The Journal of Systems & Software 169 (2020) 110667

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Examining the effects of developer familiarity on bug fixing

Chuanqi Wang, Yanhui Li *, Lin Chen, Wenchin Huang, Yuming Zhou, Baowen Xu

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Article history:
Received 1 October 2019
Received in revised form 17 May 2020
Accepted 25 May 2020
Available online 28 May 2020

Keywords:
Familiarity
Bug fixing
Software metrics
Software maintenance and evolution
Efficiency and effectiveness

ABSTRACT

Background: In modern software systems' maintenance and evolution, how to fix software bugs efficiently and effectively becomes increasingly more essential. A deep understanding of developers'/assignees' familiarity with bugs could help project managers make a proper allotment of maintenance resources. However, to our knowledge, the effects of developer familiarity on bug fixing have not been studied.
Aims: Inspired by the understanding of developers'/assignees' familiarity with bugs, we aim to investigate the effects of familiarity on efficiency and effectiveness of bug fixing.
Method: Based on evolution history of buggy code lines, we propose three metrics to evaluate the developers'/assignees' familiarity with bugs. Additionally, we conduct an empirical study on 6 well-known Apache Software Foundation projects with more than 9000 confirmed bugs.
Results: We observe that (a) familiarity is one of the common factors in cases of bug fixing: the developers are more likely to be assigned to fix the bugs introduced by themselves; (b) familiarity has complex effects on bug fixing: although the developers fix the bugs introduced by themselves more quickly (with high efficiency), they are more likely to introduce future bugs when fixing the current bugs (with worse effectiveness).
Conclusion: We put forward the following suggestions: (a) managers should assign some "outsiders" to participate in bug fixing. (b) when developers deal with his own code, managers should assign more maintenance resource (e.g., more inspection) to developers.

1. Introduction

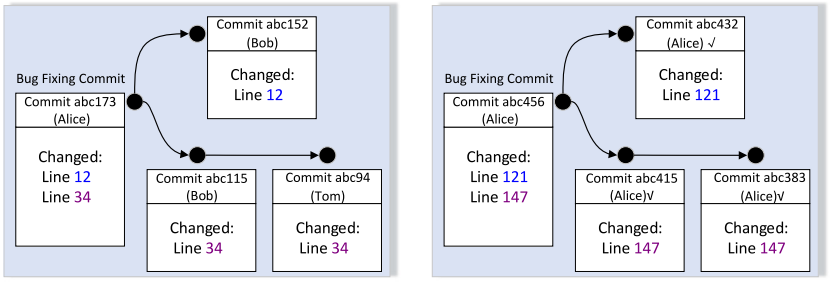
With the increasing requirements of software in modern information society, the scale of software expands and the complexity of software increases, making it tougher for software project managers and engineers to fulfil the task of software maintenance and evolution. To ensure software quality, the software project team commonly provides access for users to submit bug reports when they encounter some operational issues. The managers spend a great part of their resources to handle bug reports and allocate new bugs to developers (Hooimeijer and Weimer, 2007). In the procedure of allocation, a deep understanding of **familiarity** between bugs and developers assigned to fix bugs could help project managers make a proper allotment of maintenance resources. However, to our knowledge, the effects of familiarity on bug fixing have not been studied. Based on our discussions with project managers, they tend to believe that the developer's familiarity with buggy code has effects on the performance of bug fixing, but unfortunately they lack a quantitative manner to measure the familiarity with bug fixing, and consequently fail to conduct a correlational analysis between familiarity, and efficiency and effectiveness on bug fixing.

Interestingly, some aspects of human factors have already been known to be related to software quality or the results of software activities (Bird et al., 2009b; Cataldo et al., 2006; Greiler et al., 2015; Izquierdo-Cortazar et al., 2011; Nagappan et al., 2008). Bird et al. (2011) introduced ownership to measure large-scale software development in Windows 7 and found that higher values of ownership are associated with fewer defects. Rahman and Devanbu (2011) examined the relationship between ownership and defects at a fine-grained level. Thongtanunam et al. (2016) measured the relationship of code ownership with software quality in the scope of code review. Rahman et al. (2017) employed developer experience to predict the usefulness of code review comments.

Our approach is to focus on the code lines that have been modified by developers to fix bugs. These modified lines are considered as a metric to describe the workload for the developer during bug fixing. Moreover, "*how the developer contributed to*

* Corresponding author.
E-mail addresses: chuanqiwang@foxmail.com (C. Wang), yanhuili@nju.edu.cn (Y. Li), lchen@nju.edu.cn (L. Chen), mg1733021@smail.nju.edu.cn (W. Huang), zhouyuming@nju.edu.cn (Y. Zhou), bwxu@nju.edu.cn (B. Xu).

https://doi.org/10.1016/j.jss.2020.110667
0164-1212/



(a) A bug Alice is less familiar with (b) A bug Alice is more familiar with

Fig. 1. An example of the same developer with differently familiar bugs. Names with parentheses are authors of commits.

these modified lines before fixing" is used to measure the familiarity between the developer and the bugs he or she deals with. We assume that the more the developer contributed to these modified lines before fixing, the more the developer is familiar with the bugs. For example, as shown in Fig. 1, Alice is assigned to fix two bugs, and modifies two lines of the buggy file in each bug fixing commit (i.e., Commit abc173 in Fig. 1(a) and abc456 in Fig. 1(b)). By tracking down code evolution, we observe that:

- in the first bug fixing, the two fixed lines 12 and 34 are changed in three previous commits by two different developers Bob and Tom. Alice has no contribution to the history versions of the modified lines;
- on the contrary, in the second bug fixing, all previous commits changing line 121 and line 147 are contributed by Alice.

Based on the obvious difference of code contribution, we find that Alice is more familiar with the second bug.

Generally, our approach consists of three steps to measure the familiarity between developers and bugs, which will be detailedly described in Section 3. First, to confirm bugs and bug fixing commits, we describe the process to link the data from two databases, i.e., JIRA for bug reports and Github for code evolution history; then we track down the bug fixing commit and generate the evolution history of the modified lines in the bug fixing commits; finally we define three familiarity metrics *Recent Familiarity* (RF), *Original Familiarity* (OF), and *Developing Familiarity* (DF) to measure the familiarity when a developer deals with a bug, which focus on the cases that the developers *recently*, *firstly*, and *once* contributed to the buggy lines of the bugs correspondingly. Compared with the existing metrics of human factors (e.g., the ownership metric), our proposed familiarity metrics can distinguish the cases of different bugs fixed by the same developers.[1]

To examine the effects of familiarity on bug fixing, we structure our study by addressing the following four research questions (RQs):

RQ1: What is the distribution of familiarity metrics calculated when developers fix bugs?
We observe that in around 50% cases of bug fixing, developers deal with the buggy lines that were introduced *recently* (*firstly* or *once*) by themselves, according to the calculation of our proposed three familiarity metrics RF, OF and DF.

RQ2: Do the result of three familiarity metrics have much difference?
Based on the classification (high/low familiarity) performance of these three familiarity metrics, there are about 83% bug fixing classified into the same classes. It means that there is not much difference between the classification results of these three familiarity metrics.
We choose the first familiarity metric RF as the delegate to report the result in the following research questions and present the result of the other two familiarity metrics OF and DF in the discussion part.

RQ3: Does the developer's familiarity with buggy code have influence on the performance of bug fixing?
We observe that familiarity has complex effects on bug fixing: developers will fix the familiar bugs more quickly (with higher efficiency), meanwhile they are more likely to introduce new bugs when fixing familiar bugs (with worse effectiveness).

RQ4: Does the familiarity influence on bug fixing remain stable in different fixing periods?
We observe that the familiarity influence on bug fixing fluctuates in different fixing periods. More specifically, the familiarity influences the efficiency of bug fixing the most in around the 4th month and influences the effectiveness the most in around the 2nd month.

From the above, our study shows that during software maintenance and evolution, the developers wrote the commits that introduced the bugs, so they **borrow trouble by themselves** in the future:

- Familiarity is one of the common factors in cases of bug fixing: they are more likely to be assigned to fix the bugs introduced by themselves;
- Familiarity has complex effects on bug fixing: although they fix the bug introduced by themselves more quickly, they are more likely to introduce future bugs when fixing the current bugs.

The rest of the paper is organized as follows. In Section 2, we are going to list the related work in human factors for software quality. In Section 3, we describe three steps to measure our familiarity between developers and bugs. In Section 4, the description of data sets and the definition of the two bug fixing indicators efficiency and effectiveness are given. In Section 5, we present the experimental results according to the aforementioned four RQs. In Section 6, we make a discussion. Threats to validity of our work are presented in Section 7. Section 8 presents the conclusion.

2. Related work

In this section, first we conduct an investigation on one of the most related work "ownership". Furthermore, we present some studies about other human factors on software quality.

[1] This benefit of familiarity metrics will be detailed discussed in Section 6.3.

2.1. Ownership

One of the most related work is "ownership". Bird et al. (2011) used ownership to measure large-scale software development in Windows 7. In their study, the ownership was defined as the ratio of commits a developer made to a certain file. They found that high values of ownership and removal of low-expertise developers are associated with fewer defects. Foucault et al. (2014) replicated Bird's work on open source software and investigated the relation between ownership and module faults. Greiler et al. (2015) extended the previous two ownership studies and defined some file-level metrics to build bug prediction models. Rahman and Devanbu (2011) defined ownership as the highest contributor ratio of the code lines modified by a single developer in a certain file, to measure its impact on software defects. They found that (a) a single developer's contribution is prone to make less implicated code, and (b) an author's specialized experience in a certain file is more important than his general experience. Corley et al. (2012) discovered that the ownership characteristics tend to be the same between similar source code topics. Diaz et al. (2013) indicated that the accuracy of IR-based traceability link recovery could be improved by code ownership information. Orrù and Marchesi (2016) conducted a correlational analysis between refactoring activities and code ownership in Java software systems. In the scope of modern code review, Thongtanunam et al. (2016) measured the relationship of code ownership with software quality. Sedano et al. (2016) examined the factors that affect the sense of code ownership in a team.

Our work is different from previous ownership studies in two aspects.

- First and foremost, we consider that in the bug fixing, the familiarity is a relation between developer and buggy code. Compared with the ownership, the familiarity could be more adjustable and complex. For the same bug, different developers assigned to fix the bug may lead to different familiarity: as shown in Fig. 1, completely familiar or completely unfamiliar developers could be assigned to fix the current bugs. Besides, our proposed familiarity metrics can distinguish the cases of different bugs fixed by the same developers; on the contrary, the ownership metric cannot (see Section 6.3 for detail).
- Second, we consider the familiarity in the line-level from the whole code evolution history of modified lines to fix a certain bug, while prior studies are working on almost file-level or just tracing on the most recent commit. We use the value of "how the developer *recently*, *firstly*, or *once* contributed to these modified lines before fixing" as three familiarity metrics, and check the influence of familiarity on bug fixing performance.

