Fault-Prone Source File Analysis Focusing on the Contribution Entropy in Open Source Development

Kazuki Yamauchi*, Hirohisa Aman†, Sousuke Amasaki‡, Tomoyuki Yokogawa‡ and Minoru Kawahara†
*Graduate School of Science and Engineering, Ehime University
Matsuyama, Ehime, 790–8577 Japan
†Center for Information Technology, Ehime University
Matsuyama, Ehime, 790–8577 Japan
‡Faculty of Computer Science and Systems Engineering, Okayama Prefectural University
Soja, Okayama, 719–1197 Japan

***Abstract*—Open source software (OSS) products have been widely used for information systems, and a successful quality management of OSS development has become one of key topics in the information technology world. Since the development and maintenance of an OSS product is driven by various developers, it would be worthy to focus on their contributions and the cooperative structure. This paper proposes to measure a developer's contribution to a source file as the cumulative lines of code that he/she has changed on the file, and to evaluate the balance of contributions to the file among different developers in a form of an entropy. Through an empirical study using data collected from 10 major OSS projects, the following findings are reported: (1) a source file which has been maintained by two or more developers (entropy$>$0) is about two times more likely to be faulty than a file which has been done by only a single developer (entropy=0); (2) when two or more developers have maintained a source file and their contributions are more imbalanced (a lower entropy), the source file is more fault-prone.**

I. Introduction

In recent years, open source software (OSS) products have been more and more popular in the information technology-related business world [1]. For example, commercial software products may partially leverage OSS products, or commercial services may run on OSS-based servers. As OSS products become more popular, the importance of their successful quality managements gets higher. Source files of OSS products and their development history data (commit logs) are almost always available to everyone through their software repositories, and those source files and history data are useful research materials for the quality management of OSS products. Indeed, the mining software repositories (MSR) [2] has been one of the hottest topics in the field of software engineering, and there have been many MSR studies: for example, fault-prone program predictions based on the fix history [3], [4]; maintenance supports focusing on the logical couplings (co-change relationships) among source files [5], [6]; assessments of code change impacts [7], [8]; just-in-time quality assurance methods [9], [10], etc.

More recently, the MSR studies have focused on not only the OSS products and their development processes but also the developers [11], [12], [13], [14], [15]. That is to say, in addition to the analysis of "what was changed in a source file" and "when the change was made," MSR studies have focused on "who made the changes" and "how the developers have been involved" as well. Since a success of an OSS project usually depends on the contributions by developers, it is worthy enough to understand developers' contributions and the collaborations among different developers in more depth toward a better quality management of OSS project. As a part of a developer-oriented MSR study, we focuses on a structure for cooperation among developers in the development and maintenance of a source file, and tackle the following two questions in this paper. (1) Is there a difference in the code quality between a multi-developer file (a source file which has been developed and maintained by two or more developers) and a single-developer file (a source file which has been done by a single developer)? (2) when a source file has been maintained by two or more developers, does the balance of their contributions have an impact on the code quality?

The first question is on whether a source file has been maintained by only a single developer or not. In many OSS projects, while many developers have joined in the development and maintenance of source files, there are also source files which have been maintained by a certain developer only—no other person touched these files. We will empirically study if the difference of contribution style, "single" vs. "multi," has a significant impact on the code quality in terms of the fault-proneness.

The above second question is on an detailed insight into the contributions by two or more developers. In this paper, a developer's contribution to a source file is quantified by the cumulative number of source code lines which the developer has added to or deleted from the source file[1]. When n (≥ 2) developers have contributed to a source file, there are a lot of variations in the balance of contributions among n developers. For example, one of the developers may have dominantly contributed to the source file and the contributions by the remaining $n-1$ developers may be really small. For another example, all of n developers may have approximately equally contributed to the source file. The difference of these two example cases may cause a remarkable difference in the fault-proneness of source files. We will evaluate the balance of

[1]A code modification is interpreted as a pair of code addition and deletion.

developers' contributions by introducing a metric using the notion of entropy, and analyze the relationship between the metric value and the fault-proneness.

The remainder of this paper is organized as follows. Section II explains our research questions and proposes our measure of developers' contributions. Section III reports our data analysis and discusses the results. Section IV briefly describes the related work. Finally, Section V presents our conclusion and future work.

