit note. Then, we rely on git change sets to cluster together related commits. After that, we identify bug-fixing changes using an approach inspired by the work of Fischer et al. (2003), i.e., we select changes whose commit note match a pattern such as *bug ID, issue ID*, or similar, where *ID* is a valid bug ID from the project issue tracker. 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 issue tracker.

After that, we use the issue type/severity field to classify the issue and distinguish bug fixing changes from other issues (e.g., enhancements). Finally, we only consider bugs having the status *CLOSED* and the resolution *FIXED*. Basically, we restrict our attention to (i) issues that were related to bugs, since we use them as a measure of fault-proneness, and (ii) issues that were not duplicate nor false alarms.

3.2. Step 2: Identification of fix-inducing changes

To identify fix-inducing changes, we use an approach inspired by the SZZ algorithm of Sliwerski et al. (2005) and then improved by Kim et al. (2008). Specifically, we rely on the git *blame* 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. let $p(c)$ be the parent of the commit c; for each file f_i, $i = 1 \dots m_k$ involved in the bug fix k (m_k is the number of files changed in the bug fix k) and fixed in a commit with ID $rfix_{i,k}$, we extract the same file in the commit with ID $p(rfix_{i,k})$, as it is the most recent revision still containing the bug k.
2. starting from the revision with commit ID $p(rfix_{i,k})$, the *blame* option of git is used to identify, for each source line in f_i changed to fix the bug k, the commit 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 $n_{i,k}$ fix-inducing commit IDs $rb_{i,j,k}$, $j = 1 \dots n_{i,k}$.
3. among commits $rb_{i,j,k}$, we identify the oldest one: $rold_{i,k} = minTimeStamp(rb_{i,j,k})$, where *minTimeStamp* returns the commit among the $rb_{i,j,k}$ having the smallest timestamp.

3.3. Step 3: Distinguishing fix-inducing committers from other committers

We define the set of fix-inducing 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 fix-inducing changes.

Then, we consider all changes made to each file $f_i, i = 1 \dots m_k$ in the time interval between $rold_{i,k}$ and $rfix_{i,k}$ to identify the set G_k of committers modifying f_i, without inducing the bug fix k:

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

3.4. Step 4: Identifying file owners

Bird et al. (2011) defined a file owner as a committer that performed, on a file, a given percentage of the total number of commits occurred on that file. In this context, we consider as owners the set of top committers for a file such that, together, they performed at least half of the changes on that file.

We identify as the set of file owners for file f_i at time t the set:

$$OW(f_i, t) \equiv \{ow_1, \dots, ow_i\} \quad (1)$$

of committers that, overall, performed the 50% of commits on f_i in the time interval [0, t] (where 0 is the starting point of our period of observation), i.e.,

$$\sum_{j=1}^{|OW(f_i,t)|} CM(ow_j) \geq 0.5 \cdot TCM(f_i, t) \quad (2)$$

where $CM(ow_j)$ is the number of commits performed by ow_i on file f_i in the time interval [0, t], and $TCM(f_i, t)$ is the total number of commits performed on f_i in the time interval [0, t]. To determine the set $OW(f_i, t)$, we ranked committers based on the number of commits they performed on f_i in the time interval [0, t], and selected the minimum set of topmost committers in the ranked list that satisfies Eq. (2).

3.5. Step 5: Extraction of issue tracker, mailing list, and chat contributors social network

For each of the considered projects, we build the contributors social network by analyzing bug reports and associated comments posted on the Bugzilla[4] issue tracker of the project, messages posted in the mailing lists and communication taken place in IRC channels associated to the project development.

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 subset of structural information fields relevant for our study includes: (*i*) the bug *ID*, (*ii*) the current bug *status*, (*iii*) the bug *reported* date, (*iv*) the *reporter* name and e-mail, (*v*) the system *product* and *component* in which the bug was identified, and (*vi*) the name and email of the developer the bug was *assigned to*.

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 track of the communication occurred between contributors participating in the bug fixing process. In the following, reporters, assignees and authors are generically referred as "issue tracker contributors".

To build the contributors network for a given bug, we adopt a conservative approach, by making the assumption that the author of the jth 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

[3] https://git-scm.com/.

[4] https://www.bugzilla.org/.

her. This approach tends to overestimate the links—i.e., when responding in a thread one does not necessarily communicate with all people who posted a comment before—however in this paper we are interested to seek cases of lack of communication, therefore overestimation is preferred to underestimation.

For each project, in addition to issue trackers, also mailing lists and chats associated to the project development were analyzed to extract communication links among developers, as later specified in Section 3.7.

In mailing lists, given two contributors *Tom* and *John*, we assume a social interaction between them if the mailing list contributor *Tom* sends at least one email to *John* through the mailing list. For the purpose of our study, other than just extracting the network, we also need to keep track of all the messages exchanged by mailing list contributors over time. Mailing list contributors and their social interactions are extracted from mailing list messages by identifying the sender name and email address and, whenever available, the recipient/reply-to name and email address. Clearly, since messages can be sent to a mailing list rather than to single addresses, we assume that (i) *Tom* and *John* communicate if there is a direct message with *Tom* as sender and *John* as recipient or *vice versa*, or (ii) if *John* answers, even if replying to the mailing list, to a thread initiated by *Tom*, or to which *Tom* has previously replied.

For each bug, daily logs of development chats are parsed seeking for communication that is related to the files belonging to the fix. When a comment refers to a file included in the fix the subsequent messages, within a fixed time window,[5] are extracted along with their authors. From such messages a network is built with the same conservative approach used for BTS messages: subsequent messages answer all previous ones and represent, in the contributors social network for the bug, communication paths among the respective authors.

When building the contributors social network, we need to unify names belonging to the same person. In fact, it may happen that, although a contributor has to register on an issue tracker, the same contributor can sometimes appear with multiple accounts and slightly different names, e.g., with or without middle name, or with initials only for first/middle name. This step resolves inconsistencies and cases of people referring to themselves differently, e.g., *John Fitzgerald Smith, John F. Smith, John Smith*, or *J. Smith*, in different time. Specifically, we use an approach similar to the one used by Bird et al. (2006) that we previously used to analyze mailing lists for a different research work (Canfora et al., 2011). In the context of this work this approach has been used to unify names from all the communication channels that have been considered (i.e. BTS comments, mailing lists and chats). This approach is composed of the following steps:

1. **Remove cases and special characters:** names are converted into lower cases, and special characters, including dots ".", are removed, e.g., *John F. Smith* becomes *john f smith*.
2. **Ignore middle names:** e.g., unless there are ambiguities, *john p Smith, john paul smith*, and *john smith* are considered to be the same person. If the name is composed of more than two terms, and there are two names matching the first and last term, then we assign them to the same person, e.g., *john f Smith* and *john smith* are considered to be the same person, unless there is another person with the same first and last name, but with a different middle name, e.g., *john paul smith*.
3. **First name referred with initials only:** e.g., unless there are ambiguities, *john smith* and *j smith* are considered to be the same person.
4. **Last name only:** if the name is composed of one term only, and it corresponds to the last name of one, and only one, other person, the two names are considered to belong to the same person.
5. **Initials only:** if there is a name composed of terms of one letter only (e.g., *j f k*), and there is one, and only one, name with the same initials (e.g., *john fitzgerald kennedy*) then we assume that two names belong to the same person. This rule is also applied if there is a name composed of one term only (e.g., *jfk*), and these letters correspond to the initials of one and only person.
6. **User ID-like name:** this is a similar case, but slightly more complex, than the previous one. Sometimes people refer themselves with IDs composed of concatenated first and last names, or of first (and sometimes middle name) initials concatenated to the last name. For example, *john f. smith* could be referred as *johnsmith, jsmith*, or *jfsmith*. In this case we check if the ID could be composed by concatenating names of another person, or her initials and last name.

