RESEARCH ARTICLE

Generated Tests in the Context of Maintenance Tasks: A Series of Empirical Studies

WESLEY B. R. HERCULANO, EVERTON L. G. ALVES, AND MELINA MONGIOVI
Department of Systems and Computing, Federal University of Campina Grande, Campina Grande 58429-900, Brazil
Corresponding author: Wesley B. R. Herculano (wesleybrenno@copin.ufcg.edu.br)
This work was supported in part by the National Council for Scientific and Technological Development (CNPq), Brazil, under Grant 429250/2018-5.

ABSTRACT
Maintenance tasks are often time-consuming and hard to manage. To cope with that, developers often use failing test cases to guide their maintenance efforts. Therefore, working with good test cases is essential to the success of maintenance. Automatically generated tests can save time and lead to higher code coverage. However, they often include test smells and are hard to read. Moreover, it is not clear whether generated tests can be effective when guiding maintenance tasks, nor if developers fully accept them. This paper presents a series of empirical studies that evaluates whether automatically generated tests can support developers when maintaining code. First, we ran an empirical study with 20 real developers to compare how they perform maintenance tasks with automatically generated (Evosuite or Randoop) and manually-written tests. Then, we applied a survey with 82 developers that assessed developers’ perceptions of using refactored Randoop tests. Finally, a third empirical study with 24 developers focused on evaluating the impact of refactored Randoop tests on the performance of maintenance. Our results and statistical analyses show that automatically generated tests can be a great help for identifying faults during maintenance. Developers were more accurate at identifying maintenance tasks when using Evosuite tests and equally effective to create bug fixes (manually-written, Evosuite, and Randoop). Developers preferred refactored Randoop tests to the original ones. However, the refactorings applied did not improve their performance on fault detection. On the other hand, developers were more effective to fix the faults using refactored Randoop tests.

INDEX TERMS
Generated tests, maintenance, randoop, evosuite, test smells, refactoring.

I. INTRODUCTION
A software must be predictable and consistent, offering no surprise to the user. In this context, testing activities are important to ensure the quality of the software under development, to assess whether the program works according to its specification, and to reveal faults in advance with fewer efforts [1].

Software tests can be found in four levels: unit testing, integration testing, system testing, and acceptance testing [2]. Unit testing is the test of individual code units, or groups of related units [3]. A good unit test suite should assist developers to detect/fix the faults and be easily updated due to requirements or code changes [4].

Unit testing plays an important role during maintenance tasks [5]. It often works as a safety net to avoid fault introduction when updating a code, and as a great help for bug identification and fixing, [6]. However, building a good unit test suite is both difficult and time-consuming [7].

To cope with this issue, test-generation tools have been proposed. These tools can use different strategies for creating tests from scratch (e.g., genetic algorithms, search-based algorithms, mutation-based assertion generation, feedback-directed random generation) [8], [9], [10], [11]. From this wide range, we can highlight two of the most well-known test generation tools: Evosuite [11] and Randoop [12]. Both have...
been used as a baseline, and/or won awards, in the SBST Java Unit Testing Tool Contest [13], [14], [15].

Generated unit tests can save time and lead to higher coverage levels [11]. However, those tests are often not close to realistic scenarios and are less readable [16]. Figure 1(b) and (c) illustrates tests generated by Randoop and EvoSuite, respectively. In both tests, we can find some issues, such as verbose code (Figure 1(b) lines 2 and 4), assertions that are not easy to read (Figure 1(b) lines 6 and 7), the lack of documentation, and non-descriptive names for variables and test methods (Figure 1(b) lines 1 to 5, and Figure 1(c) lines 1 and 2). These issues make it hard for developers to understand the test code, which may prevent them from using the tests to locate/fix faults and/or maintain them.

Software maintenance is known to be costly and hard to manage. It represents up to 60% of a project’s budget [17], [18]. Previous work has evaluated automatically generated tests concerning code coverage, faults identification, and time-consumption [10], [11], [12]. However, to the best of our knowledge, there are only a few studies designed for evaluating whether automatically generated tests are maintainable and can properly support developers in software maintenance tasks.

Shamshiri et al. [19] ran an empirical study in an academic scenario where students faced maintenance tasks with the help of a failing test case. The failing test could be manually written or generated by EvoSuite. They were asked to identify and fix the cause of the failure, which could be related to implementation or a specific unit test fault. In this study, the students were more efficient at maintenance tasks when using manually written failing tests but equally effective with manually written and EvoSuite generated tests.

In our work, we perform a series of empirical studies that evaluate how developers deal with maintenance tasks using automatically generated tests. First, we ran a study with 20 developers from different companies and ask them to identify and fix the cause of a test failure. The fault could be either in implementation or test code. We used real test failures produced by developers whilst performing implementation tasks, and compared manually written tests to tests generated automatically by EvoSuite and Randoop. For this study, we reused most of the design and artifacts from Shamshiri et al.’s work [19], but we applied a more realistic scenario (real developers) and introduced an extra test generation tool (Randoop). The initial results of this study were first presented in a conference paper [20].

From this study we yielded the following main results:

- Developers were more accurate at identifying maintenance tasks when using EvoSuite tests, while they were equally accurate when using manually written and Randoop tests.
- Developers were similarly effective at producing bug fixes using the three strategies (manually written, EvoSuite, and Randoop).
- Developers were similarly efficient at executing maintenance tasks using the three strategies (manually written, EvoSuite, and Randoop).
- Developers found generated tests hard to read, specially Randoop’s. The EvoSuite test case structure was more appreciated but requires improvements.

These results reflect that, although easier to read, a manual test may not help to localize and/or understand code faults. In this sense, generated tests might be a good option. Moreover, there is a need for improvements in automatically generated tests when used for maintenance purposes, especially Randoop tests, which performed worse in our first study.

Test smells may compromise test code comprehension, readability, and maintenance [21], [22]. Generated test cases can include a number of test smells, such as non-descriptive names, assertion roulette, duplicate assert, eager test, lazy test, and magic number test [23]. All of those can often be found in Randoop tests (Figure 1(b)). This observation, together with the results of our first study, motivated us to perform a whole new study to evaluate the impact of automatic refactoring edits (Extract Method and Rename Method) on the quality of Randoop tests. For that, we conducted a survey with 82 software professionals, and ask their perceptions when comparing original Randoop tests to their automatically refactored versions.

From this survey we found that:

- Although we cannot say that developers preferred automatically generated test names over manually written ones, the automatic renaming of Randoop tests was well-received.
- Developers preferred refactored Randoop tests over original ones. However, in order to fully accepted them, they indicated the need for extra refactoring, such as variables renaming, and extract method.

The results from this survey motivated us to assess the practical impact of the use of refactored tests on the performance of maintenance tasks. For this, we performed a third study where we replicated the first one, but focusing our analysis on different versions of Randoop tests (original and refactored ones). We can summarize the results of this investigation as follows:

- The refactoring edits did not improve, nor worsen, the performance of developers determining the root cause of the faults.
- Developers were more effective in performing proper fixes when guided by refactored Randoop tests.
- When guided by refactored Randoop tests, developers took less time fixing the faults, but the same was not observed for fault identification.
- Refactored Randoop tests contributed to a better understanding of the class under testing (CUT) and facilitated the identification of maintenance activities and code fixes. However, developers felt more confident about their test fixes, when using original Randoop tests.
Thus, the present work extends our conference paper [20] in several ways:

- Extending the discussion of the results of the first study by including different analyses (e.g., Section IV-B5) and different aspects of discussions (e.g., Section IV-B5).
- Reporting a new survey with 82 developers that accessed their perspectives on different versions of Randoop test cases (Section V).
- Reporting a new empirical study on the use of refactored generated tests to guide maintenance tasks (Section VI).

The remainder of this paper is organized as follows: In the next section, we discuss a motivational example evidencing the problem dealt with in this paper. In Section III, we present a background about important topics. Section IV presents our first empirical study on how developers deal with maintenance tasks using automatically generated and manually written tests. The application of refactoring strategies to reduce test smells in generated tests is first accessed by a survey, presented in Section V. A third empirical study on how developers deal with maintenance tasks using different versions of Randoop tests is described in Section VI. Threats of validity and related works are discussed in Sections VIII and IX, respectively. Finally, in Section X we present our concluding remarks and future works.

II. MOTIVATIONAL EXAMPLE

Here, we present code snippets from the ComparatorChain class and three test cases (manually written, generated by Evosuite and generated by Randoop). The ComparatorChain class includes a fault. The for loop in the compare method iterates over all elements except the last one due to a wrong stop condition. Figure 2 illustrates a possible codefix for this fault. The test cases presented in Figures 1(a), 1(b), and 1(c) (manually written, Randoop, and Evosuite test) fail when we run them to test the faulty code of the ComparatorChain class.

The manually written test adds a new comparator in the chain object, from the ComparatorChain class, and tests it using some asserts. The Randoop test creates empty comparators and checks their sizes. The Evosuite test creates an empty comparator and expects a NullPointerException. Although the test cases exercise the same method, they have different testing purposes. All of them fail due to faulty code.

As discussed before, developers can save time by automatically generating test cases using a tool. However, a question that remains is whether automatically generated tests make maintenance tasks harder. We asked seven developers to find the fault in the code (Figure 2). Four of them used the manually written test case (Figure 1(a)) but one did not find the fault. Three developers used the generated test (Figure 1(c)) and all of them found the fault. This example may suggest the need for further evaluation in this matter. In this work, we perform a series of empirical studies to evaluate how developers deal with maintenance tasks using variations of automatically generated tests.

III. BACKGROUND

In this section, we discuss important topics related to our work.
A. TEST GENERATION TOOLS

Automatic test generation tools have gained notoriety since they can be a great alternative to reduce the costs of creating sound test cases. Two of the most well-known test generation tools are Randoop\(^2\) [12] and Evosuite\(^3\) [24].

1) RANDOOP

Randoop [12] is a tool that implements feedback-directed random test generation for object-oriented Java programs. Randoop’s generation process builds sequences of method calls to exercise the system under testing. It focuses on generating tests that check code elements that could lead to basic contract violations. For instance, a test case that verifies whether a transitive property (e.g., o1.equals(o2) && o2.equals(o3) → o1.equals(o3)) remains valid after a sequence of method calls. Following construction, it executes the testing sequences to produce results that are used for generating other tests. Figure 5 shows an example of Randoop test cases.

Randoop can be used (i) to find faults, and (ii) to create regression tests that reflect the current behavior of a given program. The Randoop project is still very active. New versions of the tool have addressed complex issues such as the generation of invalid calls to static members, and flaky tests. Several works have used Randoop and its suites in different scenarios (e.g., [25], [26], [27], [28], [29], [30]). In the empirical studies reported in this work, we used Randoop version 4.1.2.

