WHOSEFAULT: Automatic Developer-to-Fault Assignment through Fault Localization

Francisco Servant
Department of Informatics
University of California, Irvine
Irvine, CA, U.S.A.
fservant@ics.uci.edu

James A. Jones
Department of Informatics
University of California, Irvine
Irvine, CA, U.S.A.
jajones@ics.uci.edu

***Abstract*—This paper describes a new technique, which automatically selects the most appropriate developers for fixing the fault represented by a failing test case, and provides a diagnosis of where to look for the fault. This technique works by incorporating three key components: (1) fault localization to inform locations whose execution correlate with failure, (2) history mining to inform which developers edited each line of code and when, and (3) expertise assignment to map locations to developers. To our knowledge, the technique is the first to assign developers to execution failures, without the need for textual bug reports. We implement this technique in our tool, WHOSEFAULT, and describe an experiment where we utilize a large, open-source project to determine the frequency in which our tool suggests an assignment to the actual developer who fixed the fault. Our results show that 81% of the time, WHOSEFAULT produced the same developer that actually fixed the fault within the top three suggestions. We also show that our technique improved by a difference between 4% and 40% the results of a baseline technique. Finally, we explore the influence of each of the three components of our technique over its results, and compare our expertise algorithm against an existing expertise assessment technique and find that our algorithm provides greater accuracy, by up to 37%.**

***Keywords*-developer assignment; fault localization; mining software repositories; expertise assignment**

I. INTRODUCTION

Today, when a test suite is run and failures are found, a project manager (or someone equally familiar with the software) attempts to examine the symptoms of the failures and assign them to developers to find and fix the faults causing them. Each developer assignment is made utilizing (1) the symptoms of failure, (2) mental inference by the assigner of which functionality was likely to have been the root cause of that symptom, and (3) an experiential knowledge of which developers are responsible for which features. Similarly, in the process of bug-triage, faults are represented by bug reports in a bug-tracking system. In this domain, we can also see evidence of the abundance of errors in the manual assignment of developers to faults. Often a bug report is assigned, and then re-assigned, sometimes repeatedly, until it finds the proper developer that has the necessary expertise to understand, find, and fix the problem [15], [16].

We present in this paper a technique that automatically chooses the developers to fix failures, represented by erroneous executions. Utilizing such a technique, we anticipate a better assignment, sooner, thus saving the project manager's time in the assignment. We also anticipate reducing the amount of reassignments necessary, saving developer time.

Some existing techniques for developer assignment focus on finding the right developer for resolving a bug report (e.g., [3], [16], [32]). However, Bettenburg et al. [6] performed a study of 466 developers and found that bug reports are often incomplete and poorly written. Although in many cases, bug reports are adequate sources for performing developer assignments, additional techniques are warranted when they do not exist or are inadequate. Moreover, even if bug reports exist for a fault, the mapping from existing failures to bug reports may not have been identified.

Irrespective of the absence or presence of quality bug reports, testing failures or erroneous executions often provide the first evidence of program faults that need to be fixed. Testing failures can exist for extended intervals, spanning many revisions of the program, and may or may not have a related bug report. Such failures can and are often directly assigned to developers for them to fix. However, simple solutions such as assigning the failure-fixing tasks to developers who most recently committed changes are often not appropriate. As evidence, researchers have developed techniques which apply complex algorithms to finding the right developer for answering questions about source code artifacts (e.g., [13], [18], [27]).

In this paper, we present WHOSEFAULT, a technique which automatically chooses expert developers to fix the fault represented by a failing test case, and provides them with a diagnosis of the location of the fault causing the failure. Such a technique provides a series of improvements over existing techniques that assign developers to bug reports or source code artifacts: (1) In the event of a test-case failure, the test case can be directly assigned to the expert developer, without the need to write a detailed and unambiguous bug report describing the failure; (2) it provides a diagnosis for where faults may reside in the code, which brings an

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

ICSE 2012, Zurich, Switzerland

additional saving of developer time, due to more directed fault-finding tasks; (3) it assesses developer expertise not only in the faulty code, but also in other areas of the source code which interact with it, in order capture knowledge about the context of the fault.

Our approach leverages source-code history that can be found in revision-control systems and fault-localization techniques that point to locations in the source code that are likely faulty. From the source-code history, we capture every change to each line of code, the type of that change, the time of the change, and who made the change. Our technique couples this history information with the diagnosis information about the location of faults in the source code from automated fault-localization techniques. Our expertise assessment algorithm combines these two pieces of information to provide a ranked list of developers in terms of expertise in these locations.

Suggesting a list of ranked developers instead of a single expert developer is a common practice in expertise finding techniques and it has a series of advantages. The most obvious one is that if the person in the first position of the rank is unavailable, we can assign the next person in the list for the resolution of the fault. Additionally, we envision a benefit of automatically suggesting collaborators or mentors for fixing faults. Examples of such situations include faults that involve cross-component logic or the introduction of a new developer who will need guidance from an existing and knowledgeable superior.

The main contributions of this paper are:

- A novel approach for developer-to-test-case assignment that uses the results of fault-localization techniques and the source-code history of a software system. This approach is applicable for use with any fault-localization technique and any source-code history-mining technique. To our knowledge, our technique is the first to assign developers to failing test cases.
- Experimental designs that enable the evaluation of future developer-to-fault assignment techniques and their constituent components. The first experiment evaluates the effectiveness of the automated technique to predict assignments made by real developers. The second experiment compares the effectiveness of the automated technique to a simple technique that only considers the distribution of commits over the source code. The last experiment gauges the influence of the separate components that comprise the technique under evaluation. These experiments evaluate the effectiveness of the automated technique to predict assignments made by real developers.
- An implementation of our approach, WHOSEFAULT, and its evaluation, which (1) demonstrates that it can effectively recreate historical assignment choices made by real developers; (2) shows that WHOSEFAULT improves the results of a simple technique which only considers the distribution of commits over the source code; and (3) reveals the influence of the three component techniques on our experimental results. In this evaluation, we show that our expertise algorithm improves upon an existing expertise assessment technique.