3.6. Step 6: Mapping committer IDs to issue tracker contributor names

This step aims at linking issue tracker contributors with committer IDs. We use a process similar to the one used above to unify issue tracker 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 issue tracker/mailing list contributor names to be linked to git committer IDs like *johnsmith* or *jsmith*, trying to compose issue tracker contributors first and last names, or first/middle name initials and last name.

As all of the projects analyzed in our study use git as version control system for tracking changes, committer IDs also contain email addresses. In order to take into account also email addresses for discriminating contributors, we extract the email address used by the contributor in the issue tracker or mailing list, and try to match it with the git ID using the following two heuristics:

1. if there is an issue tracker contributor with an email matching the git ID (in some cases, like for Mozilla after replacing "@" by "%") then the issue tracker contributor is mapped to the committer; otherwise.
2. if there is *only one* issue tracker contributor with the email user name—i.e., characters before "@"—matching an email user name of a git ID, then the issue tracker contributor is mapped to the committer.

3.7. Step 7: Extracting discussions relevant to a bug fix

The final step of our analysis aims at identifying, for each bug fix, the portion of contributors network of interest.

Considering the entire network would be too comprehensive, as we would like to exclude communication occurred a long time before the bug fix was induced. Also, we would like to exclude communication that was related to completely different topics. For mailing lists and chats, we need to determine the set of emails that are likely related to changes on files involved in the bug fix. For each bug, we filter the communication network using the following heuristics:

1. *bug report filtering (for issue tracker communication only):* in the issue tracker we restrict our analysis to the bug report of interest and its associated comments.

[5] In this work we considered the entire daily session of the chat as we verified that smaller time windows reduce the already small number of nodes and edges contributed to the network by chats, thus making them useless.

Table 1
Information about issue tracker/mailing list contributors for the example of Fig. 1.

Issue tracker 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

2. *linking to files (for mailing lists and chats only):* for mailing lists and chats it is not possible to establish a direct link to the affected components, as we do for bug reports. Therefore, to filter emails or discussion that could possibly be relevant to a bug fix, we restrict the attention to the set of messages that mentioned any of the files changed in the bug fix. To link messages to files, we use the approach proposed by Bacchelli et al. (2010), which uses regular expressions to match file names in text.
3. *time frame filtering:* we need to restrict our time frame, and we select a time interval between the following two timestamps:
   - We consider the oldest fix-inducing change for all files f_i involved in the particular bug fixing, i.e., the $minTimeStamp(rold_{i,j})$, $\forall f_i$, $i = 1 \dots m_k$ and look back of a period of three months. Since one of our aims is to identify to what extent fix-inducing committers talked to each other, we need to start our analysis before the first fix-inducing change occurred—i.e., to see whether fix-inducing committers talked before such a change. We consider three months as a reasonable period—also representative of the minimum period between two releases for the analyzed projects—in which a communication between two developers could occur before a change was done. Thus, we consider such a date $= minTimeStamp(rold_{i,j}) - 90days$ as the starting date of the interval. We also tried longer periods without obtaining substantially different results. Conversely, choosing shorter periods increases the risk to miss relevant communications.
   - 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 fix-inducing changes (and other changes) communicated with each other *before* the bug was discovered.

3.8. Running example

Fig. 1 reports a running example illustrating our data extraction approach for an hypothetical bug *k*. Fig. 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 fix-inducing 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 fix-inducing committers (set B_k), while *Paul* and *Alice* are other committers who modified the fixed files without inducing fixes (set G_k). Fig. 1-a shows comments posted on bugs of the same component in the period considered (white area), as well as email threads mentioning the impacted files and related to the same period of interest. The resulting network is shown in Fig. 2, where filled squares represent fix-inducing committers ($\in B_k$), empty squares non fix-inducing committers ($\in G_k$), and dots other issue tracker contributors. Information on all contributors communication is reported in Table 1.

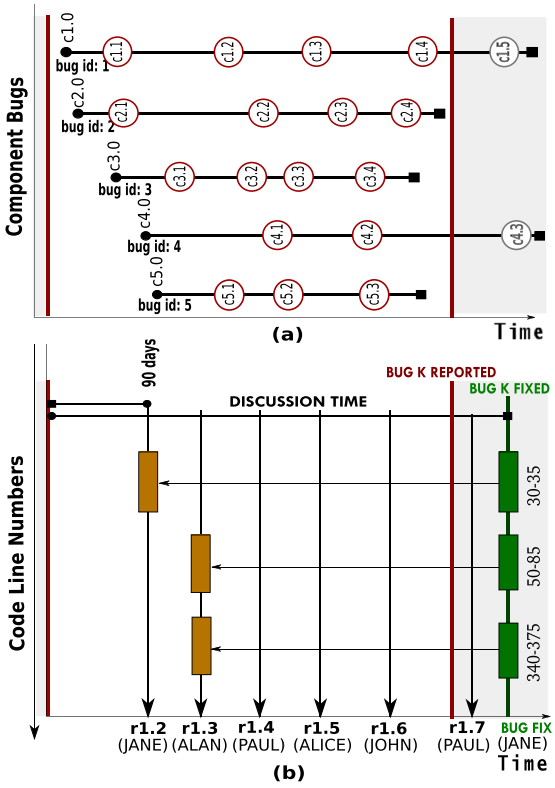


Fig. 1. Running example: issue tracker comments/email threads (top) and commits (bottom).

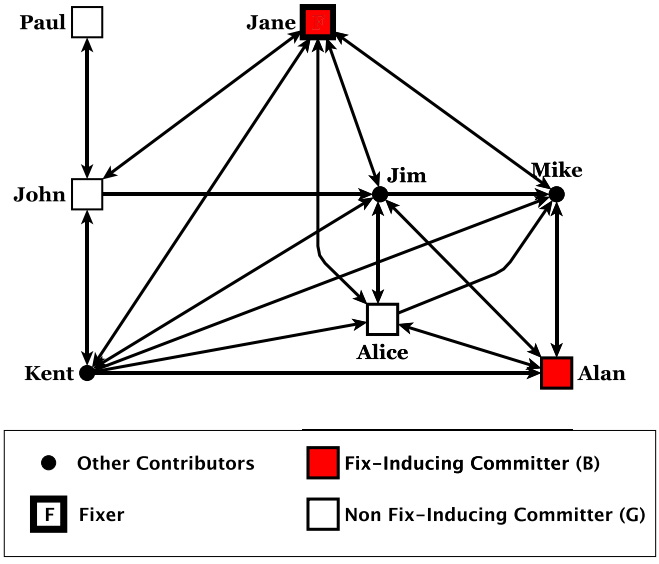


Fig. 2. Issue tracker/mailing list contributors network related to the example of Fig. 1.

4. Empirical study

The *goal* of this study is to investigate the level of communication of fix-inducing committers (compared with other committers) with the *purpose* of understanding whether a lack of communication could favor fix-inducing changes. 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 induction of fixes, but also of project managers interested to

Table 2
Characteristics of the data set used in the study.