2) EVOSUITE

 EvoSuite is a search-based automatic tool for generating JUnit test suites. It generates tests by adding assertions that summarize the system’s current behavior and enables the detection of possible behavior changes. It applies evolutionary algorithms and searches operators such as selection, mutation, and crossover to evolve the test suite. This evolution is guided by a fitness function based on coverage criteria. Figure 1 (c) shows an example of an Evosuite test case.

 EvoSuite can be used as a command line tool or as a plug-in for popular IDEs (e.g., Eclipse). Moreover, EvoSuite has been used on several industrial projects, finding potential bugs (e.g., [10], [11]). Several new and improved versions of this tool have also been released in the past years. In the empirical studies reported in this work, we used EvoSuite version 1.0.6.

B. TEST SMELLS

Similar to implementation code, unit tests can also be affected by poor design and programming practices (i.e., smells). The importance of having a well-designed test code was first discussed by Beck [31]. The term test smell was later defined and cataloged by Van Deursen et al. [32]. Test smells resemble code smells. They are anti-patterns from well-established testing practices and guidelines on how test cases should be implemented, organized, and interact with each other. The presence of test smells may negatively impact the quality of the system [33], hampering the quality and maintenance of a test suite [34], in addition to impairing its performance (e.g., flaky tests [35], [36]). Van Deursen et al. [32] cataloged 11 types of test smells. In addition, other works have defined more than 80 smells [21].

Grano et al. [22] investigated the diffuseness of test smells in automatically generated test suites. They found that Randoop and Evosuite tend to generate a high quantity of two specific test smells (Assertion Roulette and Eager Test). Moreover, Anonymous Test is a test smell type present in all tests generated by Randoop, caused by the use of stub test names (e.g., test1, test2).

In the context of our work, we focus on the three test smells described below:

- **Anonymous Test**: when a test has a meaningless and unclear method name, not expressing the purpose of the test [37] (e.g., Figure 1).
- **Assertion Roulette**: when a test has multiple assertions without explanation messages, making it difficult to read, understand, and maintain the test, and to identify the cause of a failing [32] (e.g., Figure 1 (b)).
- **Eager Test**: when a test checks several methods of the class to be tested, making it difficult to understand the test target and goal [32].

C. REFACTORING TEST CASES

Refactoring is the controlled process of modifying a program to improve its code structure without changing its external behavior [38]. The refactoring activity is known to remove code smells and improve code quality aspects such as readability, confine source code complexity, decrease coupling, and increase cohesion [39]. Fowler proposes a catalog of different refactoring types [38]. Among the most popular refactorings, we list Rename Method and Extract Method. The first is used when the name of a method does not explain what it does, therefore it should be properly renamed. The Extract Method can be used to reduce complexity and improve the code readability. For that, one should move a fragment of code from an existing method into a new method with a representative name.

Refactorings can also be used in test code. Van Deursen et al. [32] define test refactorings as transformations of test code that: (1) do not add or remove test cases, and (2) make test code more understandable/readable and/or maintainable. Therefore, refactoring can be used to remove test smells. For instance, by renaming a test case with a more representative name (See Figure 10), a tester may better understand its purpose (removing the Anonymous Test smell). By performing a series of Extract Methods based on test assertions one may remove, or reduce, the Assertion Roulette and Eager Test smells, and have less trouble locating/fixing a bug [34] (See Figure 11).
IV. A STUDY ON THE USE OF GENERATED TESTS TO GUIDE MAINTENANCE TASKS

Software maintenance involves a series of tasks (e.g., inspecting, modifying, and updating code artifacts) and it is known as complex and costly [17], [18]. To reduce the risks involved in those tasks, developers often use test cases to guide them to identify and correct undesired modifications. However, it is unclear whether generated tests can be as effective and useful as manually written ones.

To evaluate how developers deal with maintenance tasks using automatically generated and manually written tests, we ran an empirical study. A more detailed description of this study and complete results can be found in our previous work [20].

A. STUDY DESIGN

Shamshiri et al. [19] performed a study with students that investigated how they deal with maintenance tasks using EvoSuite and manually written tests. In our study, we extended Shamshiri et al.’s work [19] by performing a similar investigation. However, instead of students, we ran our study with 20 developers and we dealt with a more comprehensive set of testing generation strategies: manual, Evosuite tool, and Randoop tool. Although Shamshiri et al. did not consider Randoop in their investigation, we decided to include it in our study because other works have attested its practical benefits (e.g., [28], [29]) and have been used as baseline in the SBST Java Unit Testing Tool Contest [13], [14], [15]. Thus, our study may complement Shamshiri et al.’s work by introducing a more realistic scenario (developers) and by comparing manually written testing with two types of generated tests (Randoop and Evosuite).

This study was designed for investigating the performance and perception of developers on performing maintenance tasks guided by failing unit test cases created using different strategies (manually written or automatically generated). Our goal is to understand the outcomes and difficulties that developers have when facing a test failure and need to identify the fault and fix it, which can be related to either code or test malfunction. Moreover, we would like to understand whether generated test cases, using different strategies, are a good fit in this scenario.

To guide our investigation, we defined the following research questions:

1. **RQ1**: Do generated tests influence the effectiveness of developers in determining the source of a problem?
2. **RQ2**: Are generated tests effective to help developers find proper fixes?
3. **RQ3**: Does it take longer to execute maintenance tasks when using generated tests?
4. **RQ4**: What is the developers’ perception of using generated tests when performing maintenance tasks?

These questions were adapted from Shamshiri et al.’s work [19].
TABLE 1. Information about the selected classes. Branch coverage values refer to the test suite used to evaluate the codefixes performed by the participants.

| Class Name              | LOC Methods | Branch Cov. | Test suite LOC Manual Test LOC Evosuite Test LOC Randoop Test LOC |
|-------------------------|-------------|-------------|---------------------------------------------------------------|
| FixedOrderComparator    | 98          | 10          | 77.5%                                                       |
| ListPopulation           | 97          | 13          | 77.3%                                                       |
| ComparatorChain          | 131         | 18          | 64.3%                                                       |

Next, we ran the generated suites against the faulty versions and randomly selected a failing one as a representative for the study. Therefore, we selected three test cases per subject program (one Manual, one Evosuite, and one Randoop). For the manual test, we randomly selected a test from the original suite that fails in the faulty version.

As for the faults related to the generated tests, we followed a similar procedure. Figure 4 presents an example of a manually written faulty test case (emptyArray should be null to trigger the expected exception) and its possible fix. Finally, Figure 5 shows a faulty Randoop test (two unknown objects were not properly compared) and its fix.

One may argue that the size of objects is small. However, since we want to compare results to the ones reported by Shamshiri et al., we reused most of their artifacts (classes and faults). They were selected from open-source projects and reflect real-world faults. Shamshiri et al. argue that those artifacts were selected due to their manageable size, availability, and amenability for research purposes. Those reasons are even more important in our study because our participants are real developers with limited time. Since we asked the participants to carefully inspect all code (implementation and tests), a more complex configuration (extra classes and test suites with more than a single test) would make our study unfeasible.

We also believe the injected faults and scenario (single failing test case) emulate real maintenance tasks. When identifying and fixing faults (source code or test code), developers often focus on a single class and/or small edits. For instance, 47% of the bug fixes from Defects4J require two or fewer lines of code [41].
3) STUDY PROcedure
The procedure of our study goes as follows. Prior to the sections, each participant answered a questionnaire about her background. Moreover, the first author ran a brief tutorial about the study and tasks. Each participant was asked to perform three maintenance tasks. Each maintenance task referred to a single fault to be identified and fixed. The fault could be either related to implementation or test code. Moreover, since each participant was asked to perform three maintenance tasks, we vary the type of tests to be used (manually written, Evosuite, and Randoop). Both the task type (codefix or testfix), the order, and the received test (manually written, Evosuite, or Randoop) were randomly assigned for counterbalancing. To support each task, we provided a pre-configured environment that included an Eclipse IDE and the artifacts to perform the assigned tasks: a project with the class implementation (faulty or not) and a failing test (faulty or not).

Our study worked with the following maintenance tasks:
- \(<\text{codefix, manual} >\): Faulty implementation and a correct manually written failing test.
- \(<\text{codefix, evosuite} >\): Faulty implementation and a correct Evosuite failing test.
- \(<\text{codefix, randoop} >\): Faulty implementation and a correct Randoop failing test.
- \(<\text{testfix, manual} >\): Correct implementation and a faulty manually written failing test.
- \(<\text{testfix, evosuite} >\): Correct implementation and a faulty Evosuite failing test.
- \(<\text{testfix, randoop} >\): Correct implementation and a faulty Randoop failing test.

To avoid learning effects, no participant was assigned to the same fix type or class across sessions. For instance, a participant that was first assigned to \(<\text{codefix, manual} >\) on FixedOrderComparator class, then should be assigned to different tasks and classes in the second and third assignments (e.g., \(<\text{codefix, evosuite} >\) on ListPopulation class, and \(<\text{testfix, randoop} >\) on ComparatorChain class).

The participants were asked to perform each maintenance task with a time limit of 60 minutes and to verbalize and answer a form when identifying the problem. When participants made wrong decisions (e.g., identifying a testfix task when in fact should be a codefix one), we revealed the correct answer so they could complete the assignment. Only after that, the participants proceeded to fix the faults. All sections were performed in person, conducted by the first author, and video-recorded for later analysis. Finally, at the end of the section, each participant was asked to answer a questionnaire in which we asked questions related to the maintenance tasks and possible challenges. All artifacts used in our study, including implementations, faults, tests, and questionnaires, are available on our website\(^5\).

It is important to highlight that we did not impose any protocol for detecting/fixing the faults. When performing the maintenance, a participant could either inspect the implementation code, the test code, or both. Moreover, as either the implementation or test code was faulty, each method included a Javadoc specification. We provided this information to help participants figure the expected behavior of the methods and, therefore, to avoid misleading conclusions based on the code alone.

After the sessions, we evaluated the output artifacts of each participant. To decide whether a codefix or testfix solution was correct, we ran the process in Figure 6. A codefix was classified as correct if it did not break any additional test from the original test suite and satisfied a manual inspection. In this manual inspection, we compared the participant’s output class with our golden solution and javadoc documentation. For test-fixing tasks, we created reference solutions based on the original test purpose. Then, the first author ran the tests and inspected the code to classify the fix.

B. STUDY RESULTS
Here, we discuss the collected data and its implications for each research question.

1) RQ1: DO GENERATED TESTS INFLUENCE THE EFFECTIVENESS OF DEVELOPERS IN DETERMINING THE SOURCE OF A PROBLEM?
To answer this question, we observed the participants’ effectiveness at identifying whether the faults were in the implementation or test code. Table 2 summarizes the results of this investigation. The first three lines refer to results considering all classes, while the remaining present the results per class.

As we can see, the tasks helped by Evosuite tests presented very high rates considering all (95%), codefix (100%), and testfix tasks (90%). Those values were greater than the manual (65%, 50%, and 80%) and Randoop ones (50%, 30%, and 70%).