II. Developer's Contribution

A source code repository stores source files and their change history. Thus, for each source file, we can easily obtain the following information from the repository: "when the file was created," "when the file was changed," "what was changed in the file (the difference between before and after the change)," "who made the creation/change," etc. By analyzing these data, we can observe the contributions of developers to the development and maintenance of the source file. For example, given a source file, we can investigate "who have been involved in its development and maintenance" and "how many source lines of code have been added, deleted or changed by each of the developers." They form a history of contributions by developers. Different source files have different histories of developers' contributions. While one source file may have been developed and maintained by a certain developer only, another source file may have been done by many different developers. Moreover, for the latter kind of source file, there would be many variations in the structure for cooperation among developers. For example, suppose two developers d_A and d_B have contributed to a set of source files. While one source file may have been dominantly maintained by d_A and the contribution by d_B is a little, another source file may have been evenly maintained by both d_A and d_B. Our research interest is to analyze how such differences of contributions affect the code quality, especially, the fault-proneness of source files.

Notice that we focus only on the change history stored in the code repository in order to see a developer's contribution; there are also other types of contributions such as the code reviews, tests and discussions. Although it is ideal to take into account all kinds of contributions, it is hard to collect all of such data from any OSS project. Since the code repository is commonly available at any OSS project, we will limit our focus of "contribution" to the code changes in this paper.

A. Research Questions

According to our research interest mentioned above, we now set our research questions (RQs) as follows.

- RQ1: Is there a difference in the code quality between a source file which has been developed and maintained by two or more developers and a source file which has been done by a single developer?
- RQ2: When a source file has been maintained by two or more developers, does the balance of their contributions have an impact on the code quality?

We explain the reasons why we set the above questions.

RQ1 is a fundamental question on the difference of developers' contributions to source files. Needless to say, a source file can be classified into two categories: 1) a source file which has been developed and maintained by two or more developers, and 2) a source file which has been done by a single developer. For the sake of convenience, we will call the former source file as "multi-developer file" and the latter one as "single-developer file." An involvement of more developers to the maintenance of a source file might be better to enhance the code quality because the code would be reviewed by more people. However, there might also be an opposite effect: such an involvement might cause unnecessary confusion in the coding. Shortly, RQ1 is our simple question: "which is better" in terms of the code quality, multi-developer file or single-developer file?

Next, RQ2 is a further question on how developers contribute to the development and maintenance of source files. As mentioned above, there would be various structures for cooperation in the maintenance of multi-developer files. The aim of RQ2 is to clarify whether a difference in the structure for cooperation is related to the code quality or not, from the perspective of the balance of developers' contributions to multi-developer files.

In this paper, we will assess a code quality of a source file by the fault-proneness of the files. Through data analyses on the RQ1 and RQ2, we would be able to see what kind of source files are fault-prone and require more careful review, from a point of view on a structure of contributions by developers.

B. Contribution Entropy

For our data analysis on RQ2, we now introduce a novel metric which is called "contribution entropy." This metric focuses on the source lines which have been changed by each developer, and it quantifies the balance of these changed lines among developers as an "entropy." The notion of entropy is well-known as a measure of information randomness in the information theory [16]. We apply this notion to an evaluation of the balance among developers' contributions (the amount of changed lines) to a source file. Shortly, we consider that a source file has a high contribution entropy if the file has been evenly changed by two or more developers because the randomness of contribution is high; if the contributions are dominant by a certain developer, the randomness is low, so the contribution entropy of the file is low.

As a previous study, Taylor et al. [11] proposed the author entropy. They focused on "who is the author of each source line" in a source file. Fig. 1 presents a simple example of a code fragment showing the author of each line[2], where they considered the author of line to be who edited it last. Then, the author entropy evaluates the balance of source lines among authors (developers). While the author entropy is an attractive measure of developers' contributions to a source file, there is

[2]This example is based on the result of running "git blame" for src/main/java/io/reactivex/Completable.java in RxJava.

author	line#	code
d_1	1	public final void subscribe(CompletableObserver s) {
d_2	2	ObjectHelper.requireNonNull(s, "s is null");
d_3	3	try {
d_1	4	
d_2	5	s = RxJavaPlugins.onSubscribe(this, s);
d_1	6	
d_2	7	subscribeActual(s);
d_2	8	} catch (NullPointerException ex) {
⋮	⋮	⋮

Fig. 1. A simple example showing the author of each source line.

TABLE I
A SIMPLE EXAMPLE OF CODE CHANGE HISTORY.

revision	event
1	d_1 created a source file (100 lines).
2	d_1 deleted 10 lines, and modified 15 lines (deleted 15 & added 15).
3	d_2 modified 20 lines (deleted 20 & added 20), modified 18 lines (deleted 18 & added 18), and deleted 5 lines.
4	d_3 deleted 20 lines.
5	d_2 modified 15 lines (deleted 10 & added 15).

yet another way of evaluating the contributions focusing not only the current code but also the past code. For example, while the author of the second line in Fig. 1 is developer d_2, it was a result of rewriting the line by d_2 and its original code was written by d_3. That is to say, the contribution of d_3 to the second line seems to be overridden by d_2 when we use the author entropy. Since it would better to regard that both d_2 and d_3 have contributed to the second line in Fig. 1, we introduce another notion using the "cumulative" source lines changed by each developer.

