Received: 2 November 2022 | Revised: 5 September 2023 | Accepted: 14 September 2023

DOI: 10.1002/smr.2622

SPECIAL ISSUE - EMPIRICAL PAPER

Journal of Software: Evolution and Process WILEY

Understanding the use of spectrum-based fault localization

Higor Amario de Souza[1,2] | Marcelo de Souza Lauretto[3] | Fabio Kon[2] | Marcos Lordello Chaim[3]

[1]Department of Computing, São Paulo State University, Bauru, Brazil

[2]Department of Computer Science, Institute of Mathematics and Statistics, University of São Paulo, São Paulo, Brazil

[3]School of Arts, Sciences, and Humanities, University of São Paulo, São Paulo, Brazil

Correspondence
Higor Amario de Souza, Department of Computing, São Paulo State University, Bauru, SP, Brazil.
Email: higor.amario@unesp.br

Funding information
São Paulo Research Foundation (FAPESP), Grant/Award Numbers: 13/24992-2, 14/23030-5, 14/50937-1

Summary

Developers spend significant time locating and fixing bugs, which is often performed manually. Although spectrum-based fault localization (SFL) techniques aim at helping developers to locate faults, they are not yet used in practice. Recent studies have investigated how developers use SFL, presenting different conclusions about their effectiveness and usefulness. We carried out a user study to further enhance the understanding of SFL. We assessed whether SFL improves the developers' performance and to what extent SFL leads developers to inspect faulty code excerpts. We also investigated the intention of the developers to use SFL and how they interact with SFL. Twenty-six participants performed debugging tasks using real programs, with and without using the Jaguar SFL tool. Using SFL, more developers located and fixed the bugs. SFL also led more developers to inspect the faulty code and locate the faulty method. However, they did not spend less time locating the faults. SFL was well-accepted by the participants, who showed intention to use it in their daily activities. Our results indicate that SFL is useful even when the fault is not ranked among the first positions, leading developers to reach faulty code regions and find the bugs.

KEYWORDS
automated debugging, fault comprehension, fault localization, spectrum-based, user study

1 | INTRODUCTION

Debugging is an inherent task of software development. Once a failure occurs, developers inspect the code to *locate* and *understand* the failure's cause and, then, to *fix* the fault.[1] Testing and debugging are among the most time-consuming activities in software development.[2] Although automated testing is widely adopted by the software industry, debugging is still performed manually in practice—developers often use symbolic debuggers and print statements to identify faults.

Several automated debugging techniques have been proposed in the last decades, aiming at reducing debugging costs.[3–5] From those, spectrum-based fault localization (SFL) techniques are promising due to their low execution costs. SFL techniques provide lists of program elements (e.g., lines*, branches, and methods) more likely to contain bugs. Recent SFL techniques have been proposed to better rank faulty elements.[6–9]

Despite the importance of improvements on SFL techniques to achieve better results, few studies have tried to understand whether and how developers will use such results. Parnin and Orso[10] proposed the first study to assess how developers use SFL. They showed that only more experienced developers had better performance using SFL. They also showed that participants did not follow the inspection order suggested by SFL lists. Gouveia et al[11] conducted a user study to assess their GZoltar tool, showing that the participants were more effective and efficient using the tool. In the study of Xie et al.,[12] SFL improved the participants' effectiveness only when faults were ranked among the most suspicious statements. However, SFL decreased the participants' efficiency. Conversely, Xia et al[13] observed in their study that developers were more effective

and efficient using SFL. In the study of Horváth et al.,[14] SFL did not help developers to locate more bugs. However, less time is spent on the debugging tasks. Thus, these user studies have presented divergent results, which indicates the need for more studies to further increase knowledge on how developers actually use SFL. Such knowledge can lead to new techniques that could be used in real settings. Moreover, replication studies play an important role to improve the practice of software engineering, helping to understand whether previous results hold and in which conditions.[15]

Since debugging is a complex and intellectual task, it is too simplistic to assume that SFL can replace the developer's role in locating bugs. SFL can be useful to provide hints and to guide developers closer to the faults. A recent survey by Kochhar et al[16] asked developers about their expectations of fault localization techniques. Regarding the level of suspicious information, participants deemed methods as the preferred level. Indeed, methods enclose the logic of a program's functionalities, which may be easier to comprehend than isolated statements. Also, in Kochhar et al,[16] most participants answered that they are willing to adopt an accurate and efficient fault localization technique.

In this paper, we replicate previous studies[10–14] to investigate how developers use SFL and to what extent SFL helps to guide them to inspect suspicious code excerpts. As in the previous work mentioned above (and described in Section 2.1), we assessed if SFL improves the developers' effectiveness and efficiency in debugging. We also present our findings regarding how developers interact with an SFL tool and their opinions about SFL. Different from these previous studies, we evaluate how close to faulty code elements an SFL tool can take its users. More specifically, we assess if there is a correlation between reaching a fault site and finding a bug. Then, we assess if developers can identify the faulty methods and reach the faulty lines using SFL. In this way, an SFL tool can be useful to locate bugs or, at least, to provide hints for developers to start understanding faulty behaviors. We also assess how the developers perceive the usefulness and ease of use of SFL in practice, which indicates their willingness to use SFL in the future.

This study was conducted with a convenience sample of 26 participants[†] with different experience levels in programming. They performed two debugging tasks, one using an SFL tool (the Jaguar Eclipse IDE[‡] plug-in[§17]) and the other one using the standard Eclipse resources.

For the SFL task, we provided lists with two different levels of code information: lines and methods. We assessed not only if a developer can locate and fix a fault but also if she/he locates the faulty method and inspects the faulty line. We applied the Technology Acceptance Model (TAM)[18] to evaluate the developers' intention to use SFL in the future and their evaluations of usefulness and ease of use of the SFL tool. We also performed an open card sort[19,20] analysis to evaluate the developers' comments and observations about the use SFL and analyzed their navigation behavior while using SFL.

Our results show that SFL leads more participants to locate bugs, with statistical significance and a medium effect size (ES), even when the faults are not ranked in the first positions of the SFL lists. Twelve participants (46%) found the bugs using the tool, and five participants (19%) found the bugs without using it. SFL did not improve the fault localization efficiency with statistical significance, although the results were slightly better using the Jaguar tool. Participants took on average 22% less time to find the bugs using SFL.

SFL also improved the developers' performance in (1) locating the faulty method and (2) inspecting the faulty line. The results are statistically significant in both cases. Thus, the tool led more participants to inspect the faulty statements, even when they did not find the bug. This result is confirmed by observing the developers' comments—most of them stated that SFL helped to focus on the faulty code regions, serving as a guide to start the debugging task.

Most participants deem SFL useful and easy to use, showing willingness to use it in the future. These results include even those who did not find the bugs. About 69% responded positively regarding their intention to use the tool in the future, which suggests that participants are prone to use techniques and tools to support fault localization. Regarding the inspection of SFL lists, there are no significant differences in using the list of methods or the list of lines. Developers tend to investigate a few elements of the lists, mainly the well-ranked ones.

Our findings indicate that SFL helps developers to find and to get closer to faulty code excerpts. SFL was well-accepted by the participants of our study. Also, they would like to have more information (e.g., variables, classes, and related test cases) to help developers to understand, locate, and fix faults. Different from previous studies, our results suggest that SFL improves fault localization even when the fault is not placed at the top picks. This observation is backed by quantitative and qualitative data.

The main contributions of this paper are as follows:

- An analysis of how SFL helps developers to locate faults. We assessed the debugging performance of 26 developers with and without SFL using two faults from real programs.
- An evaluation of how SFL leads developers closer to the fault regions by assessing if they (1) locate the faulty methods and (2) inspect the faulty lines with and without SFL.
- An analysis of the developers' intention to use SFL in the future.
- An analysis of the developers' comments and observations on using SFL.
- An analysis of the developers' behavior while using SFL.

The remainder of this paper is organized as follows. Section 2 reviews previous SFL techniques and related user studies. Section 3 describes our experimental design. The results and discussion are shown in Section 4. We draw our conclusions in Section 5.

2 | RELATED WORK

Several automated debugging techniques have been proposed in the last decades. From those, SFL techniques are promising due to their relatively lightweight execution costs and their intuitive assumption of suggesting elements more likely to be faulty. To further advance the techniques for practical use, experiments with developers are needed to evaluate theoretical assumptions of SFL and to assess whether and how they use SFL. Below, we present studies that are related to this paper.

2.1 | User studies on debugging and fault localization

Several debugging techniques have been proposed since the late 1950s.[21] However, few user studies are assessing these techniques.[10] Most of these assessed the practical use of program slicing techniques.[22–26] Francel and Rugaber[24] noted that only 3 of 111 papers on slicing-based debugging techniques had addressed practical use issues by that date. Another user study assessed the human accuracy at locating faults.[27] Youngs[28] evaluated faults from 42 developers to identify what faults they made and how their experience affects the occurrence of these faults. Gould[29] showed in his user study that developers often use several information sources during debugging, from variable names to source comments. The author claimed that developers frequently search for clues in the source code while debugging.

Other studies have been proposed to understand the developers' behavior during debugging. Vessey[30] suggests that experienced developers can chunk code while debugging. They also use breadth-first strategies associated with a system view of the domain. Novice developers are less able to chunk code and use depth-first strategies. They also use breadth-first approaches but with less ability to think in terms of a system domain. Gilmore[31] stated that comprehension process is completed before debugging occurs, but comprehension and debugging are strongly related. He also claimed that debugging models proposed until then were simplistic by considering comprehension and debugging separately. By comprehending a program, developers should have their debugging tasks facilitated.

Other user studies regarding debugging tools were proposed. Ko and Myers[26] evaluated their Whyline debugging tool[¶ 32] with 20 developers. Whyline provides questions to assist developers in understanding a program's behavior. The tool uses static and dynamic slicing to formulate questions, presenting such questions graphically and interactively. Two real faults from ArgoUML (with 150 KLOC) were used in the experiments. Using Whyline, more developers located the bugs compared with those that did not use the tool. Developers were also faster using the tool. Wang et al[33] evaluated whether fault localization techniques based on information retrieval (IRFL) help developers to locate faults. The study had 58 participants who performed two debugging tasks, interchangeably using and not using the IRFL tool. The authors selected eight bugs from the SWT project[#]. The results show that bug reports with more useful information (i.e., descriptions of program entities) helped developers to identify the faulty files faster but did not improve in locating the bugs. Good IRFL suspiciousness lists (when a faulty file is ranked first) helped developers more when the respective bug report had no information about program entities.

Böhme et al[34] performed a study regarding the debugging process in which 12 professional developers were asked to manually debug 27 real bugs from two open source C programs. Their results show that 63% of the bug fixes sent by the participants were actually correct. The authors built a benchmark called DBGBench, which includes fault locations, patches, and debugging strategies provided by the participants. This benchmark can be used to evaluate fault localization techniques. Castro et al[35] performed a user study with the Pangolin tool[‖], which detects software features (i.e., units of functionality) for software maintenance and evolution. Pangolin is based on the GZoltar tool.[11] The 108 participants of the study were asked to locate a feature of the Rhino project[**]. The participants were split into two groups, one using the proposed tool and the other one using the Eclemma[††] coverage tool. Participants using Pangolin found the feature locations more precisely than those without using the tool.

2.2 | User studies on SFL

Parnin and Orso[10] proposed the first study to assess the practical use of SFL. They evaluated several theoretical assumptions that SFL studies make about ITA developers. A total of 34 participants took part in two experiments, performing two debugging tasks in Tetris[‡‡] (2.4 KLOC) and NanoXML (4.4 KLOC).[36] In the first experiment, 24 participants used SFL in one of the tasks, and in the second experiment, the 10 other ones used SFL in both tasks. The authors found that SFL improved the performance of more experienced developers. Moreover, developers did not follow the suspicious lists in order while debugging and did not find a bug immediately after inspecting it. The authors also suggest that SFL techniques should focus on ranking faulty statements among the first positions to avoid developers giving up on using such tools.

Gouveia et al[11] performed a user study to evaluate the GZoltar tool[§§].[37] GZoltar is an Eclipse plug-in that provides three different visualizations to report SFL results. These visualizations allow developers to investigate the structure of OO Java projects, showing the most suspicious namespaces, packages, classes, methods, and statements. The study was performed by 40 participants, who searched for a fault in XStream[¶¶] (17.4 KLOC). The 20 participants that used GZoltar found the bug, against 7 of 20 participants that did not use it. The developers were also faster

to locate the fault using GZoltar. The tool received positive evaluations regarding its usability. This study is more focused on the developer's performance using the tool and not in the behavior of programmers while debugging.

Xie et al[12] conducted a study with 207 participants and 17 debugging tasks. Each participant performed two tasks, with and without using an SFL tool. Seven standard Computer Science algorithms (e.g., quicksort and heapsort), with at most 500 LOC, were used in the experiments. Four of these programs contain a single file. The faults were seeded by the authors and categorized into six groups. Their results showed that SFL helped the participants to locate faults when such faults were well-ranked (i.e., ranked among the most suspicious statements). The results also showed that SFL can decrease debugging efficiency, even for the well-ranked faults. Moreover, the authors found that most participants started with a first scan (i.e., a brief overview) of the code before using the SFL lists. Xia et al[13] presented a user study about SFL with 36 professional developers. They used 16 real bugs from four open source projects. The participants were divided into three groups: using an accurate SFL tool, a mediocre SFL tool, and without using SFL. The accurate tool ranks the faults among the top 5 positions, while the mediocre tool ranks them between the top 6 and the top 10. Each participant performed four debugging tasks. Their results showed that developers using the accurate tool were more effective and efficient than those using both the mediocre one and without using SFL. The developers using the mediocre SFL tool were also more effective than those without using SFL.