To our surprise, Evosuite tests performed better than the other two strategies in 11 of the 12 analyzed scenarios. This fact may be evidence that this type of generated tests could be a good fit for identifying maintenance tasks, even better than manual tests. Developers found those tests easy to follow and helpful when identifying bugs. On the other hand, Randoop tests performed quite poorly. Since Randoop tests focus on contract checking, they tend to be less readable. Therefore, we believe that participants had a hard time understanding the tests and consequently ended up wrongly blaming them (effectiveness for codefix was only 30%).

To measure statistical significance when comparing treatments, we first ran the Shapiro–Wilks [42] normality test that did not confirm a normal distribution. Therefore, we applied the non-parametric Fisher’s exact test [43] for comparisons of correctness. With a confidence of 95% (p-value < 0.05), we were able to rank the strategies considering the results for all, and codefix and testfix tasks, individually. The last two columns of Table 2 summarize this analysis. A > B indicates

\(^5\)https://github.com/WesleyBreno/generated-tests-in-the-context-of-maintenance-tasks-a-series-of-empirical-studies/tree/main/study-with-generated-tests
that strategy A performed better than B, while $A = B$ says that they are statistically equivalent. The Evosuite strategy performed better considering all tasks together and only codefixes, but it was similar to the others when considering testfixes. On the other hand, tasks guided by Manual tests performed similarly to Randoop ones in all three analyses.

Thus, we can answer RQ1 by saying that, in general, developers were more accurate at identifying maintenance tasks when using Evosuite tests, while they were equally accurate when using manually written and Randoop tests. Those results go against Shamshiri et al.’s findings [19], in which they state there is no difference between using manual or generated tests. Our results show that not only there is a difference between those tasks, but also the generation tool seems to play an important role.

2) RQ2: ARE GENERATED TESTS EFFECTIVE IN HELPING TO FIND PROPER FIXES?
To answer this question, we followed the protocol defined at the end of Section IV-A3 and presented in Figure 6. Table 3 presents the collected results.

In general, we observed that participants were more accurate at identifying maintenance tasks when using Evosuite tests, while they were equally accurate when using manually written and Randoop tests. Those results go against Shamshiri et al.’s findings [19], in which they state there is no difference between using manual or generated tests. Our results show that not only there is a difference between those tasks, but also the generation tool seems to play an important role.

RQ1: Developers were more accurate at identifying maintenance tasks using Evosuite tests and equally accurate using manually written and Randoop tests.

In general, participants were similarly effective at producing correct fixes using the three strategies (manual, Evosuite, and Randoop). We performed a statistical analysis similar to the one described in Section IV-B1, which reinforced these conclusions (Table 3). The only exception was codefix for Randoop where only one participant was able to fix the fault using a failing Randoop test. Again, to our surprise, manually written tests did not perform better than generated ones. Thus, we can answer RQ2 by stating that, in general, generated tests are as effective as manually written ones to help to find proper fixes, regardless of the tool.

RQ2: Generated tests are as effective as manually written ones to help to find proper fixes.

3) RQ3: DOES IT TAKE LONGER TO EXECUTE MAINTENANCE TASKS WHEN USING GENERATED TESTS?
For this analysis, we evaluated the time participants took to decide whether the maintenance tasks were codefix or testfix. We also measured the time taken by the participants to perform the fixes.

On average, participants took 21 minutes to identify the faults. To better analyze the data, we first ran the Shapiro–Wilk test which indicated a not normal distribution. Therefore, we used the non-parametric Mann-Whitney U test [24] for comparing duration values. This test could not find any significant difference among the strategies when considering all classes neither for codefix nor testfix. However, when observing the classes individually, we found differences. For instance, participants took, in general, a long time to identify codefixes in the FixedOrderComparator class using manually written tests. On the other hand, Evosuite’s tests performed worse for testfixes in the ListPopulation class.
TABLE 2. Comparison of correct decisions given Manual, Evosuite, or Randoop tests. The last two columns refer to the statistical tests.

| Task Type | Class     | Manual (M) | Evosuite (E) | Randoop (R) | p-values M = E | E = R | Final Ranking |
|-----------|-----------|------------|--------------|-------------|----------------|------|--------------|
| all       | all       | 13/20 (65%) | 19/20 (95%)  | 10/20 (50%) | 0.021          | 0.5231 | 0.0018       | E > M = R |
|           | codefix   | 5/10 (50%)  | 10/10 (100%) | 3/10 (30%)  | 0.0162         | 0.6499 | 0.0015       | E > M = R |
|           | testfix   | 8/10 (80%)  | 9/10 (90%)   | 7/10 (70%)  | 0.5            | 0.291  | 1            | E = M = R  |
|           | Fixed     | 1/5 (20%)   | 7/7 (100%)   | 6/8 (75%)   | 0.0101         | 0.1026 | 0.4667       | E > M = R |
|           | codefix   | 1/3 (33%)   | 3/3 (100%)   | 2/4 (50%)   | 0.4            | 1      | 0.4286       | E = M = R  |
|           | testfix   | 0/2 (0%)    | 4/4 (100%)   | 4/4 (100%)  | 0.0667         | 0.0667 | 1            | E = M = R  |
|           | List      | 6/7 (85%)   | 6/6 (100%)   | 2/7 (28%)   | 1              | 0.1026 | 0.021        | M > E = R  |
|           | codefix   | 2/3 (66%)   | 4/4 (100%)   | 1/4 (25%)   | 0.4286         | 0.4857 | 0.1429       | E = M = R  |
|           | testfix   | 4/4 (100%)  | 2/2 (100%)   | 1/3 (33%)   | 1              | 0.1429 | 0.4          | E = M = R  |
|           | Comparator| 6/8 (75%)   | 6/7 (85%)    | 2/5 (40%)   | 1              | 0.2929 | 0.2222       | E = M = R  |
|           | codefix   | 2/4 (50%)   | 3/3 (100%)   | 0/2 (0%)    | 0.4286         | 0.4667 | 0.1          | E = M = R  |
|           | testfix   | 4/4 (100%)  | 3/4 (75%)    | 2/3 (66%)   | 1              | 0.4286 | 1            | E = M = R  |

On average, participants took 8 minutes to fix the faults. Again, the data does not follow a normal distribution and we used the Mann-Whitney U test for comparison. The test also did not find any significant difference among the strategies in general. Differences were found when observing the classes individually. For instance, participants took, in general, a long time to identify codefixes in the ListPopulation class using Evosuite tests, but Randoop tests were very effective when correcting testfixes in the same class.

Thus, we can answer RQ3 and state that, in general, we cannot say that it takes longer to execute maintenance tasks using generated tests, regardless of the used tool. However, the class under maintenance may impact the results. Again, this goes against Shamshiri et al.’s findings [19], which say that developers are more efficient at maintenance tasks using manually written tests. Our study did not find pieces of evidence in this sense.

RQ3: We cannot say that it takes longer to execute maintenance tasks when using automatically generated tests.

4) RQ4: WHAT IS THE DEVELOPERS’ PERCEPTION OF USING GENERATED TESTS WHEN PERFORMING MAINTENANCE TASKS?

To answer RQ4, we went to the questionnaire each participant answered after the sessions. Figures 7 summarize the responses for testfix and codefix tasks. In general, participants found the tasks clear (Question 1) and had enough time to finish them (Question 2). Participants found it easier to identify the fault type (Question 3) using Evosuite tests for codefixes, but not for testfixes. On the other hand, Randoop
and manual tests responses were quite similar. These results agree with the analysis and conclusions of Section IV-B1.

Question 4 asked the participants’ perceptions about the activity of fixing the bugs. As we can see, the majority of participants had a similar opinion about manual and Evo-suite tasks for codefixes and testfixes. This goes along with the actual success outcome of the participants (Table 3). However, they found it easier to fix bugs using Randoop tests. Although easier to fix according to the participants’ opinion, the results using Randoop tests were statistically similar (codefix) and worst (testfix), when compared to the other two strategies.

Participants reported higher confidence in the correctness (Question 5) and quality (Question 8) of their test fixing tasks when using manually written tests, which was expected since generated tests tend to be less readable. However, responses were quite similar for correctness when considering codefix tasks and comparing Randoop and Evo-suite.

Regarding understanding the class under test (Question 6), for codefix tasks, participants found manually written tests more helpful, followed by Evo-suite and Randoop, respectively. For testfixes, Manual and Evo-suite tests were considered better help than Randoop tests. Possible reasoning for Randoop’s poor evaluation is that its test cases focus on basic contract checking, which might not reflect direct documentation of the intended behavior of the program.

Participants found manually written tests easier to understand when used in both codefixes and testfixes, followed by Evo-suite’s and Randoop’s tests (Question 7). This suggests that participants still find generated tests not ideal to read. Moreover, it reinforces the trend that Randoop tests are hard to inspect due to test smells. For instance, a participant stated the following: “I had to go back and forth to the code to understand the test’s behavior. The test did not have a good name nor its variables, which made it hard to follow”. In the same sense, a different participant stated: “The test was full of magic numbers and names that were not related to the class”.

Finally, we found that participants had lower confidence in the quality of their fixes when using manually written tests to fix a code, but higher confidence when the fault was in the test. Since manually written tests were found easier to understand, it was not a surprise that developers were confident about their testfixes. However, they were not so sure about their code fixes. This might reflect that, although easier to read, a manual test often does not help to localize and/or understand code faults. In this sense, generated tests might be a good option. Since they use systematic approaches for test generation, this might guide developers to better understand
the code and find its weak spots. Finally, the trend was again confirmed when Evosuite tests were better evaluated than Randoop’s.

**RQ4:** Participants found it easier to identify the fault type using Evosuite tests for code fixes; easier to fix bugs using Randoop tests; easier to understand the class under test using manually written for code fix, and manual and Evosuite for test fixes; had higher confidence in the correctness and quality of their test fixes tasks when using manually written tests; and found manually written tests easier to understand in both tasks types.

### 5) ANALYSIS BY ROLES

In an additional analysis, we reanalyzed the collected data by participants’ roles. Regarding RQ1, we found no difference, between the participant’s role in the effectiveness of developers in determining the source of the problem. All of them were more effective when using the Evosuite tests. On the other hand, the Test analysts were more effective in fixing the bug using manual tests, while software developers and Software engineers were more effective using Evosuite tests (RQ2). We also analyzed the results from RQ3 and found that Test analysts, System analysts and software engineers took less time to identify the fault using Randoop tests, while Software developers were more efficient using Evosuite tests. Software developers, Software engineers, and Test analysts took less time to fix the bug using the Evosuite tests, while the system analysts were more efficient using the Randoop tests. Nevertheless, due to the small number of developers per role, these results have no statistical relevance.

### 6) DIVERGING RESULTS

The results of this investigation were, in some aspects, surprising and different from Shamshiri et al.’s [19]. For instance, we found differences in the developers’ efficiency in identifying bugs. Moreover, developers were less confident about their actions when guided by manual tests. Moreover, Evosuite tests were found as a great help in the maintenance tasks and without imposing more working time. Finally, Randoop tests, although as efficient as the other strategies, did not perform well on the developers’ perception, which found them hard to follow.