There are two key reasons why we focus on the cumulative changed lines rather than the current lines. One reason is that we consider a deletion of source code to be a contribution as well. Even if a developer just deleted some source code from a source file at a commit, it is one of essential modifications to build the current version of the source file. Thus, we should not avoid considering a code deletion in order to evaluate a developer's contribution. Another reason is that there can be two different developers before and after a code modification. Suppose developer d_i modified a source code which was originally written by different developer d_j. Although the modification is a d_i's contribution, it may be based on the original code and that is a d_j's contribution. Hence, it would be reasonable to take into account not only d_i but also d_j for evaluating their contributions.

Let us take a simple example presented in Fig. 2 and Table I. In the example, the source file was originally created by developer d_1 at revision 1 ("Rev-1" in Fig. 2). Through code modifications and deletions by three developers d_1, d_2 and d_3, we obtain the revision 5 and then d_2 is the developer who last touched all lines of the source code (see "Rev-5 by d_2" row in Fig. 2; the green parts and the red ones signify d_1's contributions and d_2's contributions, respectively). Since the previous work [11] focused only on the current version (revision 5), it evaluates that d_1 and d_3 have no contributions to this source file. This example shows an importance of our viewpoint and our motivation in this paper.

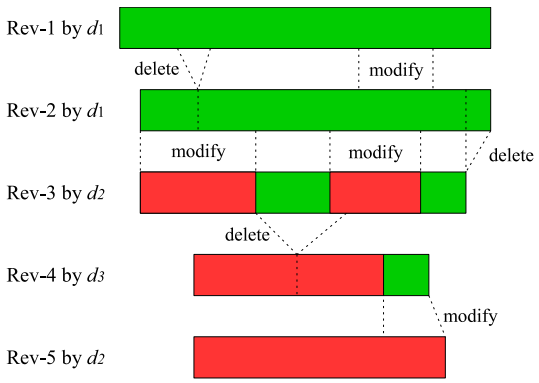

Fig. 2. A simple example of code change history.

Now we define the "contribution entropy" as follows.

Definition 1 (Contribution Entropy):

Given a source file f, and suppose that n developers d_i (for $i = 1, \ldots, n$) have been involved in the development and maintenance of f. Let $c_i(f)$ be the contribution by d_i, which is the total number of source lines added to or deleted from f by d_i until the current version[3]. Then, the following $p_i(f)$ is a measure of the relative contribution to f by d_i (for $i = 1, \ldots, n$):

$$p_i(f) = \frac{c_i(f)}{\sum_{i=1}^{n} c_i(f)} \ . \tag{1}$$

We define the contribution entropy of f as:

$$H(f) = \begin{cases} 0 & (n = 1), \\ \dfrac{-1}{\log_2 n} \displaystyle\sum_{i=1}^{n} p_i(f) \log_2 p_i(f) & (n > 1) \ . \end{cases} \tag{2}$$

Constant $\log_2 n$ in the denominator is used for normalizing the range of $H(f)$ to $[0, 1]$. When $n = 1$, we specially define as $H(f) = 0$ because $\log_2 n = 0$.

□

As we described above, if f is a single-developer file, $n = 1$ so we have $H(f) = 0$. When f is a multi-developer file, $H(f)$ varies from 0 to 1 in accordance with the balance of $p_i(f)$ among developers $\{d_i\}$. If all $p_i(f)$'s are equal, i.e., all developers' contributions to f are truly equal, we get $H(f) = 1$. As the contributions get more imbalanced among developers, $H(f)$ becomes lower. That is to say, as the contribution structure of f gets closer to a certain developer's monopoly, $H(f)$ becomes closer to 0.

[3]One line modification is regarded as one line addition after one line deletion. Thus, the contribution is expressed as two lines of changed code.

