JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2015; **00**:1–18
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

An Empirical Study on Developer Related Factors Characterizing Fix-Inducing Commits

Michele Tufano[1], Gabriele Bavota[2], Denys Poshyvanyk[1]
Massimiliano Di Penta[3], Rocco Oliveto[4], Andrea De Lucia[5]

[1]*The College of William & Mary, Williamsburg, USA* — [2]*Free University of Bozen-Bolzano, Bolzano, Italy*
[3]*University of Sannio, Benevento, Italy* — [4]*University of Molise, Pesche (IS), Italy*
[5]*University of Salerno, Salerno, Italy*
m.tufano@email.wm.edu, gabriele.bavota@unibz.it, denys@cs.wm.edu
dipenta@unisannio.it, rocco.oliveto@unimol.it, adelucia@unisa.it

SUMMARY

This paper analyzes developer related factors that could influence the likelihood for a commit to induce a fix. Specifically, we focus on factors that could potentially hinder developers' ability to correctly understand the code components involved in the change to be committed: (i) the *coherence* of the commit (i.e., how much it is focused on a specific topic), (ii) the *experience level* of the developer on the files involved in the commit, and (iii) the *interfering changes* performed by other developers on the files involved in past commits. The results of our study indicate that "fix-inducing" commits (i.e., commits that induced a fix) are significantly less coherent than "clean" commits (i.e., commits that did not induce a fix). Surprisingly, "fix-inducing" commits are performed by more experienced developers, yet, those are the developers performing more complex changes in the system. Finally, "fix-inducing" commits have a higher number of past interfering changes as compared to "clean" commits. Our empirical study sheds light on previously unexplored factors and presents significant results that can be used to improve approaches for defect prediction. Copyright © 2015 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Commits, Bug Introduction, Empirical Study

1. INTRODUCTION

Software systems are often the result of work performed by hundreds or thousands of developers often distributed around the world.

Software systems continuously evolve as new features are introduced, corrective maintenance intervention are performed, or other improvements are carried out. Also, the continuous evolution is related to the software increasing complexity [27, 26]. In certain circumstances, either because the change to perform is very complex, or for reasons related to the developers' activity, experience, or to other factors, a change could induce a bug. Recently, the research community focused on understanding this phenomenon by singling out and studying specific characteristics of commits done by developers that can impact the likelihood of introducing bugs. Some of the studied

*Correspondence to: Journals Production Department, John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK.

characteristics include the following: (i) the commits' time and day [17, 33], (ii) the ownership level of the files involved in the commit [32, 13, 31], (iii) the experience of the developer responsible for the commit [17, 32, 31], (iv) the complexity of the files involved in the commit [25], and (v) lexical properties of the commits (e.g., added and removed terms in the source code, commit's message terms, etc) [25].

Stemming from previous research in this area, we have conducted an in-depth empirical study, on five open source Java projects, aiming at analyzing three developer related factors that could influence the possibility of introducing bugs in commits. Specifically, we focus on factors that could potentially hinder developers' ability to understand the code components impacted by a change. In particular, we studied the following factors:

1. ***Coherence* of files involved in a commit**. We define *coherence* as the degree that committed changes are focused on a specific topic or subsystem. We expect commits having lower *coherence* to represent changes that are more challenging for developers to manage, thus increasing the likelihood of introducing bugs.

2. ***Experience* of the developer performing the commit**. Unlike previous studies [17, 32, 31], we do not consider general experience of developers performing commits (e.g., total number of days on a project). Instead, we evaluate developer's experience only on files involved in the commit. We expect commits checked-in by developers having less *experience* on changed files to have higher likelihood of inducing bugs.

3. ***Interfering changes***. Suppose that the developer *Sally* has modified a file F_i in a commit performed at time t_j and that, at time t_k (with $k > t$) *Sally* is working again on file F_i in order to commit other changes. If in the time period between t_j and t_k other developers have modified F_i, these changes (defined as *interfering changes*) could have affected *Sally*'s understanding of F_i, possibly leading to the introduction of bugs during the new changes she performs at time t_k. We expect commits introducing a bug to be preceded by a higher number of *interfering changes* as compared to clean commits.

It is important to note that, while there is no clear way to determine whether a commit introduces a bug, we can identify commits that modify source code lines subsequently changed in a bug fix. These commits are identified by using the SZZ algorithm [34, 25]). We refer to the remaining commits (i.e., those that do not induce a fix) as "clean" ones.

The results of our study indicate that fix-inducing commits are significantly less *coherent* as compared to clean commits and, surprisingly, fix-inducing commits are performed by developers having higher levels of *experience* than developers checking-in clean commits. However, we also observed that more experienced developers are those in charge of the most complex and, thus, risky changes. Finally, the number of interfering changes preceding fix-inducing commits is significantly higher than those ones for clean commits.

Our results, together with the results achieved in previous and related studies, provide empirical evidence for building recommendation systems aimed at helping developers and code reviewers to identify commits requiring attention during verification and validation activities.

Structure of the paper. Section 2 defines our empirical study and the research questions, and provides details about the data extraction process and analysis method. Section 3 reports the results of the study, answering our research questions, while Section 4 discusses the threats that could affect the validity of the results achieved. Section 5 discusses the related literature, while Section 6 concludes the paper and outlines directions for future work.

2. DESIGN OF THE EMPIRICAL STUDY

The *goal* of the study is to analyze developers' commits occurring over the history of a software project, with the *purpose* of investigating the influence of developer related factors on the likelihood a commit has to induce a bug fix. The *quality focus* is on software defect-proneness, which could be

Table I. Characteristics of the systems under analysis.

System	Period	Classes	KLOC	# of Commits	# of Fix-Inducing commits
Ant	Jan 2000-Sep 2013	87-2,236	8-399	12,891	1839
JMeter	Sep 1998-Nov 2013	312-2,378	43-446	10,198	1833
log4j	Nov 2000-May 2013	282-620	38-82	3,271	541
Tomcat	Mar 2006-Nov 2013	1,803-2,649	202-509	8,986	1299
Xerces-J	Nov 1999-Aug 2013	181-1,248	56-349	5,462	68

influenced by commit activities. The *perspective* is of researchers interested in investigating whether commits exhibiting specific characteristics should receive particular attention during verification and validation activities.

2.1. Context and Research Questions

The *context* of the study consists of the change history of five Java open source systems, namely `Ant`[1], `JMeter`[2], `log4j`[3], `Tomcat`[4], and `Xerces-J`[5]. `Ant` is a build tool and library specifically designed for Java applications (though it can be used for other purposes). `JMeter` is a framework designed to statically and dynamically test program performances. `log4j` is a logging library for Java while `Tomcat` is an implementation of the Java Servlet and Java Server Pages technologies. Finally, `Xerces-J` is a XML parser for Java. Table I reports characteristics of the analyzed system, namely time period analyzed and size ranges (in terms of KLOC and # of classes). Table I also reports the number of commits including those that induced fixes for each system.