We see two possible reasons for the diverging results: i) different profiles of participants, and ii) the impact of extra artifacts. Developers are likely to have faced similar tasks in their regular jobs, which might be the reason why they found generated tests useful for bug fixing, though harder to read. Students, on the other hand, are often less experienced, which might be the reason why they performed better with more readable tests (manually written). To assess whether the artifacts introduced in our study (class ComparatorChain and Randoop) influenced the results, we ran a side investigation considering a scenario identical to the original study, i.e., excluding the extra class/tool. The results remained. For instance, we found developers were as effective when performing maintenance tasks using manual or generated tests, while Shamshiri et al. found they performed better with manual tests. Moreover, they could not find differences in the effectiveness for identifying the type of the bug, we found that developers often performed better when using Evosuite tests. Therefore, we believe the diverging results are mostly due to the different profiles of participants (developers instead of students).

### V. A SURVEY TO EVALUATE DEVELOPERS’ PERSPECTIVES ON REFACTORED TESTS

Our previous study showed that developers did not accept well Randoop tests, mainly due to code quality issues related to test smells. To better understand this issue, we conducted a survey of 82 Brazilian software practitioners in order to compare their preferences on different versions of Randoop test cases. Knowing that refactoring edits can improve code quality, the goal of this survey is to have a deeper understanding of developers’ perspectives regarding test names and code, with and without refactoring. To guide this investigation, we defined the following research questions (RQ):

- **RQ5:** What is the developers’ perception concerning the names of Randoop test cases?
- **RQ6:** Do developers prefer the original Randoop tests or the refactored ones?

### A. METHODOLOGY

Our survey was designed as a three-section Google Form:6

1) **Context and Participant Background.** In this section, we introduced the goal and procedure of the survey and asked questions about the participant’s background and her previous experiences with software development and unit testing.

2) **Evaluating Test Naming.** In this section, the participant answered three questions: i) given a test case, indicate the level of agreement with a suggested name. The suggested name was generated using an automatic refactoring strategy (details in Section V-B); ii) given a test case, select the most appropriate/descriptive name from three options (original Randoop test name; automatic renaming; or a name chosen by an experienced developer); and iii) given a list of candidate names, select the one that seems fit better for a test that exercises a given code snippet. The list of candidate names includes the original Randoop test name, an automatically generated one, and a test name chosen by a developer. Figure 8 exemplifies questions from this section.

3) **Evaluating Split Tests.** In this section, we provided four versions of a given Randoop test case (A - original Randoop test; B - split Randoop test; C - renamed Randoop test; and D - split and renamed Randoop test). Versions B-D applied automatic refactorings for reducing the Assertion Roulette and Eager Test smells (details in Section V-B). We asked the participant to

---

6https://forms.gle/ESYSZLA5DMQ1PAmx8
answer two questions: i) what is the best option?; and ii) which option would you consider including in your test suite? Figure 9 exemplifies questions from this section.

It is important to highlight that, to complement the objective questions, the survey included open questions where participants justified their choices. Moreover, to avoid possible bias related to the used artifacts, we randomly associated participants with classes and test cases. Therefore, we have balanced responses.

### B. STUDY OBJECTS AND REFACTORING STRATEGIES

In the survey, we use examples of classes, tests, and test names. For that, we reused the classes (ListPopulation, FixedOrderComparator, ComparatorChain) and Randoop tests from our first empirical study (Section IV). Moreover, we generated new versions of the tests by refactoring them in order to fix the found test smells (Assertion Roulette, Eager Test, and/or Anonymous Test). The refactorings were applied using the following automatic strategies:

1) **TEST RENAMING**

This is an adaptation of the Rename Method refactoring designed to fix the Anonymous Test smell. Randoop tests receive standard test names (e.g., test23). Ermira et al. [44] propose an approach for generating names specifically for automatically generated tests. It uses coverage goals (method coverage, exception coverage, output coverage, and input coverage) for generating informative names. Coverage goals are a set of distinct objectives, such that a set of tests is considered adequate if, for each objective, there is at least one test that exercises it. While coverage goals may not describe a real test intent, they serve as reasonable approximations as they can describe what a test does. Ermira et al.’s approach was designed for EvoSuite test cases and its available on the tool’s website. As far as we know, we are the first to adapt and apply this approach for Randoop tests.

As EvoSuite captures coverage during its test generating process, we extracted its test generating approach and adapted it to make it work with Randoop tests. Figure 10 presents
an example of a Randoop test before and after its automatic renaming. To generate the names, the coverage goals for each test are ranked according to the following hierarchy: i) goals covered in the suite uniquely by this test; ii) exception coverage; iii) method coverage; iv) output coverage; and v) input coverage. Then, the N best-ranked goals are selected, and the number of selected goals can be configured and is responsible for controlling the size of the generated names. In our work, we applied its default value (two). Next, the selected goals were converted into text to be concatenated and composed as a name. Finally, to simplify the names, we used processing based on abstract text summary algorithms. The Randoop version of this renaming strategy is available on our website. 

2) SPLITTING TEST
Randoop tests are often long and include several asserts (see Figures 11). To avoid Assertion Roulette and Eager Test smells, a test can be divided based on their asserts. For that, we implemented a script that reused the Eclipse Refactoring API. It receives a given test case $t$ with $n$ assertions and performs a series of Extract Method refactorings, generating $n$ new test cases. Each Extract Method is triggered by a test assertion. For that, we analyze the test case Abstract Syntax Tree (AST) and extract to $n$ new methods each assertion statement along with its dependency statements. By using the Eclipse Refactoring Engine, we guarantee that the new set of test cases is free of compilation errors and preserve the original behavior. For instance, Figure 11 presents the original Randoop test case and its respective split suite.

We used a combination of the above-mentioned refactoring strategies (Test Renaming and Splitting Test) to automatically generate the refactored versions of the Randoop tests used in our study (B - split Randoop test; C - renamed Randoop test; and D - split and renamed Randoop test).

C. SURVEY RESULTS
Here, we discuss the participants and results of our survey by research question.

1) PARTICIPANTS DEMOGRAPHICS
For this study, we recruited volunteers by convenience (contacting developers from partner companies and universities), social networking platforms (e.g., LinkedIn), and snowball sampling [45], i.e., participants were asked to resend the survey invitation to others. All participants are developers with previous experience in Java/JUnit.
We received a total of 82 responses: six from graduate students; three from master students; two from professors; and 71 from active developers from various software companies (51 software engineers, seven software QA analysts, four software analysts, four team leaders, three data scientists, and two requirements analyst). Though this survey was sent to different mailing lists, all participants are from Brazil.

The first section of the survey helped us to understand the participants’ backgrounds (Figure 12). Most participants have at least three years of experience with Java and create unit tests regularly. Only 20% used test generating tools, mostly Randoop. Next, we answer and discuss RQ5 and RQ6.

2) RQ5: WHAT IS THE DEVELOPERS’ PERCEPTION CONCERNING THE NAMES OF RANDOOP TEST CASES?

To answer this question, we analyzed the responses of the second section. In the first question, participants were asked to set their agreement level to the suggested automatically generated test name. Most participants (51%) did not find the suggested name suitable for the presented Randoop tests, while 35% agreed with the suggested name. However, by using bootstrap [46], with 95% of confidence, we found no statistical difference to conclude that participants disagree or agree with the proposed names.

Some participants found the suggested names helpful. Here, we list some quotes collected from the open questions: “The name clearly represents the idea of the test”; “The test name already lets me know what will be tested”; and “The suggested name provides a great improvement when compared to other options”.

Although previous works have found significant improvements when renaming Evosuite tests with Ermira et al’s technique [44], our results show that although promising, the renaming strategy needs improvement when dealing with Randoop tests. Some participants found the suggested names not descriptive enough for the nature of the test code: “The test does a lot of internal things which make the name too long. The test should be split into different test cases, so the test names would be more descriptive”. This quote highlights issues related to Eager test smell and the need for smaller and more focused Randoop test cases. Moreover, some participants disagreed with part of the suggested name: “One should avoid reserved names such as List, String, Null as much as possible”.

The second question of this section asked participants to choose the most suitable name for a given Randoop test among three options: a name suggested by an invited developer, an automatically generated name (using the Test Renaming refactoring strategy), and the original Randoop name (test). As expected, no participant chose the “test” option. In general, participants preferred the names chosen by the invited developers (54%) over the generated ones (46%), except for the ComparatorChain class (47% and 53%). Again, with 95% of confidence, we can not conclude that developers preferred manually written or automatically generated test names. However, participants’ comments provide us with some reasoning and directions for improving the automatically generated names. A developer that preferred the generated option stated the chosen name was “very descriptive and reflected the purpose of the test”. Even when the name was long, one reported: “the name, despite being long, makes it clear what is being tested and expected results”. As for the ones who chose the developer’s naming, they found the names simpler and easier to understand: “By reading the names of the test I can easily understand it”, “The name reflects the test result and what caused it”.

The last question of the second section asked participants to select from three options (a name suggested by an invited developer, an automatically generated one, and the original Randoop test name) the best name for a test that would exercise a given code snippet. In general, most participants associated the given piece of code under test with the manually written names (61%), followed by the automatically generate names (39%), except for the ComparatorChain class (47% and 53%). No participant chose the “test” option. Again, our statistical analysis found no difference between the manually chosen and the automatically generated names.

We can then answer RQ5 stating that although there is a numerical advantage for manually chosen test names in our survey, we cannot say that developers preferred them over automatically generated ones. This may indicate that an automatic strategy can be a valuable option for renaming Randoop tests. However, the used renaming strategy was limited and requires further improvements for Randoop tests.

RQ5: We cannot say that developers preferred manually chosen test names over automatically generated ones, which indicates that an automatic strategy can be a valuable option for renaming Randoop tests.

3) RQ6: DO DEVELOPERS PREFER THE ORIGINAL RANDOOP TESTS OR THE REFACTORED ONES?

To answer this RQ, we analyzed the responses of the third section. In this section, given a class and four test code snippets for it (A - original Randoop test; B - split Randoop test; C - renamed Randoop test; D - split and renamed Randoop test), we asked participants: i) what is the most
difference in the developers’ preference for Code D (split-renamed tests) to all other options.

The participants’ comments on those questions may help us better understand their perspectives. Participants point out that the transformation applied in Code D improves readability and helps fault localization: "By breaking a test into different methods and renaming them according to their purposes, it makes it easier to read the code. Although the code is more extensive, the simplified parts make help understanding the whole thing”. “Although there is code duplication when a test fails, one can better understand the reasoning for the fail, because the names are descriptive and the code modularized”. When choosing option C (Randoop test automatically renamed), participants clearly favored less code duplication: “more readable code and without redundancies”. On the other hand, some participants pointed out that the refactored versions still needed improvement regarding readability: “Internal variables need to be more descriptive”, and “don’t use full class names instead of simple names (e.g., don’t use collections.comparators.FixedOrderComparator instead only FixeOrderComparator)”. Finally, one participant commented: “Although the names help to identify the purpose of the test, the body of the test is not easy to read. Maintaining these tests might be too costly, therefore, I would avoid their reuse”.