II. RELATED WORK

To provide the necessary background to motivate and explain our technique, we first overview three areas of related work: source-history mining (Section II-A), expertise-finding techniques (Section II-B), and automated fault-localization techniques (Section II-C).

A. Source-History Mining

In order to track the history of individual lines of code, Zimmerman et al. [35] proposed *annotation graphs*. Annotation graphs are multipartite graphs in which each "part" is represented as a column of nodes which represent lines of code. Edges are drawn between line nodes to represent their evolution. However, regions of contiguously changed lines are only mapped at that region-level. These regions of difference [26] introduce inaccuracies and imprecision. Several researchers (e.g., [9], [29], [31], [33], [34]) have investigated and proposed techniques that provide a finer-grained mapping of lines of code between consecutive revisions. We use such techniques in this work to provide a precise history of the evolution of each line.

B. Expertise-Finding Techniques

Existing work in the area of automatically determining developer expertise generally falls into two categories: (1) those that leverage the natural-language bug reports in a bug-tracking system to assign a developer, and (2) those that can identify the most knowledgeable developer given a location in the source code.

1) Bug-Report Expertise: The existing work that assigns a developer to a bug report (e.g., [3], [16]) utilizes the natural-language terms from the bug reports. The history of which developers were assigned to past bug reports that also used the same terms as in a current bug-report description is used to guide the selection. Such approaches do not use any information about the source code or test-case behavior other than the description of the input and symptoms described inside the bug report.

A number of researchers have investigated this area of using natural-language analyses (e.g., [2]–[4], [7], [8], [10], [16], [24]). Regarding effectiveness, Anvik et al. [3] achieved a precision between 57% and 64% when suggesting up to three developers for a bug report, and Jeong et al. [16] improved these results with additional historical bug reassignment information to achieve precision of up to 68% when suggesting three developers. Other similar but divergent techniques include: Matter et al. [28], who assign developers to bug reports based on the text of the bug reports

and developer-supplied expertise profiles; Kagdi et al [19], who link change requests to a source-code location using natural-language similarities between the change requests and source code, and then find expertise based on those locations; and Baysal et al. [5], who combines several factors such as natural-language matching and current developer workload to inform their recommendations.

These techniques also rely upon past and current bug reports to exist, to be well written, and to contain enough description to perform a sufficient matching. Moreover, even with well-written and descriptive bug reports, these techniques are sensitive to word and phrasing choices made by the various bug-report contributors. One bug-report contributor may refer to terms such as "GUI," while others may refer to "user interface," "presentation," and "window." Additionally, these approaches are incapable of providing suggestions of where in the code the fault may reside.

2) Source-Code Expertise: Another area of existing work automatically selects a developer who has expertise in a specific area of the code. In these techniques, a person chooses a location in the source code, and the technique is able to assess the developer who has the most expertise for that location. This work mines the history of the source code, capturing all changes to the system and the developers who make those changes.

A number of researchers have investigated this area of mapping "expert" developers to components of source code (e.g., [12]–[14], [18], [25], [27], [30]). Each such research endeavor has examined different factors in mapping developers to components. For example, McDonald and Ackerman [25] suggest that the developer who most recently changed a file is the expert for the entire file, and Mockus and Herbsleb [27] suggest that the expert is the developer who made most changes to a file. Another example includes Fritz et al. [13] who additionally consider factors such as frequency of reading the code.

If such techniques were to be applied to finding the most appropriate developer to fix a fault, they would require that a person can determine which parts of the program are faulty, which is a task that, itself, requires specific expertise. Moreover, the granularity of such approaches is typically at the file, class, or method level — tracking developer expertise by method or by file instead of tracking it for a set of lines of code, which could be located in different parts of the system. Also, these approaches can only point to the developers that have expertise in one particular area of code at a time — that is, they do not take a number of locations that may be related in some way and determine the developer who has the best expertise across this group. For example, for a single location, developer A may have the most expertise, and for another location, developer B may have the most expertise. However, if we are seeking the developer who has the most expertise in both of those locations, developer C may be the best choice.

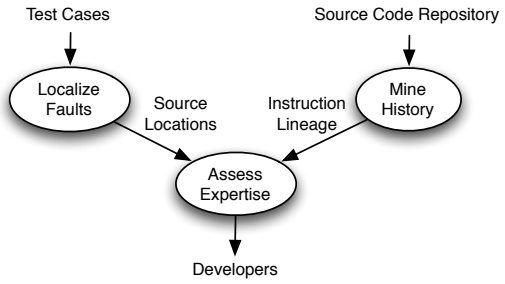


Figure 1. Process diagram of our technique.

C. Automatic Fault Localization

To provide an approach that has benefits of each of these classes of expertise assessment, while ameliorating their limitations, we utilize automated fault-localization techniques. Specifically, in this presentation, we will leverage statistical, coverage-based techniques. However, any fault-localization techniques can be utilized as long as it locates areas in the code that are determined to be likely to cause failure.

To inform the locations in the program where to query developer expertise, we will utilize a statistical, coverage-based fault-localization technique. A number of such techniques have been proposed by researchers (e.g., [17], [22], [23]). These techniques utilize information gathered about the internal behavior of the software. For example, one such technique, TARANTULA, analyzed the correlation between instruction execution and failing test cases. The intuition is that instructions that are executed primarily by failing test cases are more suspicious of being the fault causing those failures than instructions that are primarily executed by passing test cases. TARANTULA assigns a *suspiciousness* score to each instruction in the program according to this correlation.