In the example case shown in Table I, we have $c_1(f) = 140$, $c_2(f) = 106$ and $c_3(f) = 20$ just after revision 5. From Eqs.(1) and (2), we get

$$p_1(f) = \frac{140}{140 + 106 + 20} \simeq 0.5263,$$

$$p_2(f) = \frac{106}{140 + 106 + 20} \simeq 0.3985,$$

and

$$p_3(f) = \frac{20}{140 + 106 + 20} \simeq 0.0752,$$

then

$$H(f) = \frac{-1}{\log_2 3} \{\, p_1(f) \log_2 p_1(f) + p_2(f) \log_2 p_2(f) + p_3(f) \log_2 p_3(f) \,\} \simeq 0.8183.$$

While all lines of f just after revision 5 look to be ones made by only d_2 (see Fig. 2), d_1 has more contributions than d_2 if we focus on not only the current state but also the change history of f; if we focused only on the current state (revision 5) of f, the contribution is considered to be a monopoly of d_2 and $H(f)$ becomes zero.

III. Data Analysis

On our RQs mentioned in Section II-A, we conducted a data analysis using popular OSS projects. In this section, we report and discuss the results.

A. Data Source

We collected data from popular OSS projects shown in Table II. In order to ensure the generality and usefulness of our results as high as possible, we selected them: they all ranked in top 10 Java projects[4] in terms of "stars" at the GitHub.

TABLE II
Surveyed OSS projects

project	# of source files	# of commits
Butter Knife	125	279
Elasticsearch	5,527	16,671
Glide	609	1,382
Guava	3,131	4,054
Java Design Patterns	1,011	885
Kotlin	5,090	21,406
MPAndroidChart	235	1,216
OkHttp	291	1,344
Retrofit	210	535
RxJava	1,497	731
total	17,726	48,503

[4]This ranking at the end of November 2017.

B. Procedure

We conducted our data collection and analysis in the following procedure.

1) Make a copy of the repository from the GitHub[5].
2) Get the development history of each source file from the repository.
   For each source file included in the current version, we investigate its development history (change logs) by using "git log" command. In the investigation, we trace the renaming operations to the file as well.
3) Extract data items required for our analysis.
   By analyzing the commit logs that we got at step 2), we extract the following items for each commit of each source file: (a) commit hash, (b) whether it is aimed at fixing a fault or not, (c) author's name, (d) author's e-mail address, and (e) changed source lines.
   The item (b) is decided if the corresponding commit message contains a fault-fix-related keyword or not [17].
4) Organize the data collected at step 3).
   Since there are source files which were renamed through commits, we assign unique file IDs to source files in order to identify them.
   There may be an author who uses two or more different names or e-mail addresses on the repository. We tried integrating duplicated author data by the following set of rules—it is a simpler version of the algorithm proposed by Bird et al. [18]: (1) if two authors have different addresses but the same name, then we regard them as the same author; (2) if two authors have the same address but different names, then we regard them as the same author.
5) Analyze the collected data on RQs.
   For RQ1:
   a) For each source file f, we see whether f is a single-developer file or a multi-developer file at the end of the observation period (see Fig. 3), and check if a fault fix is occurred at f during the examination period or not. Here, the observation period signifies the days in which we check the development and maintenance of the source file,

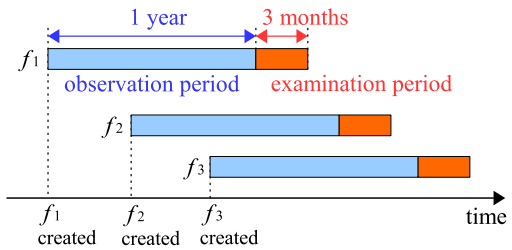

Fig. 3. Observation period and examination period.

[5]The repository URLs are https://github.com/{ akeWharton/butterknife, elastic/elasticsearch, bumptech/glide, google/guava, iluwatar/java-design-patterns, JetBrains/kotlin, PhilJay/MPAndroidChart, square/okhttp, square/retrofit, ReactiveX/RxJava}.git, respectively.

and the examination period corresponds to the days in which we decide the file is faulty or not. Since a new source file may be immature and require some fixes, we empirically set the observation period as 1 year from the day that f was newly created, and the examination period as 3 months from the end of the observation period, respectively.

b) Classify the set of source files into two subsets: the set of single-developer files and that of multi-developer ones. Then, compare their rates of faulty source files (the fault rates: FRs). A higher FR value means a higher fault-proneness of source files in the corresponding subset.

For RQ2:

a) For each source file f, compute its contribution entropy, $H(f)$, at the end of the observation period (see Fig. 3). Then, we check if a fault fix is occurred at f during the examination period or not, which is the same as the above step a) for RQ1.

b) Classify the set of source files into five subsets according to their contribution entropy: C0, C1, C2, C3 and C4. C0 is the set of source files such that $H(f) = 0$, i.e., the set of single-developer files. The remaining source files are categorized into C1 – C4 corresponding to equally divided ranges of $H(f)$: C1, C2, C3 and C4 correspond to $0 < H(f) \leq 1/4$, $1/4 < H(f) \leq 2/4$, $2/4 < H(f) \leq 3/4$ and $3/4 < H(f) \leq 1$, respectively.
After that, compare FRs among C0 – C4.