We can now answer RQ6 by stating that developers preferred the refactored version (split-renamed) that fixes most test smells. However, those versions still require improvements to be fully accepted. The most cited limitation refers to readability and reuse issues, such as the need for variables renaming and avoiding code duplication.

**RQ6**: Developers preferred refactored (split-renamed) Randoop tests over the other alternatives (original, only renamed, and only split tests).

4) **ANALYSIS BY ROLES**

When we analyzed the results by role, we found no significant differences regarding the code readability questions. Refactored codes are considered more readable and eligible to compose a test suite in all scenarios. On the other hand, there was a difference between the participant’s answers for each role regarding their preference for manually chosen test names and automatically generated ones for the Randoop tests. **Software engineers**, technical leaders, undergraduate, and **systems analysts** did not present an opinion about this topic. Notice that these results may be influenced by the sample sizes instead of the participant roles. Again, due to the small number of developers from each role, these results have no statistical relevance.

readable code snippet?; and ii) which of these codes snippets would you prefer to add to your test suite for the CUT?

Our results showed that most participants find split-renamed tests the best option (84%). Only 2% preferred the split option, 9% the renamed option, 5% found all options similar, and 0% the original Randoop test. Moreover, most participants preferred to reuse the split-renamed tests (78%). In both cases, with a 95% confidence, there is a significant

| FIGURE 11. Original and split Randoop tests for FixedOrderComparator class. |
|---|
| (a) Original Randoop test example for FixedOrderComparator class. |
| (b) Split Randoop tests from (a). |
FIGURE 12. Participants' background information.
VI. A STUDY ON THE USE OF REFAC To investigate the effectiveness of refactored Randoop tests when used for maintenance.

In this study, developers were asked to perform maintenance tasks guided by three types of failing Randoop tests: an original Randoop test, a renamed version, and a split-renamed version. The last two were created by applying the refactoring strategies presented in Section V-B for reducing test smells. Our goal was to understand whether the refactored tests can impact the use of Randoop tests in maintenance activities when they need to identify a fault and fix it.

To guide this investigation we adapted the research questions from our first study:

• RQ7: Do refactored Randoop tests improve developers’ effectiveness in determining the source of an issue? Here, we want to understand whether refactored generated tests, can better help developers to determine the source of a problem.

• RQ8: Do refactored Randoop tests improve developers’ effectiveness in performing proper fixes? We aim to investigate whether developers produce correct fixes when using refactored Randoop tests.

• RQ9: Do refactored Randoop tests improve the developers’ performance to execute maintenance tasks? Here, we investigate if refactored Randoop tests may speed up the process of identifying and performing maintenance tasks.

• RQ10: What is the developers’ perception about using refactored Randoop tests in maintenance tasks? Here, we collect developers’ opinions on the use of refactored generated tests.

A. PARTICIPANTS SELECTION AND DEMOGRAPHICS

For this study, we applied the same selection criteria as the first one (Section IV-A1). We recruited 24 volunteer active Brazilian developers (20 males and four females) from 10 different companies (12 from company 1, three from company 2, two from Company 3, and one for companies 4-10). It is important to highlight it is a different set of participants, i.e., none of them participated in our first study. They came from different companies (small and medium-sized) and perform the following roles: software engineering (17), team leader (3), data scientist (2), and test analyst (2). Similar to the first group, these developers work on projects from different natures (e.g., Mobile and Web Applications, Embedded Systems)

Prior to the study, they answered a background questionnaire (Figure 13). Half of the participants have experience with Java for at least three years. Although most find unit testing important for software development, they do not write or run unit tests very frequently. Finally, one participant has used test generation tools before.

B. STUDY OBJECTS

To run our study, we reused the implementations and faults from the first one (ListPopulation, FixedOrderComparator and ComparatorChain classes, and Randoop tests), replacing the manual and Evosuite tests by renaming Randoop tests, and split-renamed Randoop tests. To generate refactored Randoop tests we ran the strategies presented in Section V-B. While the original Randoop test and its renamed version referred to a single test, the split-renamed referred to a suite with $n$ tests, where $n$ is the number of asserts from the original Randoop test. Thus, in this study, we relied on the incorrect and faulty implementations of the ListPopulation, FixedOrderComparator and ComparatorChain classes, as well as the correct and faulty versions of the original Randoop, renamed, and split-renamed tests, for each class.

C. STUDY PROCEDURE

We replicated the procedure of the first study (Section IV) with a single difference: the maintenance tasks. By replacing manually and Evosuite generated tests by split and split-renamed Randoop suites, our study worked with the following maintenance tasks:

• < codefix, randoop >: Faulty implementation and a correct Randoop failing test.

• < codefix, split >: Faulty implementation and a correct split Randoop failing test.

• < codefix, split — renamed >: Faulty implementation and a split and renamed Randoop failing test.

• < testfix, randoop >: Correct implementation and a faulty Randoop failing test.

• < testfix, split >: Correct implementation and a faulty split Randoop failing test.

• < testfix, split — renamed >: Correct implementation and a faulty split and renamed Randoop failing test.

The participants are real developers with limited time. Therefore, to reduce the time required, we opted to not include a treatment related to only renamed tests. It is important to highlight that, this study was run between 2020-2021. Due to imposed COVID restrictions, we adapted our study to work in total remote environment. For that, we used tools
FIGURE 13. Study participants’ background information.
such as TeamViewer\textsuperscript{10} and AnyDesk\textsuperscript{11} to provide a controlled environment so the participants could perform the required tasks, and Microsoft Teams\textsuperscript{12} and Google Meet\textsuperscript{13} for remote calls. This configuration allowed the participants to work more flexible hours (the first authors still live-watched all sessions), which enabled us to recruit a relatively larger number of developers this time (24).

All artifacts used in our study, including implementations, faults, tests, and questionnaires, are available on our website.\textsuperscript{14}

1) FOLLOW-UP INTERVIEWS

To complement our analysis, after running the study, we selected a subset of participants and interviewed them for a better understanding of the found results. We divided the participants into three groups, according to the type of tests they performed better in the study. For each group, we invited three participants, however, only five were available for the interviews. During the interviews, we asked the following questions:

- What difficulties did you experience when deciding the maintenance task to be performed?
- Do you consider the provided test case name good and representative? Did it help you identify/fix the fault?
- To the ones who worked with split tests: Did you consider the non-failing tests when identifying/fixing the fault?
- Which of the three treatments (Randoop, renamed, split-renamed tests) would you consider the best option to assist the maintenance activities and why?

All interviews were conducted remotely through Google Meet or Microsoft Teams.

D. STUDY RESULTS

Here, we discuss the data analysis and results for each of our research questions, and the follow-up interviews.

1) RQ7: DO REFACTORED RANDOOP TESTS IMPROVE DEVELOPERS’ EFFECTIVENESS IN DETERMINING THE SOURCE OF AN ISSUE?

To answer this question, we compared the participants’ effectiveness at identifying whether the faults were in the implementation or test code. Table 4 summarizes the results of this investigation. The first three lines refer to results considering all classes, while the remaining present the results per class. As we can see, the tasks helped by original Randoop tests presented better rates considering all (62%), codefix (41%), and testfix tasks (83%) compared with split Randoop tests (37%, 16%, and 58%) and split-renamed Randoop ones (54%, 41%, and 66%).

To our surprise, original Randoop tests performed better than refactored ones. This fact may be evidence that the refactorings, or the treated test smells, did not bring enough improvements or guidance to correctly identify maintenance tasks. However, when we compared the refactored strategies, we found that the split-renamed strategy performed better compared to the split one, which shows that automatically generated names can impact and help developers to better understand the tests.

To measure statistical significance, we ran the Shapiro–Wilk [42] normality test, and the non-parametric Fisher’s exact test [43] for comparisons of correctness. With the confidence of 95%, we were able to statically compare strategies considering the results for all, code fix, and testfix tasks, individually (this analysis is similar to the one described in Section IV-B1). With all p-values greater than 0.05, we cannot reject the null hypothesis that treatments have similar performance for all, code fix, and testfix tasks.

Thus, we can answer RQ7 by saying that the performed refactorings did not improve (but also not worsen) the developers’ performance in determining the source of the problem. Furthermore, our results suggest that replacing bad test names has more impact on the test understanding than splitting it by assertions.

\textbf{RQ7: The refactoring performed in the original Randoop tests did not improve nor worsen the developers’ performance in determining the source of a problem.}

2) RQ8: DO REFACTORED RANDOOP TESTS IMPROVE DEVELOPERS’ EFFECTIVENESS IN PERFORMING PROPER FIXES?

To answer this question, we observed if participants were effective at producing correct fixes. Similar to the first study, we decided the correctness of a fix by following the protocol defined in Figure 6. Table 5 presents the collected results.

As we can see, in the tasks with split-renamed tests, developers had better rates in all (50%) and codefix (50%). When compared with original and split tests they had the same rates (41% for all and 25% for codefix). However, for testfix tasks, the original Randoop and split tests had slightly better rates (58%) than split-renamed (50%). Overall, these results evidenced that, while split tests have not brought any improvement to the developers’ performance in producing correct fixes, split-renamed tests have shown to be better for guiding developers to the solutions. However, our statistical analysis did not allow us to conclude that the observed difference is significant.

Therefore, we can answer RQ8 by saying that, compared to the original Randoop tests, developers’ effectiveness in producing correct fixes was not improved when using split tests. However, when guided by split-renamed tests, developers were more effective than the other two strategies. However, this improvement is not statistically significant.
TABLE 4. Comparison of correct decisions given Randoop, Randoop split or Randoop split-renamed tests. Last two columns refer to the statistical tests.

| Task Type | Class     | Randoop (R) | Split (S) | Split-renamed (SR) | R = S | p-values | S = SR | Final Ranking |
|-----------|-----------|-------------|-----------|--------------------|-------|----------|--------|--------------|
| all       | all       | 15/24 (62%) | 9/24 (37%)| 13/24 (54%)       | 0.1482| 0.7702   | 0.3852 | R = S = SR    |
| codefix   | all       | 5/12 (41%)  | 2/12 (16%)| 5/12 (41%)        | 0.3707| 1        | 0.3707 | R = S = SR    |
| testfix   | all       | 10/12 (83%) | 7/12 (58%)| 8/12 (66%)        | 0.3707| 0.6404   | 1      | R = S = SR    |
| all       | Fixed     | 5/8 (62%)   | 3/8 (37%) | 5/8 (62%)         | 0.6193| 1        | 0.6193 | R = S = SR    |
| codefix   | Fixed     | 1/4 (25%)   | 0/4 (0%)  | 2/4 (50%)         | 1     | 1        | 0.4286 | R = S = SR    |
| testfix   | Fixed     | 4/4 (100%)  | 3/4 (75%) | 3/4 (75%)         | 1     | 1        | 1      | R = S = SR    |
| all       | List      | 7/8 (87%)   | 4/8 (50%) | 4/8 (50%)         | 0.2821| 0.2821   | 1      | R = S = SR    |
| codefix   | List      | 4/4 (100%)  | 2/4 (50%) | 1/4 (25%)         | 0.4286| 0.1429   | 1      | R = S = SR    |
| testfix   | List      | 3/4 (75%)   | 2/4 (50%) | 3/4 (75%)         | 1     | 1        | 1      | R = S = SR    |
| all       | Comparator| 3/8 (37%)   | 2/8 (25%) | 4/8 (50%)         | 1     | 1        | 1      | R = S = SR    |
| codefix   | Comparator| 0/4 (0%)    | 0/4 (0%)  | 2/4 (50%)         | 1     | 0.4286   | 0.4286 | R = S = SR    |
| testfix   | Comparator| 3/4 (75%)   | 2/4 (50%) | 2/4 (50%)         | 1     | 1        | 1      | R = S = SR    |

**RQ8**: Developers were more effective in performing proper fixes when guided by split-renamed tests, however, this improvement was not statistically significant.