Li et al[38] performed a user study about their technique called Enlighten, which combines SFL, dynamic dependencies, and user feedback to indicate suspicious method calls with their input and output values. The method calls are evaluated by users in an iterative debugging process. A total of 24 participants were divided into two studies. The first study had two faults from Commons-Math[##] (16 KLOC). The second study had one fault from jsoup[||||] (10 KLOC) and the other from NanoXML. These last two faults were deemed as hard to find. In each study, the developers performed two debugging tasks, with and without Enlighten. All participants found the bugs using the proposed technique. Without it, some participants from the second study did not find the bugs. The developers were also faster using Enlighten in both studies.

Horváth et al[14] conducted a user study for their interactive fault localization technique. The technique takes into account the knowledge of developers, which can indicate program entities that are not suspicious in a ranking list. Then, the technique recalculates the suspiciousness list for the other entities. The study had 36 participants, and most of them are students. The participants were split into three groups, and each of them performed four fault localization tasks—two using the proposed techniques and two without the technique. The subject programs belong to the Defects4J database. The authors did not find differences in the users' performance to locate the bugs using the interactive fault localization technique, but the time spent on the debugging tasks was reduced when using the proposed technique. Moreover, most participants considered the technique useful to locate bugs.

These studies have presented both similar and divergent results regarding the usefulness of SFL in practice, which encouraged us to further investigate the factors that can lead developers to accept or refuse the practical use of SFL techniques. As in previous work, we investigate both effectiveness and efficiency of SFL compared with traditional debugging. We also analyze issues regarding the developers' behavior when using SFL and whether their programming experience affects their performance using SFL.

Different from previous work, we evaluate how close to faulty code elements an SFL tool can take its users. More specifically, we assess if there is a correlation between reaching a fault site and finding a bug. Then, we assess if developers can identify the faulty methods and reach the faulty lines using SFL. In this way, an SFL tool can be useful to locate bugs or, at least, to provide hints for developers to start to understand faulty behaviors.

We also assess how the developers perceive the usefulness and ease of use of SFL in practice, which indicates their willingness to use SFL in the future. Finally, we investigate whether the granularity of suspicious information impacts on effectiveness and efficiency. We use two levels of suspicious information: lines and methods. In what follows, we describe the methodology, techniques, and procedures applied in this study.

3 | STUDY DESIGN

Our study evaluates how developers use SFL through an assessment with users. We aim to understand how developers interact with an SFL tool and whether SFL improves their performance to locate bugs. We also evaluate the developers' intention to use SFL in the future. In this section, we present the details of our study design. We describe our research questions, the methodology of the user study setup. We also describe the Jaguar SFL tool, which was used in the experiments.

3.1 | Research questions

Our goal in this study is to understand how SFL techniques and tools impact users performing fault localization tasks. The research questions that address our concerns about the practical use of SFL are presented below:

RQ1: Does SFL help developers to locate faults?
RQ1 evaluates the effectiveness of using SFL to locate bugs. Can developers actually benefit from SFL to locate bugs in practice? This is a fundamental question since SFL will only be adopted in practice if it improves the developers' ability to locate faults. Also, we want to assess if the *level of information* (a list composed of suspicious lines or methods) has some impact on the developers' effectiveness. To answer RQ1, we compare the number of developers who found bugs using SFL and without using SFL.

RQ2: Do developers locate faults faster using SFL?
RQ2 evaluates the efficiency of SFL in locating bugs. A technique that reduces time spent in debugging may encourage developers to use it. To answer RQ2, we compare the time spent by participants who found bugs using and without using SFL.

RQ3: Does SFL lead more developers to inspect faulty code?
RQ3 evaluates whether SFL guides developers towards faulty code excerpts. By doing so, developers will be more prone to inspect the code for faults where they occur. Reaching faulty sites may be a first step to understand a bug. To answer RQ3, we verify if developers using SFL inspect more the faulty line and if they locate the faulty methods when compared with those without using SFL.

RQ4: Do study participants intend to use SFL in practice?
In RQ4, we investigate whether developers are actually interested in using SFL in their daily activities. This is another important question regarding the practical adoption of SFL. Developing and proposing new automated techniques to aid debugging is worthy only if developers are interested in using them. To answer this question, we use the TAM to ask participants how they evaluate usefulness and ease of use of the SFL tool and if they intend to use SFL in the future.

RQ5: Does the level of code information impact on the fault localization performance?
In RQ5, we assess whether the level of code information has some impact on the developers' ability to locate faults. To answer this question, we compared the fault localization results of developers using SFL lists of methods and lines.

Beyond the questions presented above, we also evaluate other important issues related to the practical use of SFL. First, we investigate how programming experience impacts the developers' performance using SFL. Second, we analyze and discuss the developers' comments and opinions about the use of SFL. Finally, we verify how developers behave during debugging tasks—how they navigate through the SFL lists, and how they search for bugs.

3.2 | The Jaguar SFL tool

Jaguar (JAva coveraGe faUlt locAlization Ranking)*** [39] is an open source SFL tool for Java, which is also available as an open source plug-in for the Eclipse IDE. Jaguar collects program spectra from test execution (using JaCoCo†††) and generates lists of suspicious elements for several ranking metrics (e.g., Ochiai, Jaccard, Tarantula). The ranking metrics calculate suspiciousness scores for each program element. The more an element is executed in failing test cases, the more suspicious it is. Conversely, the more such element is executed in passing test cases, the less suspicious it is.

The Jaguar plug-in presents the SFL results in lists, coloring the suspicious elements from red (high) to green (low) as done by Jones et al.[3] To assess the code granularity, we used two different views of Jaguar in this experiment: a list of suspicious methods and a list of suspicious lines. The tool also has text search and slider widgets to filter the suspicious results. Figure 1 shows the Jaguar tool, which appears at the right-side panel of Eclipse. By clicking on any statement or method in the Jaguar list, the user is taken to the corresponding code in the editor view area. The code in the editor view is also colored according to its suspiciousness. Figure 2 shows the Jaguar's lists of lines and methods.

Jaguar also tracks user actions performed within Eclipse. These actions include mouse and keyboard events, such as clicks on the plug-in area, clicks and code editing on the Editor view, breakpoint settings, and test execution. We gathered these actions to assess the behavior of participants while debugging.

3.3 | Participants

The participants' selection was made by *convenience sample*.[40] This is a common practice in user studies in the Software Engineering area due to the difficulty to obtain representative samples.

We invited graduate and undergraduate students from Computer Science and Information Systems courses at several Brazilian universities: the School of Arts, Sciences, and Humanities of the University of São Paulo (EACH-USP), the Institute of Mathematics and Statistics of the University of São Paulo (IME-USP), the Institute of Mathematics and Computer Sciences of the University of São Paulo (ICMC-USP), the Department of Computer Science of the Federal University of Technology – Paraná (UTFPR), the Department of Computing of the Federal University of

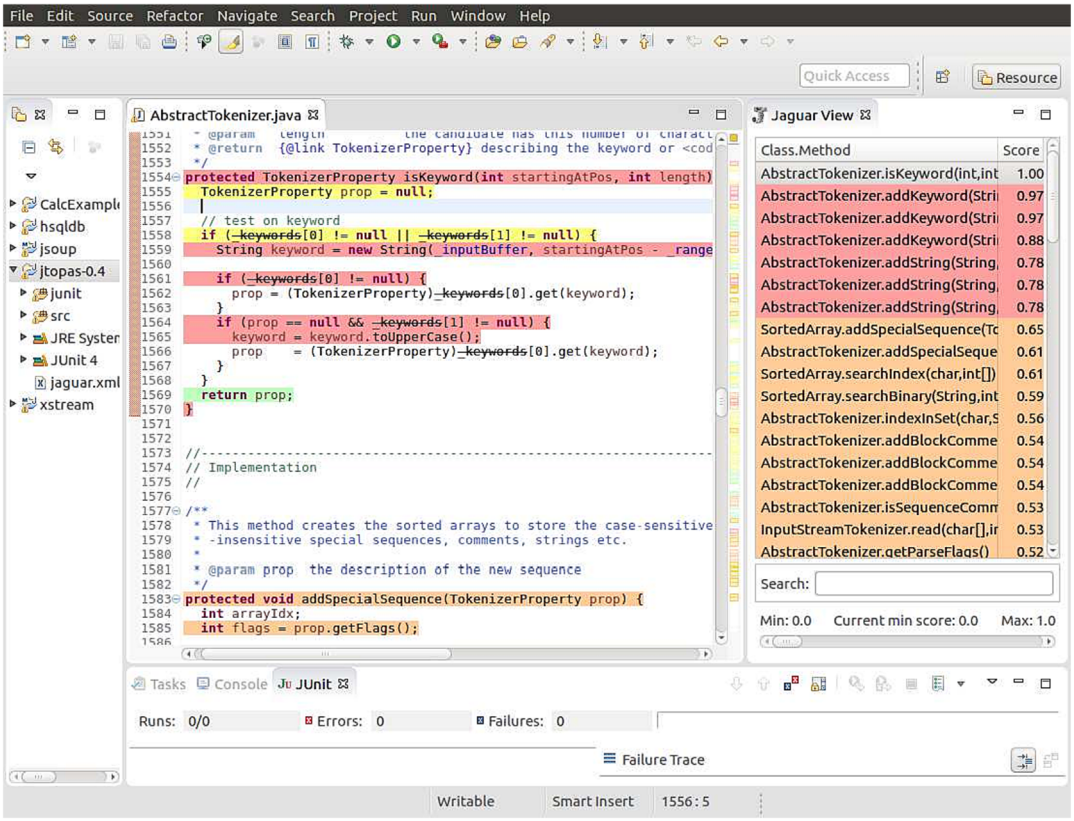


FIGURE 1 Jaguar tool.

São Carlos (DC-UFSCar), the Informatics Center of the Federal University of Pernambuco (CIn-UFPE), the National Institute for Space Research (INPE), and the Center for Mathematics, Computation and Cognition of the Federal University of ABC (UFABC).

All participants were voluntary. Students enrolled in a Software Engineering course at the EACH-USP were invited to participate in the experiment. Their participation was considered a nonmandatory activity of the course. Only students approved in the course could earn a bonus of 0.5 on the final grade (maximum of 10) by participating in the experiment. Thus, the students' participation would not improve their chances of approval in the course. Furthermore, the bonus was not related to the performance using the tool nor if the debugging tasks were finished or not. To receive the bonus, the students should send a screenshot of Jaguar in use. Their anonymity was kept as for the other participants. Thus, we do not know who are the respondents of the TAM questionnaire.

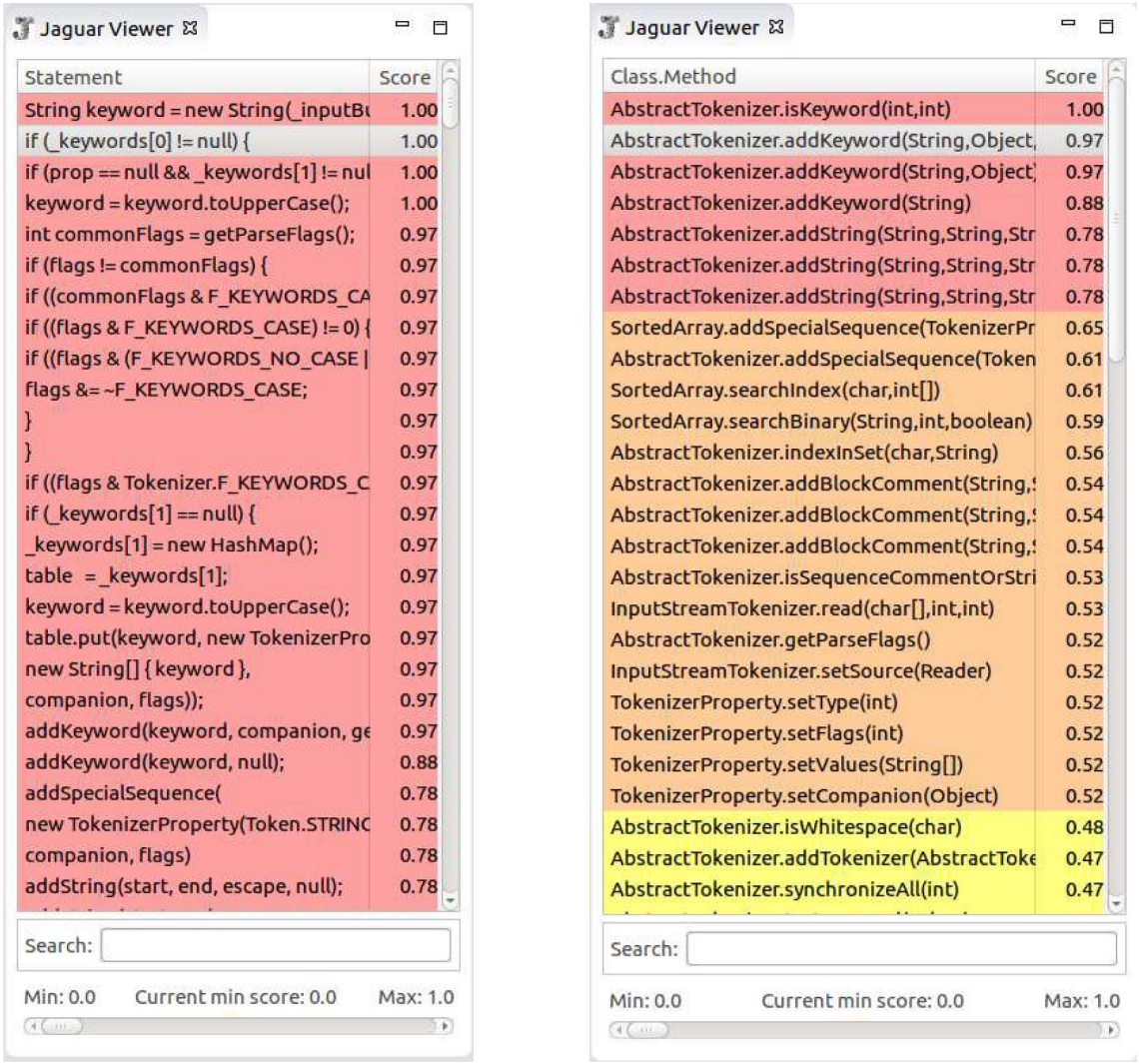
Among the volunteers willing to participate in the study, we selected those with at least a minimum previous experience (a self-declared basic level) in the Java language. Other nonexclusive criteria were a desirable previous experience with the Eclipse IDE and JUnit. Equivalent technologies, such as *TestNG*[‡‡‡], *NetBeans IDE*[§§§], and *IntelliJ IDEA*[¶¶¶], were also considered as valid experience. The participants must had no previous developing experience in the programs used in the experiment (described in the next section) to avoid the impact of previous knowledge on their performance.