C. Results for RQ1

On RQ1, we compared the set of single-developer source files and that of multi-developer ones in terms of the fault-proneness. Table III and Fig. 4 show the results. Notice that the total number of source files shown in Table III decreases from the one show in Table II because new source files younger than 15 months (the observation period plus the examination period) are not included in our analysis.

The collected source files were dominated by the multi-developer files: about 62% ($\simeq 9675/15664$) of source files were multi-developer. As a result, the multi-developer files are about two times more likely to be faulty than the single-developer one: 14.73% vs. 7.00%, and their difference was statistically significant at a 5% significance level[6].

TABLE III
COMPARISON OF SINGLE-DEVELOPER SOURCE FILES AND MULTI-DEVELOPER ONES

file type	# of source files	# of faulty ones	FR
multi-developer	9,675	1,425	14.73%
single-developer	5,989	419	7.00%
total	15,664	1,844	11.77%

[6]The null hypothesis "the FR in multi-developer files is equal to the FR in single-developer files" was rejected by a χ^2 test with $\chi^2 = 212.61$, df $= 1$ and p value $< 2.2 \times 10^{-16}$.

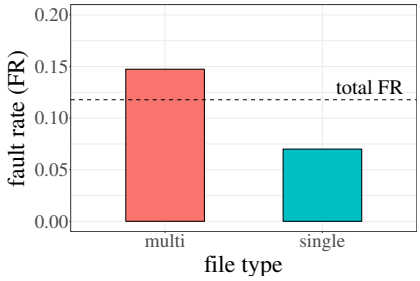

Fig. 4. Comparison of the fault-proneness: single-developer source files vs. multi-developer ones.

D. Discussions for RQ1

We compared multi-developer source files with single-developer ones in terms of the fault-proneness. The results of data analysis showed that a multi-developer file tends to be more fault-prone than a single-developer one. The fault rate of multi-developer files is about two times higher than that of single-developer ones. In general, it would not be easy to appropriately comprehend and update other people's code. Hence, such a difficulty may cause the above striking difference between the two types of source file.

Now we consider a relationship with the program size as well. The lines of code (LOC), one of the famous size metrics, is well-known metric which is highly related to the fault-proneness of programs [19]: a program having a higher LOC is more fault-prone in general. The above trend might be different according to the program size (LOC). Thus, we categorize source files by their LOC values, then compare the FRs between multi-developer files and single-developer ones within each category. Table IV and Fig. 5 show the results, where we have four categories according to the quartile of LOC: (1) LOC ≤ 12, (2) $12 <$ LOC ≤ 41, (3) $41 <$ LOC ≤ 113, and (4) $113 <$ LOC.

As Fig. 5 shows, in the relatively large-sized source files whose LOC categories are (3) or (4), the magnitude relationship of FR between multi-developer files and single-developer files is the same as the total result shown in Fig. 4. Moreover, the gap of FR between file types gets larger as the source file becomes larger. On the other hand, interestingly, the magnitude relationship of FR inverts in the relatively small-sized source files whose LOC categories are (1) or (2). The difference of

TABLE IV
COMPARISONS OF SINGLE-DEVELOPER FILES AND MULTI-DEVELOPER WITHIN LOC CATEGORIES

LOC category	file type	# of source files	# of faulty ones	FR
(1) LOC ≤ 12	multi	1,727	49	2.84%
	single	2,311	98	4.24%
(2) $12 <$ LOC ≤ 41	multi	2,145	105	4.90%
	single	1,718	123	7.16%
(3) $41 <$ LOC ≤ 113	multi	2,715	322	11.86%
	single	1,139	87	7.64%
(4) $113 <$ LOC	multi	3,088	949	30.73%
	single	821	111	13.52%

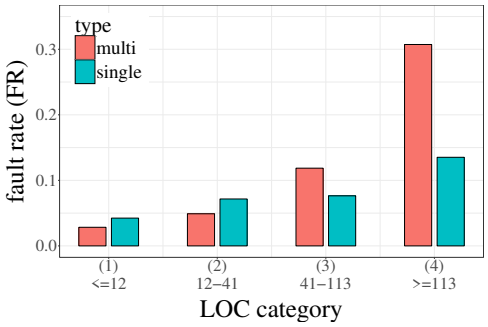

Fig. 5. Comparisons of the fault-proneness: single-developer files vs. multi-developer ones within LOC categories.