We formulated the following RQs:

- **RQ$_1$:** *Are fix-inducing commits less coherent as compared to clean commits in terms of involved code files?* This research question aims at investigating whether source code files involved in fix-inducing commits exhibit higher/lower *coherence* as compared to code files involved in clean commits. We define *coherence* of the files in the commit as the degree that the files are focused on specific topic or subsystem. For example, a commit involving three files all implementing responsibilities related to the management of users in the database should be considered as more *coherent* commit as compared to a commit involving three files implementing several unrelated responsibilities. The null hypothesis being tested is:

  H_{0_1}*: There is no significant difference among coherence values of fix-inducing and clean commits.*

- **RQ$_2$:** *Is experience level of developers involved in fix-inducing commits any different from that of developers involved in clean ones?* This research question investigates whether fix-inducing commits are performed by developers having higher/lower experience levels with respect to clean commits. As previously explained, unlike related studies, we do not focus on the general experience of developers responsible for commits. Instead, we evaluate the developers' experience only on the files for a given commit under investigation. That is, more experience mirrors better knowledge of the committed files. Thus, the null hypothesis being tested is as following:

  H_{0_2}*: There is no significant difference between experience levels of developers performing fix-inducing and clean commits.*

- **RQ$_3$:** *Do fix-inducing commits involve files that are subject to interfering changes among developers as compared to clean ones?* To explain the aim of this research question let us first introduce the concept of *interfering changes*. Suppose *Sally* performed a commit involving the code file f_i at time t_1. *Sally* also performed a commit on the same file f_i in the past (e.g., time t_0, being the very last previous commit of *Sally* before time t_1 that involved file f_i). If in the period of time between t_0 and t_1 the same file f_i had been changed and was part of a commit(s) by other developers, we define this sequence of commits as interfering changes. We are hypothesizing that changes performed by other developers on the file f_i between the last and the second-last commits performed by *Sally*, could have partially affected *Sally*'s understanding of the file f_i. This is because, for example, such changes could have modified the impact set of files being modified by *Sally*, or changed some file's semantics. This interference in her understanding of file f_i could be one of the reasons why *Sally* might introduce a bug while changing f_i. Thus, the null hypothesis being tested here is as following:

> H_{0_3}*: There is no significant difference between the number of interfering changes before fix-inducing and clean commits.*

- **RQ$_4$:** *Are the analyzed developers related factors orthogonal to structural properties of commits studied in previous works?* With this research question we want to understand whether the analyzed factors capture different characteristics of a commit with respect to previously studied commit's properties, which might help to explain the likelihood of a commit to induce a bug-fix.

2.2. Data Extraction Process

This subsection describes the data extraction process that we followed with the aim of (i) collecting developers' commits and (ii) determining whether commits could have induced subsequent bug-fixes.

Commits checked-in by developers have been collected by mining the change log of the versioning system hosting five subject projects in our study[†]. Note that in order to reduce potential noise in our analysis, we automatically excluded (i) the first commit that imports entire initial version of the system into the repository and (ii) commits involving all the files in the repository that are likely due to re-organization of files in folders or license statement changes.

Once we collected the logs, we analyzed them to identify commits that are likely to be the ones inducing a bug-fix. Firstly, we identified bug fixing commits by mining regular expressions containing issue IDs in the change log of the versioning system, e.g., *"fixed issue #ID"* or *"issue ID"*. After that, for each identified issue ID, we downloaded the corresponding issue reports from the Bugzilla issue tracking system[6] (all subject systems had the issue tracker hosted on Bugzilla) and extracted the following components from them:

- *product name*;
- *issue's type*, i.e., whether an issue is a bug, enhancement request, etc. In Bugzilla this classification is (partially) handled by the field *severity*;
- *issue's status*, i.e., whether an issue was closed or not;
- *issue's resolution*, i.e., whether an issue was resolved by fixing it, or whether it was a duplicate bug report, or a "works for me" case;
- *issue's opening date*;
- *issue's closing date*, if available.

Then, we checked each issue's report to be correctly downloaded (e.g., the issue's ID identified from the versioning system commit note could be a false positive), and that the issue, indeed, is related to the product under analysis (e.g., Apache uses the same issue tracking system for multiple products). After that, we used the issue type/severity field to classify the issue and distinguish bug fixes from other issue types (e.g., enhancements). Finally, we only considered bugs having *CLOSED* or *RESOLVED* status and *FIXED* resolution. Basically, we restricted our attention to (i) issues that were related to bugs as we used them as a measure of defect-proneness, and (ii) issues that were neither duplicate reports nor false alarms.

[†]All five systems are available in the *Git* repository.

As the final step, we identified commits that were likely to induce fixes. We used the SZZ algorithm [34, 25], which relies on the annotation/blame feature of versioning systems. In essence, given a bug-fix identified by the bug ID k, the approach works as follows:

1. For each file f_i, $i = 1 \dots m_k$ involved in the bug-fix k (m_k is the number of files changed in the bug-fix k), and fixed in its revision *rel-fix*$_{i,k}$, we extract the file revision just *before* the bug fixing (*rel-fix*$_{i,k} - 1$).

2. starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the bug k the *blame* feature of *Git* is used to identify the file revision where the last change to that line occurred. In doing that, blank lines and lines that only contain comments are identified using an island grammar parser [29]. This produces, for each file f_i, a set of $n_{i,k}$ fix-inducing revisions *rel-bug*$_{i,j,k}$, $j = 1 \dots n_{i,k}$.

2.3. *Study Variables and Analysis Method*

This subsection describes the analyses and statistical procedures that we used to address three research questions formulated in Section 2.1.

To address **RQ**$_1$, we computed the *structural* and *lexical* coherence of all the commits performed during the change history of five subject systems. The *lack of structural* coherence ($LCoh_{Str}$) of a commit C_k involving source code files $F_{C_k} = \{f_1, f_2, \dots, f_n\}$ is computed as:

$$LCoh_{Str}(C_k) = \frac{\sum_{i=1}^{|F_{C_k}|-1} \sum_{j=i+1}^{|F_{C_k}|} StrDistance(f_i, f_j)}{\frac{|F_{C_k}| \times (|F_{C_k}|-1)}{2}}$$

where $StrDistance(f_i, f_j)$ is the number of edges one should cross inside the packages graph in order to reach from file f_i file f_j (or *vice versa*). $|F_{C_k}|$ is the number of files involved in commit C_k. $LCoh_{Str} = 0$ if only one file is involved in the commit. Note that $LCoh_{Str}$ is an inverse coherence measure: the higher the $LCoh_{Str}$ the lower the *structural* coherence of the commit it is measured on.

The conjecture behind $LCoh_{Str}$ is that code files contained in different, "distant" packages in the system architecture are more likely to implement different responsibilities. Such a conjecture is inspired by the single-responsibility principle of packages, stating that a package should group together code entities implementing a single, specific responsibility [11]. Thus, commits modifying code files spread across several different packages are more likely to be incoherent, thus increasing the chances of introducing bugs.

An example of computing $StrDistance$ is reported in Figure 1. In particular, Figure 1 illustrated how $StrDistance$ between $file_1$ and $file_2$ is computed. Since five different edges (those reported in **bold**) should be crossed to reach one file from the other, $StrDistance(file_1, file_2) = 5$.