3.4 | Subject programs and faults

We selected two open source Java programs for the experiment: *jsoup*[###] and *XStream*[||||]. Jsoup is an HTML parser composed of 10 KLOC and 468 test cases. XStream is a library to (de)serialize objects to(from) XML files composed of 17 KLOC and 1457 test cases.

We set two faults neither too easy nor too hard to find. These faults are not dependent on specific knowledge about the programs' domain. The jsoup's bug is real, which we found in the program's repository. The fault is a wrong comparison in an *if* statement, which compares two elements using *equals()* instead of ==. The XStream's bug was seeded by Gouveia et al[11] for their experiment. It is a wrong comparison operator of a null condition in an *if* statement (== instead of !=).

We used the Ochiai metric to calculate suspiciousness. We opted for keeping the faulty elements in their original ranking positions. We wanted to verify how the developers inspect the code in a realistic scenario, in which SFL may or may not classify the faulty elements among the first picks.



(A) List oflines

(B) List of methods

FIGURE 2 (A, B) Jaguar's lists of lines and methods.

The faulty lines of jsoup and XStream were assigned with intermediate scores, 0.65 and 0.67, respectively. The faulty methods of jsoup and XStream were assigned with a score of 0.65 and 1.0, respectively. For jsoup, Jaguar ranks the faulty line in the 18th position, while the faulty method was ranked in the fifth position. For XStream, Jaguar ranks the faulty line and method in the 31st and 1st positions, respectively. This means that both faults are not painted in red—except for the fault in the method list of XStream—in such a way that some false positives may be inspected until reaching the bug. Thus, we are evaluating subjects in which the faulty statements were not so well-ranked by Jaguar—that is, away of the top 10 most suspicious statements.

The faulty methods were ranked among the top five suspicious ones. However, to locate the faults, it is necessary to inspect the faulty lines within the methods. For XStream, the faulty line is in the 15th position among the most suspicious lines of the faulty method. There are 11 lines in the faulty method with the maximum score (1.0) and other 26 lines with the same score of the faulty line (0.67). For jsoup, there are 13 more suspicious lines in the four methods ranked before the faulty one. Inside the faulty method, the faulty line is tied with other five lines with the same 0.65 score.

Moreover, these faults are suitable to be found without using the SFL tool. Both faults generate a few failing test cases to be inspected, 2 and 5 test cases for jsoup and XStream, respectively. The XStream fault throws an exception in which the faulty method is at the top of the stack trace information. The jsoup failing test cases call a particular method, which in turn calls the faulty method, leading directly to the faulty method in two steps. Our intention was to keep fair conditions for finding the faults using and not using SFL.

3.5 | Experimental group design

The study was devised to be performed in two debugging tasks: one task using Jaguar and the other task without using Jaguar. Thus, all participants will have contact with the tool. According to Wainer,[41] this factor avoids a bias effect on the performance of a control group, which may occur when participants use only conventional treatments during experiments. To assess RQ5, each participant should ideally perform three tasks: one using a list of lines, another using a list of methods, and a third one without using Jaguar. However, such experiment would take too long, which may lead participants to get tired and quit the experiment before its completion. So, each participant performed only one Jaguar task using one of the two available SFL lists.

Two major factors that may impact the results of our experiment are the *faults* and the *order* in which the debugging tasks are performed. The faults may present different difficulty levels of location, resulting in a threat to validity known as *instrumentation*.[42] To cope with a potential difference in the level of difficulty of the faults, they were interchanged between the Jaguar and Eclipse tasks through the experimental groups. Thus, some participants debugged one of the faults using Jaguar, while others looked for this same fault using Eclipse without the Jaguar tool. The order of the tasks may also influence the results: Always using the same treatment (task) as the second task may lead to *fatigue*. Moreover, using a conventional technique after using a new one may cause *frustration*, impairing the results of the second task. Both fatigue and frustration are related to a threat to validity known as *maturation*.[42]

We defined our groups using the *Latin Square* design,[43] which avoids the impact of *nuisance factors* in the experiment. In our case, the nuisance factors are the faults used for the experiment and the order in which the tasks are performed. In the Latin Square design, we mitigate the nuisance factors by interchanging their orders through different experimental groups.

An experiment in which each participant performs both the conventional and the proposed approaches would have four experimental groups. However, as we have two versions of Jaguar (lines or methods) and each participant uses only one version, we needed eight experimental groups. Thus, our study has two Latin Square sets: four groups using SFL with a line list (labeled as *JaguarL*) and the other four groups using SFL with a method list (labeled as *JaguarM*). The task in which Jaguar is not used was labeled as *Eclipse*. The Jaguar's line and method lists are those shown in Figures 2A and 2B, respectively. Table 1 details our group design, where the *G* columns are the group number.

Thus, our study has the following treatments: **Jaguar**: Participants can use Jaguar and all other Eclipse resources to search for the bug. It comprises participants that will use either a list of lines (JaguarL) or a list of methods (JaguarM).
JaguarL: It is the Jaguar treatment in which we consider only those that use the list of lines.
JaguarM: It is the Jaguar treatment in which we consider only those that use the list of methods.
Eclipse: Participants can use all resources of Eclipse to search for the bug, except for Jaguar.

3.6 | Questionnaires

The study has two questionnaires, which were applied before and after the experiment. The questionnaires do not ask for any personal information, ensuring confidentiality and privacy for the participants.

3.6.1 | Pretask questionnaire

Before the experiment, we applied an online questionnaire to know the experience level of those who would like to take part in the study. This questionnaire has questions regarding years of experience as professional developers, in programming, in Java, use of an IDE, and in automated testing. We also ask about their self-declared skill levels in Java, use of IDE, and in automated testing. They must choose one of the four available levels: none, basic, intermediate, and advanced. Feigenspan et al[44] claim that self-estimation is a reliable way to measure programming experience. Thus, we used the answers to the self-declared skill levels to balance the distribution through the experimental groups. Those who have at least a

TABLE 1 Experimental groups (G).

G	First task	Second task	G	First task	Second task
1	JaguarM + jsoup	Eclipse + XStream	5	JaguarL + jsoup	Eclipse + XStream
2	JaguarM + XStream	Eclipse + jsoup	6	JaguarL + XStream	Eclipse + jsoup
3	Eclipse + jsoup	JaguarM + XStream	7	Eclipse + jsoup	JaguarL + XStream
4	Eclipse + XStream	JaguarM + jsoup	8	Eclipse + XStream	JaguarL + jsoup

basic knowledge level in Java were allocated to one of the experimental groups to perform the experiment (more details on the allocation process on Section 3.7).

3.6.2 | Posttask questionnaire

We built another questionnaire to be filled out by the participants after the debugging tasks. This questionnaire is composed of questions about their performance on the tasks, their opinions, and comments about the SFL tool and the carried out tasks, the tool's usability, and to understand their intention to use SFL in the future.

We assessed the *behavioral intention* (BI) to use SFL using the TAM.[18] TAM is a widely known model used to assess how users perceive the *usefulness* and *ease of use* of new technologies, which in turn impacts their intention to use such technologies in the future. Perceived usefulness (PU) indicates a person's opinion on how useful a new technology is. Perceived ease of use (PE) is related to the easiness of using a new technology to perform her/his activities.[18] Both PU and PE are positively correlated with the BI of use.[18]

We built our TAM questionnaire based on the items presented by Davis and Venkatesh,[45] Brooke,[46] and Feigenspan et al.[44] We used a 7-point *Likert* scale to gather the participants' degree of agreement with each statement, from (1) totally disagree to (7) totally agree. We included a neutral value since participants may neither agree nor disagree with a statement. The scale items related to the TAM constructs (PU, PE, and BI) are shown in Table 2. The italic words are the main concepts addressed by each item.

Regarding usability issues, this questionnaire asks how developers evaluate its main features, such as the color scheme, filters, and the lists. The questions related to their performance ask about where are the faults, what caused them, and how to fix them. We also ask their opinions about the difficulty level of the bugs, suggestions, and comments.

3.7 | Participants' allocation

We used the *haphazard intentional allocation* (HIA) method[47] to distribute the participants through the experimental groups. The HIA method uses the *total variance*[48] to determine in which group a new participant must be allocated considering desirable *features* of the experiment and the *weights* (ω) associated with them. A perturbation parameter ($\epsilon \in [0,1]$) is used to add some randomness factor to the deterministic process. Setting $\epsilon = 0$ means no randomness at all, and the selected group is that which yields the minimum total variance after the participant's allocation. At the other extreme, setting $\epsilon = 1$ corresponds to a random allocation. Setting proper values to for ϵ, as briefly described below, provides a suitable balance between the deterministic and randomized schemes.

This allocation method allowed us to balance participants with different technical skill levels through the groups, even for a small number of participants and a large number of groups. We used three skill features in our study: knowledge levels in Java ($\omega = 2$), in usage of IDEs ($\omega = 1$), and in automated testing ($\omega = 1$), which are gathered in the pretask questionnaire.

To calibrate the perturbation parameter ϵ, we created a randomized population of 2000 individuals. We ran HIA for different samples sizes from this initial population, $n = 20,30,40,50,60,70$, with different values for ϵ ($\epsilon = 0.01,0.05,0.1,0.2,0.3$). Each combination was run four times.

TABLE 2 TAM items.

Perceived usefulness (PU)	
U1	Using Jaguar *improves* my performance in the debugging task.
U2	Using Jaguar in the debugging task increases my *productivity*.
U3	Using Jaguar enhances my *effectiveness* in the debugging task.
U4	Overall, I find Jaguar *useful* to perform the debugging task.
Perceived ease of use (PE)	
E5	*Learning* to use Jaguar is easy for me.
E6	Interacting with Jaguar does not require a lot of *mental effort*.
E7	Overall, I find Jaguar *easy to use*.
E8	I find easy to get Jaguar *to do what I want* it to do.
Behavioral intention of use (BI)	
B9	Assuming I had access to Jaguar, I *intend* to use it.
B10	Given that I had access to Jaguar, I *predict* that I would use it.

Abbreviation: TAM, Technology Acceptance Model.

The choice of ϵ was based on two criteria: *optimality* and *decoupling*. The first criterion, optimality, is based on the total variance and concerns the difference among the relative frequencies of participants in the several categories for the eight groups. The second criterion, decoupling, concerns the absence of a tendency to allocate each pair of participants to the same group or to opposite groups. In this work, we use Fleiss' Kappa coefficient, which is an agreement coefficient ranging from -1 (complete disagreement) to $+1$ (complete agreement).[49] Values close to 0 (absence of agreement or disagreement) are preferable. Based on these criteria, we set $\epsilon = 0.05$, which yielded a good balance between the total variance and Fleiss' Kappa coefficient. We carried out this procedure because we did not know a priori the number of participants that would take part in the study. The pretask questionnaire was available through a web application that implemented the HIA method, allowing us to have an automatic and interactive allocation process. Thus, each participant was allocated to a group automatically after she/he filled out the questionnaire.

3.8 | Experimental procedure

The experiment was carried out remotely using virtual machines (VMs). The VMs ensure that the participants have the same environment to perform the experiments. These VMs contain all resources required for the experiment: a *Lubuntu 14.04* operational system, Eclipse IDE with the Jaguar plug-in installed, the subject programs and faults, training material on Jaguar, and the instructions for the experiment. We created two VMs, one for Jaguar with lines and the other for Jaguar with methods. Each VM has four users, one for each experimental group. The participants needed a computer with 2GB RAM and the VirtualBox program to run the VM. Thus, they could perform the experiment from anywhere. We defined a deadline of 3 weeks from the disclosure of the experiment to finish it.