FR is statistically significant at 5% significance level in each LOC category[7].

As mentioned above, a successful comprehension and update of other programmer's code is not easy task. The difficulty would increase as the program gets larger. Therefore, it seems to be natural that the gap of FR becomes the largest one in the LOC category (4). Similarly, as the program gets smaller, a proper comprehension and update of code would become easier even if the code was written by other programmers. A smaller program originally tends to be less fault-prone, and furthermore, its quality might be further enhanced through contributions by other developers. Hence, the inversion of the magnitude relationship in FR looks a natural trend as well.

Notice that the above results are derived from the data of "fault fix" but not of "fault injection." There may be a case such that: a fault was injected into a source file when the file was a single-developer one, and then the fault was detected and fixed after the file became a multi-developer one by a participation of another developer. One of key reasons why more faults were detected and fixed in multi-developer files might be that more developers reviewed these files. In other words, a single-developer file might have latent faults which have not been detected yet because the file has been maintained by a certain developer only. Although our focus on whether a source file is a single-developer file or not seems to be noteworthy through this data analysis, we have to do a more detailed analysis in the future: a further analysis on the relationship between the fault injection and the switching of file type from single-developer to multi-developer.

E. Results for RQ2

On RQ2, we classified the source files into five categories C0, C1, ..., C4 by their contribution entropy, and compared these categories in terms of the fault-proneness. Table V and Fig. 6 show the results.

As a result, the least fault-prone category is C0 whose contribution entropy is zero, i.e., the category of the single-developer files. In contrast, the most fault-prone category is C1 which is right next to C0. Category C1 corresponds to source files in which the developers' contributions are highly imbalanced. The fault-proneness (the FR value) decreases as the contribution entropy gets higher, i.e., as the structure of contributions gets away from the most imbalanced state.

[7](1) $\chi^2 = 5.1557$, df $= 1$ and p value $= 0.02317$; (2) $\chi^2 = 8.4042$, df $= 1$ and p value $= 0.003744$; (3) $\chi^2 = 14.634$, df $= 1$ and p value $= 0.0001305$; (4) $\chi^2 = 99.438$, df $= 1$ and p value $< 2.2 \times 10^{-16}$.

TABLE V
COMPARISON OF SOURCE FILES CATEGORIZED BY THEIR CONTRIBUTION ENTROPY

category	# of source files	# of faulty ones	FR
C0	7,087	445	6.28%
C1	1,628	399	24.50%
C2	2,240	412	18.39%
C3	2,624	356	13.57%
C4	2,085	232	11.13%

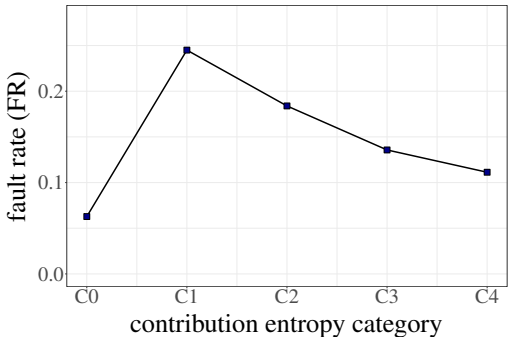

Fig. 6. Comparison of source files categorized by their contribution entropy.

F. Discussions for RQ2

In order to investigate the impact of contribution balance among developers on the fault-proneness, we categorized source files by their contribution entropy into five categories C0 – C4, and compared their FR values. As a result, multi-developer source files in category C1 are the most fault-prone. In such a source file, there are two or more developers who have had contributed to the file, but their contributions are highly-imbalanced and one of the corresponding developers is dominant. That state of source file would be made when the file type was just changed from the single-developer to the multi-developer. By a participation of another developer in the maintenance of a source file, it seems to become unstable in terms of the code quality and the most fault-prone.

As with the above discussion on RQ1, we categorized the source files by their LOC, and examined changes in FR over C0 – C4 as well. Table VI presents the results, where the highest FR within each LOC category is emphasized.