2.2. Other human factors on software quality

Many studies (Bird et al., 2009b; Cataldo et al., 2006; Nagappan et al., 2008) have shown that human factors have a significant influence on software quality. Some previous studies (Brooks, 1995; Curtis et al., 1988) indicated that with the increase of team size, the communication and coordination in the team would be worse. It may slow down the development of software and cause more defects (Cataldo et al., 2006; Espinosa et al., 2001; Herbsleb and Mockus, 2003). The similar result was also found in Windows 7 that when more developers worked on a binary, the failures of it would increase (Nagappan et al., 2008; Bird et al., 2009a). Pinzger et al. (2008) used author contributions to build contribution networks to identify defect-prone modules in Microsoft Vista System. Meng et al. (2013) extended the last commit to some earlier changes by repository graph to calculate authorship. Rahman et al. (2017) used developer experience to predict the usefulness of code review comments. Jiang et al. (2017) proposed a new bug report summarization technique with the authorship characteristics. Other studies (Fritz et al., 2010; Hattori et al., 2012; Robbes and Röthlisberger, 2013) used the data gathered from developer IDE activities to complete the authorship data which can compute developer expertise. Le et al. (2019) observed that in automatic software repair, collaborative effort is needed to distribute the expensive cost.

3. Research methodology

In this section, we will discuss three steps, which are illustrated in Fig. 2, to measure the familiarity between developers and bugs. First, we describe the process to link the data from two different databases JIRA for bug reports and Github for code evolution history; second, we show how to use these data to generate the evolution history of modified lines in the bug fixing commits; finally, based on the evolution history, we define three metrics to measure familiarity when a developer deals with a bug.

3.1. Linking process

Our data is a combination of two data sets: from JIRA we extract the bug reports, and then we link them with corresponding fixing commits on Github to get the evolution history of source code.

JIRA is a proprietary Issue Tracking System (ITS), developed by Atlassian, which provides bug tracking, issue tracking, and project management functions.[2] From the aspect of ITS, JIRA restores bug information as bug reports. Based on these bug reports, first we can extract the `BUG ID` information in these bug reports, which can help us link these bug with the fixing commit from other databases (e.g., Github). Second, we can gain `Createdtime` and `Resolvedtime` of a bug, which can help us calculate how much time developers need to fix the current bug.

Moreover, to get the source code of the bug and the code evolution history, we employ another well-known database Github,[3] which is a web-based Git and a Version Control System (VCS). In Github, based on the commit, we can get the Commitsha from its title and the `BUG ID` from its commit message. Besides, for each fixed line in the commits, we can also extract the developer of the commits to change (fix) this line.

Finally, to synchronize the bug reports from JIRA and commits from Github, we link the bug reports of JIRA with the commits of Github, according to the same `BUD IDs` appearing in both of them. This linking process has been studied by prior studies (Kim et al., 2006b; Zimmermann and Zeller, 2005). There are also many techniques to link the bug reports and the fixing commits in a more complex way, such as ReLink (Wu et al., 2011) and MLink (Nguyen et al., 2012). However, to improve the recall of linking bug report, these techniques mentioned above may affect the precision. In this paper, with the matching of the same `BUG IDs` of bug reports and commits, we can ensure that the commits we link are all bug fixing commits.

3.2. Line-level history generation

In this subsection, first we will give a brief introduction to the Github command `git log -L` used in our experiment. Furthermore, according to the code evolution history generated by the Github command, we define some symbols to facilitate the readers to understand our following experimental steps.

---

2 https://www.atlassian.com/software/jira

3 Williams, Alex (9 July 2012). GitHub Pours Energies into Enterprise Raises $100 Million From Power VC Andreessen Horowitz. TechCrunch.

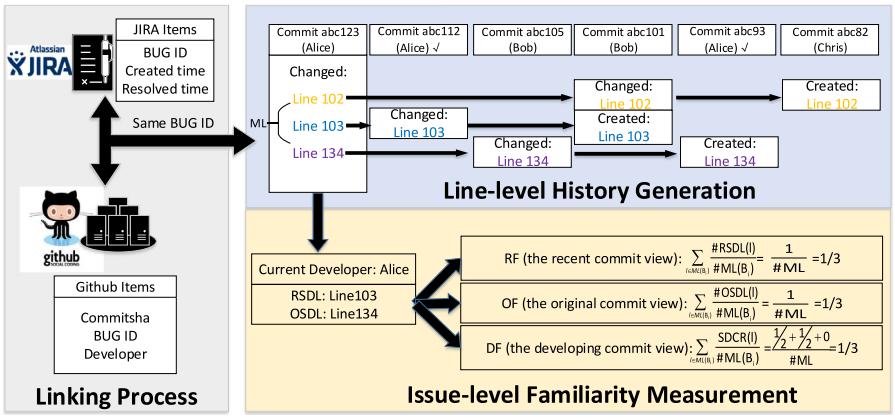


Fig. 2. An Overview of the Three Steps in Our Research Methodology. In the blue part, names with parentheses are authors of commits, e.g., Alice, Bob and Chris. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

3.2.1. Employing git log to trace

After linking bug reports of JIRA to bug fixing commits of Github, we aim to use the current fixing commit to trace down the evolution history of modified lines. For each bug fixing commit, we define the modified lines (MLs) as the hunks that are modified by the developer who deals with the bug. The number (#ML) of MLs is considered as a size metric to describe the workload for the developer during bug fixing.

Then, to trace the evolution history of all MLs in bug fixing commit, we employ the `git log -L` command in Github. For a chosen code line, `git log` can track back to original commit where it was first typed, and show us the whole evolution history of it.[4] The evolution history also can tell us the code line that is modified by which developer. The `git log` command ignores white space changes and detects copy and move, which ensures us to identify the correct code evolution history of each line. In our experiment, supposing that a commit includes several changed lines, first we record the line numbers of all the changed line. Then, for each line number in the commit, we run the `git log` command separately.

In Fig. 3, we give a real-world case of code history evolution of a single line, traced by `git log -L`. The studied line is line 108 in the fixing commit ac061467..[5] by developer AK for BUD FLINK-4149. We extract the code history evolution of line 108 by the following steps:

- The fixing commit deletes line 108 and adds line 206. Through the `git log` tracing, we observe that the deleted line 108 is created in the previous commit 57ef6c31.. by developer TR.
- Commit 57ef6c31.. deletes line 109 and adds line 108[6] with a function argument "false" in function "NFA⟨⟩()". We track from line 109 in the new commit 57ef6c31.. again.
- In the end, the whole code evolution history of fixing line 108 includes three commits, and the original commit e3759a5e.. illustrates when the buggy line was first introduced.

3.2.2. Symbols and examples

Here, we introduce some symbols to facilitate the readers to understand our following extraction steps.

- For a given bug (B_i), the fixing commit is denoted as FC(B_i). For a commit C_i, the developer (DEV) of C_i is denoted as DEV(C_i). Hence for a bug (B_i), the developer who deals with the bug is denoted as DEV(FC(B_i)). The modified lines in the fixing commits of B_i is denoted as ML(B_i).
- For any line $l \in$ ML(B_i), we can use `git log -L` to track the whole evolution history of l with a commit sequence starting with the fixing commit FC(B_i):

  $$FC(B_i) \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n$$

  where $C_1, \ldots, C_n$ are the commits which modify the line l before the bug fixing, and the first commit C_1 and the last commit C_n are called the recent commit (RC(l)) and the original commit (OC(l)), respectively.

Based on these symbols, we employ three views to describe how the developer who fixed the bugs contributes to the evolution of a fixed line $l \in$ ML(B_i):

- Recent View. The Recent Same DEV Lines (RSDLs) can be defined as the code lines with the same developer of the bug fixing commit and the recent commit: DEV(RC(l)) = DEV(FC(B_i)). It is the case that the developers are fixing the code line that they modified *recently*.
- Original View. The Original Same DEV Lines (OSDLs) can be defined as the code lines with the same developer of the bug fixing commit and the original commit that first created the code lines: DEV(OC(l)) = DEV(FC(B_i)). It is the case that the developers are fixing the buggy code that was *firstly* introduced by them.
- Developing View. The Same DEV Commits (SDCs) are defined as the previous commits C_i ($1 \leq i \leq n$) with the same developer of the bug fixing commit: DEV(C_i) = DEV(FC(B_i)). It is the case that the developers are fixing the code line that was *once* modified by them. Moreover, #SDCs(l) denotes the number of SDCs for the chosen line l. The Same DEV Commits Ratio (SDCR(l)) can be calculated as:

  $$SDCR(l) = \frac{\#SDCs(l)}{\text{the number of all previous commits changing } l}$$

As an example, Fig. 2 presents the Line-level History Generation part from the code evolution history of each line modified by the bug fixing commit abc123. There are 3 modified lines in commit abc123 for BUG-456, and the developer who deals with this bug is Alice. The number of the developers are 3 for Alice, Bob, and Chris (names are with parentheses) in this example.

[4] Another code evolution analysis tool is the `git blame` command, it was more well-known and frequently used by developers (Rahman and Devanbu, 2013). We choose to use `git log -L` instead of `git blame`, as it gives the whole evolution history which has covered the `git blame` command.

[5] For brevity, we use an abbreviation of commit IDs, which we list completely in Fig. 3. This commit modified 76 lines totally, and in this case we just focus on line 108.

[6] The line number changes from 109 to 108 on account of the other changes in this commit.

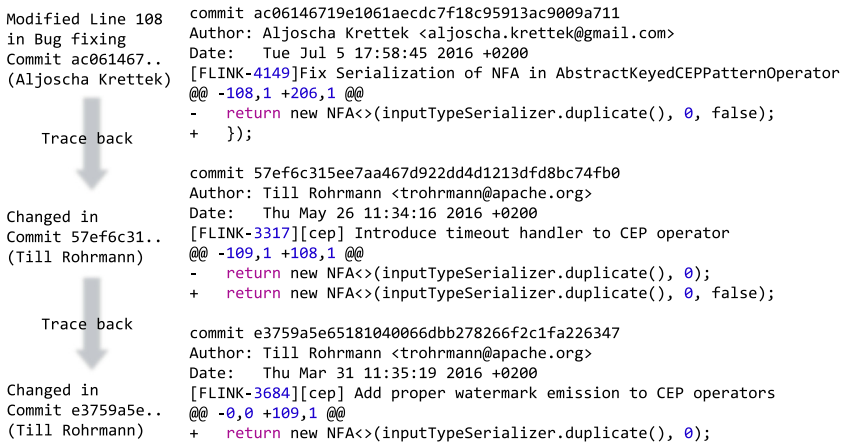


Fig. 3. Line 108 in the commit ac061467.. is traced by `git log -L`, from which we construct the evolution history of it: commit ac061467..→57ef6c31..→e3759a5e..