Figure 3 depicts the procedure of our user study. We invited students through an email consent letter. Those students who wanted to participate in the experiment filled out the pretask questionnaire. We implemented the HIA method on a web form to automatically allocate the participants. Those with at least a basic knowledge of Java were allocated to the experimental groups, receiving an ID, a user login, a password, and a link for downloading the VM. The ID was provided to keep the participants' anonymity. This ID is used to perform the debugging tasks and to fill out the final questionnaire, which allowed us to link the log files to the questionnaires of each participant.

The participants read the training material and the instructions for the experiment. The training material was composed of a document and a video explaining how to use the SFL tool and a tryout version of the tool to allow the participants to interact with it before starting the tasks. The participants were recommended to spend up to 30 min for each task. However, we let them control the time by themselves.

To start the first task, the participants need to click on a *start* button, which allows Jaguar to create the log file. To finish the second task, they should click on a *stop* button to send the log file to our server. To make sure that the log files are sent, we created a script to send the log files whenever the user session is logged off or during the VM shutdown.

After finishing the tasks, participants were asked to respond the post-task questionnaire, which was available in a web form. We then used the data collected in the questionnaires and in the log files to perform our analysis.

3.9 | Use of the collected data

We used the answers to the pretask questionnaire to perform the allocation process of participants into the experimental groups. We also used this questionnaire to evaluate the impact of experience on debugging performance.

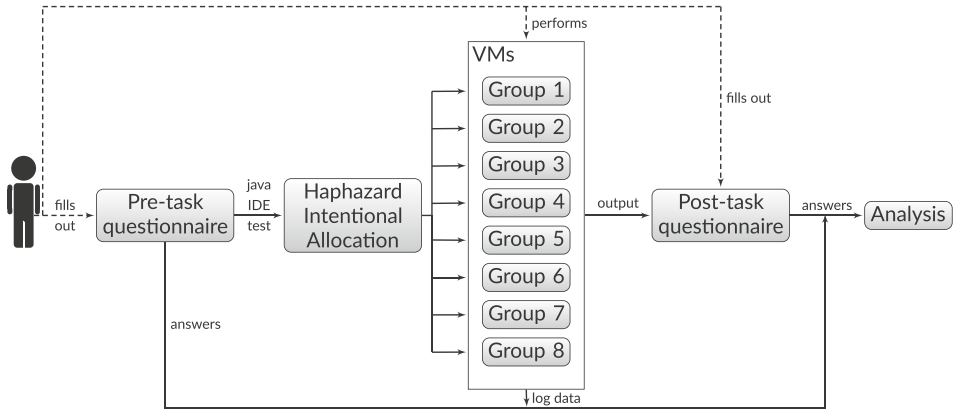


FIGURE 3 Experimental procedure.

To determine the participants who found the bugs, we considered those that correctly answered, in the post-task questionnaire, where is the bug (class, method, and line number) and how to fix it. We also used this questionnaire to identify the participants who located only the faulty method. The posttask questionnaire also enabled us to analyze the participants' comments and observations about Jaguar and evaluate issues, suggestions for improvements, and BI to use SFL.

To evaluate the time spent on the debugging tasks, we used the timestamps recorded in the navigation logs. We also used the log files to identify participants who inspected the faulty lines, to verify whether the participants gave up on using SFL, and how they performed the debugging tasks.

3.10 | Experiments' instruments

The full experimental package containing the SFL tool, subject programs, training material, HIA implementation, questionnaires, and experiment's protocol is available at github.com/saeg/user-study-sfl to enable the reproducibility of the research. The HIA and data analysis routines were implemented using the R language.

4 | RESULTS AND DISCUSSION

Before answering our research questions (see Section 3.1), we assessed how balanced is the distribution of participants through the groups regarding the study nuisance factors: the *faults* and the *order of the tasks*. By doing so, we can verify if the comparison of our results for those using the SFL tool (Jaguar) and without using it (Eclipse) has a fair distribution of the faults and the order in which the tasks were performed.

Then, we present the answers to our research questions. We also present the results regarding the impact of the developers' experience and their behavior during the tasks.

4.1 | Participant distribution

Figure 4 shows how the participants were distributed through the groups using the HIA method. In total, 41 volunteers filled out the pretask questionnaire, while 31 of them performed the experiment. The 10 volunteers who only filled up the pretask questionnaire but did not perform the experiment are represented by the gray bars. From the 31 participants, we excluded five due to incomplete data issues, such as the absence of the questionnaire or log file, as shown by the red bars. Thus, our experiment had *26 valid participants* that performed the experiment, and we have both the answers to the post-task questionnaire and the navigation log files, shown by the blue bars. Nielsen[50] recommends at least 20 users for usability studies that aim to apply statistical tests, which indicates that our sample suffices for conducting a statistical analysis. To keep the participants' anonymity, we did not ask for personal information. Thus, we do not know how many participants came from each of the invited universities.

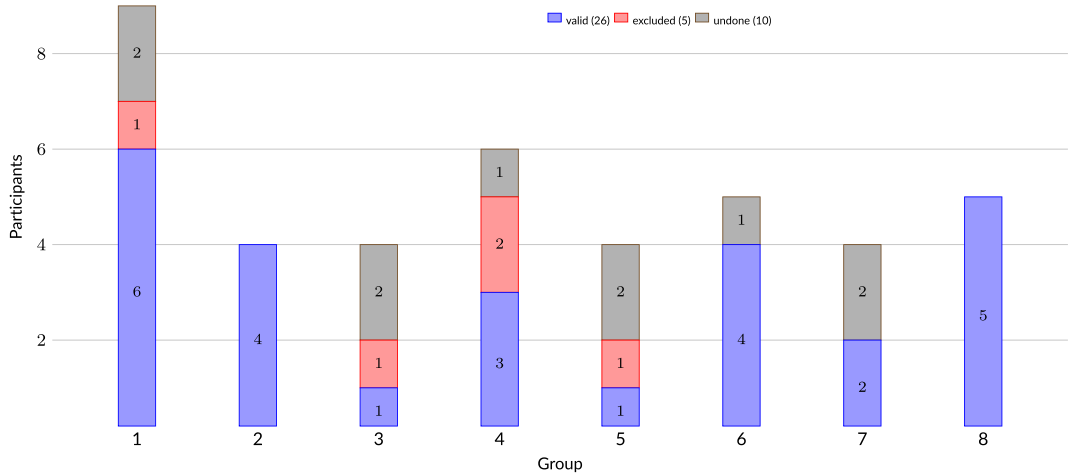


FIGURE 4 Participants' allocation.

Figure 4 shows that the HIA method was successful in allocating at least four volunteers for each group, keeping a balanced distribution for the volunteers of our study. Indeed, if a random allocation procedure were used, the probability of getting groups with less than four participants would be 94%. However, some groups had few valid participants because some of them filled out the pretask questionnaire but did not perform the experiment (e.g., groups 3 and 5). This could introduce a potential bias in data analysis due to an imbalance between groups with respect to faults and order of the tasks. Thus, we assessed whether the experimental groups were significantly different considering these two factors. Tables 3 and 4 summarize how many participants used Jaguar****, Eclipse, JaguarL, and JaguarM for each fault and in which order the tasks were performed, respectively.

To compare the distributions of participants between groups, we used the network algorithm proposed by Mehta and Patel.[51] This algorithm is an extension of Fisher's exact test for independence[52] in contingency tables larger than 2 x 2. Our null hypothesis is that the distribution is proportional to both Jaguar and Eclipse, which means that the distribution of faults and the order of the tasks do not impact the results. We used a significance level of 5% for all statistical tests performed in this study. For the distribution of faults, the *p*-value is 0.43; for the order of the tasks, the *p*-value is 0.21. Thus, despite the groups had a different number of participants, the faults and the order of the tasks were proportionally distributed among the participants. Therefore, these results indicate that the Latin Square design and HIA method were effective in keeping the experiment results free from the impact of such factors.

4.2 | RQ1: Does SFL help developers to locate faults?

SFL is statistically more effective, with a medium ES. Table 5 shows the statistical test results for RQ1 comparing the results of Jaguar, JaguarL, and JaguarM to Eclipse for both faults. Our null hypothesis (H_0) is that the use of SFL is equal to or less effective than the use of Eclipse without SFL. Bold values represent cases in which the null hypothesis is refuted, that is, SFL is more effective with statistical significance.

We applied Fisher's exact test[52] to assess the effectiveness of the techniques. We measured the ES using *odds ratio*,[53] which is suitable for our binary results of effectiveness (found or not found). According to Chen et al,[53] the ES scale for odds ratio is insignificant (I) for ES < 1.68, small (S) for 1.68 ≤ ES < 3.47, medium (M) for 3.48 ≤ ES < 6.71, and large (L) for ES ≥ 6.71.

The use of JaguarM also resulted in better effectiveness with statistical significance and medium ES. JaguarL's effectiveness was not statistically superior to Eclipse.

Table 5 also shows the number of participants who found faults using Jaguar, Eclipse, JaguarL, and JaguarM. The *jsoup + XStream* column represents the number of successful developers for both projects. This table also details how many participants found faults in each project (shown in columns jsoup and XStream). The percentages are relative to the number of participants with similar settings. For example, four of five developers using JaguarM in XStream found the bug.

TABLE 3 Distribution of participants through the faults.

Fault	Jaguar	Eclipse	JaguarL	JaguarM
jsoup	15/26 (57.69%)	11/26 (42.31%)	6/26 (23.08%)	9/26 (34.61%)
XStream	11/26 (42.31%)	15/26 (57.69%)	6/26 (23.08%)	5/26 (19.23%)

TABLE 4 Distribution of participants through the order of tasks.

Order	Jaguar	Eclipse	JaguarL	JaguarM
First task	15/26 (57.69%)	11/26 (42.31%)	5/26 (19.23%)	10/26 (38.46%)
Second task	11/26 (42.31%)	15/26 (57.69%)	7/26 (26.92%)	4/26 (15.38%)

TABLE 5 Statistical results for RQ1—effectiveness results of participants by technique and project.

Technique	*P*-value	Effect size	jsoup + XStream	jsoup	XStream
Eclipse	–	–	5/26 (19.23%)	2/11 (18.18%)	3/15 (20.00%)
Jaguar	**0.038**	3.6 (M)	12/26 (46.15%)	5/15 (33.33%)	7/11 (63.63%)
JaguarL	0.144	3.0 (S)	5/12 (41.67%)	2/6 (33.33%)	3/6 (50.00%)
JaguarM	**0.049**	4.2 (M)	7/14 (50.00%)	3/9 (33.33%)	4/5 (80.00%)

Note: Column "jsoup + XStream" has the percentage and number of participants who found the bugs in both programs. Columns "jsoup" and "XStream" detail the faults found per project.

Almost half of the participants were successful in using SFL. On the other hand, using Eclipse only, less than one-fifth of them found a bug. Four participants found the faults using both Jaguar and Eclipse; thus, only one participant who found a fault in the Eclipse task did not find the other fault in the Jaguar task. The above results show that the participants can locate more faults when using an SFL tool. Also, the effectiveness results obtained in our study corroborate previous studies, in which SFL improved fault localization.[11–13,38]

4.3 | RQ2: Do developers locate faults faster using SFL?

There are no significant differences in efficiency using SFL. As we measure efficiency by the time spent for those who found bugs, our sample size is more limited to conclude using statistical tests. The values for the statistical tests are shown in Table 6. Our null hypothesis (H_0) is that the use of SFL is equal to or less efficient than the use of Eclipse without SFL. We used the *Wilcoxon Rank-Sum* test[54] for the hypothesis tests and *Cliff's delta*[55] to measure the ES. These statistical methods are suitable for the efficiency data of our study, which is composed of non-parametric numerical values. The values for Cliff's delta range from -1 to 1. According to Macbeth et al,[56] the ES scale for Cliff's delta is insignificant (I) for $-0.2 \leq ES \leq 0.2$, small (S) for $0.21 \leq ES \leq 0.5$ and $-0.5 \leq ES \leq -0.21$, medium (M) for $0.51 \leq ES \leq 0.8$ and $-0.8 \leq ES \leq -0.51$, and large (L) for $ES > 0.8$ and $ES < -0.8$.

Figure 5 presents violin plots for the time spent in minutes by the developers to locate the bugs using Jaguar and Eclipse. The black dots are the time that each developer spent to locate the bugs. The median of time spent using Jaguar is lower than using Eclipse, respectively, 16:34 and 22:57 min, although the times using Jaguar vary more than using Eclipse. The average of time spent on the tasks is also lower using Jaguar, 15:52, than Eclipse, 20:28. These results suggest that Jaguar may help to reduce the time spent to locate bugs. However, there are also cases in which developers spent more time using SFL.

Although the time is not improved with statistical significance as in Xia et al.,[13] SFL did not impair the efficiency as occurred in Xie et al.[12] We believe that, due to the size of our programs, developers can cut off time in debugging while using SFL, since there are a large number of classes and methods to inspect. An SFL tool that classifies faulty components well can reduce the search space and, thus, improve the debugging efficiency, as shown by the studies of Li et al[38] and Horváth et al.[14]

TABLE 6 Statistical results for RQ2—column H_0 shows the null hypothesis that SFL is equal to or less efficient than the use of Eclipse without SFL, followed by their respective *p*-values and effect sizes.

H_0	*P*-value	Effect size
Jaguar ≤ Eclipse	0.799	0.1 (S)
JaguarL ≤ Eclipse	0.381	0.6 (M)
JaguarM ≤ Eclipse	0.267	−0.52 (M)

Abbreviation: SFL, spectrum-based fault localization.