Such a statistical, coverage-based fault-localization technique is used to identify areas of high suspiciousness that we can query for developer expertise. In the next section, we describe how the results from fault-localization techniques can be utilized to identify expertise in finding developers best suited for the debugging task.

III. APPROACH

In order to assign a developer to a failing test case, we follow three main steps to produce a list of the most suited developers to fix the fault represented by it. First, we utilize a fault-localization technique to determine the likely locations of the fault. In parallel, we mine the source-code repository to leverage its history — determining for every individual instruction, its lineage and all the developers that were instrumental in its development. Utilizing the results of the fault-localization technique, we weigh both the suspiciousness of each instruction and the developers who have knowledge about them to produce a list of

candidate developers, with a measure of expertise. This list of developers is then suggested as the best candidates to fix the fault represented by the test case. Figure 1 shows these steps. The following sections describe them in more detail.

A. Localize Faults

The first step of our approach consists of utilizing a fault-localization technique to determine the areas of the source code that will most likely need to be modified in order to fix the fault. Any fault-localization technique can be utilized to assess which locations are suspicious of causing failures. We can utilize approaches that identify a subset of the program (such as a slice) or approaches that assess a degree of suspiciousness for each component. In our implementation (and evaluation in Section IV), we use TARANTULA [17].

As a consequence of using a statistical, coverage-based fault-localization technique like TARANTULA, our approach takes as its input the coverage of a test case (or test cases) that tests for a fault. The coverage of a test case contains all the locations in the source code that were executed by the test case. We define a location in the source code as an executable line of code.

Once we have collected the coverage for all test cases, we can use the fault-localization technique. We use as input the coverage of the failing test case together with the coverage of all the passing test cases in the test suite. As a result, the fault-localization technique will return a suspiciousness value for all the locations in the source code. This suspiciousness value represents how likely a location is to contain the fault — these values range from 0 for the lowest suspiciousness to 1 for the highest suspiciousness. The set of all locations in the source code with their suspiciousness value is one of the inputs for the last step of our approach.

B. Mine History

As well as the localization of the fault, our approach uses the history of the source code in order to find the most suited developers to fix a fault. Therefore, we also mine the entire history of a software project to determine the lineage of each individual instruction in the code.

Previous expertise-finding projects (e.g. [12], [13], [25], [27]) suggest experts for software artifacts at the method-level or higher levels. We, however, choose a fine-grained level for mining, since it directly maps to the results of our fault-localization technique, and as such is likely to give a more precise expertise assessment. Therefore, we mine changes in the source code at the individual-instruction level by building a history graph [31] for the software project. A history graph is a variation of an annotation graph [35]. An annotation graph is a multipartite graph in which each part represents a revision of a file in the source code, and each node in a part represents a line of code in that revision. In a history graph, each node is labeled with the action that produced it and is linked to only one other node in the previous revision.

In order to build the history graph, we first collect the meta-data (including authorship and commit time) for all of the revisions that the repository contains for each file belonging to the software project. We then compare each revision of each file with its previous revision in time by using the *annotate* and *diff* features of the source code repository. We use the results of this comparison to map a line in each specific revision of a file with its corresponding line in the previous revision of the file.

However, the output of *diff* is not a perfect line to line correlation. Since *diff* uses a textual difference algorithm, it returns regions of difference called hunks [26]. Figure 2 shows a modification hunk in which line 2 was changed to become the region that now spans lines 2 to 4.

```
2c2,4
<    i = 0;
---
>    j = 2;
>    i = 5;
>    x = i + j;
```

Figure 2. Modification hunk.

By observing Figure 2, we realize that line 2 in the old revision was actually modified to become line 3 in the new revision, and that lines 2 and 4 were added. In order to provide this finer-grain mapping for lines between two revisions, we use the *line mapping* technique proposed by Williams and Spacco [34] for all modification hunks. We first obtain each possible pair of one line in the old revision and one line in the new revision. Then, we assign a weight to each of the pairs equal to their Levenshtein distance [21]. After that, we apply the Kuhn-Munkres algorithm [20] to obtain the set of pairs with optimal global similarity score. Finally, we use a threshold over the Levenshtein distance of each of the selected pairs. Previous work (e.g., [9], [34]) has determined the adequacy of a standard threshold of 0.4 for this purpose, which we also use. If the distance of an assigned pair is lower than the threshold, we classify them as a changed line. If their distance is higher than than the threshold, we classify the old line as being deleted and the new one as being added. Figure 3 illustrates an example of a small history graph.

C. Assess Expertise

In the third step of our approach, we combine the suspiciousness information described in Section III-A with the history of the source code described in Section III-B in order to infer the most experienced developers in the suspicious locations of the code.

In order to determine the expertise of a developer in a particular line of code, we consider two aspects that we

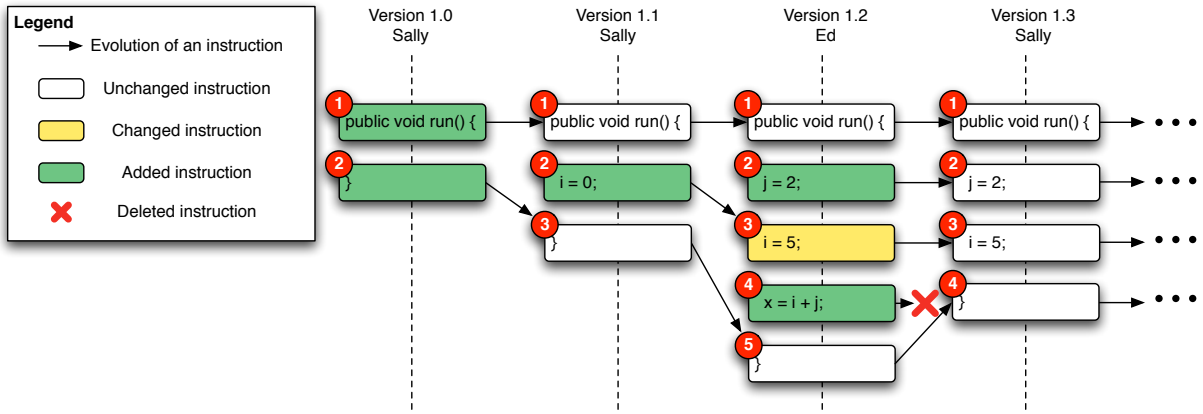


Figure 3. Example of a small history graph.

estimate to be the most important: amount of changes and recency of changes for that developer in that line of code. Our intuition is that, as a developer makes more changes to a line of code, he or she becomes more familiar with it. This intuition has already been expressed and verified by other authors [14]. Additionally, we consider that developers will be most familiar with their most recent changes, and less familiar with less recent changes. This intuition has also been previously identified by other authors [13], [24].