As the results, category C1 was always the most fault-prone regardless of LOC. Thus, we can say with more confidence that multi-developer source files with highly-imbalanced contributions are fault-prone. Such status would be created by a participation of a new developer in the maintenance of a source file. While participations of new developers are important to a successful OSS project operation, the quality of a source file may become transiently unstable when another programmer started to change a part of code. That is to say, it would

TABLE VI
COMPARISONS OF FRS OVER THE CONTRIBUTION ENTROPY CATEGORIES WITHIN LOC CATEGORIES

category		# of source	# of faulty	
LOC	entropy	files	ones	FR
	C0	2,873	98	3.41%
(1)	C1	45	3	**6.67%**
LOC $\leq$ 12	C2	96	2	2.08%
	C3	219	14	6.39%
	C4	498	13	2.61%
	C0	2,062	130	6.30%
(2)	C1	232	15	**6.47%**
12 <	C2	491	23	4.68%
LOC $\leq$ 41	C3	618	24	3.88%
	C4	694	44	6.34%
	C0	1,252	95	7.59%
(3)	C1	485	84	**17.32%**
41 <	C2	744	114	15.32%
LOC $\leq$ 113	C3	888	71	8.00%
	C4	512	52	10.16%
	C0	900	122	13.56%
(4)	C1	866	297	**34.30%**
113 < LOC	C2	909	273	30.00%
	C3	899	247	27.48%
	C4	381	123	32.23%

be useful to pay more attention to such a source file for a successful OSS project management.

We can consider the following two potential reasons why C1 is the most fault-prone category of source file: (1) a fault was created by a new developer through a code modification, or (2) a fault was detected by a new developer. In general, a code modification has a risk of a fault creation [20], [21]. Since it is not easy task to accurately comprehend and properly change a source code written by other programmers, the fault-creation risk would get higher when a code change is made by a new developer. In another perspective, a participation of a new developer includes another code review. Then, a latent fault might be detected as a result of code review by a new developer. In both cases, more faults may be detected from source files of category C1. On the other hand, source files in categories C2, C3 or C4 tend to be more changed and reviewed more times by two or more developers. Through those code changes and reviews, the source files may be polished and become less fault-prone. Because our data is based on fault-fix events in the repository, we cannot clearly analyze the above potential reasons in this study. In order to examine the reason why C1 is so fault-prone and the fault-proneness gets lower as the contribution entropy becomes higher, we need to perform a further analysis using data of "fault injections" in the future.

G. Threats to Validity

We analyzed OSS projects whose main-development language is Java. Since our data collection method does not depend on any Java-specific feature, we can perform a similar analysis to other projects developed in other languages without any changes of our method if their code is maintained with Git. Since OSS projects written in other languages may show different trends, it is a threat to the validity regrading the generality of our findings, and our significant future work.

In order to ensure quality of code, some OSS projects have strict checking (reviewing) systems for a code change proposed by a developer. Such a checking system may have a big impact on our results of data analysis. To mitigate such a threat to validity, we collected a lot of data from various OSS projects in different domains. We plan to perform a further analysis focusing on project-specific features in the future.

IV. RELATED WORK

Bird et al. [12] focused on an ownership of a source file: an ownership metric in their paper is the percentage of commits made by major contributors. They reported that a source file having lower ownership is likely to be more fault-prone. Aman et al. [22] performed a survival analysis of the time to fault fix, and proved the trend such that a source file modified by a new developer has a shorter time to the occurrence of the next fault fix. These previous work showed a risk of transiently increasing the fault-proneness of a source file by a participation of other developers in the maintenance of the file. While the notion of such a risk is common to this study, we performed a finer-grained data analysis which is at a source line-change level rather than a commit level.

Matsumoto et al. [23] proposed developer metrics and utilized those metrics for predicting faults in modules. Through an empirical study using a dataset of the Eclipse Platform project, they showed that their developer metrics can be useful explanatory variables in fault-prone module prediction models. Moreover, they reported that a module which has been touched by more developers tends to be more fault-prone. While their findings are consistent with our result for RQ1 mentioned above, we have also performed an analysis of the relationship with the source file (module) size in this paper. Furthermore, we have studied multi-developer files in terms of the balance of developers' contributions rather than the number of developers.

Posnett et al. [13] focused developers' contributions from both the perspective of developer and that of module. In the former perspective, they quantified the degree to which a developer contributes to a certain module; in the latter one, they evaluated the degree to which a module is contributed by a certain developer. Through a data analysis, they reported that a developer who has contributed to a certain module tends not to induce a fault, i.e., he/she would be a specialist of the maintenance of the module; a module which is contributed by more developers may be fault-prone. While we also have the latter perspective of their work, we analyzed the balance of contributions as well in this paper.

V. CONCLUSION

We focused on the cumulative contributions of developers to the development and maintenance of a source file, and proposed a novel metric, contribution entropy, to evaluate the balance of contributions to the file.