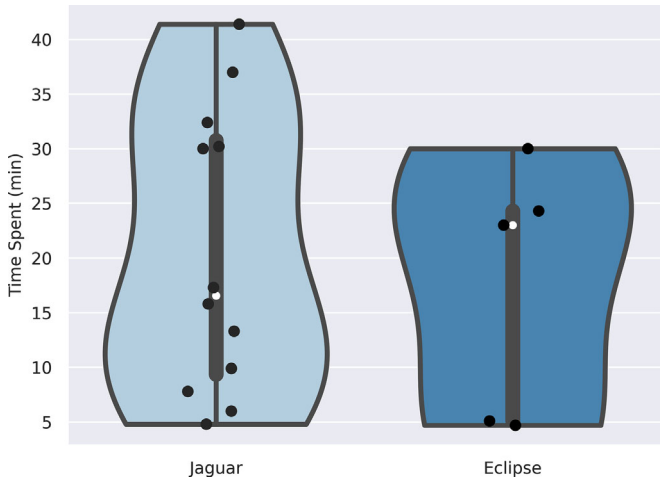


FIGURE 5 Efficiency of Jaguar and Eclipse.

4.4 | RQ3: Does SFL lead more developers to inspect faulty code?

In this question, we want to know if SFL leads developers closer to the faulty regions, which may increase their chances of finding bugs. To assess RQ3, we verify whether SFL can help developers to *inspect the faulty line* and to *locate the faulty method*.

4.4.1 | Does SFL lead more developers to inspect the faulty line?

SFL leads developers closer to the faulty lines. It is intuitive that a participant who reaches the faulty line is more likely to locate the fault. Nevertheless, we evaluated if there is some relationship between clicking on the faulty line at least once and finding the faults. Since each participant performed two tasks, we built a crosstab (see Table 7) for this comparison considering the results of both tasks. The *Yes* and *No* lines are the numbers of participants who clicked or not clicked on the faulty line at least once, respectively. The *Yes* and *No* columns are the debugging results: found or not found. Using Fisher's exact test, we obtained a *p-value of* 4.9×10^{-8} and an *odds ratio value of 6.1*, which shows that **there is a relationship between clicking on the faulty line and finding the bug with statistical significance and medium ES**.

Based on this result, we verified whether the SFL tool leads more participants to click at least once on the faulty lines. Table 8 depicts the crosstab comparing those who inspected the faulty line using Jaguar and Eclipse. The *Yes* and *No* lines are the number of participants who inspected or not inspected the faulty line at least once using Jaguar, respectively. The *Yes* and *No* columns are the number of participants who inspected or not inspected the faulty line at least once using Eclipse, respectively. We measured statistical significance and ES using Fisher's exact test and odds ratio. Our null hypothesis (H_0) for the inspection of lines is that SFL leads to an equal or small number of developers that inspect the faulty line compared with Eclipse.

The results show, with statistical significance and medium ES, that there is a relationship between the use of Jaguar and the inspection of the faulty line, as shown in Table 9 (which follows the same structure as Table 5). We also performed this comparison for JaguarL and Eclipse and for JaguarM and Eclipse. JaguarM also led more developers to inspect the faulty line with statistical significance and large ES. JaguarL did not reach statistical significance.

TABLE 7 Crosstab for clicking on the faulty line at least once and debugging result.

	Found	
Clicked	**No**	**Yes**
No	27	0
Yes	8	17

TABLE 8 Crosstab of those who clicked on the faulty line at least once using Jaguar and Eclipse.

	Eclipse	
Jaguar	**No**	**Yes**
No	7	1
Yes	12	6

TABLE 9 Statistical results for RQ3 – participants who clicked at least once on the faulty line. Column "jsoup + XStream" has the percentage and number of participants who clicked at least once on the fault line for both programs. Columns "jsoup" and "XStream" detail these numbers for each project.

Technique	P-value	Effect size	jsoup + XStream	jsoup	XStream
Eclipse	–	–	7/26 (26.92%)	4/11 (36.36%)	3/15 (20.00%)
Jaguar	**0.002**	6.11 (M)	18/26 (69.23%)	8/15 (53.33%)	10/11 (90.90%)
JaguarL	0.11	4.0 (M)	8/12 (66.67%)	3/6 (50.00%)	5/6 (83.33%)
JaguarM	**0.011**	9.16 (L)	10/14 (71.43%)	5/9 (55.55%)	5/5 (100.00%)

4.4.2 | Does SFL lead more developers to locate the faulty method?

The SFL tool leads more developers to identify the faulty method. We compared the participants who correctly answered the faulty method for both debugging tasks in the post-task questionnaire. Table 10 shows the percentage and number of participants who found the faulty methods using Jaguar, Eclipse, JaguarL, and JaguarM, where the *Total* column shows the total for both projects. The percentages are relative to the number of participants with similar settings.

For this comparison, our null hypothesis (H_0) is that SFL leads to an equal or small number of developers that locate the faulty method compared to Eclipse without SFL. We used Fisher's exact test and odds ratio for the statistical analysis. As shown in Table 10, developers using SFL were more effective in locating the faulty method with statistical significance. The differences between Jaguar and Eclipse presented a medium ES. JaguarL also reached statistical significance with a large ES.

Our analysis indicates that SFL improves the interaction with the faulty lines, which may increase the chance of finding bugs. Although clicking on the faulty line does not guarantee to find a bug,[10] it is necessary to at least reach a faulty code excerpt to understand and fix it.

SFL was also effective in helping developers to identify at least the faulty method. A method is the program entity that often contains complete excerpts of program's functionalities; it is the lowest structure containing the faulty lines. Thus, identifying a faulty method may be the first step to understand a fault. Indeed, in the survey conducted by Kochhar et al,[16] developers deemed method as the preferred code level that fault localization techniques should provide. Thus, a tool that guides developers closer to bugs can play an important role in the debugging process.

4.5 | RQ4: Do study participants intend to use SFL in practice?

Most participants showed the intention to use Jaguar in the future. We assessed the tool's PU, PE, and BI, adapting the TAM to the SFL domain. The items of our TAM questionnaire are in Table 2.

First, we verified the *internal validity* of our questionnaire regarding the TAM items in producing valid results. We evaluated the reliability of our questionnaire using the *Cronbach's alpha*,[57] which assesses whether a set of items of a test is closely related. We obtained alpha values of 0.95 and 0.97 for items of PU and PE, respectively. These values are higher than the threshold of 0.8,[57] which indicates that our questionnaire is reliable.

We also measured the *factorial validity*, which aims to verify whether the items of PU and PE form distinct constructs.[58] Items from the same construct tend to be more correlated. A threshold to indicate correlated items is 0.7.[59] Table 11 shows the correlation values for the items of PU (U1 to U4) and PE (E1 to E4), where bold values are those above the threshold. The results show that PU is highly related to the items designed for this purpose, except for U4 (usefulness), which had a value slighter lower than the threshold. PE is also highly related to items E1 to E4.

TABLE 10 Statistical results for RQ3 – participants who found the faulty methods. Column "jsoup + XStream" has the percentage and number of participants who found the faulty methods in both programs. Columns "jsoup" and "XStream" detail these numbers for each project.

Technique	P-value	Effect size	jsoup + XStream	jsoup	XStream
Eclipse	–	–	8/26 (30.77%)	4/11 (36.36%)	4/15 (26.67%)
Jaguar	**0.006**	5.06 (M)	18/26 (69.23%)	9/15 (60.00%)	9/11 (81.82%)
JaguarL	**0.042**	10.0 (L)	8/12 (66.67%)	3/6 (50.00%)	5/6 (83.33%)
JaguarM	**0.016**	3.33 (M)	10/14 (71.43%)	6/9 (66.67%)	4/5 (80.00%)

TABLE 11 Factorial analysis showing the correlation values among the TAM constructs and TAM items.

	PE	PU
U1	0.28	**0.94**
U2	0.38	**0.84**
U3	0.24	**0.84**
U4	0.61	0.66
E5	**0.84**	0.42
E6	**0.96**	0.21
E7	**0.92**	0.30
E8	**0.77**	0.51

Abbreviations: PE, perceived ease of use; PU, perceived usefulness; TAM, Technology Acceptance Model.

To ensure that the responses regarding BI to use are accurate, we assessed the *correlation* between the three TAM constructs using the *Pearson's product correlation coefficient* (*r*).[60] The *r* values vary from −1 to 1, where −1 means a negative correlation, 0 means no correlation, and 1 means a positive correlation. The results are shown in Table 12, which indicates that both usefulness and ease of use are positively correlated with BI of use. The *p*-values for each comparison indicate that the correlations are statistically significant.

Thus, our questionnaire and the TAM constructs are reliable to be used for assessing the participants' answers about the Jaguar tool.

4.5.1 | Perceived usefulness

Most participants deemed the SFL tool useful. The aggregate rating scale for the PU items ranges from 4 to 28. The values assigned by the participants had a median of 22, an average of 20.65, and a mode of 26.

Figure 6 shows the answers for usefulness of each item. The number of participants that totally agree, agree, or somehow agree varied from 15 (57.7%) for U3 (effectiveness) to 21 (80.7%) for U4 (usefulness). The maximum disagreement level was 7 (26.9%) for U3. The U3 item is directly related to task success, which may explain why this item was worst rated.

4.5.2 | Perceived ease of use

The participants rated the SFL tool as easy to use. The answers for the aggregated items had a median of 24, an average of 22.46, and a mode of 24. The rating scale ranges between 4 and 28.

Figure 7 shows the answers for each PE item. Participants who assigned some level of positive agreement varied between 20 (76.9%) for E8 (task easiness) and 21 (80.8%) for E5 (learning), E6 (mental effort), and E7 (ease of use). Only three (11.5%) participants assigned some disagreement level for each PE item.

4.5.3 | Behavioral intention of use

Most participants showed interest in using the SFL tool in the future. Their answers for the BI items had a median of 13, an average of 11.5, and a mode of 14. The rating scale ranges between 2 and 14.

Figure 8 shows the answers regarding behavioral intention of use. A total of 18 (69.2%) and 19 (73.1%) participants assigned some positive agreement level for BI9 (intention of use) and B10 (prediction), respectively. These results indicate that they are willing to use an SFL tool for their debugging activities if such a tool is available.

The SFL tool received positive ratings even from those who did not find bugs since the average of positive agreements with the TAM questionnaire is always higher than the number of developers who successfully found the bugs. This fact indicates that the participants of our study are willing to use techniques to support debugging even when they were not successful on the first try. Thus, we believe that techniques able to highlight faulty program elements in most cases will be well-accepted for practical use. Despite the need for improvements, SFL tools should be widely disseminated to increase the likelihood of being adopted in real settings.

4.6 | RQ5: Does the level of code information impact on the fault localization performance?

The *level* of code information (suspicious methods or lines) did not present differences regarding effectiveness or efficiency. Table 13 shows the statistical results for RQ5 comparing JaguarL and JaguarM. The null hypotheses are that the use of JaguarL or JaguarM has the same effectiveness and efficiency. As in RQ1, we used Fisher's exact test to check the null hypothesis and odds ratio to measure the ES of effectiveness. For the efficiency evaluation, we used the Wilcoxon rank-sum test and Cliff's delta, as in RQ2.

TABLE 12 Correlation and *p*-values among TAM constructs.

	PU x BI	PE x BI	PU x PE
r	0.87	0.61	0.69
P-value (%)	9.22×10^{-7}	8.77×10^{-2}	9.59×10^{-3}

Abbreviations: BI, behavioral intention; PE, perceived ease of use; PU, perceived usefulness; TAM, Technology Acceptance Model.

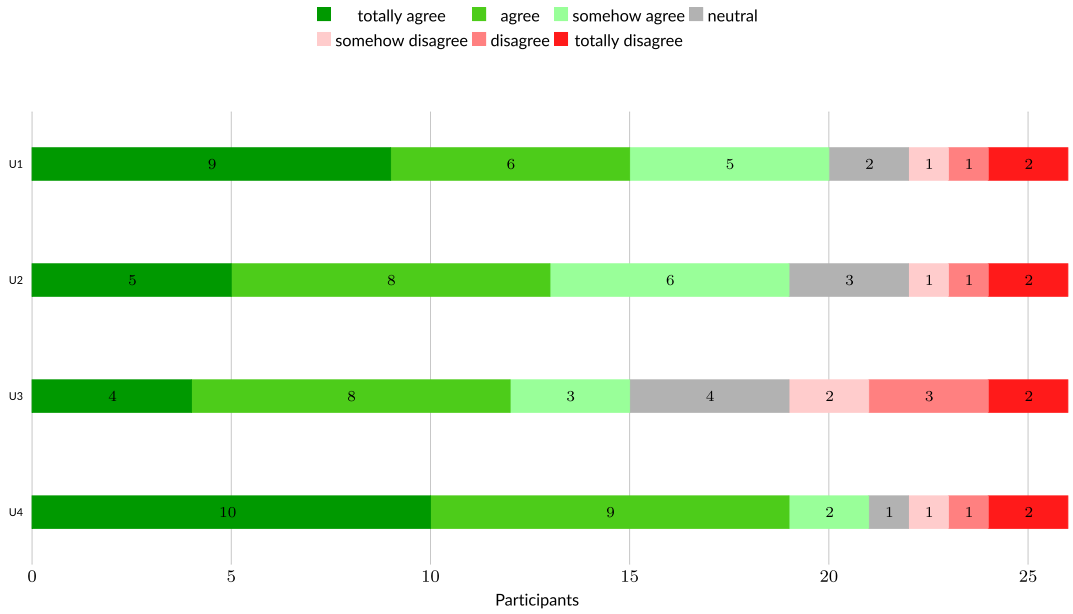


FIGURE 6 Usefulness results.