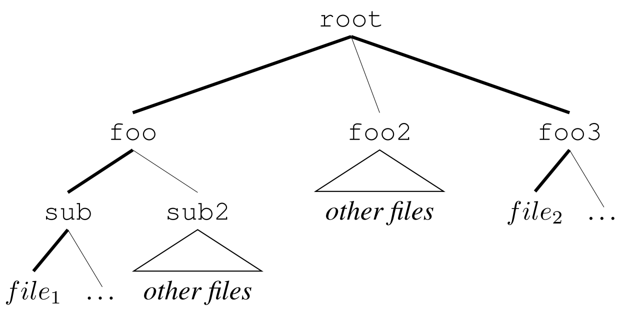

Figure 1. $StrDistance(file_1, file_2) = 5$

Concerning the *lexical* coherence (Coh_{Lex}) of a commit C_k involving the source code files F_{C_k}, it is computed as:

$$Coh_{Lex}(C_k) = \frac{\sum_{i=1}^{|F_{C_k}|-1} \sum_{j=i+1}^{|F_{C_k}|} VSM(f_i, f_j)}{\frac{|F_{C_k}| \times (|F_{C_k}|-1)}{2}}$$

where $VSM(f_i, f_j)$ is Vector Space Model (VSM) [10] cosine similarity between file f_i and file f_j. In the Vector Space Model, each document (a code file in our case) is represented as a vector of terms that occur within the corpus (all code files of a specific project) [10]. Given m the total number of different terms in the corpus, the similarity between two documents is measured by the cosine of the angle between their vectors in the m-space of the terms. Such a similarity measure increases as more terms are shared between the two vectors. When applying VSM we (i) normalized the resulting text using identifier splitting (we also kept original identifiers), (ii) applied stop words removal algorithm (i.e., we removed common English words and reserved programming language keywords), and (iii) performed stemming (we used a well-known Porter's stemmer), (iv) used the *tf-idf* weighting schema [10]. Note that unlike $LCoh_{Str}$, the higher the Coh_{Lex} the higher the coherence of the files involved in a commit. The rationale behind the definition of Coh_{Lex} is that pairs of code files exhibiting a high textual similarity are likely to implement similar responsibilities and thus, should result in a commit easier for the developer to manage. On the other hand, a commit involving pairs of classes having a low textual similarity is likely to deal with heterogeneous and less coherent responsibilities.

Finally, it is worth mentioning that the computation of both $LCoh_{Str}$ and Coh_{Lex} is optimized by avoiding the computation of the same distance multiple times. That is, we do not compute both $StrDistance(f_a, f_b)$ and $StrDistance(f_b, f_a)$, since they represent the same value. Similarly for the VSM similarity between pairs of files.

We present the results via descriptive statistics comparing the distribution of $LCoh_{Str}$ and Coh_{Lex} for fix-inducing and clean commits. Then, in order to test our null hypotheses, the results are analyzed through the Mann-Whitney test [15]. This latter is used to analyze statistical significance of the differences between fix-inducing and clean commits. The results are intended as statistically significant at $\alpha = 0.05$. We also estimated the magnitude of the measured differences by using the Cliff's Delta (or d), a non-parametric effect size measure [19] for ordinal data. We followed well-established guidelines to interpret the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$ [19].

Concerning **RQ**$_2$, we used four different measures to assess developers' experience on the files involved in a commit. The first metric is named *lexical* experience (Exp_{Lex}). Given C_k the commit performed by a developer D_s and involving the files F_{C_k}, Exp_{Lex} is computed as follows:

$$Exp_{Lex}(C_k) = \frac{\sum_{f_i \in F_{C_k}} VSM(f_i, B_{D_s})}{|F_{C_k}|}$$

where B_{D_s} represents the lexical background of a developer D_s composed of a single textual file in which all the files modified in the past by D_s are merged in. If a file has been modified n times in the past by D_s it is added n times to B_{D_s}, each time by considering the file version modified by D_s. The higher $Exp_{Lex}(C_k)$, the higher developer's experience on the files involved in C_k. Starting from the Exp_{Lex} measure, we also defined a variant of it, where the developer's background B_{D_s} is built by only considering the commits performed by D_s in the last six months. This variant, named as *recent lexical* experience (Exp_{RecLex}), aims at taking into account the fact that a file modified by a developer far in the past could (i) have been substantially changed and (ii) may not be consistent with developer's current understanding of that file.

The third measure used to capture the developer's experience is named as *frequency* experience (Exp_{Freq}) and it is computed, for a commit C_k performed by developer D_s on files F_{C_k}, as:

$$Exp_{Freq}(C_k) = \frac{|Commits(F_{C_k})|}{|Commits(D_s)|}$$

where $|Commits(F_{C_k})|$ is the number of commits performed by D_s on files in F_{C_k} in the past and $|Commits(D_s)|$ is the total number of commits performed in the past by D_s. The higher $Exp_{Freq}(C_k)$ the higher the developer's experience on the files involved in C_k. Also in this case, we adopted a variant, named as *recent frequency* experience ($Exp_{RecFreq}$), where both numerator and denominator are computed by only taking into account the commits performed by D_s in the last six months.

As done for **RQ**$_1$, we present descriptive statistics comparing the distributions of different experience measures for fix-inducing and clean commits and perform statistical analysis with Mann-Whitney test [15] and Cliff's Delta effect size [19]. The total number of commits are 25,385, of which 3,439 (13.55%) are Fix-Inducing Commits. Although the proportions of fix-inducing and clean commits are clearly different, Mann-Whitney test (differently from the parametric-equivalent procedure, i.e., t-test) does not makes any assumption on the need for having balanced sets, and given the sample size it allows for detecting differences between the medians of the two sets.

Finally, in order to address **RQ**$_3$, we report and compare the number of interfering changes (computed as explained in Section 2.1) preceding fix-inducing and clean commits as well as the size of these interfering changes (i.e., the number of LOCs modified by the changes). We perform this statistical comparison in a similar fashion as for the two previous research questions.

So far, the statistics used to address the research questions analyzed the differences, in terms of investigated factors, between fix-inducing and clean commits. However, previous work has correlated structural characteristics of a change to its likelihood to induce a fix [25]. For this reason, in order to address **RQ**$_4$, we need to ensure that developer related factors that we investigated are not just a different definition of already studied commits' characteristics. To this aim, we performed Principal Component Analysis (PCA) [23] on our metrics (i.e., $LCoh_{Str}$, Coh_{Lex}, Exp_{Lex}, Exp_{RecLex}, Exp_{Freq}, $Exp_{RecFreq}$, and $\#InterferingChanges$) and on commit's structural properties correlated to the likelihood of inducing a fix in previous work: (i) the number of lines added, (ii) the number of lines removed, (iii) the number of lines changed, (iv) the number of code hunks [8] modified, and (iv) the number of files affected by the change. As described by Alali *et al.*[8], *"a hunk basically represents the ranges of lines that contain the changes in two versions of the same file. The number of such ranges for a file approximately depends on the contiguous changed and unchanged lines"*. The number of changed hunks in a commit can be obtained by using the *GNU diff* utility and we chose it as an indicator of the size (and complexity) of the change performed by developers in commits because it takes into account spread of the performed change in the system. For instance, suppose we have two commits: C_1 involving source code file f_i, and C_2 involving source code files f_i and f_j. In C_1 lines of code 3-16 were modified (for a total of 14 LOCs), while in commit C_2 lines of code 3-4 and 35-37 were modified in f_i and lines of code 22-23 and 81-84 were modified in f_j (for a total of 11 LOCs). By using LOCs as indicator of the size (and complexity) of the change, commit C_1 can be perceived as more complex as compared to commit C_2. Instead, considering the number of hunks tells us that commit C_2 is likely to be more complex than commit C_1, with four vs. one hunks modified.