We performed a data analysis for 15,664 source files from 10 popular OSS projects. As the results, the following trends were founds: (1) a source file maintained by two or more developers tends to be more fault-prone than a file maintained

by a certain developer only. The fault-proneness of the former-type source file is about two times higher than that of the latter-type one; (2) when a source file has been maintained by two or more developers and the developers' contributions are more imbalanced, the source file is more fault-prone. From the results, the fault-proneness of a source file seems to be transiently increased when the file type changes from single-developer to multi-developer, i.e., when another programmer started to change a part of code. A successful OSS quality management would be prompted by focusing on such changes as well.

In order to understand a more detailed impact of another programmer's participation, we plan to check not only fault-fix commits but also fault-inducing commits, and analyze who made the corresponding code changes as our significant future work. Moreover, to predict the code quality stability, we will examine the impact of number of contributing developers on the quality, and investigate changes in the contribution entropy over time. Other our future work includes: (1) a further analysis using OSS projects whose development languages are other than Java to examine the generality of our findings, and (2) an application of our results to the just-in-time quality assurance studies.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI #16K00099 and #18K11246. The authors would like to thank the anonymous reviewers for their helpful comments to an earlier version of this paper.

REFERENCES

[1] Black Duck Software, "The 2017 open source 360° survey," https://www.blackducksoftware.com/open-source-360deg-survey, June 2017.

[2] A. E. Hassan, "The road ahead for mining software repositories," in *Proc. Frontiers of Softw. Maintenance*, Sept. 2008, pp. 48–57.

[3] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "Bugcache for inspections : Hit or miss?" in *Proc. 19th ACM SIGSOFT Symp. & 13th European Conf. Foundations Softw. Eng.*, Sept. 2011, pp. 322–331.

[4] C. Lewis and R. Ou, "Bug prediction at google," http://google-engtools.blogspot.jp/2011/12/bug-prediction-at-google.html, Dec. 2011.

[5] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, June 2005.

[6] N. Ajienka and A. Capiluppi, "Understanding the interplay between the logical and structural coupling of software classes," *J. Syst. & Softw.*, vol. 134, pp. 120 – 137, Dec. 2017.

[7] A. T. Misirli, E. Shihab, and Y. Kamei, "Studying high impact fix-inducing changes," *Empir. Softw. Eng.*, vol. 21, no. 2, pp. 605–641, June 2016.

[8] S. Suzuki, H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "An application of the pagerank algorithm to commit evaluation on git repository," in *Proc. 43rd Euromicro Conf. Softw. Eng. & Advanced Applications*, Aug. 2017, pp. 380–383.

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

[10] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Tran. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, June 2013.

[11] Q. C. Taylor, J. E. Stevenson, D. P. Delorey, and C. D. Knutson, "Author entropy: A metric for characterization of software authorship patterns," in *Proc. 3rd Int'l Workshop Public Data about Softw. Dev.*, pp. 42–47, Sept. 2008.

[12] 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 *Proc. 19th ACM SIGSOFT Symp. & 13th European Conf. Foundations of Softw. Eng.*, Sept. 2011, pp. 4–14.

[13] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *Proc. 35th Int'l Conf. Softw. Eng.*, May 2013, pp. 452–461.

[14] K. Yamashita, S. McIntosh, Y. Kamei, and N. Ubayashi, "Magnet or sticky? an oss project-by-project typology," in *Proc. 11th Working Conf. Mining Softw. Repositories*, May 2014, pp. 344–347.

[15] S. Onoue, H. Hata, and K. Matsumoto, "Software population pyramids: The current and the future of OSS development communities," in *Proc. 8th Int'l Symp. Empir. Softw. Eng. & Measurement*, Sept. 2014, pp. 34:1–34:4.

[16] C. E. Shannon, "A mathematical theory of communication," *The Bell System Tech. J.*, vol. 27, no. 3, pp. 379–423, July 1948.

[17] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005.

[18] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proc. Int'l Workshop Mining Softw. Repositories*, May 2006, pp. 137–143.

[19] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, Aug. 2000.

[20] E. N. Adams, "Optimizing preventive service of software products," *IBM J. Research & Development*, vol. 28, no. 1, pp. 2–14, Jan. 1984.

[21] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *Proc. 1st Int'l Symp. Empir. Softw. Eng. & Measurement*, Sept. 2007, pp. 364–373.

[22] H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "A survival analysis of source files modified by new developers," in *Product-Focused Software Process Improvement*, M. Felderer et al. Eds. Cham: Springer, Dec. 2017, pp. 80–88.

[23] S. Matsumoto, Y. Kamei, A. Monden, K. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proc. 6th Int'l Conf. Predictive Models in Softw. Eng.*, Sept. 2010, pp. 18:1–18:9.