System	Apache httpd	GNU GCC	Mozilla Firefox	Xorg Xserver
Language	C	C/C++	C/C++, JavaScript, CSS	C
Source of information	issue tracker + mailing list + chat	issue tracker + mailing list + chat	issue tracker + mailing list + chat	issue tracker + mailing list + chat
Components considered	all	ada, bootstrap, c, debug, java, rtl-optimization, libgcj	all	all
Observed period (oldest and latest fix date for bugs in the dataset)	Feb 2002-May 2017	No. 1999-May 2017	Aug 2003-Jul 2017	Dec 2004-Apr 2017
# of bugs in the sample	976	3,503	12,972	294
mean # of edges per mailing list network	257	393	40	379
mean # of edges per issue tracker network	110	89	110	141
mean # of edges per chat network	9	11	15	12
mean # of nodes per mailing list network	32	57	7	28
mean # of nodes per issue tracker network	14	19	7	14
mean # of nodes per chat network	6	12	5	3
# of committers	218	880	6,821	554
# of contributors	696	8,802	11,212	656
# of committers ∩ contributors	75	257	1435	287

promote a better communication, through suitable tools or better project management, with the objective of avoiding to introduce bugs because of lack of communication.

The *context* consists of data extracted from versioning systems, issue trackers, mailing lists, and IRC chats of four open source projects: Apache httpd, GNU GCC, Mozilla Firefox (or the former "Mozilla browser"), and Xorg Xserver. For all the analyzed systems, the source code is versioned under git. All projects use Bugzilla as issue tracker.

Apache httpd[6] is a well-known open source http server. GNU GCC[7] (GNU Compiler Collection) is a compiler system produced by the GNU Project supporting various programming languages and a key component of the GNU toolchain and the standard compiler for most Unix-like Operating System. Mozilla Firefox[8] is a Web browser developed by the Mozilla Foundation that supports plug-in and add-on program packages called extensions, written in C/C++. Xorg Xserver[9] is the free and open source implementation of the display server for the X Window System stewarded by the X.Org Foundation.

Table 2 reports, for the four projects, some overall information relevant for our study. We selected 976 bugs from Apache httpd, 3,503 from GNU GCC, 12,972 from Mozilla Firefox, and 294 from Xorg Xserver, spanning each project's considered lifetime interval. Specifically, (i) we considered issues on the issue tracker that were related to bug-fixing and not to other maintenance activities (although—see Section 7—the issue tracker classification can be often imprecise), (ii) we considered issues labeled as CLOSED and RESOLVED, and (iii) we considered issues for which, in the versioning system, it was possible to identify fix-inducing changes using the SZZ algorithm (Sliwerski et al., 2005; Kim et al., 2008). This allowed us to exclude issues fixed early in the project, which, according to the SZZ algorithm, resulted as induced by the first revision of each file.

Other than general information about the project, the period of observation, and the number of considered bugs, Table 2 reports (i) the mean number of edges and nodes of the networks extracted, how many edges and nodes are contributed, as average for each bug, by each source of information (mailing list, issue tracker, and chat); and (ii) the number of committers, issue tracker/mailing list/chat contributors, and their intersection.

4.1. Research questions

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

- **RQ1:** *What is the social importance of committers that worked on the files of a fix, if compared to those that did not?* The first research question aims at understanding the level of importance of fix-inducing committers in the communication about the considered project, taken place in issue tracker, mailing list, and chat, if compared with other committers and with the bug-fixing committers. It could either happen that fix-inducing committers have, in general, a very poor role in the communication, and because of that induce bug fixes, or that, despite their high communication (and thus importance) they induce bugs more than others.
- **RQ2:** *What is the level of communication of committers that induced fixes, if compared with other committers?* The second research question aims at investigating to what extent fix-inducing committers *talk to each other*, compared to other committers. This would help to capture cases where multiple persons modify the same artifacts without coordinating each other, thus inducing a fix.
- **RQ3:** *To what extent are file owners/top committers involved in inducing bug fixes?* The third research question focuses on the activities of file owners, to determine the extent to which fix-inducing changes and bug fixing could depend on file ownership.
- **RQ4:** *Who are the bug-fixing committers? Are they the same developers who induced the bug fix, or, instead, other committers?* The fourth research question aims at understanding whether a bug was fixed by: (*i*) one of the committers that induced it (and thus someone who realized his/her own mistake and corrected it); (*ii*) committers who changed the file in the meantime, without however inducing fixes; (*iii*) or, instead, by someone else that jumped in the discussion and solved the problem.

[6] httpd.apache.org.
[7] gcc.gnu.org.
[8] www.mozilla.org/firefox.
[9] www.x.org/wiki/XServer/.

4.2. Analysis method

This section describes the social network analysis measures used to characterize the communication occurring through the issue tracker, mailing lists, and chats, as well as the statistics used to analyze the data. All analyses were performed using the *R* statistical environment[10] and, specifically, the packages *igraph* and *sna* to perform social network analysis.

To address **RQ1**, we consider the network extracted for each bug (see Section 3.7), and compute—for each issue tracker/mailing list/chat contributor of the network—a series of social network measures that characterize his 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.
- *betweenness:* this is a generalized concept of "centrality" for an actor; 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_i, a_j), the fraction of shortest paths between a_i and a_j that pass through the actor in question, and by summing such percentages over all pairs.

For all the bugs considered in the study, we compare the above measures across the three groups of committers: (*i*) any of the committers who likely induced the bug (denoted as *B*); (*ii*) committers that worked on the file during the same period when the bug was induced, without however inducing fixes (denoted as *G*); or (*iii*) other committers not modifying the fixed files in the same period (denoted as *O*). 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 (Holm, 1979). 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 (Grissom and Kim, 2005), 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 is considered negligible for $|d| < 0.147$, 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 **RQ2**, we identify the following restrictions from the contributors network of each bug: (*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 a set of nodes in a graph is. The Krackhardt's connectedness for a digraph *DG* is defined as the fraction of all pairs of 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 (connected graph).

Then, we perform a statistical comparison of the connectedness of the subsets of committers involved in files that were changed during the bug fixing: specifically we compare—using boxplots, Mann-Whitney test, and Cliff's delta—the connectedness between *B* committers with that of *G* committers. Note that in this case we do not consider other committers (*O*) as we want to focus our attention on people who worked on the affected file(s) before the bug was discovered and then fixed. Finally, we repeat the analysis for file owners, to determine whether their level of connectedness is different with respect to other committers.

To address **RQ3**, we simply report the number and percentage of bugs induced and fixed by file owners and by other committers. Specifically, we compute the number and percentage of bugs (i) which have been induced by file owners, (ii) that were fixed by file owners, and (iii) which were induced and then fixed by file owner(s).

To address **RQ4**, we compute the proportion of bugs fixed by the same kind of contributors evaluated in addressing RQ1 (i.e., the B, G and O sets defined above). We test whether the proportions significantly differ using χ^2 goodness-of-fit test, and also use the odds ratio (OR) effect size measure (Sheskin, 2007) 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 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: $OR = (p/(1-p))/(q/(1-q))$.

5. Results and implications

This section reports and discusses results of the empirical study defined in Section 4, with the aim of addressing the four research questions formulated in Section 4.1. Raw data and working data sets are available for replication purposes.[11] The section ends with a discussion of practical implications related to main findings of the study.

5.1. RQ1: What is the social importance of committers that worked on the files of a fix, if compared to those that did not?