All the measures used in the PCA have been computed on the total set of 40,808 commits coming from the five open source systems in our study. We applied PCA as done in previous work [18], including procedures for identifying outliers and rotating principal components.

The PCA allows to (i) identify different dimensions that describe certain phenomenon (in our case, the likelihood of a commit to induce a bug-fix), (ii) obtain an indication of the importance of each dimension (i.e., the proportion of variance) in the description of this phenomenon, and (iii) understand which measure captures which dimension (note that the same dimension can be captured by different measures). Thus, using PCA we can verify if the factors studied in this paper capture the same dimension(s) as structural properties of commit analyzed in previous studies, or rather they capture orthogonal dimensions helping to better explain the phenomenon.

Table II. Lack of Structural Coherence ($LCoh_{Str}$) of fix-inducing and clean commits: Descriptive statistics, Mann-Whitney test (p-value) and Cliff's delta (d). The lower $LCoh_{Str}$ the higher the commit's coherence.

System	Fix-inducing Commits					Clean Commits					p-value	d
	Min	Mean	Median	Max	St. Dev.	Min	Mean	Median	Max	St. Dev.		
Ant	0.00	3.20	1.52	16.00	3.53	0.00	1.92	0.00	18.00	3.44	<0.01	-0.25 (Small)
JMeter	0.00	3.34	1.00	17.00	3.84	0.00	1.28	0.00	20.00	3.00	<0.01	-0.33 (Medium)
log4j	0.00	2.79	2.00	13.00	2.89	0.00	1.17	0.00	14.00	2.39	<0.01	-0.40 (Medium)
Tomcat	0.00	2.76	1.87	14.00	2.93	0.00	1.46	0.00	14.00	2.62	<0.01	-0.29 (Small)
Xerces-J	0.00	0.55	0.00	4.69	1.05	0.00	0.34	0.00	9.00	0.98	<0.01	-0.16 (Small)

Table III. Lexical Coherence (Coh_{Lex}) of fix-inducing and clean commits: Descriptive statistics, Mann-Whitney test (p-value) and Cliff's delta (d). The higher Coh_{Lex} the higher the commit's coherence.

System	Fix-inducing Commits					Clean Commits					p-value	d
	Min	Mean	Median	Max	St. Dev.	Min	Mean	Median	Max	St. Dev.		
Ant	0.03	0.58	0.47	1.00	0.30	0.00	0.78	1.00	1.00	0.31	<0.01	0.33 (Medium)
JMeter	0.00	0.61	0.48	1.00	0.33	0.00	0.83	1.00	1.00	0.29	<0.01	0.34 (Medium)
log4j	0.09	0.50	0.37	1.00	0.30	0.00	0.74	1.00	1.00	0.33	<0.01	0.39 (Medium)
Tomcat	0.02	0.53	0.40	1.00	0.29	0.00	0.74	1.00	1.00	0.32	<0.01	0.33 (Medium)
Xerces-J	0.15	0.69	0.59	1.00	0.29	0.00	0.83	1.00	1.00	0.26	<0.01	0.26 (Small)

2.4. *Replication Package*

All the data used in our study are publicly available at our online appendix [7]. Specifically, we provided the *R* scripts and working data sets used to run the statistical tests and produce the plots and tables reported in this paper.

3. ANALYSIS OF THE RESULTS

This section discusses the results achieved in our study aimed at responding to our three research questions.

3.1. *RQ$_1$: Are fix-inducing commits less coherent as compared to clean commits in terms of involved code files?*

Tables II and III show the results achieved when comparing fix-inducing and clean commits with respect to their *lack of structural coherence* ($LCoh_{Str}$) and *lexical coherence* (Coh_{Lex}), respectively. In particular, Tables II and III report the descriptive statistics, the results of the Mann-Whitney test (p-value), and the Cliff's d effect size. Note that the lower $LCoh_{Str}$ the higher the commit's coherence, while the higher Coh_{Lex} the higher the commit's coherence.

In terms of *structural* coherence (see Table II) clean commits always show higher coherence (i.e., lower $LCoh_{Str}$ values). This is true for all the systems with a percentage difference going, on average, from a minimum of +62% structural cohesion in favor of clean commits registered on `Xerces-J` (0.34 vs 0.55) up to a maximum of +161% registered on `JMeter` (1.28 vs 3.34). Moreover, the results of the Mann-Whitney test always report statistically significant difference between the $LCoh_{Str}$ of fix-inducing commits and clean commits (p-value<0.01) with a medium effect size on two systems (i.e., `JMeter` and `log4j`) and a small effect size on the remaining three systems.

Looking at Table II, the much lower structural coherence of commits is rather evident in Xerces-J, both for fix-inducing and for clean commits. The reason is due to the peculiar modularization of Xerces-J. Table IV shows the number of files and packages for each of the subject systems as well as the average number of files per packages. From Table IV it is clear that Xerces-J has a higher average number of files per package with respect to the other systems. This means that a commit performed in Xerces-J is more likely to involve files only spread in few packages, thus leading to low values of $LCoh_{Str}$ (i.e., high structural coherence). On the other side, systems exhibiting a low average number of files per package are more likely to exhibit high values for $LCoh_{Str}$.

Table IV. Files and Packages of the Subject Systems

System	Files	Packages	Avg. Files per Package
Ant	2257	322	7.0
JMeter	2400	393	7.5
log4j	642	107	6.0
Tomcat	2671	420	6.3
Xerces-J	1270	107	11.9

Table V. Number of involved files in fix-inducing and clean commits.

System	Fix-inducing Commits Mean	Median	St. Dev.	Clean Commits Mean	Median	St. Dev.	p-value	d
Ant	4.68	3.00	5.99	2.88	1.00	4.84	<0.01	0.34 (Medium)
JMeter	4.64	2.00	6.64	2.10	1.00	3.47	<0.01	0.36 (Medium)
log4j	7.35	4.00	8.60	3.06	1.00	4.92	<0.01	0.46 (Medium)
Tomcat	4.85	3.00	6.61	2.47	1.00	3.81	<0.01	0.38 (Medium)
Xerces-J	7.16	2.00	10.84	2.52	1.00	4.39	<0.01	0.32 (Small)

Number of Files

As a further check, we also tried removing commits only modifying a single file from our analysis (and thus from both clean and fix-inducing commit sets). This was done to check if the achieved results are just due to the fact that clean commits are more often focused on a single source code file (achieving the lowest possible value of $LCoh_{Str}$) than fix-inducing commits. The performed analysis nearly entirely confirmed the results achieved by considering all the commits:

- all the systems' clean commits have higher *structural* coherence as compared to fix-inducing commits;
- there is statistically significant difference on four out of five subject systems (i.e., all but `Xerces-J`).

To even further analyze possible influences of the commits' size on the results achieved for $LCoh_{Str}$, Table V reports descriptive statistics of the number of files involved in fix-inducing and in clean commits. As we can see, fix-inducing commits generally involve a higher number of files. Since the $LCoh_{Str}$ formula (see Section 2.3) poses the number of files modified in a commit ($|F_{C_k}|$) to the denominator, a higher number of files modified in a commit pushes toward a lower $LCoh_{Str}$ value, and thus to a higher structural coherence. Thus, if there is some bias due to the commit size, this contributes to increasing the structural coherence of buggy commits, clearly reinforcing our findings. Indeed, despite the higher number of files modified in fix-inducing commits, the clean ones still exhibit a higher structural coherence (see Table V).