- When we track down the evolution history of line 102 (highlighted in brown), the recent commit that modified this line is commit abc101 given by Bob, and the original commit is commit abc82 given by Chris. Alice does not contribute to line 102 in the past commits, which means that line 102 is neither a RSDL nor a OSDL. Thus, we may consider that Alice is not familiar with line 102.
- For line 103 (highlighted in blue), Alice contributes to it with the recent commit (line 103 is labelled as a RSDL), which means that she is familiar with this line in the view of recent measurement.
- Finally, Alice contributes to line 134 (highlighted in purple) with the original commit (line 134 is labelled as a OSDL), which means that Alice is familiar with the created version of the buggy code line.

3.3. Familiarity measurement

According to the code evolution history of a modified line $l \in \mathrm{ML}(B_i)$ in bug fixing commits, we can define three familiarity metrics to measure the developer's familiarity with the bug (B_i):

- **Recent Familiarity (RF)** — For a fixed bug (B_i), RF describes how much percent of code lines are the cases that the developer is fixing the code line he or she modified *recently*. It can be calculated as:

$$\mathrm{RF} = \frac{\#\mathrm{RSDLs}}{\#\mathrm{ML}(B_i)}$$

- **Original Familiarity (OF)** — For a fixed bug, OF describe how much percent of code lines are the cases that the developer is fixing the code line that they introduced *firstly*. It can be calculated as:

$$\mathrm{OF} = \frac{\#\mathrm{OSDLs}}{\#\mathrm{ML}(B_i)}$$

- **Developing Familiarity (DF)** — For a fixed bug, the DF can be calculated as the average of all SDCRs of MLs:

$$\mathrm{DF} = \frac{\sum_{l \in \mathrm{ML}(B_i)} \mathrm{SDCR}(l)}{\#\mathrm{ML}(B_i)}$$

  It shows the developer's familiarity with the developing history of the fixed bug, based on the cases that the developer is fixing the code that they modified *once*.

Familiarity Measurement part in Fig. 2 describes an example of calculating the familiarity metrics. In this figure, we can see that the number of RSDLs equals to 1 for line 103, and the number of OSDLs equals to 1 for line 134. Therefore, RF can be calculated as the number of RSDLs divided by the number of MLs, which equals to 1/3, while OF can be calculated as the number of OSDLs divided by the number of MLs, which equals to 1/3. The SDCR of line 103 equals to 1/1, that of line 134 equals to 1/2, and that of line 102 equals to 0/2. Thus, DF can be calculated as the average of SDCRs of line 103, line 134 and line 102: $(1 + 1/2 + 0)/3 = 1/2$.

4. Experiment setup

4.1. The data sets

In this study, we collect bug reports of 6 projects from Apache Software Foundation (ASF) that are all managed in JIRA ITS. The projects of Apache Software Foundation have well-structure issue tracking system in JIRA and version control system in Github. Moreover, these projects are from diverse application domains (e.g., data serialization and web application), and each of them has over 1000 bug reports in JIRA. From the source code and commits of these 6 projects in Github, we can obtain their code evolution history which can be used to calculate the familiarity metrics.

Table 1 presents the detail of the 6 studied projects. The first and second columns are the names and the description of them. The third column shows the date from the starting time of the first linked commit to the end time (August 2017) of our data collection. It shows that the project lifecycle ranges from three years to over ten years. The fourth column lists the numbers of the modified lines in all bug reports of these six projects, which are from 24 KLOC (thousands lines of code) to 182 KLOC. The fifth column shows the numbers of all bug reports (Bug(A)) available in JIRA. The sixth column lists the numbers of the linked bug (BUG(L)) that can be linked with the same `BUG IDs` between JIRA and Github. The result shows that, after the linking data process, most (9111 out of 15409, about 60%) of the bugs can be linked between JIRA and Github. The seventh column presents the numbers of distinct developers in the code evolution history of all linked bugs. The last column includes the numbers of distinct commits in the code evolution history of all linked bugs.

To sum up, we have totally over 9000 linked bugs. The number of commits in these bug reports' evolution history is over 29000. These commits include over 374KLOC modified lines and involve about 1200 distinct developers. These data sets are sufficient to support our following experiment. In the following experiment, the project "ALL" represents the whole data of 6 projects and the single project name represents the data of the single project.

Table 1
The 6 studied project.

Name	Description	Date	ML	Bug(A)	Bug(L)	Developer	Commits
NiFi	NiFi is a software project which enables the automation of data flow between systems.	2014.12–2017.8	38K+	1843	935	163	3059
Avro	Avro is a data serialization system.	2009.4–2017.8	31K+	1094	926	414	1386
Bigtop	Bigtop is a tool for comprehensive packaging, testing, and configuration of the leading open source big data components.	2011.7–2017.8	45K+	2195	1007	129	1868
Flink	Flink is an open source stream processing framework.	2014.6–2017.8	182K+	2763	2740	399	7552
Tuscany	Tuscany is software project for developing and running software applications using a service-oriented architecture.	2006.4–2017.8	24K+	1993	672	27	3970
Wicket	Wicket is a lightweight component-based web application framework for the Java programming language.	2006.10–2017.8	52K+	5521	2831	67	11595
Totals			374K+	15409	9111	1199	29430

4.2. Fixing efficiency and fixing effectiveness

In this section, to measure the familiarity influence on bug fixing, we introduce two indicators: fixing efficiency and fixing effectiveness.

Fixing Efficiency. From JIRA, we calculate the fixing effort of bugs as the time-window between `Createdtime` and `Resolvedtime` of them, which are widely used in previous works (Akbarinasaji et al., 2018; Assar et al., 2016; Thung, 2016). Moreover, as the number (#ML) of modified lines can be considered as a size metric to describe the workload for the developer during the bug fixing. We calculate the fixing efficiency of each bug as average fixing effort:

$$\text{average fixing effort} = \frac{\text{fixing effort}}{\#\ \text{ML}}$$

It shows that when the developer deals with a bug, in average sense how much fixing effort will be spent in fixing a single buggy line. Note that *lower* average fixing effort means *higher* efficiency.

The reason why we use the average fixing effort is that, within a project, the numbers of fixing code lines vary a lot in different bugs. To exclude the influence of the size factor on fixing efficiency, we use the average fixing effort. Moreover, in the view across projects, the differences of the numbers of modified lines may even enlarge. We also utilize the average fixing effort to compare the fixing efficiency of bugs in different projects fairly.

Fixing Effectiveness. From the scope view of a popular bug-introducing changes identification approach proposed by Sliwerski, Zimmermann and Zeller ("SZZ" for short) (Sliwerski et al., 2005) and its complementary work (Kim et al., 2006a; Misirli et al., 2016; Sinha et al., 2010), the bug-inducing commit is defined as the last commit of buggy code lines. This inspires us to measure fixing effectiveness by checking whether the bug fixing commit will be considered as a bug-inducing commit for a new bug in the future. If the answer is "Yes", it means that the fixing procedure is incorrect or incomplete. The bugs with these incorrect or incomplete bug fixing commits are labelled as "re-fixed".

Fig. 4 presents an example of how to identify these incorrect or incomplete fixing commits. The old Bug ZXC-1026 in JIRA is linked with the bug fixing commit abc123 in Github. Commit abc123 modifies line 102 that is fixed by another bug fixing commit abc156 linked with a new Bug ZXC-1053. From the above definition, commit abc123 is considered as incorrect or incomplete, and bug ZXC-1026 is considered as re-fixed, which means the effectiveness of fixing ZXC-1026 is worse.

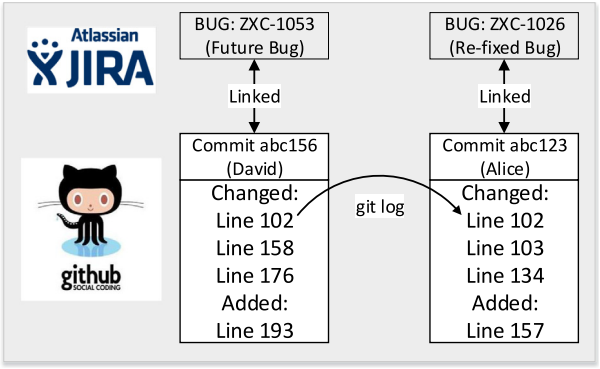


Fig. 4. An example to identify the incorrect or incomplete bug fixing.

4.3. High/low familiarity classification

Based on these three familiarity metrics mentioned in Section 3.3, we can classify the developer's familiarity with a bug fixing into two classes. Specifically, we use the median values[7] of three familiarity metrics RF, OF and DF in our data sets ('ALL' and 6 single project) to classify bug fixing into two groups as low-familiarity (equal to or less than the median) and high-familiarity (greater than the median). We argue that this progress is general and can split the data as uniform as possible in different projects, i.e., the percent of low-familiarity group/high-familiarity group is almost the same (about 50%). This uniform partition will be used to compare different influence with different familiarity classes in the following RQs of Section 5.

[7] The median in the data set is the value such that half of the data set is less than the median and half is greater than it.

In the following paper, to simplify the expression of familiarity, when we say a bug fixing with high/low familiarity (i.e., in high/low familiarity group), it means that the developer who solves this bug currently is more/less familiar with the buggy code based on our familiarity classification.

5. Experiment results

We begin with an overall statistic of the distribution of RF, OF, and DF on 6 studied projects in RQ1. Moreover, in RQ2, we have an investigation on the classification difference of three familiarity metrics in high-familiarity group and low-familiarity group. Then, in RQ3, we compare the developers' performance (i.e., fixing efficiency and effectiveness) in different bug fixing groups with high familiarity and low familiarity. Finally, we focus on whether the influence of familiarity on bug fixing is stable in different fixing periods in RQ4.

RQ1: What is the distribution of familiarity metrics calculated when developers fix bugs?

As we have extracted the whole evolution history of bug fixing to calculate the familiarity metrics, it makes sense to have an investigation on the distribution of these three familiarity metrics RF, OF, and DF in the 6 studied projects.

Fig. 5(a) gives the ratio of developers with different RF. In view of the recent commit, RF = 0 (grey block) means all the modified lines that the current developer deal with are not his own recently modified code; 0 < RF < 1 (orange block) means that a part of the modified lines that the current developer deal with are his own recently modified code; correspondingly RF = 1 (red block) means the current developer just deals with his own recently modified code.

From Fig. 5(a), we can find that in around 50% cases of the bug fixing, developers deal with the buggy lines that were introduced *recently* by themselves: (a) around 50% bug fixing (orange block plus red block in Fig. 5(a)) are the cases that the developers fix the buggy code that includes at least one line created by themselves recently. (b) around 25% bug fixing (red block in Fig. 5(a)) are the cases that the developer is fully fixing his own recently modified code.