Fig. 3 shows a comparison of the degree social network measure for three different kinds of committers, i.e., fix-inducers (*B*), non fix-inducers, i.e., committers who worked on the fixed files without inducing fixes (*G*), and other committers (*O*). 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, the differences found between the different groups of committers are perfectly consistent between degree, in-degree and out-degree. Similar boxplots—related to betweenness—are shown in Fig. 4. Table 3 reports results of a pairwise comparison of degree distributions (left side) and betweenness distribution (right side) for the three types of committers, specifically Mann-Whitney test p-values (adjusted with the Holm's correction) and Cliff's *d* effect sizes.

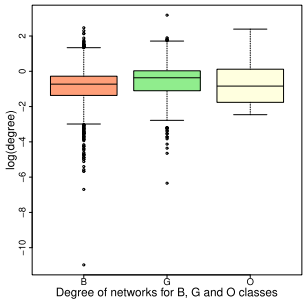
It is particularly relevant to compare developers who worked on the affected files and induced (*B*) or not induced (*G*) a fix.

For all projects, *G* committers tend to have a significantly higher degree than *B* committers. The effect size is *small* for Apache httpd ($d = 0.18$) and GNU GCC ($d = 0.21$), while it is large for Mozilla Firefox ($d = 2.12$) and Xorg Xserver ($d = 0.98$). This means that, especially for the last two projects, those who induced a fix have a significantly smaller degree than those that did not induce a fix. Also, *G* committers tend to have significantly high degree than *O* committers, with a large effect size in all cases but for GNU GCC, where the effect size is negligible.

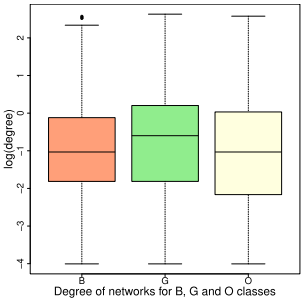
Results are less conclusive in terms of betweenness, where the comparison between *G* and *B* shows statistically significant differences only for Xorg Xserver. Recall that the betweenness measures the extent to which a node is in the shortest path between two

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

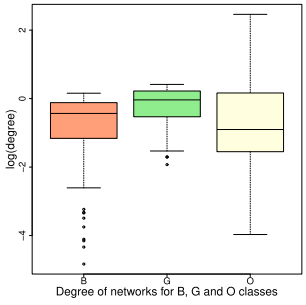
[11] https://github.com/mlbresearch/talking-data.



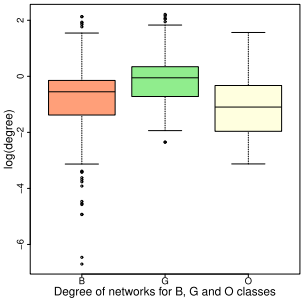
(a) Apache httpd



(b) GNU GCC

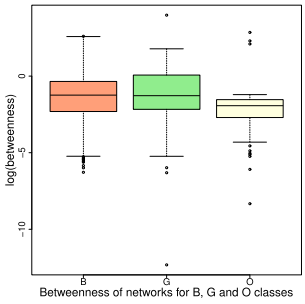


(c) Mozilla Firefox

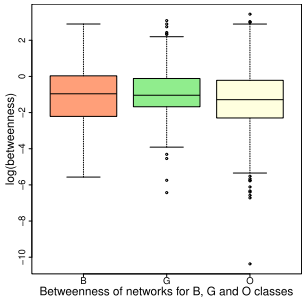


(d) Xorg Xserver

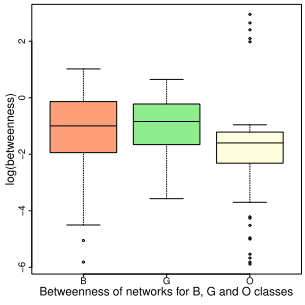
Fig. 3. Degree for different kinds of committers. B: fix-inducing committers, G: non-bug fix inducing committers, O: others.



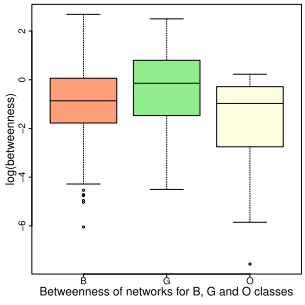
(a) Apache httpd



(b) GNU GCC



(c) Mozilla Firefox



(d) Xorg Xserver

Fig. 4. Betweenness for different kinds of committers. B: fix-inducing committers, G: non fix-inducing committers, O: others.

Table 3
Comparison between Degree and Betweenness of different groups of committers: adjusted Mann-Whitney p-value and Cliff's delta effect size.

Apache httpd				
	Degree		Betweenness	
Comparison	adj. p	Cliff's delta	adj. p	Cliff's delta
G vs. B	< 0.01	0.18	0.88	−0.04
G vs. O	< 0.01	0.85	< 0.01	0.92
B vs. O	< 0.01	0.08	< 0.01	0.86
GNU GCC				
	Degree		Betweenness	
Comparison	adj. p	Cliff's delta	adj. p	Cliff's delta
G vs. B	< 0.01	0.21	0.60	0.14
G vs. O	< 0.01	0.04	< 0.01	0.02
B vs. O	< 0.01	1.12	< 0.01	0.10
Mozilla Firefox				
	Degree		Betweenness	
Comparison	adj. p	Cliff's delta	adj. p	Cliff's delta
G vs. B	< 0.01	2.12	0.16	0.12
G vs. O	< 0.01	1.64	< 0.01	1.07
B vs. O	< 0.01	1.06	< 0.01	1.01
Xorg Xserver				
	Degree		Betweenness	
Comparison	adj. p	Cliff's delta	adj. p	Cliff's delta
G vs. B	< 0.01	0.98	< 0.01	1.72
G vs. O	< 0.01	2.20	< 0.01	1.97
B vs. O	< 0.01	1.42	< 0.01	0.95

other nodes. It is possible that such a relationship is less relevant as the amount of *direct* communication between developers is the one that makes a difference here.

We can conclude **RQ1** stating that in general committers who worked on the files impacted by a bug fix and that did not induce the fix (*G* committers) tend to have a higher social importance (in terms of degree) than those that induced the fix (*B* committers) and others that did not work on such files (*O*). Results are inconclusive in terms of betweenness.

Table 4
Comparison between connectedness of *G* and *B* committers: Mann-Whitney p-value and Cliff's delta effect size.

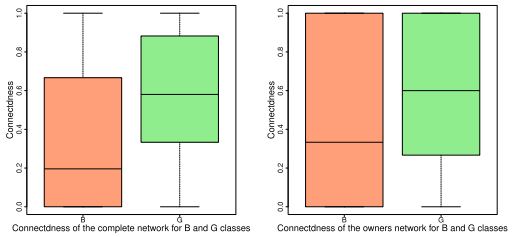
Project	Committers	p-value	Cliff *d*
Apache httpd	All	< 0.01	3.28
	Owners	< 0.01	2.43
GNU GCC	All	< 0.01	1.62
	Owners	< 0.01	0.42
Mozilla Firefox	All	< 0.01	2.56
	Owners	< 0.01	0.38
Xorg Xserver	All	< 0.01	4.85
	Owners	< 0.01	5.11

5.2. RQ2: What is the level of communication of committers that induced fixes, if compared with other committers?

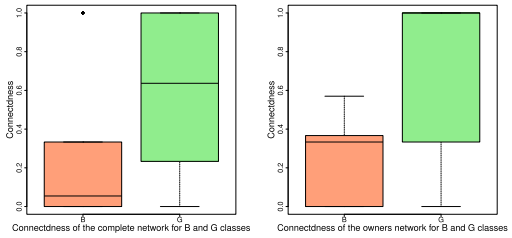
Fig. 5 shows the boxplots of the connectedness for fix-inducing committers (*B*), and committers who worked on the fixed files without inducing a bug (*G*). Results are reported on the left-side for all committers, and on the right-side for file owners only.