The higher coherence of clean commits with respect to fix-inducing commits is also confirmed after analyzing the results of the *lexical* coherence metrics (see Table III). Indeed, on all five systems, the *lexical* coherence of clean commits is higher (i.e., higher Coh_{Lex} values) and in particular, it goes from the +20% (on average) achieved on `Xerces-J` (0.59 vs 0.83) up to the +48% achieved on `log4j` (0.50 vs 0.74). The results of the Mann-Whitney test confirm a significant difference in terms of Coh_{Lex} in favor of clean commits on all the subject systems with a small effect size on `Xerces-J` and medium effect size on the other four systems. As previously done for the *structural* cohesion, also in this case we recomputed results by excluding commits involving only one source code file. Again, the results were almost inline with what achieved if considering all the commits:

- all the systems' clean commits have higher *lexical* coherence as compared to fix-inducing commits;
- there is statistically significant difference on four out of five systems considered in our study (i.e., all but `Xerces-J`).

Table VI. Developer's experience in fix-inducing and clean commits: Descriptive statistics, Mann-Whitney test (p-value) and Cliff's delta (d).

Lexical Experience (Exp_{Lex})

System	Fix-inducing Commits			Clean Commits			p-value	d
	Mean	Median	St. Dev.	Mean	Median	St. Dev.		
Ant	0.39	0.35	0.17	0.33	0.29	0.16	<0.01	-0.26 (Small)
JMeter	0.28	0.26	0.12	0.24	0.22	0.12	<0.01	-0.24 (Small)
log4j	0.37	0.34	0.17	0.31	0.27	0.17	<0.01	-0.19 (Small)
Tomcat	0.32	0.29	0.13	0.29	0.26	0.14	<0.01	-0.15 (Small)
Xerces-J	0.42	0.43	0.18	0.33	0.30	0.16	<0.01	-0.31 (Small)

Recent Lexical Experience (Exp_{RecLex})

System	Fix-inducing Commits			Clean Commits			p-value	d
	Mean	Median	St. Dev.	Mean	Median	St. Dev.		
Ant	0.47	0.43	0.18	0.41	0.38	0.18	<0.01	-0.22 (Small)
JMeter	0.32	0.30	0.12	0.27	0.24	0.13	<0.01	-0.25 (Small)
log4j	0.43	0.39	0.19	0.36	0.32	0.18	<0.01	-0.22 (Small)
Tomcat	0.37	0.33	0.15	0.34	0.30	0.16	<0.01	-0.14 (Small)
Xerces-J	0.45	0.45	0.17	0.37	0.34	0.16	<0.01	-0.28 (Small)

Frequency Experience (Exp_{Freq})

System	Fix-inducing Commits			Clean Commits			p-value	d
	Mean	Median	St. Dev.	Mean	Median	St. Dev.		
Ant	0.04	0.01	0.08	0.02	0.01	0.06	<0.01	-0.18 (Small)
JMeter	0.04	0.01	0.06	0.02	0.01	0.06	<0.01	-0.26 (Small)
log4j	0.04	0.02	0.06	0.03	0.01	0.06	<0.01	-0.22 (Small)
Tomcat	0.05	0.01	0.09	0.04	0.01	0.09	<0.01	-0.12 (Small)
Xerces-J	0.09	0.02	0.17	0.04	0.01	0.09	<0.01	-0.19 (Small)

Recent Frequency Experience ($Exp_{RecFreq}$)

System	Fix-inducing Commits			Clean Commits			p-value	d
	Mean	Median	St. Dev.	Mean	Median	St. Dev.		
Ant	0.06	0.02	0.09	0.04	0.01	0.08	<0.01	-0.09 (Small)
JMeter	0.05	0.02	0.07	0.03	0.01	0.06	<0.01	-0.26 (Small)
log4j	0.06	0.03	0.09	0.04	0.01	0.08	<0.01	-0.23 (Small)
Tomcat	0.07	0.01	0.12	0.06	0.01	0.12	<0.01	-0.11 (Small)
Xerces-J	0.10	0.02	0.18	0.04	0.01	0.10	<0.01	-0.22 (Small)

Summary for RQ$_1$. The achieved results allow us to reject the null hypothesis H_{0_1}, stating that *clean commits have structural and lexical coherence significantly higher than fix-inducing commits*. This finding is also confirmed on four out of five subject systems when just considering commits involving more than one source code file.

3.2. *RQ$_2$: Is experience level of developers involved in fix-inducing commits any different from that of developers involved in clean ones?*

Table VI reports the descriptive statistics, the results of the Mann-Whitney test (p-value) and the Cliff's delta (d), obtained when comparing the developers' experience (measured using four metrics described in Section 2.3) for clean and fix-inducing commits.

All four experience metrics, on all five subject systems, reveal an unexpected story: *developer's experience is higher for fix-inducing commits than for clean commits*. This difference is also always statistically significant, even if all the comparisons report a small effect size (see Table VI). Note that a similar finding is also discussed by Zeller in his book *Why programs fail*, where the author describes the result of a past experiment reporting Erich Gamma, the master developer of Eclipse,

Table VII. Kendall's τ_B rank correlation between developer's experience and the size of the change.

System	Experience Measure	τ	p-value
Ant	Exp_{Lex}	0.20	<0.01
	Exp_{RecLex}	0.21	<0.01
	Exp_{Freq}	0.22	<0.01
	$Exp_{RecFreq}$	0.18	<0.01
JMeter	Exp_{Lex}	0.18	<0.01
	Exp_{RecLex}	0.19	<0.01
	Exp_{Freq}	0.24	<0.01
	$Exp_{RecFreq}$	0.23	<0.01
log4j	Exp_{Lex}	0.24	<0.01
	Exp_{RecLex}	0.25	<0.01
	Exp_{Freq}	0.32	<0.01
	$Exp_{RecFreq}$	0.30	<0.01
Tomcat	Exp_{Lex}	0.15	<0.01
	Exp_{RecLex}	0.15	<0.01
	Exp_{Freq}	0.17	<0.01
	$Exp_{RecFreq}$	0.15	<0.01
Xerces-J	Exp_{Lex}	0.21	<0.01
	Exp_{RecLex}	0.22	<0.01
	Exp_{Freq}	0.22	<0.01
	$Exp_{RecFreq}$	0.21	<0.01

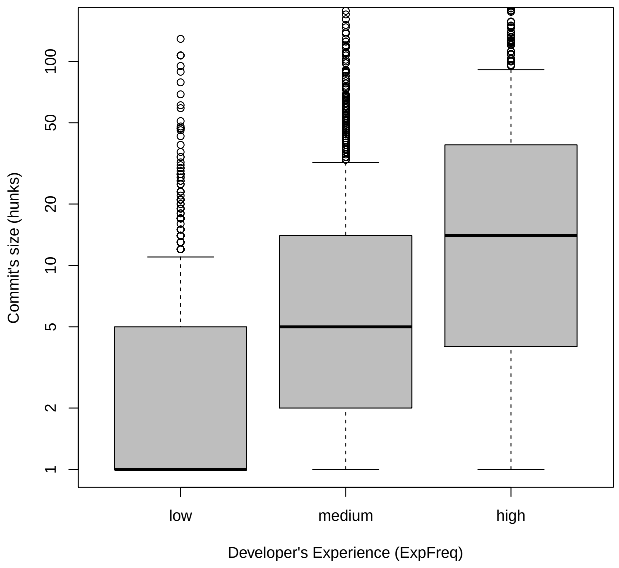

Figure 2. Boxplots of commit's size for developers having different levels of experience (`log4j`). The boxplots, in log-scale, are produced using 768, 1,535 and 768 observations respectively for low, medium and high experience.

as the second most defect-prone Eclipse developer [37]. The explanation provided by Zeller to this surprising result is that more experienced developers tend to perform more complex and critical tasks, thus their commits may be more prone to inducing bugs.