We then define the recency of a change c as such:

$$recency_c = 1 - \frac{\text{today} - \text{date}_c}{\text{today} - \text{beginning date}} = \frac{\text{date}_c - \text{beginning date}}{\text{today} - \text{beginning date}} \quad (1)$$

In Formula 1, the recency of the change c is the amount of time that has passed since the beginning of the software project until the change was committed, compared to the total age of the software project. We calculate the recency metric as a distance from the beginning of the project (as opposed to a distance from now) so that an increase in time from the change to now is reflected as a decrease in the recency metric. The recency value ranges from 0 to 1. Figure 4 depicts the recency calculation.

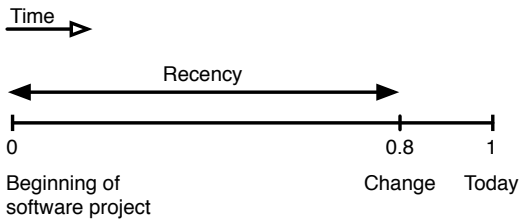


Figure 4. Depiction of the calculation of the recency metric. The value of the recency metric increases as the difference of the change date and now decreases.

Once recency has been defined, Formula 2 represents the expertise of a developer for fixing a fault. We define the expertise value of a developer d for fixing a fault as the sum of the recency of all changes $c_{d,l}$ made by that developer d to any suspicious line l in the source code, multiplied by the suspiciousness value of that line. In this formula, we reward the suspiciousness of lines, recency of changes and amount of changes. As any of these variables increases, the total expertise value of a developer also increases.

$$expertise_d = \sum_{c_{d,l}=1}^{M} \sum_{l=1}^{N} recency_{c_{d,l}} \times suspiciousness_l \quad (2)$$

We implement our expertise algorithm by first applying history slicing [31] to the set of suspicious lines returned by the fault-localization step. Thus, we walk the history graph that we built during the mining of the source-code repository, obtaining all the equivalent lines to the suspicious lines in all the past versions of the software project in which they were modified. Finally, we apply Formula 2 to this output in order to obtain an expertise score for each developer that made changes to the suspicious lines.

This expertise metric produces values for each developer for a given program and fault-localization diagnosis. The value of this metric for a specific developer is only meaningful relative to the values of the metric for the other developers — we only use it as a means to compare the relative expertise of all developers for a specific bug. Once Formula 2 has been applied for all developers, they are sorted from highest to lowest expertise value. The developer in position 1 of the list is the most appropriate to be in charge of fixing the fault, according to our technique. The next most appropriate developer is in position 2, and so on. Developers that did not make any changes to the suspicious lines of code (and therefore have no expertise value) are not included in the list.

IV. Evaluation

To evaluate our technique for automatic developer assignment to test cases, we implemented it in our tool WhoseFault and conducted two experiments over a real-world project. With these experiments, our goal is to answer the following research questions:

- RQ1: How often does the automatic assignment of developers to test cases find the right developer and in which position?
- RQ2: How much does our automatic technique improve over a naïve approach?
- RQ3: How does each component of our automatic technique affect its results?

A. Experimental Subject

In order to perform our evaluation, we need access to a software project's test cases and source-code repository. First, we need a test suite to execute for any revision of the software and identify the test cases that fail. Second, we need a source-code repository to mine the entire history of the project and obtain the history of every line of source code as described in Section III-B.

We selected the AspectJ open-source project [1] as the subject for our experiments, since it provides access to both of the previously mentioned resources. AspectJ is an aspect-oriented extension to the Java programming language. We mined the source code history of AspectJ and extracted 10,454 revisions, which had been created over 8.5 years of development. We also extracted AspectJ's test suite, which contained an average of 1,585 test cases per revision.

To assist in the selection, compilation and execution of test cases, we utilized the iBugs project [11]. The iBugs project processes the bug-tracking system and source-code repository of a software project in order to identify the source-code changes that fixed a bug and the test cases that were committed with those changes. Its original purpose is to determine the changes that fixed bugs in real software projects in order to provide an oracle to benchmark fault-localization techniques.

B. Experimental Setup

In order to perform our experiments, we need a set of failing test cases and an oracle to tell us who is the developer with the most expertise to fix the fault represented by each of them. We select this set of failing test cases by exploring the source-code repository of AspectJ and identifying committed changes that fixed the fault represented by a test case. We refer to these changes as the *fix* for that failing test case. We use the developer who committed the *fix* to the source code repository as the oracle for who is the developer with most expertise to fix the fault represented by a test case. We refer to this developer as the *expert* for the test case.

In order to identify committed changes that fix a failing test case, we used the iBugs project. iBugs reports 350 bugs for AspectJ and the changes committed to its source code repository that fixed each of these bugs. iBugs also provides the revision of the source code of AspectJ just before (*pre-fix*) and just after (*post-fix*) each of the bug-fixing changes was committed to the source code repository. These bug-fixing changes often include a test case to test for the bug in the future. Therefore, we can expect some failing test cases in a pre-fix revision to become passing test cases in the post-fix revision.