From the familiarity of the original view, Fig. 5(b) gives the ratio of developers with different familiarity measured by OF. We have observed the similar findings. The percentage of "OF > 0" is around 50% (orange block plus red block in Fig. 5(b)), which means in the view of original measurement, most developers fix some of the buggy code that was *firstly* created by them. Although the OF = 1 group is a little less than the RF = 1 group, there are still 20% bug fixing (red block in Fig. 5(b)) when the developers fix the buggy code that totally firstly created by themselves.

From the familiarity of developing history of the code line, Fig. 5(c) presents the ratio of developers with different familiarity measured by DF. Over 60% bug fixing (orange block plus red block in Fig. 5(c)) are the cases that the current developer *once* participated in the previous commits of the buggy code lines. Moreover, the 0 < DF < 1 (orange block) part is much larger than the corresponding part of RF and OF. It means that in the developing familiarity view, there are more developers are dealing with their own code. From the definition of DF, DF provides a long-term view, compared with the other two metrics RF and OF, which causes that for most bugs, the developers assigned to fix them contributed, but not totally, to the previous versions of bug lines. Therefore, in Fig. 5, the percent of the cases with 0 < DF < 1 is much more than that of the other two metrics. Finally, there are around 10% bug fixing (red block in Fig. 5(c)) when all the commits in evolution history are contributed by only one developer.

> Familiarity is one of the common factors in cases of bug fixing. We observe that in around 50% of the bug fixing, developers deal with the buggy lines that were introduced *recently* (*firstly* or *once*) by themselves according to our proposed three familiarity metrics RF, OF, and DF. This result also indicates that, although managers lack a quantitative manner to measure the familiarity, they tend to assign familiar developers to fix the current bugs.

RQ2: Do the three familiarity metrics have much difference?

As we have defined three familiarity metrics RF, OF, and DF in Section 3, it is essential to have an investigation on the classification of bug fixing in the view of each metric. It can illustrate whether these three metrics vary a lot from each other.

Table 2 gives the classification of three familiarity metrics on the six studied projects. In Table 2, the symbol "L" means the bug fixing has been classified into **l**ow-familiarity class, where the symbol "H" means it has been classified into **h**igh-familiarity class.[8] In Table 2, the classification results of bug fixing are listed as triples containing "L" or "H" in the order of the classification according to RF, OF, and DF. If a bug fixing has been classified into the same familiarity classes with three familiarity metrics (labelled as "HHH" or "LLL"), this bug fixing will be labelled as "no difference". The percent of the "no difference" labels tells us that most of the bug fixing activities have been classified into the same familiarity classes, which are ranged from 76% to 92% for a single project and 83% for all 6 studied projects.

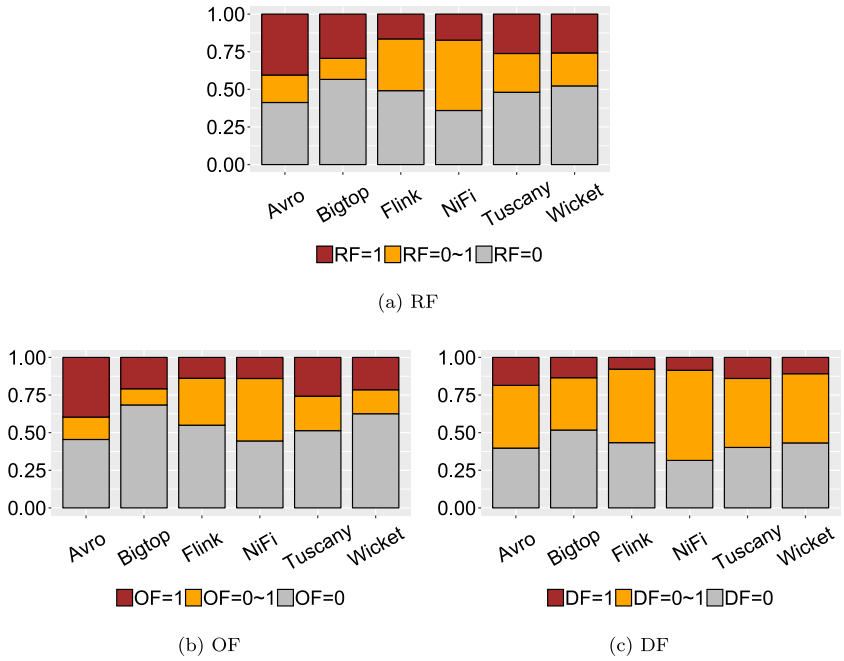
To examine the significance of the association between these familiarity classifications (i.e., RF vs OF, RF vs DF and DF vs OF), we present a statistical analysis of Fisher's exact test (Fisher, 1992) on the classification results of the high and low familiarity groups according to different familiarity. Fisher's exact test calculates the *p*-value for evaluating the correlation of the familiarity group. In the last two rows of Table 2, the result shows that the *p*-values of Fisher's exact test are very low ($\ll 0.001$), which indicates that, there would be a statistically significant association between these two familiarity classifications.

In the following RQs, we choose the RF metrics as the delegate to report the result of our research.[9] It is for three reasons:

- First and foremost, as several studied projects have long evolution history, some of them are migrated from other version control systems to Github. The bug commits with long evolution history will lose some information through migration. Thus, using the most recent indicator RF to measure the familiarity is more accurate.
- Moreover, to get the most recent indicator RF, we only need to trace one step or one commit in the code evolution history. This one step tracing is applied in many previous studies (Sliwerski et al., 2005; Kim et al., 2006a; Misirli et al., 2016; Sinha et al., 2010), which means the RF metric based on recent commit is more accessible.
- In the above analysis, we have an investigation on the difference of the familiarity metrics RF, OF and DF. The result tells us that there is not much difference in classification between them.

[8] The detail of classification is presented in Section 4.3.

[9] We will present the result of the other two familiarity metrics OF and DF in Section 6.1.



(a) RF

(b) OF

(c) DF

Fig. 5. The distribution of 3 metrics RF, OF and DF on 6 studied projects. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 2
Detail of three metrics familiarity classification in each project.

Project	NiFi	Avro	Bigtop	Flink	Tuscany	Wicket	ALL
LLL	0.399	0.463	0.517	0.447	0.421	0.448	0.437
LLH	0.004	0.002	0.027	0.018	0.016	0.026	0.011
LHL	0.055	0.026	0.000	0.010	0.028	0.005	0.031
HLL	0.027	0.015	0.000	0.019	0.019	0.037	0.021
LHH	0.043	0.014	0.022	0.025	0.036	0.044	0.028
HLH	0.071	0.021	0.140	0.065	0.057	0.114	0.068
HHL	0.020	0.002	0.000	0.023	0.033	0.013	0.013
HHH	0.382	0.457	0.294	0.392	0.390	0.313	0.392
No difference class (LLL+HHH)	0.781	0.920	0.811	0.839	0.811	0.761	0.830
RF vs OF (p-value)	$\ll$0.001	$\ll$0.001	$\ll$0.001	$\ll$0.001	$\ll$0.001	$\ll$0.001	$\ll$0.001
RF vs DF (p-value)	$\ll$0.001	$\ll$0.001	$\ll$0.001	$\ll$0.001	$\ll$0.001	$\ll$0.001	$\ll$0.001
OF vs DF (p-value)	$\ll$0.001	$\ll$0.001	$\ll$0.001	$\ll$0.001	$\ll$0.001	$\ll$0.001	$\ll$0.001

Based on the three familiarity metrics, there are about 83% bug fixing classified into the same classes. It means that there is not much difference between the classification of them. We choose the RF as the delegate to report our following research in RQ3 and RQ4.

RQ3: Does the familiarity have influence on the performance of bug fixing?

As we discussed in RQ1, when the developers deal with the buggy code lines, a large part of them are introduced by the developers themselves. As the above observation "familiarity is usual", we will discuss the influence of familiarity on the performance (efficiency and effectiveness) of bug fixing.

Fig. 6 shows the boxplots of the efficiency of low-familiarity and high-familiarity bug fixing. In average sense, the developers have much higher efficiency in fixing the bugs they are familiar with. Furthermore, when we check the comparison for each single project, the result shows that the high-familiarity developers also fix the bugs faster (with low average fixing effort). It tells us that the high-familiarity bug fixing has the higher working efficiency.

The efficiency (average fixing effort) of high-familiarity bug and low-familiarity bug are shown in Table 3. The first column is the project name. The second and the third columns give the average fixing efficiency to fix a high-familiarity or low-familiarity bug in each project. To compare the difference of fixing efficiency between high-familiarity group and low-familiarity group, we define Fixing Efficiency Ratio which equals to the average fixing efficiency of low-familiarity bug fixing divided by that of high-familiarity bug fixing. The fourth column shows the result of the Fixing Efficiency Ratio. In the fifth column, we present the result of the Wilcoxon Signed-rank Test (Wilcoxon, 1946) (the null hypothesis states that "the efficiency of high-familiarity bug group is equal to that of low-familiarity bug group"). All the p-values are very low ($<$0.05), which are conclusively rejecting the null hypothesis. Compared to the high-familiarity bug fixing, the average fixing effort of the low-familiarity bug fixing is 1.599 times higher on average for all projects.

For the measurement of effectiveness, we also calculate the re-fixed ratio of bugs in the two familiarity group. The re-fixed

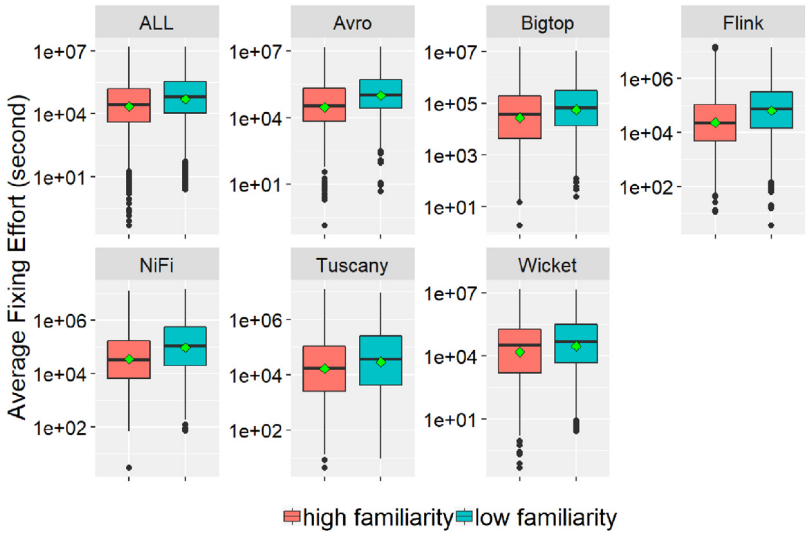


Fig. 6. The boxplot of fixing efficiency (average fixing effort) in high-familiarity and low-familiarity bug fixing, grouped by median RF (*lower values means higher efficiency*). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 3
Fixing efficiency in high-familiarity (High) and low-familiarity (low) bug fixing (*Lower values means higher efficiency*).