Table VIII. Number and size (LOCs) of interfering changes preceding fix-inducing and clean commits: Descriptive statistics, Mann-Whitney test (p-value) and Cliff's delta (d).

Number of interfering changes

System	Fix-inducing Commits			Clean Commits			p-value	d
	Mean	Median	St. Dev.	Mean	Median	St. Dev.		
Ant	5.34	1.00	17.09	2.70	0.00	11.66	<0.01	-0.21 (Small)
JMeter	2.00	0.00	9.92	0.77	0.00	3.25	<0.01	-0.09 (Small)
log4j	1.25	0.00	5.59	0.85	0.00	4.44	<0.01	-0.09 (Small)
Tomcat	4.86	1.00	36.02	2.56	0.00	20.25	<0.01	-0.10 (Small)
Xerces-J	1.38	0.00	6.10	1.15	0.00	2.19	0.03	-0.10 (Small)

Size of interfering changes (LOCs)

System	Fix-inducing Commits			Clean Commits			p-value	d
	Mean	Median	St. Dev.	Mean	Median	St. Dev.		
Ant	71.31	1.00	232.31	37.99	0.00	198.11	<0.01	-0.20 (Small)
JMeter	31.94	0.00	213.24	10.51	0.00	67.19	<0.01	-0.09 (Small)
log4j	21.12	0.00	96.87	16.85	0.00	111.43	<0.01	-0.09 (Small)
Tomcat	43.91	1.00	245.89	26.16	0.00	170.07	<0.01	-0.10 (Small)
Xerces-J	50.00	0.00	163.08	40.91	0.00	226.73	0.02	-0.11 (Small)

Thereby, we empirically verify whether the relationship between developers' experience and the likelihood introducing bugs is also valid in the projects we have studied. Specifically, we investigate whether developers tend to perform larger (and thus, likely more complex) changes on source files they have more experience with and smaller (and thus, likely simpler) changes in commits involving code components they have less experience with. To this aim, we compute the Kendall's τ_B rank correlation [36] between the four investigated indicators of developer's experience and the size of the change in terms of code hunks [8]. We employ Kendall's τ_B rank correlation because it makes adjustments for ties as opposed to other procedures such as Spearman's ρ and Kendall's τ_A rank correlation. Cohen *et al.* [14] provided a set of guidelines for interpreting correlation coefficients. It is assumed that there is no correlation when $0 \leq \tau_B < 0.1$, small correlation when $0.1 \leq \tau_B < 0.3$, medium correlation when $0.3 \leq \tau_B < 0.5$, and strong correlation when $0.5 \leq \tau_B \leq 1$. Similar intervals also apply for negative correlations.

The results of the Kendall's τ correlation are reported in Table VII. As we can see, there is always a positive correlation between the four exploited indicators of developer's experience and the size of the change in terms of code hunks. The correlation is almost always small, except in two cases, i.e., Exp_{Freq} and $Exp_{RecFreq}$ for log4j, and always statistically significant (p-value<0.01). To some extent, and within the limits of the small correlations being found, *the higher the developer's experience, the larger (and likely more complex) the changes that she performs during commits*. To have a visual representation of this result, Figure 2 reports the box plots[‡] depicting the distribution of change size (hunks) performed by developers at different levels of experience (Exp_{Freq}) on the `log4j` system. Given Q_1, Q_2, and Q_3 the first, the second, and the third quartile, respectively, of developers experience on `log4j`, we clustered developers into *low* ($Exp_{Freq} < Q_1$), *medium* ($Q_1 \leq Exp_{Freq} < Q_3$), and *high* ($Exp_{Freq} \geq Q_3$) experience. The box plots clearly show an upper trend of commit's size when the developer's experience increases. A similar trend was also observed on the other four systems.

Finally, to check possible impact of hunks and LOC on the results, we also performed the same analysis by considering LOCs as the driver metric for assessing the commit's size. The results were inline with what was obtained by using hunks.

[‡]The y-axis is limited to 150 for the sake of readability.

Summary for RQ$_2$. The achieved results allow us to reject the null hypothesis H_{0_2}, stating that, surprisingly, *fix-inducing commits are performed by developers having higher levels of experience than developers performing clean commits*. However, developers tend to perform slightly larger (and likely more complex) changes when they have higher experience on the code files in the commit. This could partially explain the results of our **RQ$_2$**.

3.3. *RQ$_3$: Do fix-inducing commits involve files that are subject to interfering changes among developers as compared to clean ones?*

Table VIII reports the number of interfering changes (and their size in terms of LOCs) preceding clean and fix-inducing commits. The analysis of Table VIII shows that:

- fix-inducing commits are preceded by a higher number of interfering changes as compared to clean commits. Moreover, the size of such interfering changes is larger in fix-inducing commits. Indeed, the average size of such interferences in terms of LOCs is 44 for fix-inducing commits versus 26 for clean commits (+69%);
- the difference between the number and size of interfering changes between fix-inducing and clean commits is always statistically significant even with small effect size.

Thus, in fix-inducing commits a developer generally works on source code files that have been modified more times and in a sharper way, from the last commit she performed on the same files, with respect to code files involved in clean commits. As discussed in Section 2.3, the large number of changes performed on such files could have hindered the developers' understanding of these files.

Summary for RQ$_3$. The achieved results allow us to reject the null hypothesis H_{0_3}, stating that *the number and size of interfering changes preceding fix-inducing commits are significantly higher than clean commits*. However, the factor has always a small-effect size. Thus, its influence could be limited in explaining the phenomenon of fix-inducing commits.

3.4. *RQ$_4$: Are the analyzed developers related factors orthogonal to structural properties of commits studied in previous works?*

Table IX shows the results of the PCA. In particular, the first row reports the proportion of variance of different components or, in other words, the percentage of the observed phenomenon (i.e., the likelihood for a commit of inducing a bug-fix) that they capture. We only report data for the first four principal components resulting from the PCA (i.e., PC1, PC2, PC3, and PC4), since those are the most important ones, capturing alone at least 10% of the phenomenon. The second row reports the cumulative proportion captured by the principal components: the four components capture 74% of the variance in the dataset.

Finally, Table IX reports the loadings of each measure in each rotated component. The achieved results suggest that the first component (PC1), accounting for the 32% of the variance in data, is mainly captured by structural measures related to the size of the performed commit. The factors proposed in this paper capture instead the remaining three components. In particular:

- measures related to the *developer's experience* are those better capturing the second component (PC2), accounting for 20% of the variance in the data;
- the number and size of the *interfering changes* are totally orthogonal to the other measures, capturing the third component (PC3) accounting for 12% of variance;
- the *coherence* measures capture the fourth principal component (PC4), responsible for 10% of the variance in the data.

Thus, PCA results allow us to draw two main conclusions:

Table IX. Results of the Principal Component Analysis.