We selected our set of failing test cases from the 350 pre-fix revisions of AspectJ provided by iBugs. We copy the test suite from each of the post-fix revisions to their corresponding pre-fix revision. Then, we execute the test suite in both the pre-fix and post-fix revisions. Finally, we select all test cases that fail in the pre-fix revision and pass in the post-fix revision as candidate test cases for our experiments. The expert for each of these test cases is the developer who committed the bug-fixing changes for their corresponding pre-fix revision. Out of the 350 pre-fix revisions reported by iBugs, 245 contained test cases that fail in the pre-fix revision and pass in the post-fix revision.

In order to be able to use a fault-localization technique, we apply a source-code instrumenter (Cobertura) over each compiled pre-fix revision. This is so that we can capture the coverage of each test case when running the test suite. We performed the instrumentation over the 245 pre-fix revisions that contained test cases that fail in the pre-fix revision and pass in the post-fix revision. In 34 of these pre-fix revisions, the instrumentation process failed for the files involved in the fix for the failing test cases. This happened because the fixes were contained in a library that had been compiled without debugging information, and the source code was not available. This situation is similar to using test cases that test third-party libraries, for which neither debugging information or source code is available. Our approach is not applicable to such situations, because if the faulty source code were not available to the parties performing the instrumentation, they could neither instrument nor fix the fault. As a consequence, we discarded these 34 revisions.

Finally, we executed the test suite on the 211 remaining pre-fix revisions for which we could successfully instrument the files involved in the fix. We also executed the test suite on their corresponding post-fix revision. Some failing test cases did not produce coverage information, because they crashed as soon as their execution started. We cannot provide a developer recommendation for these test cases, since they practically do not interact with the source code of the application.

Thus, we selected as candidate test cases for our experiments a total of 889 test cases, which: (1) failed in one of the 211 pre-fix revisions, (2) passed in its corresponding post-fix revision, and (3) produced some coverage information.

For each individual candidate failing test cases, we applied a fault-localization technique to its coverage plus that of all

the passing test cases. As a result of this step, we obtained, for each individual failing test case, a suspiciousness value for each line of code. In parallel, we mined the source-code history of the project as explained in Section III-B and obtained the history of each line of code. Afterward, we ran our expertise algorithm and stored the expertise value for each candidate developer for each test case.

Then, we ranked the list of developers in terms of expertise values, from highest to lowest, which was our prediction for each failing test case. If the actual developer who fixed the failing test case (the author that committed the changes that fixed it) was within the top-ranked developers, we considered our approach successful.

C. Experimental Variables

The primary object of our experiments is to assess the degree to which our technique automatically selects the developer that actually fixed a failing test case, according to historical record. To this end, we assess the position in the ranked list of developers for the actual developer that wrote and committed the changes that fixed the failing test case. A position of "1" in our ranked list indicates that our technique precisely chose the correct developer according to the commit log of the version-control system. We also evaluate the further positions in the ranked list: a position of "2" in our ranked list indicates that we chose the correct developer within the top two positions; and, so on. We note here that our assessment of the "correct" developer is based on the actual developer that wrote and committed the changes that fixed the failing test case, which is an undoubtedly subjective and imperfect measure of true expertise. The developer with the greatest expertise in a particular failing test case could have been busy working on other tasks, on vacation, or otherwise unavailable. In addition, other developers could have volunteered to fix failing test cases to learn more about the project. Nonetheless, we believe using the historical artifact as our oracle for the correct assignment is a reasonable one as it was chosen by the people most knowledgeable about the project — the actual developers.

D. Experiment 1

For our first experiment, we mined the source-code history at the line level, using the Levenshtein distance for our line-mapping technique. Then, we used the TARANTULA fault-localization technique and our expertise algorithm for the 889 test cases that we identified as candidates. After running the experiment, we obtain the results in Figure 5.

These results show that WHOSEFAULT identifies the expert in the first position of its ranked list for 35.21% of the failing test cases that we studied. This result increases up to 50.17% when we consider the first two positions of the suggestions and up to 81.44% when we consider the first three. We considered as candidate developers all developers who had committed changes to the source-code repository

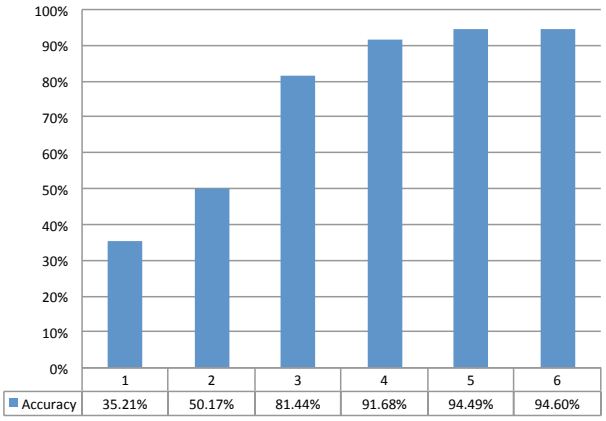


Figure 5. Effectiveness of our developer assignment technique.

before the date of the pre-fix revision of the test case considered. For each test case considered, WHOSEFAULT considered an average of 8.2 candidate developers, with a standard deviation of 2.3.