Project	High(second)	Low(second)	Fixing efficiency ratio	*p*-value
NiFi	399,902	530,959	1.328	= 0.001
Avro	326,098	742,081	2.276	≪0.001
Bigtop	361,664	466,359	1.289	≪0.001
Flink	243,355	499,586	2.053	≪0.001
Tuscany	378,510	435,197	1.150	= 0.015
Wicket	402,004	554,320	1.379	≪0.001
ALL	337,997	540,388	1.599	≪0.001

ratio of a group of bug fixing is defined as:

$$\text{Re-fixed Ratio} = \frac{\text{the number of re-fixed bugs}}{\text{the number of all bugs}}$$

Table 4 shows the re-fixed ratio of the studied 6 projects. To compare the difference of fixing effectiveness in high-familiarity group and low-familiarity group, we define a Fixing Effectiveness Ratio which equals to the Re-fixed Ratio of high-familiarity bug fixing divided by the Re-fixed Ratio of low-familiarity bug fixing. The result is contrast to our intuition,[10] in the view of all projects, the re-fixed ratio of low-familiarity bug fixing is 0.432 and that of high-familiarity bug fixing is 0.604. Moreover, the fixing effectiveness ratio of each project ranges from 1.200 to 2.400 (all over 1). The *p*-value of Wilcoxon Signed-rank Test on fixing effectiveness of high-familiarity group and low-familiarity group is less than 0.05. It tells us that when developers deal with the buggy code that they are more familiar with, the fixing effectiveness will decrease and these fixed bugs are more likely to be re-fixed in the future.

> Familiarity has complex effects on the bug fixing: developers will fix the familiar bugs more quickly, meanwhile they are more likely to introduce new bugs when fixing familiar bugs.

[10] Our intuition tells us that higher familiarity may cause better fixing performance.

Table 4
Re-fixed ratio in high-familiarity (high) and low-familiarity (low) bug fixing (*Lower values means better effectiveness*).

Project	High	Low	Fixing effectiveness ratio
NiFi	0.588	0.245	2.400
Avro	0.684	0.513	1.333
Bigtop	0.558	0.432	1.290
Flink	0.756	0.503	1.505
Tuscany	0.387	0.220	1.754
Wicket	0.501	0.418	1.200
ALL	0.604	0.432	1.397

RQ4: Does the familiarity influence on bug fixing remain stable in different fixing periods?

The former result shows that developer's familiarity with fixing code has a large influence on the performance of fixing bugs. In software development, we observe that different bugs may have different fixing periods (i.e., different resolution time required for bugs of different difficulty): for some tricky problems, developers will spend a lot of days or weeks to deal with; and for some trivial bugs, they only take several days to fix. There are many previous studies (Akbarinasaji et al., 2018; Assar et al., 2016; Zhang et al., 2013) on this topic, which construct prediction models to predict the fixing periods of the incoming bug fixing tasks. Thus, we intend to investigate whether the familiarity effect on the bug fixing remains stable as they have different fixing periods. Moreover, if it does not remain stable, it is essential to find out in which fixing period the familiarity influences the most. The result of this RQ can help team managers assign the bug fixing tasks with different fixing periods to the proper developers.

To find out the familiarity influence on bug fixing with different fixing periods, we group the studied data into 6 parts as the 1st-month (1–30 days), the 2nd-month (31–60 days), …, the 5th-month (121–150 days), and over 5 month (≥151 days). Fig. 7 shows the fixing efficiency ratio of low-familiarity divided by high-familiarity in the 6 different periods. The result tells us that the familiarity effect on bug fixing efficiency fluctuates in different fixing periods. Furthermore, the peak of the boxplots is the 4th month, which means that the familiarity has the most influence on the bugs of fixing period from the 91st day to the 120th day. In the view of all projects, the average fixing efficiency of high-familiarity is 2.397 times (the green point of the 4th

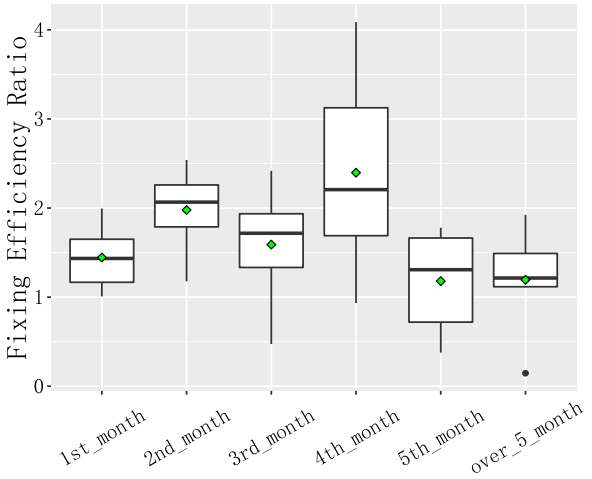


Fig. 7. The boxplot of fixing efficiency ratio in 6 periods. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

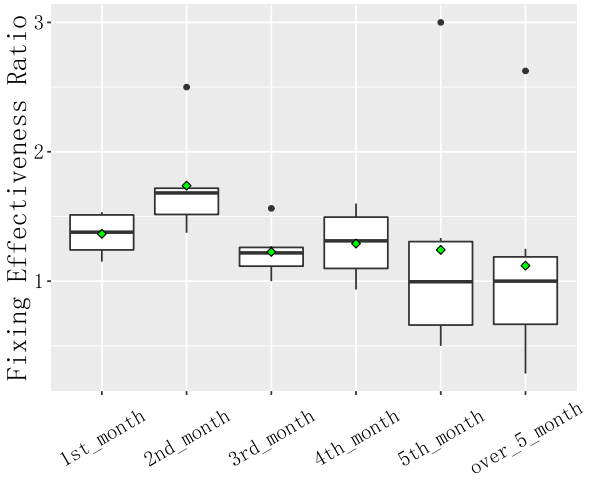


Fig. 8. The boxplot of fixing effectiveness ratio in 6 periods. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

month boxplot in Fig. 7) higher than low-familiarity developers during the 4th month. We also present the Wilcoxon Signed-rank Test on the data of fixing efficiency from the 4th month and the other month data. The result shows that the p-value $= 0.039 < 0.05$, which shows that the influence on fixing efficiency during the 4th month is significantly different from that during other fixing periods. It tells us that, to achieve the most efficient bug fixing, we should assign the bugs in this fixing period to high-familiarity developers.

Fig. 8 shows the fixing effectiveness ratio of high-familiarity divided by low-familiarity in the same 6 different periods. It shows the similar trend with the measurement of efficiency: the familiarity effect on bug fixing effectiveness fluctuates in different fixing periods. Specifically, the familiarity has the most influence on the bugs of fixing period in the 2nd month (around the 31st day to the 60th day). The p-value of Wilcoxon Signed-rank Test of fixing effectiveness on the 2nd month data and the other month data is 0.004, and the effectiveness of low-familiarity group performs (1.74 times) better than high-familiarity group. It tells us that the influence of familiarity on fixing effectiveness during the 2nd month is significantly different from that during other fixing periods. Based on the above observation, managers should

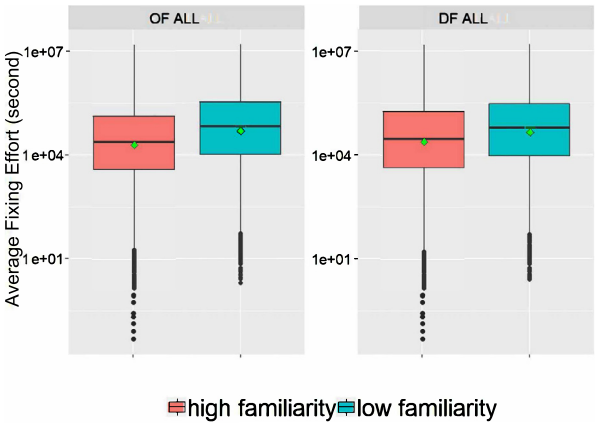


Fig. 9. The boxplot of fixing efficiency in high-familiarity and low-familiarity bug fixing, grouped by the median of OF and DF (*lower values means higher efficiency*). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

assign more maintenance resource to the bugs fixed by familiar developers in this fixing period, to improve the correctness of bug fixing.

> The fixing influence on bug fixing fluctuates in different fixing periods. The familiarity influences the efficiency the most in around the 4th month and influences the effectiveness the most in around the 2nd month.

6. Discussion

In this section, we start with the discussion about the influence of the other two familiarity metrics OF and DF on the performance of bug fixing. We intend to find whether these metrics have similar results with the result mentioned above of RF metric in RQ3. Moreover, we present a case study to show re-fixed bugs and a case study to show the benefits of familiarity metrics in measuring the developer's familiarity with buggy code. Furthermore, we exclude the confounding effects and investigate if our conclusion on RQ3 is the same. Besides, we try to give some explanations to the experiment result of developer familiarity's effects on fixing efficiency and fixing effectiveness. Finally, we provide some recommendations for utilization of maintenance resources and improvement of software quality.

6.1. Results of the other familiarity metrics OF and DF

In Section 5, we have discussed the classification of the three familiarity metrics. At the end of RQ2, as these metrics do not have much difference from each other in classification, we choose to use RF to report the experiment result of RQ3. In this section, we conduct the same experiments under the other two familiarity metrics OF and DF.

Fig. 9 shows the boxplots of fixing efficiency of high-familiarity and low-familiarity bug fixing, which are grouped by the medians of OF and DF. In the view of all 6 studied projects, the fixing efficiency ratio is 2.09 for OF and 1.54 for DF. It is similar to the result of RF in the previous experiment that developers with high-familiarity of the buggy code would fix the bug with high efficiency.

In the aspect of effectiveness, we also have an investigation on the re-fixed ratio in high-familiarity and low-familiarity bug fixing. Fig. 10 gives the boxplots of the re-fixed ratio of the

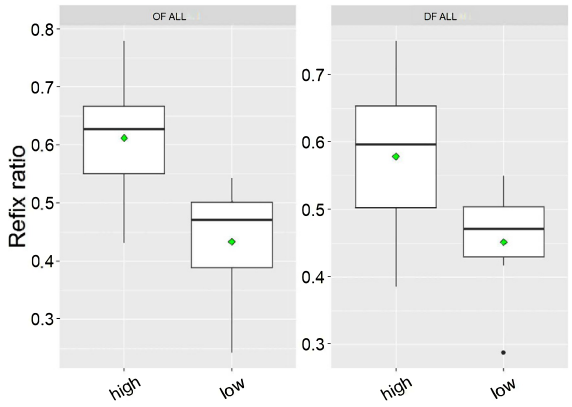


Fig. 10. The boxplot of fixing effectiveness in high-familiarity and low-familiarity bug fixing, grouped by the median of OF and DF, (*lower values means better effectiveness*). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

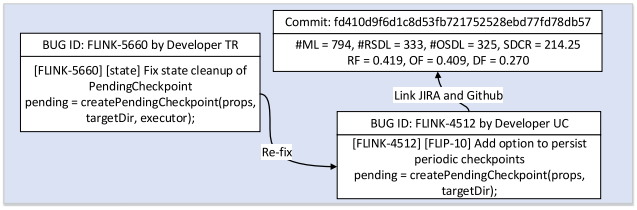


Fig. 11. A real-world case of re-fixed bug fixing.

studied 6 projects. Similar to the RF result, the result of two metrics OF and DF also tells us that the developer deal with high-familiarity bugs with worse fixing effectiveness.