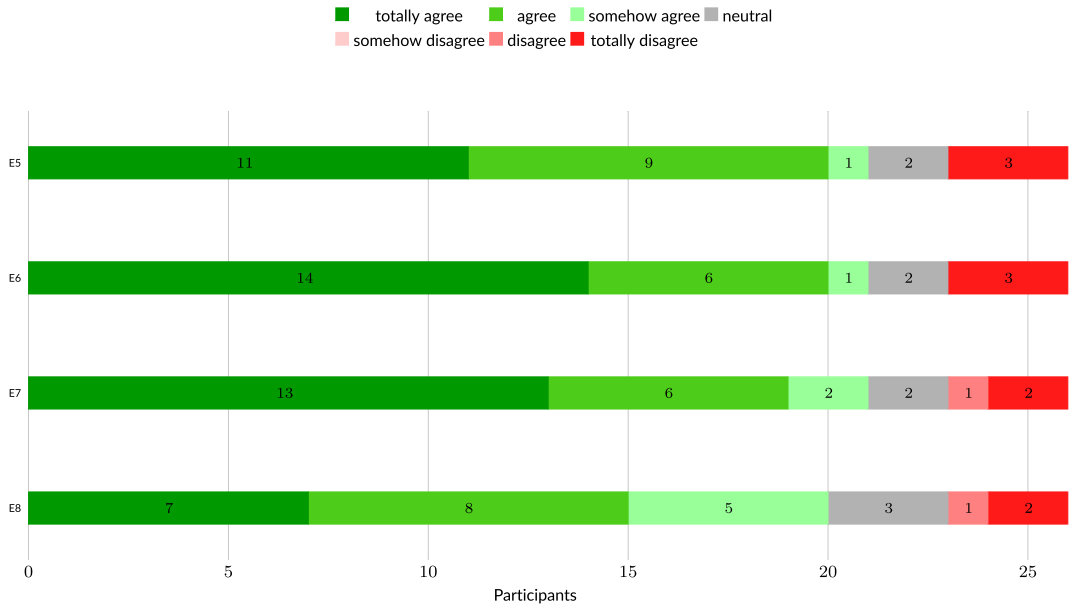


FIGURE 7 Ease of use results.

In both cases, the null hypothesis cannot be rejected. The *p*-values indicate that there were no differences when comparing JaguarL and JaguarM for both effectiveness and efficiency. The ESs are insignificant and small, respectively. Thus, SFL with information of method or lines were more effective than debugging without using SFL, but they presented similar performance between them.

4.7 | Programming experience

Table 14 shows the participants' years of experience in Java, professional programming, use of IDEs, automated testing, and general experience in programming. The *None* column shows those who had no experience, while the *0* column shows those who had less than one year of experience. The *10+* column shows those who had 10 or more years of experience. Few participants had a large experience. Regarding Java, only 3 of them

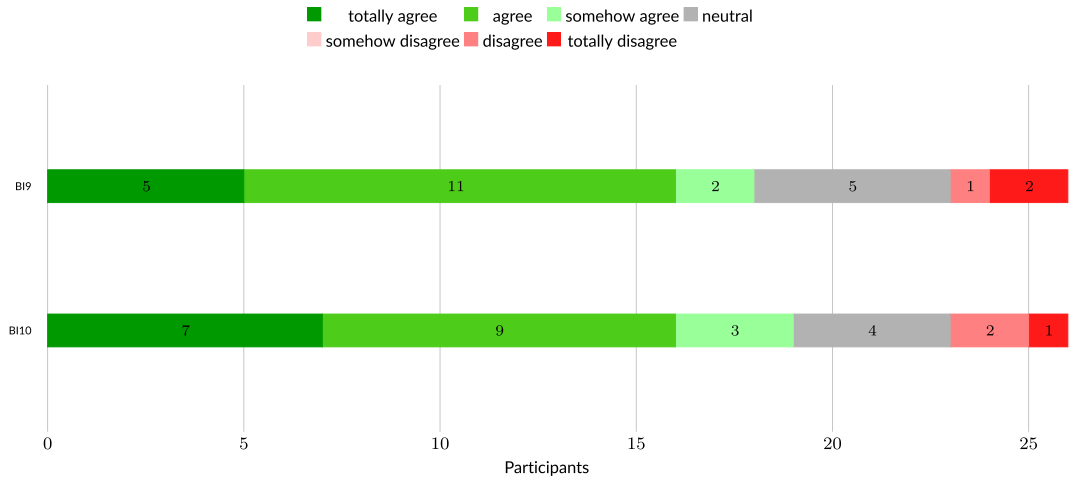


FIGURE 8 Behavioral intention of use results.

TABLE 13 Statistical results for RQ5—*p*-values and effect sizes for effectiveness and efficiency comparing JaguarL and JaguarM.

	Effectiveness		Efficiency	
H_0: JaguarL = JaguarM	*P*-value	Effect size	*P*-value	Effect size
	0.724	0.71 (I)	0.755	−0.14 (S)

TABLE 14 Participants' experience in years.

Experience	None	0	1	2	3	4	5	6	7	8	9	10+
Java	0	2	3	2	10	3	3	0	0	1	0	2
Professional	5	5	6	4	1	0	3	0	0	0	1	1
IDE	0	2	6	7	5	1	2	0	0	1	1	1
Testing	7	8	9	0	0	0	1	0	0	0	0	1
Programming	0	1	1	1	7	6	6	1	0	0	0	4

had more than 5 years of experience, while 16 had 3 to 5 years. Most participants had at least a minimum of experience with professional programming. Also, almost all participants had little or no experience in automated testing.

Most developers who found a bug using Jaguar had 2 or 3 years of experience in Java. Two of three developers with 8 or more years of experience in Java found bugs using SFL, and one of them did not find the other bug without using SFL. From those who found at least one bug, four had no previous experience with automated testing and seven had at most 1 year. Regarding professional experience, only one developer with no experience found a bug, three have less than 1 year, and five of them had between 1 year and 2 years as developers.

Figure 9 shows the distribution of participants' years of experience in Java, professional programming, IDE usage, and automated testing. The data are split into those who found or did not find the faults using Jaguar and Eclipse. These results indicate that less experienced developers performed better using Jaguar. For example, regarding experience in Java (Figure 9A), those who found bugs using Jaguar are more concentrated from 1 to 5 years of experience, while this concentration varies from 2 to 5 years for those who found using Eclipse. More experienced developers also were more effective using Jaguar. Figure 9A–C shows that there are fewer developers with large experience who did not find the bug using Jaguar compared with Eclipse.

It is important to note that the debugging tasks performed in this study used real programs, and the developers had no previous knowledge of the code. Moreover, we kept the faulty line and methods in the positions assigned by the Ochiai ranking metric. Thus, the lines were ranked in intermediate positions, as we reported in Section 3.4.

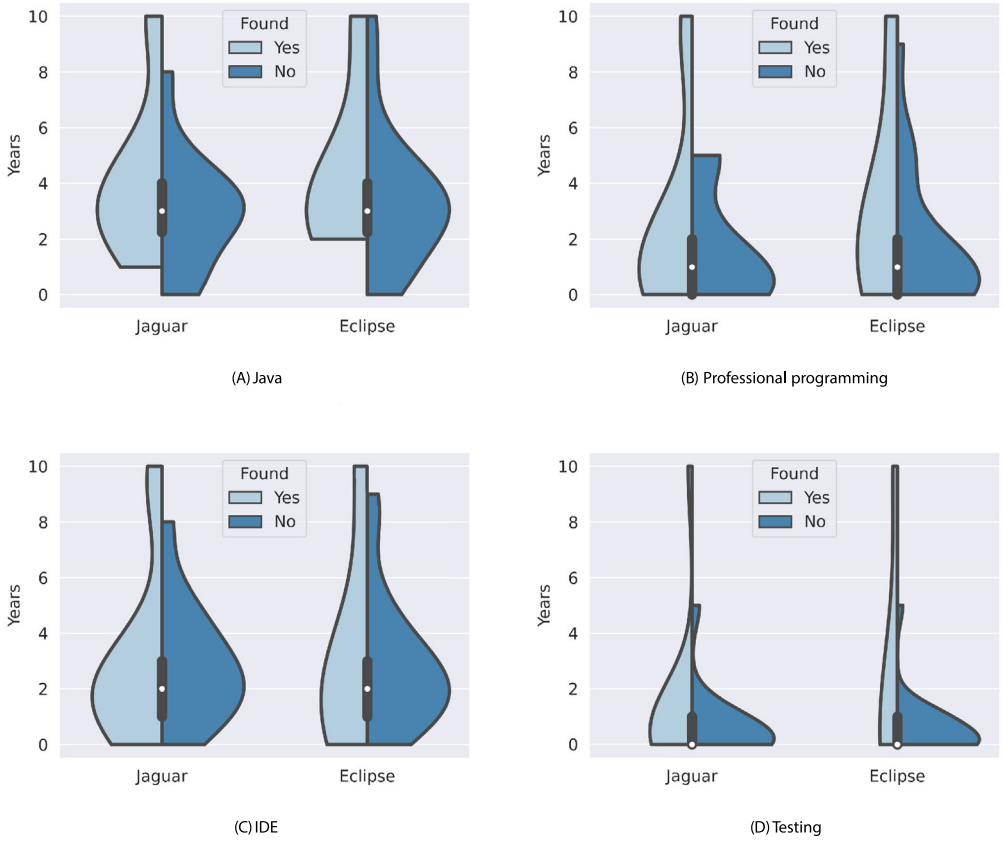


FIGURE 9 (A–D) Participants' years of experience.

4.8 | Participant comments and observations

In the post-task questionnaire, the participants answered several questions regarding the SFL tool and its resources and provided suggestions for improvement and their opinions about the experiment settings.

We performed an open card sort[19,20] to identify common characteristics of the participants' opinions on how SFL was (or was not) helpful during the debugging tasks in which Jaguar was used. We took the responses for question (1) "Describe how the tool helped you to find the bug. Otherwise, simply answer 'It did not help'.". We also analyzed the answers for the related question (2) "What information provided by the tool was more useful to perform the debugging task? Otherwise, simply answer 'None'.".

Two of the authors identified the possible groups of responses individually and then we discussed those groups to achieve a consensus. Our card sorting resulted in seven groups. All answers to question (1) are presented below. We also added those participants' answers for question (2) that brought complementary information regarding question (1); these additional answers appear after the → symbol. For each answer, we use symbols to indicate when the participant found the fault only in the Jaguar task (✓), only in the Eclipse task (●), in both tasks (✚), or when none fault was found (✗). The black and gray colors indicate whether the developers used Jaguar in jsoup or XStream, respectively.

G1: The tool helps to focus on the most likely faulty lines. The answers in this group indicate that participants perceived the SFL tool as useful for highlighting suspicious parts of the code, giving them hints about code excerpts that should be inspected in a reduced search space. Some participants commented that the tool helped them to perceive that something in the code was wrong, leading them to find the bug. Even some participants who did not find the bug also had this opinion, since by looking at their qualitative comments, they were positive regarding the Jaguar features (e.g., highlighting the suspicious code, suspiciousness scores, the fault list) that may be useful for debugging. There are two such comments right below (marked as ✗).

> ✚ *"It correctly indicated the point of the code that contained the fault."→ "I think that was the demarcation of the suspicious code excerpts. In the end, I sort of repaired the code by chance, since changing the conditional [operator] was an attempt since the original condition did not seem to make sense."*

✓ *"The fact that Jaguar marks line codes that supposedly may show some "oddity" served me as a great filter to search for [the bug] since I did not have any knowledge about the code."→ "All marked lines were of functions [methods] that at some point use the function that was in fact faulty."*
✓ *"The tool highlighted the [faulty] code line as very likely to contain the fault, which allowed me to find it."*
✓ *"Jaguar showed the code snippets where there was more chance of finding the fault."*
✗ *"Even not finding the bug, I think that Jaguar may be useful, showing code lines that should have the programmer's attention."*
✗ *"It pinpoints the code lines that are likely to be faulty and it helps us in searching for the fault without getting lost among so many lines of code."*

G2: The tool gives a start for the bug search. This group is similar to the G1 group but the participants' answers commented more specifically that the tool is useful to give them a starting point for debugging. Thus, SFL can serve as a way to corroborate with the bug understanding process, leading developers to specific code excerpts that should be focused.