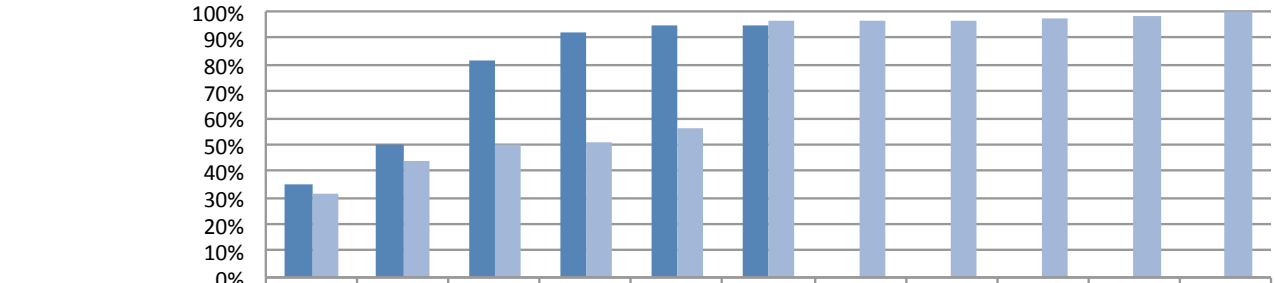
To put our results into context, consider that random selection over all active developers would yield results of only about 12%, 24%, and 36% (i.e., the probability of randomly choosing the one correct developer out of an average of 8.2 developers is $1/8.2 \approx 12\%$, and so on). Thus, the results of WHOSEFAULT provide a considerable improvement over a random selection.

Note that, for 5.40% of the considered test cases, the expert was not found inside the ranked list of developers. In these cases, the person who fixed the failing test case had not committed any changes to any of the suspicious lines in the past. A possible explanation could be that our fault-localization technique was not successful at finding the context of the fault. Another possibility is that the person who fixed the failing test cases was actually not the most appropriate one, given that our oracle is not perfect. We describe this and other threats to validity in Section V.

> RQ1: Our automatic developer assignment technique finds the expert within the top 3 candidate developers for 81.44% of the considered test cases, which indicates that our automatic recommendations can be valuable and accurate.

E. Experiment 2

In our second experiment, we compare the results of our automatic technique to a baseline of the distribution of expertise throughout the source code. By this comparison, we check whether the success of our technique is due to a skewed distribution of expertise. If a small number of developers commit most changes to the source code history, then any technique that depends on the amount of commits made by a developer can be successful at finding the expert. Such a scenario would imply that a technique based on simply ranking all developers in terms of the amount of commits that they made to the source-code repository would obtain a high accuracy.



	1	2	3	4	5	6	7	8	9	10	11
WhoseFault	35.21%	50.17%	81.44%	91.68%	94.49%	94.60%					
Distribution of commits	31.16%	43.53%	49.61%	50.73%	56.47%	96.29%	96.40%	96.40%	97.64%	97.98%	100.00%

Figure 6. Effectiveness of our developer assignment technique, compared to the distribution of changes committed by developers.

We obtained the distribution of expertise by building a simple technique that retrieves all developers who committed changes to the source-code repository before the date of the pre-fix revision of the test case. Then, it sorts this list of developers in terms of the number of commits that each developer made, from high to low.

We executed this simple technique for the 889 candidate test cases. Figure 6 displays the accuracy obtained by this simple technique in comparison with the results of our automatic assignment-technique in WHOSEFAULT.

These results show that, when considering only the first or the first two suggested developers, our automatic technique provides more accurate results than the *distribution of commits* technique by finding the expert in 4.05% and 6.64% more test cases, respectively. Moreover, when considering the first three, four, and five developers, our automatic technique is more accurate at finding the expert for 31.83%, 40.95%, and 38.02% more test cases, respectively, than the *distribution of commits* technique.

Our results also show a sudden increment in the accuracy of the *distribution of commits* technique when the first six developers are considered. Upon investigation of this phenomenon, we discovered a significant number of failing test cases were fixed within a relatively short period of time, by a developer who was the sixth most active developer. As a consequence, the accuracy of the *distribution of commits* technique increases significantly when including the sixth position. However, our automatic technique recommended the correct developer at a higher position for these failing test cases, because its localization algorithm causes it to focus on a particular area of the code as opposed to the *distribution of commits* technique, which considers equally all parts of the entire program. Additionally, our automatic technique weighs developer expertise according to our expertise formula, assigning higher importance to more recent changes. This phenomenon demonstrates one strength of our technique: its resilience to such uneven expertise and fault distribution.

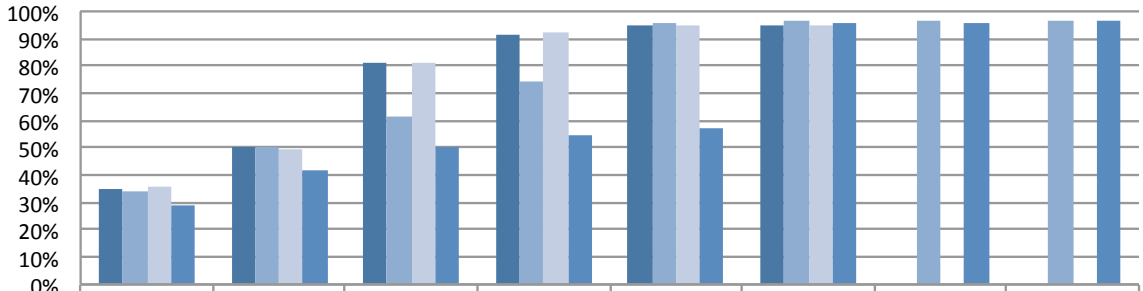
We should note that high accuracy obtained after considering more than four positions does not have a high impact, given that an average of eight developers were considered as candidates. Considering more than four positions would mean, on average, considering more than 50% of the candidate developers. As such, although we can see that the distribution of commits finds the expert for more test cases than our automatic technique when considering the first six suggested developers, we interpret these results to not have a high impact, because it would mean having to consider 75% of the candidate developers.

Finally, for some test cases (5.40% in our experiment), our automatic technique could not find the expert in any of the suggested positions, which means that it could not reach 100% accuracy, no matter how many positions were considered. In contrast, the *distribution of commits* technique always suggests all developers, which allows it to reach 100% accuracy, even if it means inspecting all positions in its ranked list of developers (our experiment considered a maximum of 11 candidate developers).