From above, we can conclude that the influence of OF and DF on efficiency and effectiveness of bug fixing shows similar results to that of RF.

6.2. A case study of re-fixed bug fixing

Fig. 11 presents a real case of re-fixed bugs. It shows that developer UC resolved a bug FLINK-4512[11] "Add option to persist periodic checkpoints". By linking this bug from JIRA to its commit in Github, we find that commit fd410d9f.. is the kind of bug fixing commits when the developer deals with his own code. It is a bug fixing with high-familiarity where the RF equals to 333/794 = 0.419, the OF equals to 325/794 = 0.409 and the DF equals to 0.270. Moreover, some of the modified code lines in FLINK-4512 was fixed by another bug fixing commit for a new bug FLINK-5660[12] which reports that "Fix state cleanup of PendingCheckpoint".

To conclude, in this example, the bug fixing with high-familiarity causes re-fixing, which illustrates that the bug fixing with high-familiarity is considered to have not good performance in effectiveness measurement.

6.3. Benefits of familiarity metrics (vs. ownership)

Fig. 12 presents a real case to show the benefits of familiarity metrics. In this case, developer TR contributes 46 lines in the file PendingCheckpointTest.java (394 lines in total). Moreover, using the buggy code's changing history, we can calculate the familiarity metrics. When dealing with the two different bugs (bug FLINK-5085 and bug FLINK-5660) in this file, the developer TR has the same experience in ownership measurement.[13] However, when using familiarity metrics, we can find that the developer TR's familiarity with the bug FLINK-5085 is different from that of bug FLINK-5660 in the familiarity metrics' view. In detail, in the view of bug FLINK-5085, the RF equals to 0.25, the OF equals to 0.25, and the DF equals to 0.26, respectively. In contrast, the RF, OF and DF of bug FLINK-5660 are 0.2, 0 and 0.033, respectively. It tells that when dealing with the bug FLINK-5085, the developer has higher familiarity than dealing with the bug FLINK-5660.

To conclude, our proposed familiarity metrics can distinguish the cases of different bugs fixed by the same developers; on the contrary, the ownership metric cannot.

6.4. Excluding the confounding effects of fixing effort

In the previous RQs, we have presented the familiarity metrics' effects on both fixing efficiency and fixing effectiveness. It is worthwhile pointing out that the fixing efficiency (i.e., average fixing effort) is a common factor that may affect the fixing effectiveness. However, we did not mention any concerns with the potential confounding effects of fixing efficiency on fixing effectiveness, which may affect even further the validity of the results. Fig. 13 shows the path diagram of the confounding effect of fixing efficiency on the association between familiarity metrics and fixing effectiveness. Our familiarity metrics could be considered as an independent variable, and the familiarity metrics may affect the dependent variable fixing effectiveness through the third variable fixing efficiency.

In this section, we first build a linear regression model[14] (Zhou et al., 2014) to remove the confounding effect, and check whether our conclusions of the previous RQ3 (the familiarity's effects on fixing effectiveness) will change after excluding the potentially confounding effect of the third variable fixing efficiency.

Table 5 shows the re-fixed ratio of the 6 studied projects after excluding the potentially confounding effect. In the view of all projects, the re-fixed ratio of low-familiarity bug fixing is 0.461, and that of high-familiarity bug fixing is 0.576. Besides, the fixing effectiveness ratio of each project ranges from 1.123 to 1.609 (all over 1) and the p-value of the Wilcoxon Signed-rank Test is less than 0.05. It illustrates that after excluding the potentially confounding effect, when developers solve the bug fixing tasks that they are more familiar with, the fixing effectiveness still decrease and these fixed bugs are more likely to be "re-fixed" in the future.

Overall, our conclusion on RQ3 is mostly the same, if the potentially confounding effect of fixing efficiency is removed.

6.5. Some explanations of experiment results

From the experiment results, we observe that a bug fixing with higher familiarity has higher fixing efficiency and lower fixing effectiveness. We try to give some explanations about our observation in this section.

For the fixing efficiency, some previous works analysed the features that have effects on enhancing the fixing efficiency.

11 https://issues.apache.org/jira/browse/FLINK-4512

12 https://issues.apache.org/jira/browse/FLINK-5660

13 The ownership metrics is measured by the developer's contribution to the file.

14 Suppose that X is a familiarity metric (the independent variable) and Z is fixing efficiency (the third variable). The linear regression model is $X = a+bZ+c$, where a, b, and c represent the population regression intercept, slope and residual of the model. Based on the model, we calculate the sample estimates $\hat{a}$ and $\hat{b}$ for a and b, respectively, and have the predicted estimate $\hat{X}$ for X: $\hat{X} = \hat{a}+\hat{b}Z$. Finally, we subtract $\hat{X}$ to remove the confounding effect: $X' = X-\hat{X}$.

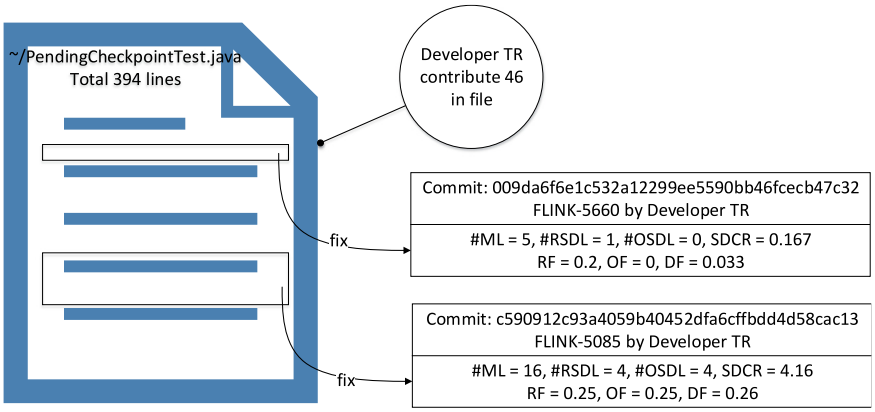


Fig. 12. Familiarity metrics in PendingCheckpointTest.java.java.

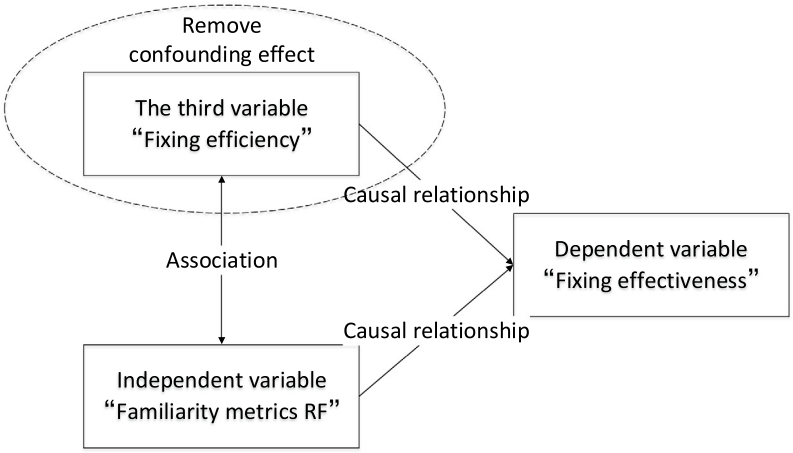


Fig. 13. Path diagram of confounding effect..

Table 5
Re-fixed ratio in high-familiarity (high) and low-familiarity (low) bug fixing after excluding the potentially confounding effect (*Lower values means better effectiveness*).

Project	High	Low	Fixing effectiveness ratio
NiFi	0.565	0.372	1.520
Avro	0.719	0.447	1.609
Bigtop	0.515	0.454	1.133
Flink	0.702	0.566	1.241
Tuscany	0.354	0.277	1.280
Wicket	0.489	0.435	1.123
ALL	0.576	0.461	1.248

Zhang et al. indicate that the "Submitter" and "Owner" are the top two important features (Zhang et al., 2013) that affect the fixing effort of a bug fixing. In addition, Guo et al. observed that "Assignee" and "Opener" also have influence on bug fixing effort (Guo et al., 2010, 2011). We extend these previous studies and focus on the familiarity effort on fixing efficiency. The positive effort of familiarity on efficiency may rely on the following reasons: (a) high familiar developers may have better understanding of the local/global structure of code they contributed to; (b) high familiar developers may locate the buggy code in the suspicious code region more quickly; (c) high familiar developers may be more experienced to adjust the buggy code, to avoid reported errors.

For the fixing effectiveness, in the previous discussion, we experiment to exclude the confounding effects of fixing efficiency and find out that the developer familiarity still has significant effects on fixing effectiveness. That means our familiarity has direct effects on fixing effectiveness. Moreover, after discussing with some project managers in industry, we try to analyse the reasons for the negative effects of familiarity on fixing effectiveness: (a) high familiarity may lead to over-confidence of developers in some cases of bug fixing: familiar developers may focus on the direct and local revision of code to avoid the errors, instead of carefully reviewing the global structure of the code and finding out the hidden causes of errors; (b) managers and reviewers may trust on familiar developers to fix the bugs, and may not consider the code change submitted by them carefully.

6.6. Additional analysis of RQ1: Why zero values of familiarity metrics?

According to the result of RQ1, we observe many bug fixing activities are the cases that the developers deal with the code they are unfamiliar with (i.e., three familiarity metrics of them have zero values). Before discussing the reasons, we first investigate the difference of unfamiliar cases among three familiarity metrics RF, OF, and DF. Compared to RF and OF, the unfamiliar percent of DF (the grey parts in Fig. 5.c) is less than that of the other two metrics (the grey parts in Fig. 5.a and .b). The reason is that DF supports a wide perspective to summarize the whole history of code change. The difference cases between DF and RF/OF are in the following two folds. Some unfamiliar cases of RF are that the developer is familiar with the code in the previous commit (not the recent one). Similarly, some unfamiliar cases of OF are that

Table 6
The classification grouped by the mean values of three familiarity metrics.