Mann-Whitney test results—reported in Table 4—indicate that, for all projects, committers $\in G$ have a significantly higher connectedness (p-value < 0.01) than those $\in B$, in all cases with a large effect size ($d > 0.474$).

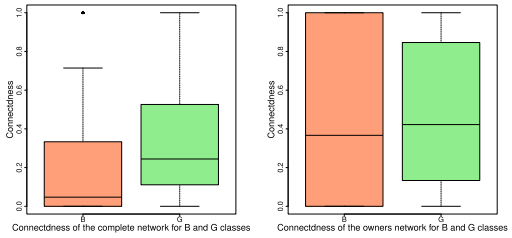
If we restrict our attention to file owners, we notice that the behavior is similar: committers $\in G$ have a higher connectedness than committers $\in B$. The effect size remains large for Apache httpd and Xorg Xserver, while it becomes medium ($0.33 \leq d < 0.474$) for GNU GCC and Mozilla Firefox. Looking at the connectedness of fix-inducing committers we see that it is significantly lower than the median connectedness of other committers working on the fixed files.



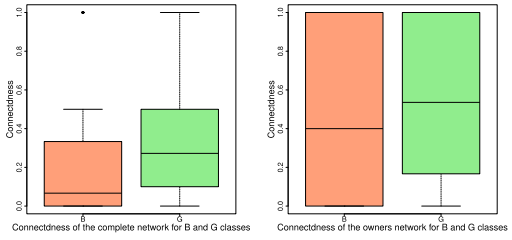
(a) Apache httpd



(b) Xorg Xserver



(c) GNU GCC



(d) Mozilla Firefox

Fig. 5. Connectedness for B (fix-inducing committers) and G (non fix-inducing) - The complete networks are reported on the left and the owner networks on the right.

We can conclude **RQ2** stating that, since effect size is large for all the analyzed systems as reported in Table 4, the level of communication correlates with committer's class (G and B). This is true also committers that are file owners.

Results of **RQ2**, compared with results of **RQ1**, indicate that, despite fix-inducing committers having an important role in the bug discussion, the communication between them is scarce: *people that work on the same file and induce fixes communicate less than others.*

5.3. RQ3: To what extent are file owners/top committers involved in inducing bug fixes?

Table 5 reports, for the four projects, the number and percentage of fixes performed and of fixes induced by file owners, as defined in Section 3.4.

As it can be noticed from the first data column of the table, a percentage of bug fixes ranging from 25% of Apache Httpd to 39% of Xorg Xserver were performed by file owners. This, once again, indicates that fixes are committed by other people than those who mostly worked on those files. The second data column of the table reports the number and percentage of fixes induced by file owners. Comparing these percentages with those of performed fixes on the left side of the table it results that in Mozilla Firefox and GNU GCC file owners induced fewer fixes than the number of fixes they performed, while in smaller projects (Apache httpd and Xorg Xserver) it applies the opposite. The percentages of induced fixes range between 15% of GNU GCC and 44% of Xorg Xserver. Such percentages tell that, in general, fixes are more often induced by "occasional" committers than by file owners.

As it appears from the third data column, a small number and percentage of fixes—i.e., 117 (12%) for Apache httpd, 354 (10%) for GNU GCC, 2,282 (18%) for Mozilla Firefox, and 57 (19%) for Xorg Xserver—were both induced and performed by file owners.

In summary, results of **RQ3** indicate that bug fixing is an activity that goes beyond file ownership, as only a minority of bugs has been fixed by file owners and (ii) in confirmation to what previously found by Bird et al. (2011) only a reduced percentage of fixes (less than 25% for the two larger projects we considered and less than 44% for smaller ones) is induced by file owners, suggesting that lack of knowledge/confidence about source code could increase the risk of inducing bug fixes. This could also be due to the fact that often file owners work on new features/enhancements, while a separate maintenance team works on bug fixes.

5.4. RQ4: Who are the bug-fixing committers? Are they the same developers who induced the bug fix, or, instead, other committers?

Fig. 6 reports for each project the number and percentage of bugs fixed by (i) *B*: fix-inducing committers, (ii) *G*: non fix-inducing committers, i.e., committers working on the same files without inducing fixes, and (iii) *O*: other committers, i.e., committers not having worked before on the files of the fix.

The χ^2 test indicated a significant difference, in proportions, among the three sets of fixings (p-value < 0.001) for all the systems.

Results for Apache httpd shows that fix-inducing committers (*B*) fixed 28% of the bugs, while 71% of the bugs were fixed by (*G*) committers who worked on the same files without inducing fixes. Only 1% of the bugs were fixed by other committers (*O*). This is likely an indication of the presence of developers responsible for quality assurance and for committing patches every time bugs were closed. Committers $\in G$ have OR=2.5 times the chances of fixing a bug than those $\in B$; committers $\in G$ have 86.5 times the chances to fix a bug than those $\in O$. Finally, committers $\in B$ have 34.5 times the chances to fix a bug than those $\in O$

In the case of Mozilla Firefox, committers $\in G$ have almost the same chances (OR=1.017) to fix a bug of those $\in B$. Committers $\in G$ have OR=412.7 times the chances to fix a bug than those $\in O$. Finally, committers $\in B$ have OR=283 times the chances to fix a bug than those $\in O$

For GNU GCC, committers $\in G$ have OR=1.34 times the chances of fixing a bug than those $\in B$; in this case committers in $\in O$ did not perform any bug fixes.

This applies also to Xorg Xserver for which committers $\in G$ have OR=1.7 times the chances of fixing a bug than those $\in B$ and there are no committers $\in O$ that performed bug-fixes.

In summary, we can conclude **RQ4** stating that a considerable percentage of bug fixes (between 28% and 49%) is performed by the same committers who induced them. However, a high number of bug fixes (between 50% and 71%) is applied by other people who previously worked on the affected files (but that did not induce the

Table 5
Number of fixes performed and induced by file owners.