✚ *"The highlighting of excerpts more likely to contain bugs helped a lot to start the search for the bug, giving a good starting base. The presentation mechanism [the tool's list] was excellent to control the debugging [task]."*
✓ *"It pointed out the first part of the faulty code in a red situation, from there I was able to understand the class that was presenting the problem and, together with the JUnit test, I found out the problem itself."→ "Pinpointing what most likely is the problem when starting to debug a code totally unknown was very useful. In the XStream's bug without the tool I got lost trying to understand all that could be behind the problem [fault] and I ran out of time. In the jsoup's bug, after running Jaguar and the JUnit tests I realized quickly where the bug could be and after a brief code reading I understood the necessary methods and what was going wrong."*
✓ *"Listing the fault probabilities." → "It gives a direction where to start by listing the highest chances of fault."*
✓ *"Basically, I read the code excerpt pointed out by the tool and at the first sign of something strange I changed [the code] and it worked."*
● *"I did not find the fault, but I believe Jaguar directed me to at least find out the problem."*
✗ *"I did not find the bug, but I was able to have a good notion from the indication made by Jaguar."*

G3: The tool helps to identify faulty class and method. Most answers in this group are from participants that did not find the bug itself. However, they perceived the tool as helpful to figure out the class and/or method where the bug was, which is a first step to find and fix it.

✓ *"The tool positioned me at first in the method where the error [bug] was, although the [faulty] line was not the most suspect, the most suspicious lines were in this method."*
✗ *"I did not exactly find the bug but Jaguar helped me to figure out the class and the method that would contain the fault."*
✗ *"Jaguar indicated the method that contained the fault, it is easier to find, but for lack of time, I could not fix it."*
✗ *"It helped to prioritize methods that could contain the fault."*

G4: The list of suspiciousness is helpful. This group contains answers from those participants that pointed out more generically the SFL list itself as helpful for debugging. There are also comments about the suspiciousness scores assigned to the code and the use of colors to distinguish them.

✚ *"The fact of placing hierarchically the code with a higher probability of error is a very interesting factor of the tool. In addition, the contextualization of the painted code was crucial in the analysis of the code."*
✓ *"Indicating the probability of finding the bug in the code [score] greatly facilitates the localization of the code excerpt containing the bug."*
✗ *"Simply by the list of methods it provides! It really helps a lot to find the problem. It goes far beyond that being a guide to the tester."*

G5: The tool is similar to stack trace. One participant deemed Jaguar as useful as the Eclipse stack trace console, deeming Jaguar more useful for highlighting the suspicious code using colors in the Eclipse's editor area.

✚ *"Jaguar helped to locate the problem in the code as much as the error's stack trace in JUnit [Eclipse]." → "It only helped to focus on the general area of the code where the problem was, the "hottest" areas indicated the method to look at."*

G6: The tool does not help to find bugs. Five participants who did not find any bug deemed that Jaguar was not helpful in their tasks. They wrote exactly the same answer as suggested in the question for those who did not deemed Jaguar as useful.

✗ *"It did not help." (3 answers)*

✗ *"It did not help." (2 answers)*

G7: The developer does not understand how to use the tool. One of the participants did not understand how to use Jaguar, which indicates that (1) the use of suspiciousness lists may not be intuitive for everybody or (2) the lack of knowledge of the faulty program may impair its use.

✚ *"I did not know the tool and I got lost in using it."*

We also asked the participants what other information they would like to see in the tool and if they had any other comments or observations about the tool. Several participants suggested improvements for Jaguar. Some of them would like to see why the tool deemed a line as suspicious. Another frequent suggestion was including information about suspicious variables. The developers also want to see information about the relationship between suspicious elements and failed tests. Another feature they want to see is the suggestion of potential fixes, which can be obtained by integrating SFL and program repair techniques.[61,62] Regarding other Jaguar resources, few participants used the text search and the slider filters provided by the tool. Also, some participants complained about the limited time to perform the tasks, while others commented on the difficulty of debugging without previous knowledge of the code. Indeed, such factors may have impaired their performance.

The comments and observations made by the participants indicate that SFL is useful to guide developers to inspect suspicious code excerpts. It is also useful as a starting point to understand the bug causes. However, the participants pointed out that they want to see more information to help them understand what causes a fault. Thus, the use of SFL with other existing development resources may help even more developers to find bugs.

4.9 | Developer behavior

Using the log files, we observed how the participants interacted with Jaguar and without it in the experiment. In what follows, we describe the main findings regarding their behavior during the debugging tasks.

4.9.1 | Use of SFL during the tasks

Most participants used Jaguar throughout the task. To evaluate this issue, we divided the task time performed by each participant into three parts to assess how long they interacted with the Jaguar during their tasks. Of the 26 participants, 24 used Jaguar in the first part, 20 of them in the second part, and 21 used it in the third part. Thus, **most participants did not give up on using SFL**, even when the fault was not found. This result is opposite to that shown by Parnin and Orso,[10] in which developers would give up on using SFL in the presence of false positives. None of the participants of our study reported having given up on Jaguar, even those who answered that the tool did not help them. Only two participants who did not find the bug in the SFL task gave up on the task in a short time, both after spending around 15 minutes.

Contrary to the results of Xie et al,[12] most developers in our experiment inspected the SFL list before inspecting the source code. As the programs used in Xie et al[12] are small, some of them with a single file, it seems reasonable to look at the code before (first scan) using an SFL tool. However, the programs we used are large, with hundreds of class files. Thus, such a first scan pattern may not hold when debugging more complex programs. The behavior pattern of the developers in our experiment was somehow similar to that observed in the study of Xia et al.[13] In general, the developers (1) clicked on some suspicious lines/methods in the SFL lists, (2) navigated through the source code, and then (3) looked at the test cases. This process was repeated by most of them until finding the bug or the time ran out. Several participants changed the code and ran the tests aiming at fixing the faults.

Most developers used both SFL and JUnit together during the SFL task. We observed that some of them switched between SFL and JUnit several times. Such a fact suggests that relating suspicious elements to their respective failed tests, as suggested by some participants (and also pointed out by Parnin and Orso[10]), may help to understand possible causes of a fault. Few developers used the Eclipse's debugging resources (i.e., JDT Debug) during the tasks. Only five participants used breakpoints and the Eclipse Debugger in the SFL task and six of them used in the other task.

4.9.2 | Inspection of the SFL lists

In general, developers inspected only the most suspicious elements in the SFL lists. These most suspicious ones were inspected several times during the tasks. The number of distinct elements inspected by the participants has a median value of 6. 16 participants (61.5%) inspected at most 7 distinct elements, and 23 (88.5%) inspected at most 20 distinct ones. Regarding the suspiciousness scores, 13 (50%) participants inspected at

most elements within the two highest scores. Twenty-one (80.1%) participants inspected at most elements with six different scores. The median for distinct scores is 3.

The developers did not inspect the elements in the order proposed by the SFL lists. Twenty-one (80.1%) of them started the task by inspecting the top 1 element, but only 3 (11.5%) followed the list in order until the fourth element. Most developers inspected the high-ranked elements several times. The average for the most inspected element by participant is 9.61, with a median of 6. We did not observe differences in the navigation between the SFL lists of lines and methods.

The results presented in the TAM analysis suggest that developers are willing to spend time using an SFL tool for their debugging activities. However, lists with a great number of program elements are useless, since they inspect at most 7 distinct ones in most cases and no more than 20 elements for almost all participants. Our results show that most participants interact with Jaguar by clicking on the suspicious elements and then by navigating through the source code around them. Thus, the suspicious elements act as starting points to understand and reach the faults. Even inspecting a few suspicious elements, SFL helped more participants to find the bugs, to find the faulty methods, and to inspect the faulty lines. This fact suggests that SFL may be useful even when the faulty line is not ranked among the top ones.

Moreover, the order in which the elements are presented does not seem to be important. Thus, other ways to present the most suspicious elements can be explored, for example, presenting elements that are more closely related together, suspicious variables, small subsequences, and so on.

4.10 | Factors in favor of the practical use of SFL

Throughout our research questions, we identified some factors that suggest that SFL can be used in practice:

(1) SFL improved the effectiveness to locate faults, even when a fault is not ranked among the top ones. Moreover, even less experienced developers were able to find bugs using SFL. The efficiency was not significantly better for SFL; the time to complete the tasks using Jaguar was just slightly shorter.
(2) There is a correlation between clicking on a faulty line and finding its bug. Although the perfect bug understanding is not accomplished (as shown in previous studies[10,12,13]), reaching the faulty site helps to locate bugs.
(3) SFL led more participants to inspect the faulty line and also led developers to locate the faulty methods, even for those who did not find the bug. Thus, SFL can be useful to provide hints for a developer investigating possible causes of faulty behaviors.
(4) Participants deemed Jaguar as easy to use and useful. Moreover, they showed the intention to use it in the future, including those that did not find the bug. We believe that most developers in the software industry do not know about SFL techniques, and disseminating them is fundamental for its adoption. Also, researchers should propose new ways to report suspicious results to provide techniques more suitable for practical use.

4.11 | Factors against the practical use of SFL

By observing the navigation logs and from the participants' comments and suggestions, we identified factors that impair the adoption of SFL:

(1) Long lists of suspiciousness are useless and may confuse developers. Most developers did not inspect elements below the top 20. Thus, excessive information should be avoided by SFL tools.
(2) Developers missed information explaining why elements are deemed as suspicious. Also, several participants would like to see additional information beyond the SFL lists, such as variables and test cases related to suspicious elements, which means that they need more information to understand the bugs. Thus, these developers' observations suggest that strategies to add contextual information for SFL can be useful in practice.
(3) Most participants in our study have little or no experience with automated testing, which is not an uncommon fact in real settings. Indeed, several software companies do not make use of automated testing or make limited use of it. This fact limits the possibilities of adoption of SFL techniques in such scenarios.[63]

4.12 | Threats to validity

The internal threats to validity are the Jaguar tool and the SFL lists it provides. We opted to use single lists of faulty methods and lines that are similar to other Eclipse's views. Our objective in this study is not to evaluate the Jaguar tool by itself but a general SFL tool. However, the use of other visualization schemes may lead to different results.

Regarding the posttask questionnaire, the TAM questions are framed positively, which may lead participants to more positive responses. We followed the proposed structure of TAM using a 7-level Likert scale, including a neutral option, to give the participants more fine-grained possibilities for their responses.

A threat to construct validity is the time to perform the tasks. We set a time limit of 30 min for each task, which may impact the participants' performance, especially for less experienced developers. However, the time is the same for both tasks. A long period for the tasks can also impact the performance since some participants would get tired and lose concentration in the second activity.

None of the participants had previous knowledge about the programs used in our experiment. This scenario represents the case where a developer has to debug an unknown program. Even a developer who had worked on a project may lose the context of a code when s/he is away from such a project for a couple of months. However, a scenario in which developers have previous knowledge about the code may lead to different results. Regarding the criteria to participate in the study, we accepted participants with experience using Eclipse IDE, NetBeans IDE, and IntelliJ IDEA since they have similar testing and debugging tools. However, participants experienced in NetBeans IDE or IntelliJ IDEA may have different performance while using Eclipse IDE.

We used the HIA method to allocate the participants in real-time to the groups. Such an allocation procedure aimed at a twofold goal: groups with similar participants' profiles and bias avoidance by means of random perturbation. The method was successful in yielding groups with at least four participants and similar profiles; however, some participants dropped out of the experiment, while others were excluded due to incomplete data. To tackle this threat, we applied Fisher's exact test for independence. It showed that the difference among the groups is not significant regarding the faults and the order of the tasks.

Another threat to the construct validity is the position of the faults in the SFL lists. We opted for keeping the faulty lines and methods in the original positions where they were assigned by Ochiai. We also ran the faulty programs using Tarantula and Jaccard to avoid the influence of a ranking metric in the results. These metrics assigned the same positions for both faults.

Regarding the external threats, we chose two faults that are neither too easy nor too hard to find. Both faults were independent of knowledge about the program domain. The assessment of the time to locate the faults considers both faults together. Although the faults may have different difficulty levels, the results do not indicate such a difference. However, using a different number of faults, faults with distinct characteristics, or faults from other programs may lead to different results. To avoid biases, we also used the Latin square design to distribute the faults across the techniques and the order in which they are performed.

Nearly half of the participants found the bugs using SFL. We believe that factors such as experience level, reduced time for the debugging tasks, and lack of knowledge about the code led to this modest success rate. Moreover, the statistical analysis comparing JaguarL and JaguarM (RQ5) has nearly half participants (14 using JaguarM and 12 using JaguarL). Although we have used the Wilcoxon rank-sum test, which is suitable for small samples, more participants could lead to more distinguishable results.

A final threat to the external validity is related to the participants of the study. They are undergraduate and graduate students, which may not generalize to a representative sample of the population of developers. Most of them have some professional experience, and few of them have large professional experience. We used the HIA method to keep the distribution of developers balanced through the experimental groups.

5 | CONCLUSIONS AND FUTURE WORK

This paper presented our user study that aimed to understand how developers use SFL and to what extent SFL can be useful. We conducted a user study with 26 developers who performed two debugging tasks using two faults from two real programs. We investigated several factors, some of them were already investigated (e.g., effectiveness) and others factors not previously assessed to the best of our knowledge (e.g., guidance to lead developers towards faulty excerpts, user acceptance).

We found that SFL improved the developers' fault localization ability, leading more developers to locate faults. However, the time spent to find the bugs was not reduced using SFL.

We also found that there is a relationship between inspecting a faulty line and locating its bug. Although it seems obvious, we show that this relationship occurs with statistical significance. Thus, finding the code region where faults occur may be the first step to locate bugs. In our study, SFL leads more participants to inspect the faulty lines and to locate the faulty methods, which may increase their chances of finding bugs. Indeed, the developers' comments on the usefulness of Jaguar for debugging indicate that most of them deemed the tool as useful to reach the faulty code excerpts. Conversely, some participants did not perceive Jaguar as useful since they did not find the bugs using it.