**RQ9**: When guided by split-renamed tests, developers took less time in fault fixing, but the same was not observed in fault identification.

**RQ10**: WHAT IS THE DEVELOPERS’ PERCEPTION ABOUT USING REFACTORED RANDOOP TESTS IN MAINTENANCE TASKS?

To answer RQ10, we looked at participants’ responses to the questionnaire at the end of the study. Figure 14 summarizes the answers for *testfix* and *codefix* tasks. In general, participants found the tasks clear (Question 1) and they had enough time to finish the tasks (Question 2). Participants found it easier to identify the fault type (Question 3) using split-renamed tests for both maintenance tasks. However, they found it easier to identify the fault type when guided by the original Randoop than split tests. These results go against those presented in Table 4, which shows that developers were more assertive at identifying faults when guided by Randoop tests. However, our statistical analyses did not show that the strategies have similar performance at this point.

Question 4 asked the participants’ perceptions about the activity of fixing faults. For codefix, the participants found it easier to perform a bug fix using refactored tests, especially the split-renamed ones. These results agree with those presented in Table 5, which shows that the developers were more effective in code fixes when guided by split-renamed tests. This goes according to our expectations since a less smelly test tends to be easier to follow. However, statistical tests have shown that these differences have no relevant significance.

Participants reported higher confidence about the correctness (Question 5) of their code fixes when using refactored tests, but the same was not observed for test fixes, where participants were more confident with original Randoop tests in those scenarios. Again, these results are in accordance with those classified in Table 4.

Developers found that was easier to understand the class under test (Question 6) and the tests (Question 7) when they were guided by original Randoop tests. For test fixes, they found it easier when guided by the refactored tests. In question 8, for code fix tasks, participants pointed out that the refactored tests were more useful in understanding the
CUT, especially using split-renamed tests. As for the test fixes tasks, they believed that they produced a better solution for the faults present in the Randoop tests.

In summary, we can answer RQ10 by saying that refactored tests contribute to a better understanding of the CUT, facilitate the identification of maintenance activities (for both tasks), and facilitate code fixes. But developers were more confident in their test fixes when using original Randoop tests.

RQ10: Refactored tests contribute more to the understanding of CUT, and facilitate the identification of maintenance activities and code fixes, but developers feel more confident in their test fixes when using original Randoop tests.

5) ANALYSIS BY ROLES
All participant roles were more effective in determining the source of a problem using the original Randoop tests, except software engineers, which were more effective using the refactored (renamed and split) tests (RQ7). These results may be influenced by the sample sizes. There were 17 software engineers, while for the other roles there were less than four participants. Regarding RQ8, data scientists and software engineers were more effective in fixing the bug using the refactored (renamed and split) tests. On the other hand, technical leaders and test analysts were more effective using the original Randoop tests. Finally, technical leaders and software engineers took less time to identify the fault using the refactored (renamed and split) tests, while test analysts and data scientists were faster using the original Randoop tests (RQ9). Again, these results may be influenced by the sample sizes instead of the participant roles. Moreover, due to the small number of developers from each role, these results have no statistical relevance.

6) FOLLOW-UP INTERVIEWS
Here we discuss the results of our guided interviews:

- What difficulties did you experience when deciding the maintenance task to be performed?

Most interviewees pointed out poor test code readability as their main difficulty. Non-descriptive variable names, long test statements, and confusing assert messages were mentioned as reasons for wrongly performed maintenance tasks. These test smells were not addressed by our refactored tests. Therefore, we can say that even those versions require improvements considering other test smells.

- Do you consider the provided test case name good and representative? Did it help you identify/fix the fault?

All interviewees pointed out the generated names (renamed refactoring) as suitable for the test cases. Some respondents stated that those tests “guided them in solving the problem and understanding the tests”. However, one participant stated that some of the test names were too long, which ended up working as a confusing factor.

- To the ones who worked with split tests: Did you consider the non-failing tests when identifying/fixing the fault?

Most interviewees stated that they focused only on the failing test. One participant reported that he scanned the other tests but only for understanding the structure of the assertions. This shows that the refactoring strategy Split tests makes it easier for developers to focus on a smaller code snippet where the fault is evidenced.

- Which of the three treatments (Randoop, renamed, split-renamed tests) would you consider the best option to assist the maintenance activities and why?

Although some disadvantages were listed (e.g., duplicate and unused code, number of test cases, test suite execution time), all interviewees preferred the split-renamed option over the original Randoop tests. The main advantages listed were the opportunity to focus on a single objective (one assert per test), and test names that facilitated the understanding of what the test was doing. On the other hand, most mentioned the fact that by dividing a test per asserts some code duplication was added. When splitting a test, we use static analysis
to find variable dependencies related to the asserts. To avoid compilation errors or behavioral change, any statement found related to the asserts’ was copied to the split tests. This process may lead to some duplicate and unwanted code in the test.

The results of this investigation were quite interesting. Although developers agree that splitting and renaming a Randoop test improve its quality (Sections V-C and VI-D6), we could not find many practical improvements when they faced a maintenance task with refactored tests. However, the use of refactored Randoop tests did not worsen the results. Moreover, they brought gains to the time spent fixing bugs, both in CUT and in test suites, which shows that they can contribute to reducing the costs of maintenance tasks. Furthermore, the best results were obtained when removing both test smells together (split-renamed tests), which demonstrates that the more test smells are removed, the better their adoption should be in practice. These results may guide further studies focused on improving tests automatically generated by Randoop and similar tools.

VII. IMPLICATIONS
Our work is based on a series of empirical studies. We believe that those studies may provide valuable insights for developers and future research.

We compared different generated tests (Evosuite, Randoop, and Randoop refactored) regarding their impact on software maintenance. Our results showed that they can be a great help for supporting developers in such activity. This information can help developers decide which and how to adopt the use of generated tests in their projects. For instance, the participants of our studies recognize the importance of unit tests. However, most of them admitted that they often do have enough time to write tests due to tight deadlines and/or issues related to the nature of the project. In such a scenario, we believe that generated test suites could be very
useful, especially EvoSuite’s tests, as they can contribute to the identification and correction of faults effectively (see RQ1 and RQ2). In scenarios where manual suites are available, both tools could be used, since they are helpful for guiding maintenance (RQ1 and RQ2), may lead to better code coverage, and do not increase the time spent fixing the fault (see RQ3).

Although generated tests are often discarded and replaced by new ones over the course of the project, some tests that exercise code parts not covered by manually written tests may be chosen to stay permanently. For instance, we found that developers perceived Randoop tests as the best option to fix the bugs (see RQ4). It might be related to its white-box generation strategy that may cover execution flows not easily identified by a tester. Those are candidates to be part of the project’s regression suite.

We also demonstrated that well-known refactoring techniques can be applied to reduce test smells occurrences in generated tests, and how this can be a good way to improve the quality and usage of them in support of developers in software maintenance. Moreover, we captured the developers’ perceptions about generated tests. Researchers can use this information for developing new approaches to improve the algorithms of existing tools or propose new tools that generate suites with a lower incidence of test smells. Another way of working is to develop post-processing tools that remove test smells in existing generated suites.

Finally, we believe that our studies may inspire other works to investigate the use of refactored generated tests during software maintenance.

VIII. THREATS TO VALIDITY

Here, we discuss the main threats to our conclusions.

In terms of construct validity, both studies (Sections IV and VI) reused most of the artifacts (classes and faults) from other work [19]. We decided to reuse those artifacts to be able to compare results. The added extra class and fault were also inspired by previous unit testing empirical study [40]. Although limited, those artifacts were selected from open-source projects and reflected real-world faults. Moreover, a more complex configuration (more classes and test suites with more than a single test) would be impractical. Even with such limited artifacts, participants took an average of 30 minutes to find and fix the faults per session (a total of 1.5h per participant). It is important to remember that participants were real developers, which often have limited time to participate in such studies.

We also believe that the used artifacts in each maintenance task (single class and failing test case) emulate real scenarios. When identifying and fixing faults (source code or test code), developers often focus on a single class and/or small edits [41]. That said, our results do not generalize beyond our dataset of subject programs, faults, and tests. For instance, a different set of test cases may lead to different results. However, by selecting a test that fails after fault injection, we guarantee it relates to the fault and, therefore, it can help detect and fix the fault. Thus, we believe the selected artifacts are good representatives for maintenance tasks.

We did not assess the quality of the code nor select test cases, as it was not our goal. We used developers’ output artifacts, video recordings, and multiple-choice questions to investigate aspects such as effectiveness and perception. Other strategies could be used in this sense, however, our goal was to see the practical aspects of a maintenance task using failing test cases. In addition, we counterbalanced the order and task assignment to mitigate learning effects.

We adapted two refactoring strategies (Rename and Extract Method) for improving Randoop tests (Test Renaming and Split Tests). To apply them, we created a script that reused the Eclipse Refactoring Engine and adapted Ermira et al.’s renaming strategy [44]. The Eclipse Refactoring Engine was used by other works [47] and is known to have a robust test suite that validates its transformations. Moreover, to validate our implementation, a series of tests were conducted and the results were manually validated by the authors.

As for conclusion validity, our studies dealt with a limited number of participants from a single country. Again, since we chose to work with real developers, we were subjected to the availability of developers from partner companies. However, we selected participants from different projects, with different roles and levels of experience. We believe that by working with real-world developers, we apply a more practical investigation. Works on empirical software engineering (e.g., [48], [49]) reinforce the need for real-world participants in empirical studies. Moreover, our study ended up providing interesting conclusions that even went against a similar study that used students as participants [19].