Project	Committers	Performed fixes		Induced fixes		Induced and Performed fixes	
Apache httpd	Owners	242	(25%)	295	(30%)	117	(12%)
	Other committers	734	(75%)	681	(70%)	859	(88%)
GNU GCC	Owners	1007	(29%)	508	(15%)	354	(10%)
	Other committers	2496	(71%)	2995	(85%)	3149	(90%)
Mozilla Firefox	Owners	4775	(37%)	3283	(25%)	2282	(18%)
	Other committers	8197	(63%)	9689	(75%)	10,690	(82%)
Xorg Xserver	Owners	116	(39%)	129	(44%)	57	(19%)
	Other committers	178	(61%)	165	(56%)	237	(81%)

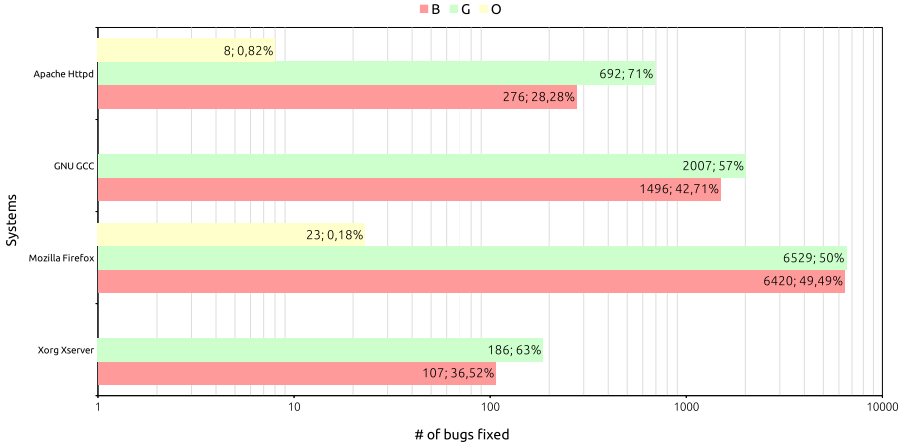


Fig. 6. Number (represented on a logarithmic scale) and percentage of fixes for B: fix-inducing committers, G: non fix-inducing committers, and O: other committers.

bug fix). Only a very small percentage of fixes ($< 1\%$) is handled by people that did not work recently on files affected by the bug.

5.5. Implications

One of the main findings of our study concerns the relation between the low level of connectedness of fix-inducing committers' social networks and the induction of fixes. This result suggests that increasing the level of communication among them could reduce the number of induced fixes in the software project.

To this aim, the communication between committers working on the same files should be fostered. This will have no impact on G committers that already have good level of communication, but improves the connectedness of B ones.

In issue trackers this could be obtained by showing, for each file of the project, the list of related committers as the key persons that should be contacted for discussing and validating changes to the file before introducing them. This list could be ranked to show committers that modified the file most recently and more frequently first. The same feature could also be implemented in the preferred IDE (e.g., as a plug-in for Eclipse), by using the results of an analysis similar to the one conducted in our study and updated on a daily base.

For the same purpose, messages sent to the project mailing list by fix-inducing committers and mentioning a file of the project could be emphasized to other fix-inducing committers for that file.

The above tactics could mitigate the effects of the lower levels of communication that characterize fix-inducing committers and foster a more informed and shared participation of them in the project.

Fostering communication between developers, and in particular between fix-inducing committers, goes in the same direction taken by the everyday more adopted software development practice of modern code-review (Bacchelli and Bird, 2013), which, however, is not yet a standard de-facto (at least not for in the software projects analyzed in our study).

6. Some qualitative examples

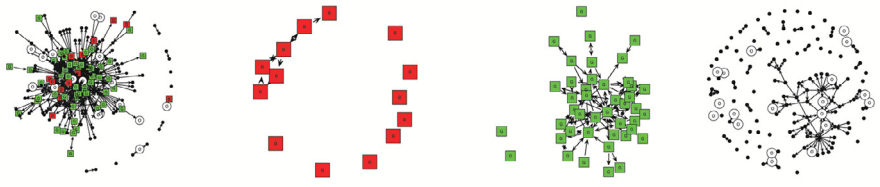
This section provides some qualitative analysis aimed at corroborating the quantitative evidence provided in Section 5. The analysis has been performed by means of a purposeful sampling aimed at reporting and discussing cases in which the lack of written communication between developers occurred in contexts where bugs were introduced as well as cases that highlight the importance of feedback received by developers communicating with each other.

In some cases, we found comments mentioning that the bug was fixed after clarifying the problem with another person, i.e., very likely there was lack of communication. For confidentiality reasons, we omit contributors' names from the reported comments.

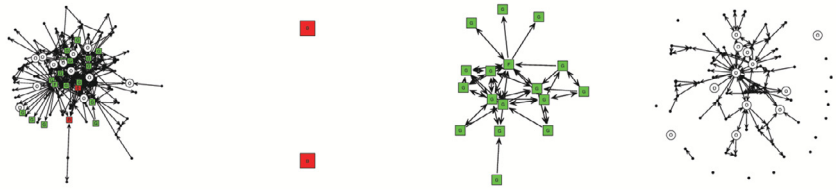
For example, in Apache httpd we found such evidence in the discussion of bug **#38034**[12]: *"Then in the next month, after discussing this bug with XX at ApacheCon, YY attempted to revive the thread and proposed another possible approach endorsed by XX. No one replied to YY's email."*, bug **#45994**: *"Here's a better patch. Thanks to XX for pointing out another problem that was lurking."*, or bug **#15132**: *"Since this is a message from XX, I think you'll need to ask them what they are talking about. I have no idea what they are referring to."*.

For Mozilla Firefox, consider for example bug **# 104075**[13]: *"Please talk with me before doing so.*, bug **#124485**: *"I know next to nothing about menus. You need to talk to XX."*, bug **#159359**: *"new patch after talk with XX and YY"*, and bug **#163645**: *"Okay after talking with XX, I think I have a handle on how to properly fix"*.

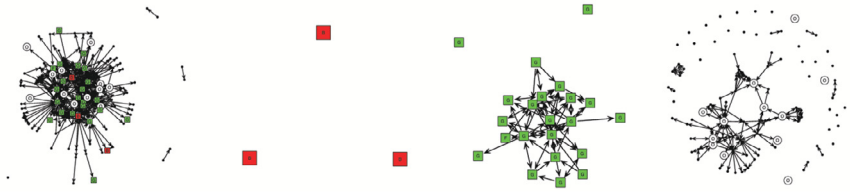
[12] https://issues.apache.org/bugzilla/show_bug.cgi?id=7180.
[13] https://bugzilla.mozilla.org/show_bug.cgi?id=104075.



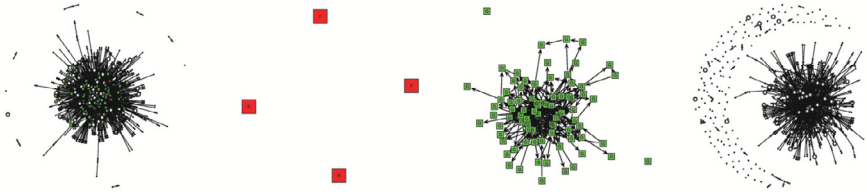
(a) Mozilla Firefox Bug #1162205



(b) Xorg Xserver Bug #58398



(c) Apache Httpd Bug #31759



(d) GNU GCC #29544

Fig. 7. Example discussion graphs (in order, from left to right, are reported Full, B, G and O networks).

6.1. Examples of communication networks

Fig. 7 reports, for the four projects, examples of communication networks, with the aim of highlighting the social importance and the connectedness of the different kinds of committers.

Fig. 7(a) shows the discussion related to Mozilla Firefox bug #1162205 *"Empty Chrome imported cookie values cause Twitter and Gmail redirect loops"*. It contains four networks, from left to right: the full network with all communication channels, the B network containing only fix-inducers, the G network containing only non fix-inducing committers and the O network containing only other committers and contributors to the discussion that are not committers. 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 in the time interval between the fix-inducing change (No. 2011) and the bug fix (Feb 2015) on the component (*Migration*) impacted by the bug. As it can be noticed, while *G* committers (green squares) mostly communicate with each other, the *B* fix-inducing committers (red squares) are basically disconnected. This is better highlighted by the B network used to evaluate connectedness since it does not contain the links through other issue tracker/mailing list/chat contributors (the small black dots that are present in Full and O networks).

