How Do Automatically Generated Unit Tests Influence Software Maintenance?

Sina Shamshiri*, José Miguel Rojas†, Juan Pablo Galeotti‡, Neil Walkinshaw† and Gordon Fraser§
*Department of Computer Science, University of Sheffield, UK
†Department of Informatics, University of Leicester, UK
‡Department of Computer Science, University of Buenos Aires, Argentina
§Chair of Software Engineering II, University of Passau, Germany
*sina.shamshiri@sheffield.ac.uk, †{j.rojas, nw91}@leicester.ac.uk, ‡jgaleotti@dc.uba.ar, §Gordon.Fraser@uni-passau.de

***Abstract*—Generating unit tests automatically saves time over writing tests manually and can lead to higher code coverage. However, automatically generated tests are usually not based on realistic scenarios, and are therefore generally considered to be less readable. This places a question mark over their practical value: Every time a test fails, a developer has to decide whether this failure has revealed a regression fault in the program under test, or whether the test itself needs to be updated. Does the fact that automatically generated tests are harder to read outweigh the time-savings gained by their automated generation, and render them more of a hindrance than a help for software maintenance? In order to answer this question, we performed an empirical study in which participants were presented with an automatically generated or manually written failing test, and were asked to identify and fix the cause of the failure. Our experiment and two replications resulted in a total of 150 data points based on 75 participants. Whilst maintenance activities take longer when working with automatically generated tests, we found developers to be equally effective with manually written and automatically generated tests. This has implications on how automated test generation is best used in practice, and it indicates a need for research into the generation of more realistic tests.**

I. Introduction

Developers can save time and effort by generating unit tests automatically. Automatically generated tests can be integrated into the code base of the program under test just like manually written tests, where they support developers during software maintenance [1]. Good tests should not merely detect *undesired* modifications of the code by failing, but should also provide guidance to the developer in correcting the undesired modification to make the test pass again. Furthermore, *desired* code changes may require the test code to be modified in order to avoid spurious test failures. Accordingly, a good test case should not only be sensitive to deviations from the intended behavior, but should also be maintainable in its own right; it should be easy to understand so that it can be readily adapted to changes in the rest of the code base as it evolves.

It is far from clear that automatically generated tests satisfy these latter requirements. Instead of capturing realistic scenarios (as manually written tests commonly do), automatically generated tests tend to simply concatenate expressions that have the narrower aim of leading to code coverage. This is cause for concern, as software maintenance is commonly accepted to not only be difficult [1], but also to account for around 60% of the total software costs [2]. What if the time saved by generating tests automatically comes at a net cost? What if automatically generated tests end up requiring more time to understand, and offer less reliable indicators to the whereabouts of bugs in the source code? What if manual tests, though more expensive to construct, are ultimately better able to support the developer during routine software maintenance tasks?

Common approaches to evaluate automated test generation tools cannot answer these questions, as they focus on test adequacy measures such as code coverage or estimates of their fault detection ability (e.g., mutation scores). These measures neither evaluate whether the tests help developers to distinguish between desired or undesired modifications, nor do they evaluate whether the tests themselves are maintainable and support developers when maintaining code. To investigate this unexplored aspect of automatically generated tests, we conducted an empirical study to specifically evaluate their influence on developers' effort and effectiveness when performing software maintenance tasks.

The scenario of our controlled experiment represents a developer facing a test failure. The developer has to identify and fix the cause of this failure. If the test fails because the code does not satisfy the specification, the code needs to be fixed to make the test pass again. Otherwise, if the test fails in error, the developer must fix the test to match the specification. We instantiated this maintenance scenario using real test failures produced by developers whilst performing implementation tasks, and compared manually written tests to tests generated automatically with the EvoSuite [3] tool.

In detail, the contributions of this paper are as follows:

- **Experiment Design:** We present a comprehensive controlled experiment design, refined using two pilot studies with 20 participants, to evaluate automated test generation in terms of maintenance effort and effectiveness (Sec. III).
- **Experiment Results:** We present the results of our experiment and two replications, resulting in a total of 150 data-points based on 75 participants (Sec. IV).

Our experiments yield the following key results:

- **Developers are as effective at maintenance tasks with generated tests as they are with manually written tests**. This observation is an important reinforcement of existing and ongoing work in automated test generation.

- **Developers are less efficient at performing maintenance tasks with generated tests**. This suggests that future research is necessary to identify the best ways to integrate automated test generation into the developer workflow. For example, integrating generated tests into the code base might make more sense when software is close to completion, whereas before that they could be re-generated on demand, rather than maintained.
- **Developers are more skeptical of generated tests than manual tests**, and would tend to blame test failures on the tests as opposed to the program code. This provides evidence for common presumptions about generated tests, and reinforces ongoing research on improving or explaining generated tests (e.g., [4]–[6]).
- **Developers find automatically generated tests harder to understand**, despite these tests being less complex, because they execute unrealistic scenarios. Thus, beyond ongoing efforts to improve readability or to explain tests, there is a need to investigate ways in which generated tests can be made to resemble *realistic* scenarios.

Since our findings raise new questions on how to best use automated test generation tools, we provide a thorough description of our experimental setup and methodology and a replication artifact. We hope to thereby lay the ground for further replications and related studies to investigate these questions, and to learn more about the effects of automated test generation at different stages during software development.