To mitigate internal validity, before the participants started their maintenance tasks, we ran a short tutorial on the procedure of the tasks. Moreover, they were familiar with the general aspects of a Java/JUnit application and identifying and fixing bugs. The participants were not familiar with the tests and CUTs before the study sessions. However, this scenario resembles very common in real projects, where developers need to maintain others or even legacy code. Furthermore, we cannot generalize our findings to contexts where developers maintain familiar code. Finally, during the sections, the first author was available for questions regarding the study procedure and provided environment.

IX. RELATED WORK

Test case generation is known for reducing the burden and costs of creating tests. Several works attest to the practical benefits of using test generation tools (e.g., [10], [11], [25], [28]). Although there is a wide range of tools for generating tests from Java programs [12], [13], [14], [15], [50], in our studies, we use two of the most well-known (EvoSuite and Randoop). Test generating tools can also be found for other languages (e.g., Python [51], C/C++ [52], .NET [53]). Though not used in our studies, we believe that our results might be valid for similar contexts, i.e., languages that share
characteristics with Java and/or tools that resembles the generation strategies of either Randoop or Evosuite.

Regarding comparing manual and generated test cases, there are works that are worth mentioning. Fraser et al. [54] and Rojas et al. [40] compare the behavior of participants when writing tests to the use of test generators. Alves et al. [25], [29] investigate whether generated tests (Randoop and Evosuite) can be used to find specific refactoring faults. Panichella et al. [16] run an empirical study with developers to investigate test understandability when comparing regular generated tests and generated tests with textual test summaries. They concluded that developers find twice as many bugs using tests with summaries. Daka et al. [55] investigate the effect of test readability on the time developers take to predict generated test outputs. They conclude that readability has a significant impact on the time developers need to reach a decision. These results corroborate our findings since participants of our study complained that some generated tests require improvements regarding code readability.

Our empirical study was inspired by Shamshiri et al.’s work [19]. The authors run a study with students to investigate how they perform maintenance tasks using Evosuite and manually-written tests. In our first study, we focused on real developers and we deal with a more comprehensive set of strategies: study 01 - manual, Evosuite tool, and Randoop tool; study 02 - original and refactored versions of Randoop tests. As discussed before, on several points, our conclusions differed from Shamshiri et al.’s. Therefore, we believe both works are complementary, they apply an investigation in different contexts and treatments. Moreover, we expanded the investigation by running a survey with developers and a new study on the effectiveness of generated tests, now focusing on different versions of Randoop test cases that reduced test smells.

About strategies that use refactoring-like transformations in test cases, we can discuss a series of works. Xuan and Monperrus [56] propose an approach that divides test cases with the goal of improving fault localization. Stefano et. al. [57] present DARTS, an IntelliJ plug-in that detects and refactors tests based on multiple asserts. They extract asserts to a private method that is called the original test. Zhang et al. [58] propose an approach to generate names for unit tests based on common test structures. Given a test, it identifies the action (e.g., the method being tested), the testing scenario (e.g., the parameters and context of the action), and the expected result (e.g., an assertion). Though the paper does not focus on generated tests, we believe their strategy might not work well with those tests. Testing scenario identification often depends on variables with descriptive names and the expected result is assumed to be a single assertion. Moreover, generated tests often cover several methods in a single test case, which may confuse the renaming strategy.

Allamanis et al. [59] apply a log-bilinear neural network model that suggests method names based on source code features. Again, this is a technique that might be hard to apply in generated tests, since those tests tend to use short sequences of calls and less descriptive names.

Ermira Daka et al. [44] present an approach for generating Evosuite test names. It uses coverage goals to create the names. As as far as we know, this is the first approach that deals with generated tests. Therefore, we adapted it to use with Randoop tests (V-B). Moreover, we ran a survey (Section V) and an empirical study (Section VI) to evaluate it in this novel scenario.

Our results show that, although important, applying extract and rename refactorings might not be enough to improve its quality. Other works have proposed strategies for refactoring test cases focused on other quality aspects, such as improving identifies names, code simplification, and quality metrics. For instance, Thies and Roth [60] propose an approach based on static analysis to support identifiers renaming. Allamanis et al. [61] proposed NATURALIZE, an approach based on an n-gram language model that suggests new names for identifiers. The n-gram model predicts the probability of the next token, considering the previous n-1 tokens. NATURALIZE learns coding conventions from the code base, promoting consistency in the use of identifiers. Lin et al. [62] evolve NATURALIZE with an approach that combines code analysis and n-gram language models.

Another way to improve the quality of unit tests is to simplify them. Search-based approaches can be used to make tests more understandable by generating more realistic scenarios [63], closer to natural language [64], or with better quality metrics (e.g., coupling and cohesion) [65]. Those strategies are yet to be evaluated when dealing with generated tests.

X. CONCLUSION
Developers often use failing test cases to guide maintenance tasks. However, good and trustworthy test cases are not always available. Generated suites have become an option to cope with this problem. The goal is to reduce the burden of creating sound test cases. However, we need to assess how developers perform and perceive the use of generated tests during maintenance. In this context, we ran three empirical studies with a total of 126 real developers.

The first study compared how 20 developers performed maintenance tasks with two types of automatically generated (Randoop and Evosuite) and manually-written test cases. We found that developers were more accurate at identifying maintenance tasks when using Evosuite tests, while they were equally accurate when using manually written and Randoop tests. Moreover, they were similarly effective at producing correct bug fixes using the three strategies (manually written, Evosuite, and Randoop). Regarding their perspectives, developers were more confident that produced correct outputs when using Evosuite tests, but they found manual tests a better proxy for the classes under the test’s behavior.

Test smells may be the main factors that impair test code comprehension, readability, and maintenance. Randoop tests
often include a number of test smells, such as non-descriptive names, assertion roulette, duplicate assert, eager test, lazy test, and magic number. This observation motivated us to perform our second study, a survey with 82 developers, that evaluated developers’ perceptions about refactored Randoop tests. We found that automatic renaming is well-received. Moreover, they preferred refactored Randoop tests over original ones. They also reported the need for more refactorings (e.g., better variable names) in order to fully accept Randoop tests.

Finally, our third study replicated the first one now focusing on evaluating whether refactored Randoop tests have an impact on the performance of maintenance tests. We found that refactorings did not improve the performance when finding the faults. However, developers were more effective to fix the faults with refactored Randoop tests. This task was also less time-consuming.

Based on those results we can conclude that automatically generated tests, specially Evosuite’s, can be a great help for identifying/fixing faults during maintenance, which differs from previous findings [19]. Randoop tests, although effective for fault identification, require improvements to be better accepted in practice. Refactoring transformations might be a good way for improving them. The suggested refactored tests (split-rename) were better appreciated by developers, but have shown little improvements in performance. We believe that other refactoring edits could also be used in this context, such as variables renaming and combined method extractions.

As future work, we plan to extend our studies with a larger group of participants and consider other configurations (e.g., working with an entire test suite instead of a single failing test). We also intend to improve the generation tools (Evosuite and Randoop) in order to solve some of the issues that may cause test smells. Moreover, we plan to investigate the use of different test refactoring transformations (e.g., variable/field Renaming) and evaluate their impact on the maintenance tasks.

REFERENCES

[1] G. Myers, C. Sandler, and T. Badgett, “The Art of Software Testing,” 3rd ed. Hoboken, NJ, USA: Wiley, 2012.
[2] M. Umar, “Comprehensive study of software testing: Categories, levels, techniques, and types,” Int. J. Adv. Res., Ideas Innov. Technol., vol. 5, pp. 32–40, Nov. 2019.
[3] IEEE Standards Coordinating Committee, “IEEE Standard Glossary of Software Engineering Terminology,” IEEE Standard 610.12-1990, IEEE Computer Society, Los Alamitos, CA, USA, vol. 169, 1990.
[4] M. J. Harrold and M. L. Souffia, “An incremental approach to unit testing during maintenance,” in Proc. Conf. Softw. Maintenance, 1988, pp. 362–367.
[5] G. Bavota, A. Quesef, R. Oliveto, A. De Lucia, and D. Binkley, “An empirical analysis of the distribution of unit test smells and their impact on software maintenance,” in Proc. 28th IEEE Int. Conf. Softw. Maintenance (ICSM), Sep. 2012, pp. 56–65.
[6] A. K. Onoma, W.-T. Tsi, M. Poonawala, and H. Suganuma, “Regression testing in an industrial environment,” Commun. ACM, vol. 41, no. 5, pp. 81–86, May 1998.
[7] E. Daka and G. Fraser, “A survey on unit testing practices and problems,” in Proc. IEEE 25th Int. Symp. Softw. Rel. Eng., Nov. 2014, pp. 201–211.
[8] P. McMinn, “Search-based software test data generation: A survey,” Softw. Test., Verification Rel., vol. 14, no. 2, pp. 105–156, 2004.
[9] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in Proc. 29th Int. Conf. Softw. Eng. (ICSE), May 2007, pp. 73–84.
[10] G. Fraser and A. Arcuri, “Evosuite: On the challenges of test case generation in the real world,” in Proc. IEEE 6th Int. Conf. Softw. Test., Verification Validation, Mar. 2013, pp. 362–369.
[11] G. Fraser and A. Arcuri, “A large-scale evaluation of automated unit test generation using Evosuite,” ACM Trans. Softw. Eng. Methodology, vol. 24, no. 2, pp. 1–42, Dec. 2014.
[12] C. Pacheco and M. D. Ernst, “RANDOOP: Feedback-directed random testing for Java,” in Proc. Companion 22nd ACM SIGPLAN Conf. Object Oriented Program. Syst. Appl. Companion (OOPSLA), 2007, pp. 815–816.
[13] S. Bayersfeld, T. Vos, and K. Lakhotia, “Unit testing tool competitions—Lessons learned,” in Future Internet Testing, Cham, Switzerland: Springer, 2014, pp. 75–94.
[14] F. Kifetew, X. Devroey, and U. Rueda, “Java unit testing tool competition—Seventh round,” in Proc. IEEE/ACM 12th Int. Workshop Search-Based Softw. Test. (SBST), May 2019, pp. 15–20.
[15] U. R. Molina, F. Kifetew, and A. Panichella, “Java unit testing tool competition: Sixth round,” in Proc. 11th Int. Workshop Search-Based Softw. Test., May 2018, pp. 22–29.
[16] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. Gall, “The impact of test case summaries on bug fixing performance: An empirical investigation,” in Proc. 38th Int. Conf. Softw. Eng. (ICSE), 2016, pp. 547–557.
[17] K. Bennett and V. Rajlich, “Software maintenance and evolution: A roadmap,” in Proc. Conf. Future Softw. Eng., 2000, pp. 73–87.
[18] R. Glass, Software Engineering: Facts and Fallacies. Reading, MA, USA: Addison-Wesley Longman Publishing, 2002.
[19] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser, “How do automatically generated unit tests influence software maintenance?” in Proc. IEEE 11th Int. Conf. Softw. Test., Verification Validation (ICST), Apr. 2018, pp. 250–261.
[20] W. Herculano, M. Mongiovi, and E. Alves, “Manually written or generated tests? A study with developers and maintenance tasks,” in Proc. 34th Brazilian Symp. Softw. Eng., 2020, pp. 273–282.
[21] V. Garousi and B. Kucuk, “Smells in software test code: A survey of knowledge in industry and academia,” J. Syst. Softw., vol. 138, pp. 52–81, Apr. 2018.
[22] G. Grano, F. Palomba, D. Di Nucci, A. De Lucia, and H. C. Gall, “Scented since the beginning: On the diffuseness of test smells in automatically generated code,” J. Syst. Softw., vol. 156, pp. 312–327, Oct. 2019.
[23] T. Virginio, L. A. Martins, L. R. Soares, R. Santana, H. Costa, and I. Machado, “An empirical study of automatically-generated tests from the perspective of test smells,” in Proc. 34th Brazilian Symp. Softw. Eng., 2020, pp. 92–96.
[24] G. Fraser and A. Arcuri, “Evosuite: Automatic test suite generation for object-oriented software,” in Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng. (SIGSOFT/FSE), 2011, pp. 416–419.
[25] I. P. S. C. Silva, E. L. G. Alves, and P. D. L. Machado, “Can automated test case generation cope with extract method validation?” in Proc. 33rd Brazilian Symp. Softw. Eng. (SBES), 2018, pp. 152–161.
[26] M. Mongiovi, “Scaling test refactoring engines,” in Proc. 38th Int. Conf. Softw. Eng. Companion, May 2016, pp. 15–17.
[27] M. Mongiovi, R. Gheyi, G. Soares, M. Ribeiro, P. Borba, and L. Teixeira, “Detecting overly strong preconditions in refactoring engines,” IEEE Trans. Softw. Eng., vol. 44, no. 5, pp. 429–452, May 2018.
[28] C. Pacheco, S. K. Lahiri, and T. Ball, “Finding errors in.NET with feedback-directed random testing,” in Proc. Int. Symp. Softw. Test. Anal. (ISTTA), 2008, pp. 87–96.
[29] I. P. S. C. Silva, E. L. G. Alves, and W. L. Andrade, “Analyzing automatic test generation tools for refactoring validation,” in Proc. IEEE/ACM 12th Int. Workshop Autom. Softw. Test. (AST), May 2017, pp. 38–44.
[30] G. Soares, “Making program refactoring safer,” in Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. (ICSE), Mar. 2010, pp. 521–522.
[31] K. Beck, Test Driven Development By Example. Reading, MA, USA: Addison-Wesley, 2002.
[32] A. Deursen, L. Moonen, A. Bergh, and L. Teixeira, “Understanding test smells and their impact on software maintenance,” in Proc. 39th Int. Conf. Object-Oriented Program. Syst. Appl. Companion (OOPSLA), 2014, pp. 201–211.
[33] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, “On the relation of test smells to software code quality,” in Proc. IEEE/ACM Int. Conf. Software Maintenance Evol. (ICSM), Sep. 2018, pp. 1–12.
[34] G. Meszaros, xUnit Test Patterns: Refactoring Test Code. Upper Saddle River, NJ, USA: Prentice-Hall, 2006.