Most participants showed intention to use SFL in the future, deeming the Jaguar SFL tool useful and easy to use. Even those that did not find bugs using SFL showed this intention. Also, they would like to see more information related to the suspicious elements, such as variables, test cases, and explanations about faulty behaviors. Such additional information may help them to understand the context where a bug occurs.

Regarding the developers' behavior using SFL, our user study shows that developers do not inspect large amounts of suspicious code in the SFL lists. They focused more on the first picks, as already observed in previous work.[10,13] In our study, most developers inspected up to 20 different program entities. This fact shows the importance of proposing new ways to reduce the amount of code to be inspected to locate bugs. By

doing this, we can develop techniques more suitable for practical use. However, even when the faults are not ranked among the first positions, SFL serves as a guide to reach faulty code regions and to find the bugs when the suspicious code that is close to the faulty line is well-ranked.

Also, our study corroborates the findings of previous studies.[10,13] Developers do not identify a bug immediately after clicking on the faulty entity. In most cases, they inspect the faulty entity more than once to identify the fault. Also, they do not follow the SFL lists in their exact order, but the first picks are often investigated.

Debugging is a strictly intellectual activity, which often requires a lot of mental effort. Thus, SFL techniques should not try to replace the developer's role. Instead, they should provide more precise and useful information to support developers to locate bugs. It is important to develop tools that are easy to use and that can be used in real software projects. Integration with IDEs can also facilitate the adoption of those tools. Despite the need for improvements in SFL, we believe that it is necessary to create and publicize SFL tools in the software communities to attract the interest of practitioners.

As future work, it would be interesting to evaluate the use of information of suspicious variables. Also of interest would be to perform experiments in which developers have previous knowledge of the code to see how useful SFL is in this scenario. Exploratory studies and case studies should also be conducted, assessing the use of SFL in software companies.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Qianqian Wang and Professor Alessandro Orso for their collaboration in our user study design. We also would like to thank Professors Rafael Oliveira, Marcelo D'Amorim, Ivanilton Polato, Eduardo Guerra, and Auri Vincenzi for helping us in inviting participants for this study. We thank the anonymous referees for their valuable input for this paper. The authors acknowledge the support granted by the São Paulo Research Foundation (FAPESP), processes 13/24992-2, 14/23030-5, and 14/50937-1.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in SFL user study setup (https://github.com/saeg/user-study-sfl).

ORCID

Higor Amario de Souza https://orcid.org/0000-0003-4233-5987
Marcos Lordello Chaim https://orcid.org/0000-0001-7157-5141

ENDNOTES

* We use *lines* and *statements* interchangeably.

† We use *developers* and *participants* as synonyms in this work.

‡ eclipse.org

§ github.com/saeg/jaguar

¶ github.com/andyjko/whyline

eclipse.org/swt

‖ tqrg.github.io/pangolin

** developer.mozilla.org/en-US/docs/Rhino

†† www.eclemma.org

‡‡ percederberg.net/games/tetris

§§ gzoltar.com

¶¶ x-stream.github.io/repository.html

commons.apache.org/proper/commons-math

‖‖ github.com/jhy/jsoup

*** github.com/saeg/jaguar

††† eclemma.org/jacoco

‡‡‡ testng.org

§§§ netbeans.org

¶¶¶ jetbrains.com/idea

github.com/jhy/jsoup

‖‖‖ x-stream.github.io

**** Whenever we refer to Jaguar or SFL, we are considering both JaguarL and JaguarM.

REFERENCES

1. Myers GJ. *The Art of Software Testing*: John Wiley & Sons, Inc.; 1979.
2. Hailpern B, Santhanam P. Software debugging, testing, and verification. *IBM Syst J*. 2002;41(1):4-12.
3. Jones JA, Harrold MJ, Stasko J. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, ICSE'02 IEEE; 2002:467-477.
4. Zeller A. Isolating cause-effect chains from computer programs. In: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, FSE '02 ACM; 2002:1-10.
5. Abreu R, Zoeteweij P, van Gemund AJC. On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques, MUTATION'07 IEEE; 2007:89-98.
6. Laghari G, Murgia A, Demeyer S. Fine-tuning spectrum based fault localisation with frequent method item sets. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE'16 IEEE/ACM; 2016:274-285.
7. B. Le T-D, Lo D, Le Goues C, Grunske L. A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 2016 International Symposium on Software Testing and Analysis, ISSTA'16 ACM; 2016:177-188.
8. Beszédes A, Horváth F, Di Penta M, Gyimóthy T. Leveraging contextual information from function call chains to improve fault localization. In: 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020 IEEE; 2020:468-479.
9. Zou D, Liang J, Xiong Y, Ernst MD, Zhang L. An empirical study of fault localization families and their combinations. *IEEE Trans Softw Eng*. 2021;47(2):332-347.
10. Parnin C, Orso A. Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA'11 ACM; 2011:199-209.
11. Gouveia C, Campos J, Abreu R. Using html5 visualizations in software fault localization. In: 1st IEEE Working Conference on Software Visualization, VISSOFT'13 IEEE; 2013:1-10.
12. Xie X, Liu Z, Song S, Chen Z, Xuan J, Xu B. Revisit of automatic debugging via human focus-tracking analysis. In: Proceedings of the 38th International Conference on Software Engineering, ICSE'16 ACM/IEEE; 2016:808-819.
13. Xia X, Bao L, Lo D, Li S. "automated debugging considered harmful" considered harmful: a user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In: Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution, ICSME'16 IEEE; 2016:267-278.
14. Horváth F, Beszédes A, Vancsics B, Balogh G, Vidács L, Gyimóthy T. Experiments with interactive fault localization using simulated and real users. In: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution, ICSME'20 IEEE; 2020:290-300.
15. Shull FJ, Carver JC, Vegas S, Juristo N. The role of replications in empirical software engineering. *Empir Softw Eng*. 2008;13(1):211-218.
16. Kochhar PS, Xia X, Lo D, Li S. Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA'16 ACM; 2016:165-176.
17. Ribeiro HL, de Souza HA, de Araujo RPA, Chaim ML, Kon F. Jaguar: a spectrum-based fault localization tool for real-world software. In: Proceedings of the 11th International Conference on Software Testing, Verification and Validation, ICST'18 IEEE; 2018:404-409.
18. Davis FD. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quart*. 1989;13(3):319-340.
19. Spencer D. *Card Sorting*: *Designing Usable Categories*: Rosenfeld Media; 2009.
20. Lo D, Nagappan N, Zimmermann T. How practitioners perceive the relevance of software engineering research. In: Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'15 ACM; 2015:415-425.
21. Gilmore JT. Tx-o direct input utility system. Lincoln Laboratory, MIT; 1957.
22. Weiser M. Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, ICSE'81 IEEE; 1981:439-449.
23. Weiser M. Programmers use slices when debugging. *Commun ACM*. 1982;25(7):446-452.
24. Francel MA, Rugaber S. The value of slicing while debugging. *Sci Comput Programm*. 2001;40(2–3):151-169.
25. Kusumoto S, Nishimatsu A, Nishie K, Inoue K. Experimental evaluation of program slicing for fault localization. *Empir Softw Eng*. 2002;7(1):49-76.
26. Ko AJ, Myers BA. Finding causes of program output with the Java Whyline. In: Proceedings of the Sigchi Conference on Human Factors in Computing Systems, CHI'09 ACM; 2009:1569-1578.
27. Fry ZP, Weimer W. A human study of fault localization accuracy. In: Proceedings of the 26th IEEE International Conference on Software Maintenance, ICSM '11 IEEE; 2011:1-10.
28. Youngs EA. Human errors in programming. *Int J Man-Machine Studies*. 1974;6(3):361-376.
29. Gould JD. Some psychological evidence on how people debug computer programs. *Int J Man-Machine Studies*. 1975;7(2):151-182.
30. Vessey I. Expertise in debugging computer programs: a process analysis. *Int J Man-Machine Studies*. 1985;23(5):459-494.
31. Gilmore DJ. Models of debugging. *Acta Psychol*. 1991;78(1–3):151-172.
32. Ko AJ, Myers BA. Debugging reinvented. In: Proceedings of the 30th International Conference on Software Engineering, ICSE'08 ACM/IEEE; 2008:301-310.
33. Wang Q, Parnin C, Orso A. Evaluating the usefulness of IR-based fault localization techniques. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA'15 ACM; 2015:1-11.
34. Böhme M, Soremekun EO, Chattopadhyay S, Ugherughe E, Zeller A. Where is the bug and how is it fixed? An experiment with practitioners. In: Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'17 ACM; 2017:117-128.
35. Castro B, Perez A, Abreu R. Pangolin: an SFL-based toolset for feature localization. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE'19 IEEE/ACM; 2019:1130-1133.
36. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empir Softw Eng*. 2005;10:405-435.
37. Campos J, Riboira A, Perez A, Abreu R. GZoltar: an eclipse plug-in for testing and debugging. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE'12 IEEE/ACM; 2012:378-381.

38. Li X, Zhu S, d'Amorim M, Orso A. Enlightened debugging. In: Proceedings of the 40th International Conference on Software Engineering, ICSE'18 ACM; 2018:82-92.
39. Ribeiro HL. On the use of control- and data-flow in fault localization. *Master thesis*: School of Arts, Sciences and Humanities, University of São Paulo, São Paulo, Brazil; 2016.
40. Creswell JW. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*: Sage publications; 2013.
41. Wainer J. Experiment in collaborative systems. In: Pimentel M, Fuks H, eds. *Collaborative Systems*: Elsevier-Campus-SBC; 2011:405-432. In Portuguese.
42. Campbell DT, Stanley JC. Experimental and quasi-experimental designs for research. In: Gage NL, ed. *Handbook of Research on Teaching*: Rand McNally; 1963:171-246.
43. Cochran WG, Cox GM. *Experimental Designs*: John Wiley & Sons; 1957.
44. Feigenspan J, Kästner C, Liebig J, Apel S, Hanenberg S. Measuring programming experience. In: Proceedings of the 2012 IEEE 20th International Conference on Program Comprehension, ICPC'12 IEEE; 2012:73-82.
45. Davis FD, Venkatesh V. A critical assessment of potential measurement biases in the technology acceptance model: three experiments. *Int J Human-Comput Studies*. 1996;45(1):19-45.
46. Brooke J. SUS a quick and dirty usability scale. *Usabil Eval Indust*. 1996;189(194):4-7.
47. Fossaluza V, de Souza Lauretto M, de Bragança CAP, Stern JM. Combining optimization and randomization approaches for the design of clinical trials. In: Polpo A, Louzada F, Rifo RLL, Stern MJ, Lauretto M, eds. *Interdisciplinary Bayesian Statistics: EBEB 2014*: Springer International Publishing; 2015: 173-184.
48. Aitchison J. Principal component analysis of compositional data. *Biometrika*. 1983;70:57-65.
49. Fleiss JL. Measuring nominal scale agreement among many raters. *Psycholog Bull*. 1971;76(5):378-382.
50. Nielsen J. *How Many Test Users in a Usability Study?* Neilsen Norman Group; 2012. nngroup.com/articles/how-many-test-users. Accessed: 2015-11-03.
51. Mehta CR, Patel NR. ALGORITHM 643: FEXACT: a FORTRAN subroutine for fisher's exact test on unordered r×c contingency tables. *ACM Trans Math Softw*. 1986;12(2):154-161.
52. Fisher RA. The logic of inductive inference. *J R Stat Soc*. 1935;98(1):39-82.
53. Chen H, Cohen P, Chen S. How big is a big odds ratio? Interpreting the magnitudes of odds ratios in epidemiological studies. *Commun Stat – Simul Comput*. 2010;39(4):860-864.
54. Wilcoxon F. Individual comparisons by ranking methods. *Biometr Bull*. 1945;1(6):80-83.
55. Cliff N. Dominance statistics: ordinal analyses to answer ordinal questions. *Psycholog Bull*. 1993;114(3):494-509.
56. Macbeth G, Razumiejczyk E, Ledesma RD. Cliff's delta calculator: a non-parametric effect size program for two groups of observations. *Universitas Psycholog*. 2011;10:545-555.
57. Carmines EG, Zeller RA. *Reliability and Validity Assessment*, Vol. 17: Sage publications; 1979.
58. Laitenberger O, Dreyer HM. Evaluating the usefulness and the ease of use of a web-based inspection data collection tool. In: Proceedings of the 5th International Software Metrics Symposium, Metrics 1998 IEEE; 1998:122-132.
59. Jae-On Kim CWM. *Introduction to Factor Analysis: What It Is and How To Do It*: Sage publications; 1978.
60. Cohen J. *Statistical Power Analysis for the Behavioral Sciences*: Academic Press; 1977.
61. Le Goues C, Nguyen T, Forrest S, Weimer W. GenProg: a generic method for automatic software repair. *IEEE Trans Softw Eng*. 2012;38(1):54-72.
62. Nguyen HDT, Qi D, Roychoudhury A, Chandra S. SemFix: program repair via semantic analysis. In: Proceedings of the 35th International Conference on Software Engineering, ICSE'13 IEEE/ACM; 2013:772-781.
63. Abreu R. The bumpy road of taking automated debugging to industry. arXiv:221201237; 2022.

How to cite this article: Amario de Souza H, de Souza Lauretto M, Kon F, Lordello Chaim M. Understanding the use of spectrum-based fault localization. *J Softw Evol Proc*. 2024;36(6):e2622. doi:10.1002/smr.2622