> RQ2: Our automatic developer assignment technique performs better than a naïve approach based on the distribution of commits among developers. It produces recommendations that are more accurate (by at least 4.05%) when considering the top two ranks, and remarkably more accurate when the recommendations include between three and five developers (up to 40.95%).

F. Experiment 3

In our third experiment, we evaluated the influence over our results of each of the individual steps that comprise our approach. For that goal, we individually substituted the techniques that we selected in each of the steps described in Section III for a simple technique. Then, we collected the results for each of these variations of our approach and checked how much they vary in comparison with the original configuration. The results of this experiment are displayed in Figure 7. The WHOSEFAULT label represents the results of our original configuration.



	1	2	3	4	5	6	7	8
WhoseFault	35.21%	50.17%	81.44%	91.68%	94.49%	94.60%		
Mining file level	33.75%	49.94%	61.75%	74.24%	95.50%	96.40%	96.51%	96.85%
Coverage	35.55%	49.49%	81.10%	92.24%	94.49%	94.60%		
Experience Atoms	29.36%	41.39%	50.17%	55.01%	57.03%	95.39%	95.73%	96.85%

Figure 7. Effectiveness of different variants of our developer assignment technique, when each step is substituted by a simple one.

1) Mining History at the File Level: First, we substituted our original technique of mining the history of the source-code repository at the line-of-code level for a simpler technique that retrieves changes in files, but does not store which lines were modified by each change. Figure 7 displays the results for this variation under the label *Mining file level*.

As a consequence of this modification, our expertise algorithm includes any author that committed changes to any of the files identified by the fault-localization technique as suspicious, regardless of whether they modified suspicious lines or any other lines of code. This is also represented in the results by the fact that *Mining file level* finds the expert for a higher number of test cases than WHOSEFAULT when considering all suggested developers (96.85% vs. 94.60%, respectively). However, this advantage has low impact, because it involves considering a high percentage of the candidate developers (as described in Section IV-E).

The results obtained for the variant *Mining file level*, although lower, are very similar to those obtained by our original configuration, WHOSEFAULT, for the first one and two positions. However, they are quite worse when we consider the first three and four positions (by 19.69% and 17.44%, respectively). Therefore, we conclude that mining the source code history at the line-of-code level is important to achieve the results of our approach.

2) Coverage Fault Localization: In our second variant, we substitute the TARANTULA fault localization technique for a simpler technique that only captures those lines that were executed by the failing test case and assigns the highest suspiciousness level to all such lines. Since all the lines executed by the failing test case have the same suspiciousness value, all of them acquire the same importance when they are processed by our expertise technique. The results for this variation are displayed in Figure 7 under the label *Coverage*.

In this case, the results for the variant *Coverage* are practically the same as for WHOSEFAULT (less than 1% difference in all positions). This result implies that the suspiciousness value of the TARANTULA fault-localization technique had little impact on the results of WHOSEFAULT, beyond the coverage of the failing test cases that inform it. As such, it seems that the passing test cases did not significantly contribute to the results. We speculate that unlike this evaluation in which only a *single* failing test case was used, the inclusion of multiple failures may be helpful to allow the suspiciousness metric to differentiate the failing coverage, and thus improve the results. However, further studies need to be performed to verify this hypothesis.

3) Experience Atoms: Finally, we substitute our expertise algorithm with the simpler technique used by Expertise Browser [27]. By this technique, the developer with the highest amount of *experience atoms* is the expert for a source-code entity. Experience atoms are measured by atomic changes that a developer commits to the source-code repository. We implemented this technique by obtaining all developers who had committed changes in the source-code repository to any of the suspicious files, and sorting them by the number of commits that each developer made, from high to low. We display the results of this variant in Figure 7 under the label *Experience Atoms*.

Our original configuration achieves much more accurate results than those achieved by this variant (up to 37.46% for the first 5 positions) in all positions, except for the case of considering the first six developers suggested. Because *Experience Atoms* is based on the distribution of commits over a set of files, it shows the same peculiarities described for the *distribution of commits* technique in Section IV-E.

Therefore, we can conclude that our original expertise algorithm played an important role in our results. The unique features of our original expertise algorithm that are

not included in the *Experience Atoms* approach are the consideration of only commits that modified suspicious lines and the measure of recency of each commit.

RQ3: The results obtained by our automatic developer-assignment technique were mostly influenced by our expertise algorithm, and also influenced by mining the history of the source code at the line-of-code level, but not significantly influenced by the suspiciousness values provided by the fault-localization technique.

V. Threats to Validity

Threats to internal validity arise when factors affect the dependent variables without the researchers' knowledge. It is possible that some implementation flaws could have affected the results. However, we are confident in the correctness of our results, given that we repeated our technique for 889 test cases, and the results were consistent among them. In addition, we exercised diligence in testing and manually checking our results at each step of the process.

Threats to external validity arise when the results of the experiment are unable to be generalized to other situations. In this experiment, we evaluated the technique on only one program, and thus we are unable to definitively state that our findings will hold for programs in general. However, we conducted our experiment on a real project that is used world-wide. The test cases on which we evaluated represent actual faults with actual failures. The AspectJ project is over eight-years old, of which the test cases that we used spanned five years. The results may not generalize to other programs, development teams and their practices, and test cases. However, we have a strong evidence that our approach has promise in practice.

Threats to construct validity arise when the metrics used for evaluation do not accurately capture the concepts that they are meant to evaluate. In our case, we measure the location of the correct developer in a ranked list. However, "correct" in our experiment is defined as the developer who actually fixed the fault represented by the test case — this could have been the developer performing the fix for any number of reasons. For example, the second-best developer could have been chosen because the most experienced developer was unavailable; an inexperienced developer was chosen because she wanted to learn about that function in the code; or, simply the wrong choice was made by the person doing the assignment. Nonetheless, the actual person chosen by the people who know the most about the codebase (the actual development team) is likely to be among the most experienced, most of the time.