	PC_1	PC_2	PC_3	PC_4
Proportion of Variance	**0.32**	**0.20**	**0.12**	**0.10**
Cumulative Proportion	**0.32**	**0.52**	**0.64**	**0.74**
added_lines	**0.32**	0.21	-0.07	-0.08
modified_lines	**0.36**	0.23	-0.03	-0.05
removed_lines	0.27	0.21	-0.07	-0.12
modified code_hunks	**0.39**	0.25	-0.04	-0.07
#files	0.28	0.05	0.15	0.28
Exp_{RecLex}	0.24	**-0.41**	-0.10	-0.17
Exp_{Lex}	0.24	**-0.41**	-0.11	-0.14
$Exp_{RecFreq}$	0.22	**-0.40**	-0.15	-0.10
Exp_{Freq}	0.23	**-0.40**	-0.16	-0.08
Coh_{Lex}	-0.22	0.12	-0.15	**-0.59**
$LCoh_{Str}$	0.16	-0.14	0.14	**0.60**
#interfer. ch.	0.08	-0.09	**0.65**	-0.23
size interfer. ch.	0.10	-0.08	**0.65**	-0.24

1. the factors investigated in this paper are orthogonal to structural properties of commits analyzed in previous work, and, thus, capture different principal components;

2. the four analyzed experience measures capture the same principal component (PC2) as well as the two investigated coherence measures (PC4). This suggests that they are highly correlated and thus considering just one of them (i.e., one for the developer's experience and one for the commit's coherence) should be enough to highlight risky commits requiring particular attention during verification and validation activities.

Summary for RQ$_4$. The achieved results allow us to say that the analyzed developers related factors are orthogonal to structural properties of commits studied in previous works.

4. THREATS TO VALIDITY

This section discusses the threats that might affect the validity of the results of our empirical study. In the context of our study, threats to *construct validity*—that concern the relation between the theory and the observation—are mainly due to the measurements that we performed. Specifically, construct validity threats are related to:

- *missing or wrong links between bug tracking systems and versioning systems* [12]: although not much can be done for missing links, as explained in the design we verified that links between commit notes and issues are correct;

- *imprecision in issue classification made by issue-tracking systems* [9]: all the systems considered in our study used Bugzilla as an issue tracking system. Thus, we were able to distinguish between bugs and other issues (e.g., enhancement) by looking at the issue type/severity field. Still, possible misclassifications in the issue tracker (e.g., classifying a bug as an enhancement) are possible and could affect our results [22].

- *approximations due to identifying fix-inducing changes using the SZZ algorithm* [25]: at least we used heuristics to limit the number of false positives by excluding blank lines and comments from the set of fix-inducing changes.

- *approximation of the amount of change*: we used hunks instead of simple LOC to measure the amount of change performed by a developer on a file. However, other (possibly more precise) techniques could have been used, such as AST Diff or LSDiff. Studying to what extent such techniques impact the measurement of the amount of changes is part of the agenda of our future work.

Other threats that could affect the validity of our results are related to external factors we did not consider that could affect the variables being investigated (*internal validity*). Besides the three factors analyzed in our study, there are other characteristics of commits that can impact the likelihood of introducing bugs. However, the aim of this study is to provide a new reading key of the phenomenon taking into account factors that are related to developer's mental model. Also, we used PCA to statistically analyze the complementarity of the factors analyzed in this paper with structural characteristics of commits previously studied. We plan to combine our factors with those in the related work aiming at building a comprehensive prediction model for detecting fix-inducing changes.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. Although this is mainly an observational study, wherever possible we used appropriate statistical procedures, integrated with effect size measures that, besides the significance of the differences found, highlight the magnitude of such differences.

Threats to *external validity* concern the generalization of results. We analyzed five different systems. However, other systems should be analyzed to confirm or contradict our conclusions. In future studies, we are also planning to use diversity measures to guide the selection of subject systems to maximize the generalizability of case study results [30].

5. RELATED WORK

Our study is mainly related to studies on mining software change history aimed at identifying buggy patterns and/or performing bug prediction/identification.

Eyolfson *et al.* [17] analyzed the impact on the commits' bugginess of three characteristics pertinent to commits: (1) time of the day, (2) day of the week, and (3) developer's experience. Note that their definition of developer's experience considered the days of participation in the project and then is different from the one we used in this paper, which directly links experience with the changed files. Their results show that: (i) late-night commits are buggier than others, (ii) no day is buggier than another, and (iii) more experienced developers introduce bugs less frequently. This latter result contradicts the results obtained in our study. However, this is likely due to the different definitions of experience used. Also, the finding related to the absence of a buggier day of the week is in contrast with what observed by Sliwerski *et al.* [33], that identified Friday as the day when developers are likely to introduce bugs.

Rahman and Devanbu [32] investigated the impact of the ownership and the developer experience on the likelihood of introducing bugs. The authors mined software repositories looking for pieces of code modified to fix a bug; this code was tagged as *implicated code*. The analysis of this implicated code highlights that: (i) implicated code has higher ownership levels than non-implicated code, (ii) implicated code owner has lower contribution at a file level, and (iii) general experience of an author has no clear association with implicated code. Also, in this work the definition of developer's experience is different from the one used in our paper. Indeed, even though the authors defined a specialized experience on a particular file, this metric does not take into account the developer's background. Similar observations hold for the ownership metric, that besides ignoring the developers' background does only refer to the highest contributor of that file.

Williams and Hollingsworth [35] described an approach to refine the search for bugs in existing tools. Their technique is based on source code change history mining and focuses on specific type of bugs related to functions' return type. Firstly, their approach checks if during the history of a system under analysis a conditional statement has been added before the use of the value returned by a function. This results in a list of functions that have been subject of a *function-return-value* bug fix. If a change is performed to a piece of code invoking one of the functions present in the list and the returned value is not checked, a bug warning is generated.

Hassan and Holt [20] presented four heuristics aimed at computing the ten most fault-prone subsystems in a project. The information exploited by these four heuristics are based on information mined from software repositories, and in particular they focus on subsystems most frequently (recently) modified and most frequently (recently) fixed.

Livshits and Zimmerman [28] presented DynaMine, a tool combining revision history mining and dynamic analysis techniques to find bugs. DynaMine learns common patterns from the versioning system history and identifies violations to these patterns during software evolution.

Kim *et al.* [25] used a machine learning-based classifier to determine if a performed change is more likely to be a fix-inducing change or a clean one. Their approach takes into account commit's characteristics different from those considered in this paper, and in particular the terms added and deleted while performing the change (i.e., added and deleted delta), the name of the files subject to the commit activities, the terms in the commit message, the author and time of the commit, and the complexity of the modified files.

Bird *et al.* [13] mined change history of `Windows Vista` and `Windows 7` to verify the existence of a relationship between code ownership and software quality. They found that high levels of ownership are associated with fewer bugs. This result is somewhat related to what we observed in our third research question (**RQ**$_3$); changes performed by different developers on the same files increase the likelihood of introducing bugs when working on them.