These considerations are even more evident for the Xorg Xserver bug #58398 (Fig. 7(b)) in which the two fix-inducing committers are almost isolated in the communication network.

Similar considerations can be drawn from the communication graph reported in Fig. 7(c) for bug #31759 of Apache httpd. In this case, we have three fix-inducing committers with no direct nor indirect path between them, which highlights scarce communication between each other.

Such considerations are also valid for cases where the communication networks are larger. This happens for example in GNU GCC for bug #29544, reported in Fig. 7(d). In this case there is a big network of connected *G* committers and a very small (only four) fix-inducing committers that are completely isolated like most of the other contributors.

7. 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:

- We use an implementation of the SZZ algorithm (Sliwerski et al., 2005) to identify fix-inducing changes. While SZZ has been used in other pieces of research (e.g., Bavota et al., 2012; Izquierdo-Cortazar et al., 2011; Kim et al., 2008), and while our implementation ignores cosmetic changes or changes to comments, it has been reported by DaCosta et al. how SZZ can still produce inaccurate results (da Costa et al., 2017). One possibility of improving the results, as pointed out

by DaCosta et al. , would be to use the "affected version" field available on issue trackers, however this piece of information might not be always available.
- 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 author of the patch. 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 other, i.e., their communication can still be considered an important factor related to fault-proneness. A related problem is also the migration between versioning systems (CVS/SVN) where committers acted as gatekeepers and distributed versioning systems (git) reporting information about authors. This problem has been discussed in a work by Delorey et al. (2007).
- The mapping between committer IDs and contributor names—explained in Section 3—can be imprecise. We mitigated this threat by manually validating the mapping. Also for all the considered projects the git repository contained the developer full name and email address which simplified the mapping.
- As shown by Aranda and Venolia (2009) for industrial projects, software repositories do not fully capture the rationale around a bug fixing activity; we found this in some cases to be true also for open source projects. For example, in Eclipse there were bug reports referring to IRC communication, e.g., *"Peter asked on IRC whether I wanted the startup patch"* (message in Xserver mailing), *"After consulting on this with some folks in the apache IRC channel"* (bug #34094 in Apache httpd), *"This rather complex testcase was auto-reduced and provided on IRC"* (bug #81525 in GNU GCC). To mitigate this threat to validity additionally to bug reports in issue trackers in our analysis we also consider communication taken place among contributors in mailing lists and in IRC channels.
- The way a social network is built from issue trackers, mailing lists and chats only represents an approximation of what is the real link between people and—as explained by Nia et al. (2010)—heuristics used to reconstruct the network often are cause of omissions and imprecisions. As explained in Section 3.5, when capturing the communication in a thread we followed a conservative approach, privileging an overestimation of links than an under-estimation.
- Finally, we are aware that the linking between commits and bug reports through bug IDs in commit notes is far from being complete (Bachmann et al., 2010), although this represents the techniques mostly adopted in mining software repositories studies. Also, we are aware (Antoniol et al., 2008; Bird et al., 2009a) that there could be issues—not explicitly marked as enhancements—that might not be related to bug fixing.

Conclusion validity concerns the relationship between the treatment and the outcome. As explained in Section 4.2, where opportune, we used appropriate statistical procedures to support claims to address the paper research questions, including non-parametric tests and effect-size measures.

Threats to *internal validity* concern factors that can influence our observations. In this study, we do not claim a cause-effect relationship between the lack of communication—and the introduction of bugs. Rather, we provide statistically significant evidence that committers inducing bug fixes communicate between themselves less than others. In addition, Section 6 complements the quantitative results with qualitative ones, showing examples of discussion networks and explaining why the lack of communication could have induced bug fixes, as well as excerpt from mailing lists or bug reports highlighting evidence of problems due to lack of communication and misunderstanding between developers.

Threats to *external validity* concern the generalization of our findings. This paper reports analyses related to four open source projects having different size, characteristics, and belonging to different domains, considering in all cases discussions from issue trackers, mailing lists and chats. Nevertheless, replication on further projects to confirm or contradict the obtained results is still desirable.

8. Conclusion and work-in-progress

This paper reported an empirical study aimed at analyzing how the "social importance" and the communication level—in issue trackers, mailing lists and chats—of committers relate with their proneness to induce bug fixes. The study has been conducted over a dataset comprising bugs from four open source projects, namely: Apache httpd, GNU GCC, Mozilla Firefox, and Xorg Xserver.

Results indicate that:

- the majority of bugs are fixed by committers who did not induce them, a smaller but substantial percentage of bugs are fixed by committers that induced them, and very few bugs are fixed by committers that were not directly involved in previous changes on the same files of the fix. This indicates the presence of quality assurance people who commit patches and, on the other hand, the absence of persons "jumping in" to help out and solve the problem.
- in general, fix-inducing committers and non fix-inducing committers have a higher social importance in the communication concerning the files impacted by a fix than committers not modifying those files.
- the level of connectedness of fix-inducing committers is significantly lower than for other committers. This indicates that fix-inducing committers communicate less with each other (with respect to non fix-inducing ones) and this likely induces the fixes. The same consideration applies to file owners that induce fixes: also in this case their level of connectedness is significantly lower than that of non file owners.

In summary, while this study agrees with previous studies showing that fix-inducing changes occur in the context of high communication (Abreu and Premraj, 2009) and is seldom correlated with file ownership (Bird et al., 2011), it also suggests that a limited communication between committers could be related to fix-inducing changes. As an implication, increasing the level of communication among fix-inducing committers could reduce the number of induced fixes.

There are several directions for future work. First, we would like to extend the analysis to other sources of information, for example chat histories (where available), to try capturing the developers' communication as much as possible. Also, we would use the information obtained in this paper to better build fault-prediction models, as well as recommenders for warning developers about lack of communication.

References

Abreu, R., Premraj, R., 2009. 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. ACM, New York, NY, USA, pp. 153–158.

Antoniol, G., Ayari, K., Penta, M.D., Khomh, F., Guéhéneuc, Y.-G., 2008. Is it a bug or an enhancement?: a text-based approach to classify change requests. In: Proceedings of the 2008 Conference of the Centre for Advanced Studies on Collaborative Research, October 27–30, 2008, Richmond Hill, Ontario, Canada. IBM, p. 23.

Aranda, J., Venolia, G., 2009. 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, pp. 298–308.

Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 712–721.

Bacchelli, A., Dal Sasso, T., D'Ambros, M., Lanza, M., 2012. Content classification of development emails. In: Proceedings of the 2012 International Conference on Software Engineering. IEEE Press, pp. 375–385.

Bacchelli, A., Lanza, M., Robbes, R., 2010. Linking e-mails and source code artifacts. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1–8 May 2010. ACM, pp. 375–384.

Bachmann, A., Bird, C., Rahman, F., Devanbu, P.T., Bernstein, A., 2010. The missing links: bugs and bug-fix commits. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7–11, 2010. ACM, pp. 97–106.

Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., Strollo, O., 2012. When does a refactoring induce bugs? An empirical study. In: 12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23–24, 2012, pp. 104–113.

Bernardi, M.L., Canfora, G., Di Lucca, G.A., Di Penta, M., Distante, D., 2012. Do developers introduce bugs when they do not communicate? The case of eclipse and mozilla. In: 2011 15th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, pp. 139–148.

Bettenburg, N., Hassan, A.E., 2010. 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, pp. 124–133.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P.T., 2009a. Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24–28, 2009. ACM, pp. 121–130.