VI. Conclusions

In this paper, we presented a novel technique that automatically assigns a developer to a fault, represented by a failing test case. The technique is more automated than previous techniques that relied upon a person to first determine the location in the codebase that needed an expert. It also works at a finer level of granularity — at the individual line level. The technique is more automated than previous techniques that relied upon well-written and consistent bug reports to describe the witnessed failures to find an expert. In addition, we provide a diagnosis of the locations in the code for the expert developer to begin her search for the fault.

We implemented our approach in a tool called WhoseFault and demonstrated its effectiveness on a well-known, real-world application that has been in active development for over eight years. Our results show that WhoseFault chooses the correct developer in the first suggested position for 35% of the test cases studied, for 50% when considering the top two positions, and for 81% when considering the top three positions.

We also evaluated WhoseFault against a simple distribution of the commits performed by each developer to check whether our results are caused by a skewed distribution of commits over the source code. WhoseFault improved by a difference between 4% and 40% the results obtained by a simple technique that sorts the developers in terms of number of commits performed in the source-code repository.

Finally, we explored the influence of each of the components of our approach over its results. We found that our results were mostly influenced by our expertise algorithm and moderately influenced by the algorithm that we chose to mine the history of the source code. We also found that the fault-localization technique that we utilized did not have a high influence in our results. As a consequence, we found that, by using a simpler fault-localization technique, which utilizes only the coverage of the failing test case, we could retain our effectiveness and require less input and potentially less runtime overhead. As part of this experiment, we also compared our expertise algorithm against an existing expertise assessment technique and found that our algorithm provided greater accuracy, by up to 37%.

We envision WhoseFault being used with automated regression-testing or continuous-integration environments, for use as: (1) a recommendation for which developer to assign, (2) a guide for the eventually assigned developer to find other experts to consult and collaborate, and (3) a diagnosis for areas to explore during the debugging process.

In the future, we will expand our studies to additional subjects and test cases. We also intend to experiment with applying different techniques in each of the components of our approach — using different line-mapping techniques, fault-localization techniques, and expertise techniques — and using multiple failures to inform the approach.

VII. Acknowledgements

This material is based upon work supported by the National Science Foundation under award CCF-1116943, and by a Google Research Award.

REFERENCES

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

[2] J. Anvik. Automating bug report assignment. In *International Conference on Software engineering*, pages 937–940, 2006.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.

[4] J. Anvik and G. C. Murphy. Determining implementation expertise from bug reports. In *International Workshop on Mining Software Repositories*, 2007.

[5] O. Baysal, M. Godfrey, and R. Cohen. A bug you like: A framework for automated assignment of bugs. In *International Conference on Program Comprehension*, pages 297 – 298, 2009.

[6] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Foundations of Software Engineering*, pages 308–318, 2008.

[7] G. Canfora and L. Cerulo. How software repositories can help in resolving a new change request. In *Workshop on Empirical Studies in Reverse Engineering*, 2005.

[8] G. Canfora and L. Cerulo. Supporting change request assignment in open source development. In *ACM Symposium on Applied computing*, pages 1767–1772, 2006.

[9] G. Canfora, L. Cerulo, and M. D. Penta. Identifying changed source code lines from version repositories. In *International Workshop on Mining Software Repositories*, pages 14–21, 2007.

[10] D. Cubranic. Automatic bug triage using text categorization. In *International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, 2004.

[11] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *International Conference on Automated Software Engineering*, pages 433–436, 2007.

[12] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Foundations of Software Engineering*, pages 341–350, 2007.

[13] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *International Conference on Software Engineering*, pages 385–394, 2010.

[14] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Workshop on Principles of Software Evolution*, 2005.

[15] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. "Not my bug!" and other reasons for software bug report reassignments. In *Computer Supported Cooperative Work*, 2011.

[16] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Foundations of Software Engineering*, pages 111–120, 2009.

[17] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering*, pages 467–477, 2002.

[18] H. Kagdi, M. Hammad, and J. Maletic. Who can help me with this source code change? In *International Conference on Software Maintenance*, pages 157 –166, 2008.

[19] H. Kagdi and D. Poshyvanyk. Who can help me with this change request? In *International Conference on Program Comprehension*, 2009.

[20] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2), 1955.

[21] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, pages 707–710, 1966.

[22] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Programming Language Design and Implementation*, PLDI '05, pages 15–26.

[23] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *Foundations of Software Engineering*, pages 286–295, 2005.

[24] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *International Working Conference on Mining Software Repositories*, pages 131–140, 2009.

[25] D. W. McDonald and M. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Computer Supported Cooperative Work*, pages 231–240, 2000.

[26] W. Miller and E. W. Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, 1985.

[27] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *International Conference on Software Engineering*, pages 503–512, 2002.

[28] M. M. Rahman, G. Ruhe, and T. Zimmermann. Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects. *Empirical Software Engineering and Measurement*, 0:439–442, 2009.

[29] S. P. Reiss. Tracking source locations. In *International Conference on Software Engineering*, pages 11–20, 2008.

[30] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *International Working Conference on Mining Software Repositories*, pages 121–124, 2008.

[31] F. Servant and J. A. Jones. History slicing. In *International Conference on Automated Software Engineering (ASE)*, pages 452–455, 2011.

[32] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *Foundations of Software Engineering*, 2011.

[33] C. Williams and J. Spacco. Szz revisited: verifying when changes induce fixes. In *Workshop on Defects in large software systems*, pages 32–36, 2008.

[34] C. C. Williams and J. W. Spacco. Branching and merging in the repository. In *International Working Conference on Mining Software Repositories*, pages 19–22, 2008.

[35] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, Jr. Mining version archives for co-changed lines. In *International Workshop on Mining Software Repositories*, pages 72–75, 2006.