Project	LLL	LLH	LHL	HLL	LHH	HLH	HHL	HHH	No difference
ALL	0.551	0.013	0.013	0.033	0.037	0.067	0.002	0.285	0.835

the developer is familiar with the code in the following commit (not the original one).

We try to analyse the reasons there are many cases that developers deal with the totally unfamiliar bugs: (a) the code evolution history of some files is not long (e.g., the number of commits for this file is small). That leads to fewer authors contributing to this file, which may improve the probability of the cases that developers are assigned to fix the bugs in the file he or she does not contribute to; (b) as mentioned in Section 1, project managers lack a quantitative manner to measure the familiarity with bug fixing, and may ignore the familiarity factor in assigning the developer to fix the bugs. That also leads to developers participating in unfamiliar bug fixing.

6.7. Results under mean values

In Section 4.3, we use the **median** values of three familiarity metrics RF, OF, and DF in our data sets to classify bug fixing into two groups as low-familiarity (equal to or less than the median) and high-familiarity (greater than the median) and conduct the following experiment. In this section, we introduce another cutoff the **mean** values[15] and check whether the experiment results will be changed. In detail, we will re-conduct the following three experiments.

(a) Classification Evaluation in RQ2. Table 6 gives the classification of three familiarity metrics on all the studied projects when the high-familiarity and low-familiarity bug fixing are grouped by the mean values. In Table 6, the symbol "L" means the bug fixing has been classified into **l**ow-familiarity class, where the symbol "H" means it has been classified into **h**igh-familiarity class. In Table 6, the classification results of bug fixing are listed as triples containing "L" or "H" in the order of the classification according to RF, OF and DF. If a bug fixing has been classified into the same familiarity classes with three familiarity metrics (labelled as "HHH" or "LLL"), this bug fixing will be labelled as "no difference". The percent of the "no difference" labels for all studied projects is 0.835, which tells us that most of the bug fixing activities have been classified into the same familiarity classes.

(b) Fixing efficiency in RQ3. Fig. 14 shows the boxplots of fixing efficiency of high-familiarity and low-familiarity bug fixing, which are grouped by the mean values of RF. In the view of all studied projects, the fixing efficiency ratio is 1.161. It is similar to the result of RF in the previous experiment that developers with high-familiarity of the buggy code would fix the bug with high efficiency.

(c) Fixing effectiveness in RQ3. In the aspect of effectiveness, we also have an investigation on the re-fixed ratio in high-familiarity and low-familiarity bug fixing. Table 7 summarizes the re-fixed ratio of all the studied projects. Similar to the result in RQ3, the result also tells us that the developer deal with high-familiarity bugs with worse fixing effectiveness.

From above, we can conclude that the experiment results under the new cutoff points (mean) are similar to that under the original cutoff points (median).

[15] As mentioned above, using median is general and can split the data as uniform as possible in different projects; on the contrary, using mean may lead to imbalanced distribution of low-familiarity and high-familiarity group.

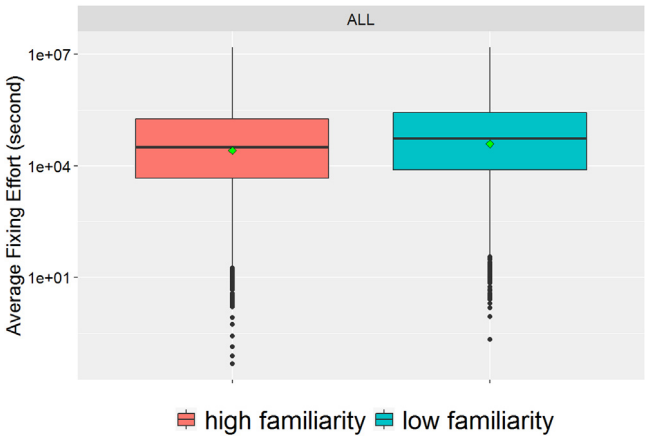


Fig. 14. The boxplot of fixing efficiency in high-familiarity and low-familiarity bug fixing, grouped by the mean values of RF (*lower values indicates higher efficiency*). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 7
Re-fixed ratio in high-familiarity (high) and low-familiarity (low) bug fixing, grouped by the mean values of RF (*Lower values indicates better effectiveness*).

Project	High	Low	Fixing effectiveness ratio
ALL	0.567	0.486	1.167

6.8. Some recommendations

In this study, we propose 4 research questions to answer whether the familiarity influences the performance of bug fixing. The result mentioned above shows that in around 50% of the bug fixing, developers deal with the buggy lines that were introduced *recently* (*firstly* or *once*) by themselves according to our proposed three familiarity metrics RF, OF, and DF. Furthermore, in average sense, the developers' familiarity with buggy code does have influences on the performance of bug fixing. Especially in the fixing period of the 4th month, the familiarity affects the fixing efficiency the most, and in the 2nd month affects the fixing effectiveness the most. To better utilize resources and improve software quality, there are some recommendations:

- **If the bug fixing activity is urgent, manager should assign the task to developers with high-familiarity as they have high fixing efficiency.** — Furthermore, from our above investigation, the fixing efficiency varies from different fixing period. Numerous previous studies (Akbarinasaji et al., 2018; Assar et al., 2016; Zhang et al., 2013) can help us predict the fixing period of bugs. Thus, through assigning the bugs with four months fixing period to developers with high-familiarity, the fixing efficiency of them will be doubled.
- **More fixing resources are recommended to be assigned to bug fixing tasks with high familiarity.** — After discussing with project managers, we find that in most cases they are more concerned about the fixing effectiveness, which is an essential part of software quality (Thongtanunam et al., 2016; Bosu et al., 2017; Kononenko et al., 2016). From the result in this study, to ensure the better effectiveness of bug fixing, we recommend that managers should assign some "outsiders" to participate in bug fixing. In other word, if the current fixing matches the pattern "developers fix the bug created by themselves", more fixing resources are recommended to be assigned to this bug fixing task with "high familiarity".

7. Threats to validity

This study provides an investigation on whether the developer's familiarity with buggy code has influence on the performance of bug fixing. However, there are some threats to validity that should be taken into consideration.

7.1. External validity

External threats to validity are concerned with the degree to which the result of the familiarity can be generalized. In this paper, the main external threat is related to project selection. All the data sets used in our experiments are well-known open source projects of Apache Software Foundation. The changed lines and the numbers of developers are varying from project to project. Moreover, the 6 open source projects are all programmed by Java-related language. The bug fixing activity in other programming languages (e.g., Python and C) may have differences in observing the familiarity influence.

7.2. Internal validity

Internal threats are concerned with the degree to which our conclusions can be drawn from the variables that we used in our experiment.

Fixing effort calculation. The fixing effort is calculated by `Resolvedtime` and `Createdtime` that JIRA's bug reports provide, i.e., the time window between `Resolvedtime` and `Createdtime`. The result may not be the exact time that the developer spends on bug fixing (e.g., it may ignore the inspection efforts, discussion efforts and casual time). Compared to the open-source projects, the commercial data sets may be more accurate in calculating fixing effort. This threat has also been presented in many other literatures (Zhang et al., 2013; Echeverria et al., 2016; McLeod and MacDonell, 2010) and remains an open challenge.

Considering only modified code. This paper calculated the three familiarity metrics by the code lines that developers modified. However, some other important factors (e.g., the *discussion* about bugs and the *inspection* of buggy code) are ignored in measuring familiarity in our study. Our results may vary when considering the above factors into measuring developers' familiarity with bugs. To alleviate the threat, we randomly select 500 bugs from our studied 9000+ bugs, extract the comments of them, and calculate the numbers of comments as an indicator to measure the discussion about bugs, the result of our additional experiment shows that the numbers of comments are slightly negatively correlated with the effectiveness of bug fixing, i.e., more number of comments may be correlated with more probabilities to be re-fixed. Nonetheless, extension studies about applying the discussion and the inspection information are needed to validate our results.

7.3. Construct validity

Construct threats to validity are concerned with the extent to which the variable studied measures the concept it claims to measure.

Discovering bug-inducing changes. We employed the commonly used SZZ algorithm (Sliwerski et al., 2005) to discover bug-inducing changes, which is applied in many previous studies of mining software repositories. There are many techniques to match the bug reports and the fixing commits in a more complex way, such as ReLink (Wu et al., 2011) and MLink (Nguyen et al., 2012). However, to improve the recall of linking bug report, these techniques mentioned above may affect the precision. Besides, Le et al. proposed a novel approach to identifying bug fixing commits in Github (Le et al., 2016). They utilized the test case of a commit and the number of change lines of a commit to determine whether a commit is a bug fixing commit. They also used the key words (i.e., "fix" and "bug fix") to identify bug fixing commits, but excluded keywords like "fix typo", "fix build" or "non-fix".

In our study, to identify bugs and synchronize the two databases JIRA and Github, we use a strong constraint on linking bugs and commits, i.e., they can only be matched by the same `BUG ID`. This process guarantees the correctness of our data set, but may ignore some bugs that do not show `BUG ID` in Github. Besides, the discovered bug-inducing changes may be not accurate in some cases (e.g., a newly detected old bug), which is a potential threat to the construct validity. Indeed, this is an inherent problem to most, if not all, studies that discover bug-inducing changes by mining software repositories, not unique to us.

Identifying the contributors of code. First, in a single line's code evolution history, to determine the contributor of commits, we use the developer names to match. If a developer changed his name in the code evolution history, our matching process might cause false negative. Second, to measure the familiarity in original view and developing view, we use the metrics OF and DF. Bug commits with long evolution history will lose some information through migration, which affect the results of OF and DF. Specifically, in the view of OF, a new code line added by one developer may not actually indicate that the code is firstly introduced by the developer. This introduces the threat that the values of OF and DF may not be fully aligned with what was claimed in the definition of OF and DF. Besides, re-used (copied) code is another potential threat to identify the contributors of code. For example, if the developer (A) submits a commit of re-used code contributed by another developer (B), the re-used code is considered as the contribution of the former (A). Hence, the familiarity of the former may be overestimated. To evaluate this, we randomly select 500 bug-inducing commits from our data set, extract the buggy code submitted by each commit and check whether it contains more than 5 consecutive lines of re-used code, i.e., more than 5 consecutive lines of code are the same in the current bug-inducing commit and a previous commit. Our result shows that only 17 out of 500 bug-inducing commits contain more than 5 consecutive lines of re-used code.

8. Conclusion

In this study, based on the evolution history of buggy code lines, we propose three metrics to evaluate the familiarity between the bugs and the developers. Our approach consists of three steps: links the bugs with the same `BUG ID` from both JIRA and Github; tracks down the evolution history of modified lines from a bug fixing commit; and extracts three familiarity metrics RF, OF, and DF.