Bird, C., Gourley, A., Devanbu, P., Gertz, M., Swaminathan, A., 2006. Mining email social networks. In: Proceedings of the 2006 international workshop on Mining software repositories. ACM, New York, NY, USA, pp. 137–143.

Bird, C., Nagappan, N., Gall, H., Murphy, B., Devanbu, P.T., 2009b. 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, pp. 109–119.

Bird, C., Nagappan, N., Murphy, B., Gall, H., Devanbu, P.T., 2011. 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, pp. 4–14.

Bird, C., Pattison, D., D'Souza, R., Filkov, V., Devanbu, P., 2008a. Latent social structure in open source projects. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 24–35.

Bird, C., Pattison, D., D'Souza, R., Filkov, V., Devanbu, P., 2008b. Latent social structure in open source projects. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 24–35.

Canfora, G., Cerulo, L., Cimitile, M., Di Penta, M., 2011. 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, pp. 143–152.

Cataldo, M., Herbsleb, J.D., Carley, K.M., 2008. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In: Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, October 9–10, 2008, Kaiserslautern, Germany. ACM, pp. 2–11.

Cataldo, M., Wagstrom, P., Herbsleb, J.D., Carley, K.M., 2006. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In: Proceedings of the 2006 ACM Conference on Computer Supported Cooperative Work, CSCW 2006, Banff, Alberta, Canada, November 4–8, 2006, pp. 353–362.

da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A.E., 2017. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. IEEE Trans. Softw. Eng. 43 (7), 641–657. doi:10.1109/TSE.2016.2616306.

Delorey, D., Knutson, C.D., Chun, S., 2007. Do programming languages affect productivity? A case study using data from open source projects. In: First International Workshop on Emerging Trends in FLOSS Research and Development, 2007. FLOSS'07, p. 8.

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

Grissom, R.J., Kim, J.J., 2005. Effect Sizes for Research: A Broad Practical Approach, 2nd edition Lawrence Earlbaum Associates.

Guzzi, A., Bacchelli, A., Lanza, M., Pinzger, M., van Deursen, A., 2013. Communication in open source software development mailing lists. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR'13, San Francisco, CA, USA, May 18–19, 2013. IEEE / ACM, pp. 277–286.

Holm, S., 1979. A simple sequentially rejective Bonferroni test procedure. Scand. J. Stat. 6, 65–70.

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

Kim, S., Whitehead, E.J., Zhang, Y., 2008. Classifying software changes: clean or buggy? IEEE Trans. Softw. Eng. 34 (2), 181–196.

Kim, S., Zimmermann, T., Pan, K., Whitehead, E.J., 2006. 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, pp. 81–90.

Nia, R., Bird, C., Devanbu, P.T., Filkov, V., 2010. 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, pp. 201–209.

Panichella, S., Bavota, G., Di Penta, M., Canfora, G., Antoniol, G., 2014a. How developers' collaborations identified from different sources tell us about code changes. In: 30th International Conference on Software Maintenance and Evolution, ICSME 2014. Victoria, British Columbia, Canada, September 28, October 3, 2014, pp. 251–260.

Panichella, S., Canfora, G., Di Penta, M., Olivieto, R., 2014b. How the evolution of emerging collaborations relates to code changes: an empirical study. In: 22nd International Conference on Program Comprehension, ICPC 2014. Victoria, British Columbia, Canada, September 28, October 3, 2014, pp. 177–188.

Pohl, M., Diehl, S., 2008. 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. ACM, New York, NY, USA, pp. 81–84.

Scott, J.P., 2000. Social Network Analysis: A Handbook, (2nd edition) Sage Publications Ltd, Englewood Cliffs, NJ.

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

Singh, P.V., 2010. The small-world effect: the influence of macro-level properties of developer collaboration networks on open-source project success. ACM Trans. Softw. Eng. Methodol. 20 (2).

Sliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005. ACM.

Wagstrom, P., Herbsleb, J., Carley, K., 2005. A social network approach to free/open source software simulation. In: Proceedings of the 1st International Conference on Open Source Systems. Genova, Italy.

Zhou, M., Mockus, A., 2011. Does the initial environment impact the future of developers. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21–28, 2011. ACM, pp. 271–280.

Mario Luca Bernardi received the Master degree in Computer Science Engineering from the University of Naples "Federico II", Italy, in 2003 and the Ph.D. degree in Information Engineering from the University of Sannio in 2007. Since 2003 he has been a researcher in the field of software engineering and he is author of more than 60 papers published in journals and conference proceedings. His main research interests include software maintenance, evolution and testing with particular interest on software architecture and design. He has served as reviewer of papers submitted to conferences and journals in the field of software engineering, software maintenance and program comprehension.

Gerardo Canfora is a professor of computer science at the Faculty of Engineering of the University of Sannio, Italy. He serves on the program and organizing committees of a number of international conferences. He was general chair of WCRE'06 and CSMR'03, and program co-chair of ICSE'15, WETSoM'12 and '10, ICSM'01 and '07, IWPSE'05, CSMR'04 and IWPC'97. He is co-editor of the Journal of Software: Evolution and Processes (former: the Journal of Software Maintenance, Research and Practice). He has authored more than 200 research papers. His research interests include software maintenance and evolution, security and privacy, empirical software engineering, and service-oriented computing.

Giuseppe A. Di Lucca received the Laurea degree in Electronic Engineering from the University of Naples "Federico II", Italy, in 1987 and the Ph.D. degree in Electronic Engineering and Computer Science from the same University in 1992. He is currently an Associate Professor of Computer Science at the Department of "Ingegneria" of the University of Sannio. Since 1987 he has been a researcher in the field of software engineering and his list of publications contains more than 80 papers published in journals and conference proceedings. His main research interests include software engineering, software maintenance, software testing, reverse engineering, software reuse, software reengineering, software migration, aspect-oriented software development, and web engineering. He serves both as a member of the program and organizing committees of conferences, and as reviewer of papers submitted to some of the main journals and magazines in the field of software engineering, software maintenance and program comprehension.

Massimiliano Di Penta is an associate professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and service-centric software engineering. He is author of more than 230 papers appearing in international journals, conferences, and workshops. He has served as organizing and program committee member of more than 100 conferences such as ICSE, FSE, ASE, ICSM, ICPC, GECCO, MSR WCRE, and others. He has been a general co-chair of various events, including SCAM'10, SSBSE'10, and WCRE'08. Also, he has been a program chair of events such as WCRE'06 and '07, SSBSE'09, ICSM'12, MSR'12 and '13, and ICPC'13, and other workshops. He is currently a member of the steering committee of ICSME, MSR, SSBSE, and PROMISE. Previously, he has been a steering committee member of other conferences, including ICPC, SCAM, and WCRE. He is in the editorial board of the IEEE Transactions on Software Engineering, the Empirical Software Engineering Journal edited by Springer, and of the Journal of Software: Evolution and Processes edited by Wiley. He is a member of the IEEE.

Damiano Distante is Associate Professor of Computer Science at the University of Rome Unitelma Sapienza, Italy. He holds a PhD in Information Engineering and a Master Degree in Computer Science and Engineering from the University of Salento, Italy. His main field of research is software engineering in general and software evolution and web engineering in particular. His research interests include: design and model-driven development of Web applications, evolution of Web systems, e-learning methodologies and technologies, data min- ing and information retrieval techniques. He co-authored more than 50 papers in referred international journals and proceedings of international conferences. He is member of the IEEE Computer Society.