[35] F. Palomba and A. Zaidman, “Notice of retraction: Does refactoring of test smells induce fixing flaky tests?” in Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME), Sep. 2017, pp. 1–12.

[36] F. Palomba and A. Zaidman, “The smell of fear: On the relation between test smells and flaky tests,” Empirical Softw. Eng., vol. 24, no. 5, pp. 2907–2946, Oct. 2019.

[37] S. Reichhart, T. Gibra, and S. Ducasse, “Rule-based assessment of test quality,” J. Object Technol., vol. 6, no. 9, p. 231, 2007.

[38] M. Fowler, Refactoring: Improving the Design of Existing Code. Reading, MA, USA: Addison-Wesley, 1999.

[39] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, “A case study on the impact of refactoring on quality and productivity in an agile team,” in Proc. IFIP Central East Eur. Conf. Softw. Techn., 2007, pp. 252–266.

[40] J. M. Rojas, G. Fraser, and A. Arcuri, “Automated unit test generation during software development: A controlled experiment and think-aloud observations,” in Proc. Int. Symp. Softw. Test. Anal., Jul. 2015, pp. 338–349.

[41] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for Java programs,” in Proc. Int. Symp. Softw. Test. Anal. (ISSTA), 2014, pp. 437–440.

[42] N. Razali and B. Yap, “Power comparisons of Shapiro–Wilk, Kolmogorov–Smirnov, Lilliefors and Anderson–Darling distributions,” J. Stat. Model. Anal., vol. 2, no. 1, pp. 21–33, 2011.

[43] R. Fisher, “On the interpretation of $X^2$ from contingency tables, and the calculation of $P$,” J. Roy. Stat. Soc., vol. 85, no. 1, pp. 87–94, 1922.

[44] E. Daka, J. Rojas, and G. Fraser, “Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?” in Proc. 26th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA), 2017, pp. 57–67.

[45] C. Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering,” in Proc. 18th Int. Conf. Eval. Assessment Softw. Eng. (EASE), 2014, pp. 1–10.

[46] T. Hesterberg, “Bootstrap,” in Wiley Interdisciplinary Reviews: Computational Statistics, vol. 3, Hoboken, NJ, USA: Wiley, 2011, pp. 497–526.

[47] E. L. G. Alves, P. D. L. Machado, T. Massoni, and S. T. C. Santos, “A refactoring-based approach for test case selection and prioritization,” in Proc. 8th Int. Workshop Autom. Softw. Test. (AST), May 2013, pp. 93–99.

[48] J. Carver, L. Jaccheri, S. Morascha, and F. Shull, “Issues in using students in empirical studies in software engineering education,” in Proc. 5th Int. Workshop Enterprise Netw. Comput. Healthcare Ind., 2003, pp. 239–249.

[49] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. Edam, and J. Rosenberg, “Preliminary guidelines for empirical research in software engineering,” IEEE Trans. Softw. Eng., vol. 28, pp. 721–734, 2002.

[50] C. Csalnner and Y. Smaragdakis, “XCrasher: An automatic robustness tester for Java,” Softw., Pract. Exper., vol. 34, no. 11, pp. 1025–1050, Sep. 2004.

[51] S. Lukacszyk and G. Fraser, “Pygunit: Automated unit test generation for Python,” in Proc. IEEE/ACM 44th Int. Conf. Softw. Eng., Companion Proc. (ICSE-Companion), May 2022, pp. 163–166.

[52] H. Yoshida, G. Li, T. Kamiya, I. Ghosh, S. Rajan, S. Tokumoto, K. Munakata, and T. Uehara, “KLOVER: Automatic test generation for Java,” Empirical Softw. Eng., vol. 34, no. 5, pp. 30–37, Sep. 2017.

[53] Industrial Software Systems at ABB Corporate Research. Randoom.Net, 2015. Accessed: Oct. 9, 2022. [Online]. Available: https://www.google.com/search/client=firefox-b-d&p=Randoom.Net%2C2+2015

[54] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, “Does automated unit test generation really help software testers? A controlled empirical study,” ACM Trans. Softw. Eng. Methodol., vol. 24, no. 4, pp. 1–49, Sep. 2015.

[55] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, “Modeling readability to improve unit tests,” in Proc. 10th Joint Meeting Found. Softw. Eng., Aug. 2015, pp. 107–118.

[56] J. Xuan and M. Monperrus, “Test case purification for improving fault localization,” in Proc. 22nd ACM SIGSOFT Int. Symp. Softw. Eng., Nov. 2014, pp. 52–63.

[57] S. Lambiase, A. Cupito, F. Pecorrelli, A. De Lucia, and F. Palomba, “Just-in-time test smell detection and refactoring: The DARTS project,” in Proc. 28th Int. Conf. Program Comprehension, Jul. 2020, pp. 441–445.

[58] B. Zhang, E. Hill, and J. Clause, “Towards automatically generating descriptive names for unit tests,” in Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng., Aug. 2016, pp. 625–636.

[59] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Suggesting accurate method and class names,” in Proc. 10th Joint Meeting Found. Softw. Eng., Aug. 2015, pp. 38–49.

[60] A. Thies and C. Roth, “Recommending rename refactorings,” in Proc. 2nd Int. Workshop Recommendation Syst. Softw. Eng. (RSSE), 2010, pp. 1–5.

[61] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Learning natural coding conventions,” in Proc. 22nd ACM SIGSOFT Int. Symp. Softw. Eng., Nov. 2014, pp. 281–293.

[62] B. Lin, S. Scalabrino, A. Mocci, R. Oliveto, G. Bavota, and M. Lanza, “Investigating the use of code analysis and NLP to promote a consistent usage of identifiers,” in Proc. IEEE 17th Int. Work. Conf. Source Code Anal. Manipulation (SCAM), Sep. 2017, pp. 81–90.

[63] G. Fraser and A. Zeller, “Exploiting common object usage in test case generation,” in Proc. 4th IEEE Int. Conf. Softw. Test., Verification Validation, Mar. 2011, pp. 80–89.

[64] S. Afshan, P. McMinn, and M. Stevenson, “Evolving readable string test inputs using a natural language model to reduce human Oracle cost,” in Proc. IEEE 6th Int. Conf. Softw. Test., Verification Validation, Mar. 2013, pp. 352–361.

[65] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Luca, “Automatic test case generation: What if test code quality matters?” in Proc. 25th Int. Symp. Softw. Test. Anal. (ISTTA), 2016, pp. 130–141.

WESLEY R. R. HERCULANO received the bachelor’s degree in computer science from the Federal University of Campina Grande, Brazil, in 2016, where he is currently pursuing the master’s degree in computer science. His research interests include software testing and software maintenance. He has been a fellow at the Research, Development, and Innovation Center in Information, Communication, and Automation Technology (VIRTUS) acting as of the Quality Assurance Analyst, since 2016.

EVERTON L. G. ALVES received the bache- lor’s, master’s, and doctorate degrees in computer science from the Federal University of Campina Grande, UFCG, Brazil, in 2009, 2011, and 2015, respectively. He has been a Professor with the Computing and Systems Department, UFCG, since 2016. Since 2005, he has been produced a number of contributions in the area of software development and testing, including research projects, publications, tools, supervising, national/international cooperation and teaching. His research interests include software testing, software analysis, software maintenance, component based software development, model-driven development, and data analysis.

MELINA MONGIOVI received the bachelor’s, master’s, and doctorate degrees in computer science from the Federal University of Campina Grande, UFCG, Brazil, in 2011, 2012, and 2016, respectively. She has been a Professor with the Computing and Systems Department, UFCG, since 2016. Since 2011, she has been produced a number of contributions in the area of software engineering, including research projects, awards, publications, tools, supervising, national/international cooperation and teaching. Her research interests include program refactoring, software testing, static and dynamic analysis, formal methods, and program synthesis.