Posnett *et al.* [31] analyzed the impact of developer's focus on bug introduction. In particular, their "focus" metrics are based on the idea that a developer performing most of her activities on a single module (a module could be a method, a class, etc.) has a higher focus on the activities she is performing and is less likely to introduce bugs. Following this conjecture, they defined two symmetric metrics, namely the *Module Activity Focus* metric (shortly, *MAF*), and the *Developer Attention Focus* metric (shortly, *DAF*) [31]. The former is a metric which captures to what extent a module receives focused attention by developers. The latter measures how focused are the activities of a specific developer. Their results show that project leaders and top committers tend to be less focused on specific aspects of the system; this result is related to our second research question (**RQ**$_2$), where we found that more expert developers work on larger (and likely more complex) tasks. Also, they showed that developers having specific focus introduce fewer bugs. While this finding is related to our **RQ**$_3$, note that we analyzed the focus of single commits, while Posnett *et al.* [31] analyzed the overall focus of developers.

Hassan and Zhang [21] investigated the introduction of integration bugs in large teams working on the same software product. They proposed an approach based on decision trees able to correctly predict 69% of builds that will cause integration bugs. While related, our work has clearly different scope, focusing on factors that might influence the introduction of any type of bug.

Khoshgoftaar et al. [24] used classification and regression trees to predict faulty software modules by exploiting as predictor variables source code characteristics, configuration management transactions, and problem reporting transactions. These factors are different with respect to the ones investigated in our study.

6. CONCLUSION AND FUTURE WORK

In recent and past years, researchers studied factors that could potentially influence the likelihood of developers introducing bugs when performing changes. The results achieved so far indicate that commits' time and day [17, 33], the ownership level of the files involved in the commit [32, 13, 31], the experience of a developer responsible for the commit [17, 32, 31], the complexity of the files involved in the commit [25], and the lexical properties of the commits are all the factors that play important roles to discriminate between "clean" and "fix-inducing" commits.

We conjectured that when studying the characteristics of fix-inducing changes, there are other important factors beyond those previous researchers have analyzed. To the best of our knowledge, none of the previous studies has taken into account the role of the developer's understanding of the change to be committed. Indeed, it is reasonable to think that an inconsistent understanding of a developer could be one of the reasons for introducing a bug. For this reason, we defined three new measures aimed at capturing three different, *previously unexplored*, factors that could potentially hinder developers' ability to correctly understand the change to be performed. Specifically, we analyzed (i) the structural and lexical coherence of the files involved in the commit, i.e., the degree that files are focused on a specific topic/subsystem; (ii) the experience of the developer on the files involved in the commit; (iii) the interfering changes performed by other developers on the files in the commit.

The results of our study can be summarized as follows:

- *Clean commits have lexical and structural coherence (the degree that files are focused on a specific topic/subsystem) significantly higher than the one of bug-inducing commits*. This means that the likelihood to introduce a bug is higher when a developer has to work on different things/topics at the same time;
- *Fix-inducing commits are performed by developers having higher levels of experience than developers performing clean commits*. At the first glance, such result could be quite counterintuitive. However, we further observed that experienced developers are usually in charge of performing more complex tasks. Here the reason why more experienced developers are usually the authors of bug-inducing commits. This confirms the findings achieved by Zeller *et al.* when studying the Eclipse ecosystem [37]; and
- *Fix-inducing commits are preceded by a higher number of interfering changes performed by other developers*.

While our study is mainly observational in nature, its findings can help researchers and practitioners in designing tools able to intercept "risky" commits (i.e., commits performed under conditions promoting the introduction of bugs) which might require particular attention during verification and validation activities. As future work, we are planning to replicate our study on other software systems in order to corroborate our findings. In addition, we plan to further analyze the complementarity of the factors defined in this paper together with those defined in the literature. We are also planning to further investigate developers' experience, in particular, the usage of *localised expertise* of the changes performed by a developer in a commit. Such an analysis could better capture the real developer's expertise on a file. Indeed, especially for large source code files, a developer could have only a partial understanding of the code, limited to the code fragments she actually maintains. Moreover, we plan to investigate whether our defined metrics can actually capture the developer's ability to build a correct mental model of the change she performs in the software system. This would require controlled experiments with real developers where we analyze the impact of commits' coherence and interfering changes. Last but not least, we are planning on using all these factors in order to build a comprehensive predictive model for classifying "clean" and "fix-inducing" commits on top of existing approaches for detecting/predicting defects [16]. Particular attention will be devoted to carefully assessing the advantages provided by the defined factors over existing predictors, by introducing the new factors only when the increase in performances of the prediction model justifies the consequent increase in the model complexity (due to the need for computing the new factors). Such a model could be integrated in a recommender system that may be particular useful to highlight commits (and, thus, components) that require particular attention during verification and validation activities.

REFERENCES

1. http://ant.apache.org/.
2. http://jmeter.apache.org/.
3. http://logging.apache.org/log4j/.
4. http://tomcat.apache.org/.
5. http://xerces.apache.org/xerces-j/.
6. http://www.bugzilla.org/.
7. http://www.cs.wm.edu/semeru/data/JSEP15-Fix-Inducing-Commits/.
8. A. Alali, H. Kagdi, and J. I. Maletic. What's a typical commit? a characterization of open source software repositories. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 182–191. IEEE Computer Society, 2008.

9. G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. 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*, page 23. IBM, 2008.
10. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.
11. G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto. Using structural and semantic measures to improve software modularization. *Empirical Software Engineering*, 18(5):901–932, 2013.
12. C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu. 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*, pages 121–130. ACM, 2009.
13. C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 4–14. ACM, 2011.
14. J. Cohen. *Statistical power analysis for the behavioral sciences*. Lawrence Earlbaum Associates, 1988.
15. W. J. Conover. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998.
16. M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, Aug. 2012.
17. J. Eyolfson, L. Tan, and P. Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 153–162, New York, NY, USA, 2011. ACM.
18. M. Gethers and D. Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems. In *Proceedings of the 26th IEEE International Conference on Software Maintenance, 2010*, pages 1–10, 2010.
19. R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.
20. A. E. Hassan and R. C. Holt. The top ten list: dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005*, ICSM '05, pages 263–272. IEEE Computer Society, 2005.
21. A. E. Hassan and K. Zhang. Using decision trees to predict the certification result of a build. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 189–198, Washington, DC, USA, 2006. IEEE Computer Society.
22. K. Herzig, S. Just, and A. Zeller. It&#039;s not a bug, it&#039;s a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.
23. I. Jolliffe. *Principal Component Analysis*. Springer, 1986.
24. T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Data mining for predictors of software quality. *International Journal of Software Engineering and Knowledge Engineering*, 9(5):547–563, 1999.
25. S. Kim, E. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.
26. M. M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.
27. M. M. Lehman, J. F. Ramil, P. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *4th IEEE International Software Metrics Symposium (METRICS 1997), November 5-7, 1997, Albuquerque, NM, USA*, page 20, 1997.
28. B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 296–305. ACM, 2005.
29. L. Moonen. Generating robust parsers using island grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 13–22, 2001.
30. M. Nagappan, T. Zimmermann, and C. Bird. Diversity in software engineering research. In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, August 2013.
31. D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov. Dual ecological measures of focus in software development. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 452–461. IEEE Press, 2013.
32. F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 491–500, 2011.
33. J. Sliwerski, T. Zimmermann, and A. Zeller. Don't program on fridays! how to locate fix-inducing changes. In *Proceedings of the 7th Workshop Software Reengineering*, May 2005.
34. J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005*. ACM, 2005.
35. C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering (TSE)*, 31(6):466–480, 2005.
36. J. H. Zar. Significance testing of the spearman rank correlation coefficient. *Journal of the American Statistical Association*, 67(339):pp. 578–580, 1972.
37. A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.