Based on the experiment conducted within 9000+ confirmed bugs on 6 well-know Apache Software Foundation projects, we find that: developers are more likely to be assigned to fix the bugs introduced by themselves; they fix the bug introduced by themselves more quickly; and they are more likely to introduce future bugs when fixing familiar bugs. As a result, we recommend that manager should assign some "outsiders" to participate in the bug fixing and they should assign more maintenance resource to the bugs that developer deals with his own code.

Further work includes the following aspects: (a) extending the definition of familiarity metrics; (b) applying familiarity on other software activities; (c) calculating familiarity based on other factors, e.g., the discussion and inspection.

CRediT authorship contribution statement

Chuanqi Wang: Methodology, Software, Writing - original draft. **Yanhui Li:** Conceptualization, Methodology, Writing - original draft. **Lin Chen:** Methodology, Writing - review & editing. **Wenchin Huang:** Software. **Yuming Zhou:** Methodology, Writing - review & editing. **Baowen Xu:** Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The work is supported by National Key R&D Program of China (Grant No. 2018YFB1003901) and the National Natural Science Foundation of China (Grant No. 61872177 and 61772259). We thank the anonymous referees for their helpful comments on this paper.

References

Akbarinasaji, S., Caglayan, B., Bener, A., 2018. Predicting bug-fixing time: A replication study using an open source software project. J. Syst. Softw. 136, 173–186.

Assar, S., Borg, M., Pfahl, D., 2016. Using text clustering to predict defect resolution time: a conceptual replication and an evaluation of prediction accuracy. Empir. Softw. Eng. 21 (4), 1437–1475.

Bird, C., Nagappan, N., Devanbu, P.T., Gall, H.C., Murphy, B., 2009. Does distributed development affect software quality? An empirical case study of Windows Vista. In: 31st International Conference on Software Engineering, ICSE 2009, Vancouver, 16–24 May, pp. 518–528.

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

Bird, C., Nagappan, N., Murphy, B., Gall, H.C., 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: 13th European Software Engineering Conference, ESEC-13, Szeged, Hungary, 5–9 September, pp. 4–14.

Bosu, A., Carver, J.C., Bird, C., Orbeck, J.D., Chockley, C., 2017. Process aspects and social dynamics of contemporary code review: insights from open source development and industrial practice at microsoft. IEEE Trans. Softw. Eng. 43 (1), 56–75.

Brooks, Jr., F.P., 1995. The Mythical Man-Month - Essays on Software Engineering, second ed. Addison-Wesley.

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, 4–8 November, pp. 353–362.

Corley, C.S., Kammer, E.A., Kraft, N.A., 2012. Modeling the ownership of source code topics. In: IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, 11–13 June, pp. 173–182.

Curtis, B., Krasner, H., Iscoe, N., 1988. A field study of the software design process for large systems. Commun. ACM 31 (11), 1268–1287.

Diaz, D., Bavota, G., Marcus, A., Oliveto, R., Takahashi, S., Lucia, A.D., 2013. Using code ownership to improve IR-based Traceability Link Recovery. In: IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20–21 May, pp. 123–132.

Echeverria, J., Pérez, F., Abellanas, A., Panach, J.I., Cetina, C., Pastor, O., 2016. Evaluating bug-fixing in software product lines: an industrial case study. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016, Ciudad Real, Spain, 8–9 September, pp. 24:1–24:6.

Espinosa, J.A., Kraut, R.E., Lerch, F.J., Slaughter, S., Herbsleb, J.D., Mockus, A., 2001. Shared mental models and coordination in large-scale, distributed software development. In: Proceedings of the International Conference on Information Systems, ICIS 2001, New Orleans, Louisiana, USA, 16–19 December, pp. 513–518.

Fisher, R.A., 1992. Predicting bug-fixing time: A replication study using an open source software project. In: Breakthroughs in Statistics. pp. 66–70.

Foucault, M., Falleri, J., Blanc, X., 2014. Code ownership in open-source software. In: 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, 13–14 May, pp. 39:1–39:9.

Fritz, T., Ou, J., Murphy, G.C., Murphy-Hill, E.R., 2010. A degree-of-knowledge model to capture source code familiarity. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1–8 May, pp. 385–394.

Greiler, M., Herzig, K., Czerwonka, J., 2015. Code ownership and software quality: A replication study. In: 12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, 16–17 May, pp. 2–12.

Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B., 2010. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, May 1–8.

Guo, P.J., Zimmermann, T., Nagappan, N., Murphy, B., 2011. "Not my bug!" and other reasons for software bug report reassignments, In: Proceedings of the 2011 ACM Conference on Computer Supported Cooperative Work, CSCW 2011, Hangzhou, China, March 19–23.

Hattori, L., Lanza, M., Robbes, R., 2012. Refining code ownership with synchronous changes. Empir. Softw. Eng. 17 (4–5), 467–499.

Herbsleb, J.D., Mockus, A., 2003. An empirical study of speed and communication in globally distributed software development. IEEE Trans. Softw. Eng. 29 (6), 481–494.

Hooimeijer, P., Weimer, W., 2007. Modeling bug report quality. In: The Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, pp. 34–43.

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

Jiang, H., Zhang, J., Ma, H., Nazar, N., Ren, Z., 2017. Mining authorship characteristics in bug repositories. Sci. China Inf. Sci. 60 (1), 12107.

Kim, S., Zimmermann, T., Pan, K., Jr., E.J.W., 2006. Automatic identification of bug-introducing changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006, 18–22 September, Tokyo, Japan, pp. 81–90.

Kim, S., Zimmermann, T., Pan, K., Whitehead, E.J., 2006. Automatic identification of bug-introducing changes. In: Ieee/Acm International Conference on Automated Software Engineering, pp. 81–90.

Kononenko, O., Baysal, O., Godfrey, M.W., 2016. Code review quality: how developers see it. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14–22.

Le, X.D., Bao, L., Lo, D., Xia, X., Li, S., Pasareanu, C.S., 2019. On reliability of patch correctness assessment. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, pp. 524–535.

Le, X.D., Lo, D., Le Goues, C., 2016. History driven program repair. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14–18, pp. 213–224.

McLeod, L., MacDonell, S.G., 2010. Stakeholder perceptions of software project outcomes: an industry case study. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ESEM 2010, 16–17 September, Bolzano/Bozen, Italy.

Meng, X., Miller, B.P., Williams, W.R., Bernat, A.R., 2013. Mining software repositories for accurate authorship. In: 2013 IEEE International Conference on Software Maintenance, Eindhoven, the Netherlands, 22–28 September, pp. 250–259.

Misirli, A.T., Shihab, E., Kamei, Y., 2016. Studying high impact fix-inducing changes. Empir. Softw. Eng. 21 (2), 605–641.

Nagappan, N., Murphy, B., Basili, V.R., 2008. The influence of organizational structure on software quality: an empirical case study. In: 30th International Conference on Software Engineering, ICSE 2008, Leipzig, Germany, 10–18 May, pp. 521–530.

Nguyen, A.T., Nguyen, T.T., Nguyen, H.A., Nguyen, T.N., 2012. Multi-layered approach for recovering links between bug reports and fixes,

Orrù, M., Marchesi, M., 2016. A case study on the relationship between code ownership and refactoring activities in a Java software system. In: Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics, WETSoM@ICSE 2016, Austin, TX, USA, 14 May, pp. 43–49.
Pinzger, M., Nagappan, N., Murphy, B., 2008. Can developer-module networks predict failures? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, 9–14 November, pp. 2–12.
Rahman, F., Devanbu, P.T., 2011. Ownership, experience and defects: a fine-grained study of authorship. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, 21–28 May, pp. 491–500.
Rahman, F., Devanbu, P.T., 2013. How, and why, process metrics are better. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, 18–26 May, pp. 432–441.
Rahman, M.M., Roy, C.K., Kula, R.G., 2017. Predicting usefulness of code review comments using textual features and developer experience. In: Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, 20–28 May, pp. 215–226.
Robbes, R., Röthlisberger, D., 2013. Using developer interaction data to compare expertise metrics. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, 18–19 May, pp. 297–300.
Sedano, T., Ralph, P., Péraire, C., 2016. Practice and perception of team code ownership. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, EASE 2016, Limerick, Ireland, 1–3, June, pp. 36:1–36:6.
Sinha, V.S., Sinha, S., Rao, S., 2010. BUGINNINGS: identifying the origins of a bug. In: Proceeding of the 3rd Annual India Software Engineering Conference, ISEC 2010, Mysore, India, 25–27 February, pp. 3–12.
Sliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? ACM SIGSOFT Softw. Eng. Notes 30 (4), 1–5.
Thongtanunam, P., McIntosh, S., Hassan, A.E., Iida, H., 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, 14–22 May, pp. 1039–1050.
Thung, F., 2016. Automatic prediction of bug fixing effort measured by code churn size. In: Proceedings of the 5th International Workshop on Software Mining, SoftwareMining@ASE 2016, Singapore, Singapore, September 3, pp. 18–23.
Wilcoxon, F., 1946. Individual comparisons of grouped data by ranking methods. J. Econ. Entomol. 39 (6), 269.
Wu, R., Zhang, H., Kim, S., Cheung, S., 2011. ReLink: recovering links between bugs and changes. In: SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference, ESEC-13, Szeged, Hungary, 5–9 September, pp. 15–25.
Zhang, H., Gong, L., Versteeg, S., 2013. Predicting bug-fixing time: an empirical study of commercial software projects. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, 18–26 May, pp. 1042–1051.
Zhou, Y., Xu, B., Leung, H., Chen, L., 2014. An in-depth study of the potentially confounding effect of class size in fault prediction. ACM Trans. Softw. Eng. Methodol. 23 (1), 10:1–10:51.
Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? In: International Workshop on Mining Software Repositories, pp. 1–5.

Chuanqi Wang is a Ph.D. student in the Department of Computer Science and Technology at Nanjing University. His current research direction is the analysis, evaluation and prediction of bug fixing.

Yanhui Li received the BS, MS and PhD degrees in Computer Science from Southeast University, China. He is currently an assistant professor in the Department of Computer Science and Technology at Nanjing University. His main research interests include empirical software engineering, software analysis, knowledge engineering and formal methods. He is a member of the IEEE and the ACM.

Lin Chen received Ph.D. degree in computer science from Southeast University in 2009. He is currently an associate professor in the Department of Computer Science and Technology at Nanjing University. His research interests are software analysis and software maintenance.

Wenchin Huang is a master student in the Department of Computer Science and Technology at Nanjing University. His current research direction is bug fixing and defect prediction.

Yuming Zhou is a professor in the Department of Computer Science and Technology at Nanjing University. His main research interests are empirical software engineering.

Baowen Xu received the BS, MS, and Ph.D. degrees in computer science from Wuhan University, Huazhong University of Science and Technology, and Beihang University, respectively. He is currently a professor in the Department of Computer Science and Technology at Nanjing University. His main research interests are programming languages, software testing, software maintenance, and software metrics. He is a member of the IEEE and the IEEE Computer Society.