II. Background

A. Automated Test Generation

Unit tests for object oriented software consist of sequences of calls to classes under test. Randomly generating such sequences can be effective at finding undeclared exceptions [7] or violations of code contracts [8], [9]. By adding assertions that capture the state of the current program [10], [11] these tests can also be used for regression testing [8], [12], [13]. A multitude of search-based (e.g., [14]–[18]) and symbolic execution-based (e.g., [19]–[21]) test generators exist, which intend to overcome some of the intrinsic limitations of random testing by aiming to cover as much code as possible with the smallest possible test suites. EvoSuite is a competitive example of such advanced test generation tools (e.g., [22]).

B. Test Evolution

Test code has to evolve alongside the rest of the source code. Changes to the application code can render tests obsolete or require the addition of new tests. As such, test code is subject to the same problems that arise in conventional software maintenance; the code can evolve, can deteriorate in quality, and become harder to understand and fix as time passes [23].

The need for maintenance action is often triggered by a failing test. Dealing with a failing test can potentially be a time consuming task. Although some automated solutions have been proposed, these tend to focus on relatively specific tasks, such as changing assertions to make failing tests pass [24], [25], fixing test compilation errors after the code under test has been changed [26], [27], or deciding whether a test failure is due to a problem with the test code or the code of the program under test [28]. Although promising, these approaches have not passed into widespread use. By and large, in practice the bulk of the process—diagnosing a failing test case, determining a cause (either in the test code or program code) and applying a fix—is a manual one.

C. Evaluating Generated Tests

The most common approach to evaluating automatically generated test sets is in terms of their capacity to expose faults. Accordingly, their evaluation tends to focus on conventional test adequacy metrics such as code coverage or mutation scores. Whilst many of the test generators that have been evaluated in these terms have been shown to be effective, it appears that their uptake in practice has been low. This indicates barriers to adoption that have not been highlighted in previous studies.

In order to discover what these barriers might be, test generators need to be evaluated in a more realistic context, where developers need to interact with them continuously and to maintain and evolve test code and application code in tandem. There are some noteworthy studies that measured the influence of automatically generated tests on developers (e.g., [4], [5], [29], [30], discussed in more detail in Section VI). However, studies that attempt to go beyond test adequacy to assess generated tests are rare, and none of them have provided insight into our specific problem, which is assessing the value of automatically generated test cases in a maintenance context.

III. Experimental Setup

This paper investigates the performance of developers on maintenance tasks that are triggered by the failure of manually written or automatically generated unit tests. To this end we conducted a controlled experiment intended to recreate a typical software maintenance scenario: a test has failed, and the developer has to identify and solve the problem, either by fixing the application code, or by updating the test code.

By collecting data on the correctness and duration of maintenance activities as well as qualitative survey responses, we aim to answer the following research questions:

How do automatically generated tests influence...

RQ1 ***The effectiveness of developers when identifying maintenance tasks?*** Does the accuracy of developers at determining the source of a problem upon a test failure change when using generated tests?

RQ2 ***The effectiveness of developers when performing maintenance tasks?*** Do developers produce correct fixes more or less often when using generated tests?

RQ3 ***The efficiency of developers when identifying and performing maintenance tasks?*** Does it take longer to execute maintenance tasks when using generated tests?

RQ4 ***The developers' perception of maintenance tasks?*** Do developers find it easier to maintain when using generated tests, and are they more confident in their solutions?

The remainder of this section describes the experimental setup and the procedure for the controlled experiment in detail.

A. Participant Selection

We recruited participants by emailing invitations to Software Engineering and Computer Science undergraduate and graduate students at the University of Sheffield (UK) and the University of Leicester (UK). The main requirements to participate were basic programming skills and a basic command of the Java language and the JUnit framework. Moreover, participants were required to take and pass (answer correctly at least 3 out of 5 questions) an online quiz specifically designed to test their Java and JUnit expertise. As a result, 75 participants (55% undergraduate (BSc), 45% postgraduate (MSc, PhD, PostDoc)) took part in the study and were remunerated GBP20 in cash for their work. Participants had diverse backgrounds: 92% had two or more years of programming experience, 68% had two or more years of experience with Java, 83% had used JUnit before, and 67% had performed some software maintenance task before. Eleven students failed the qualification quiz and hence were not invited to take part in the study.

B. Object Selection

To conduct the experiment we required realistic, faulty implementations and faulty tests (both manually and automatically generated). Although mutants can serve as proxy for real faults when evaluating test effectiveness [31], their suitability for software maintenance experiments is less clear. Therefore, we aimed to find real faults. Although some repositories exist that provide seeded and real code faults [32], [33], we are not aware of any similar dataset for test faults. This lack of dedicated artifacts led us to the study by Rojas et al. [34] which, to the best of our knowledge, is the only study where both code and unit tests were produced by human participants. Their 46 participants were asked to manually develop implementations and write unit tests for four reasonably-sized Java classes according to their provided JavaDoc specifications. Besides these 46 sets of implementations and test suites, the dataset also contains the original open-source implementations and test suites for these four classes, referred to as *golden* implementations and test suites. Their artifacts have already been used independently by other researchers, e.g., for automated oracle generation [35].

We selected our experimental objects by searching for faulty implementations and test suites in this dataset. In particular, we were interested in *minimally-faulty* implementations and test suites produced by developers who performed well in their implementation and testing tasks but made subtle mistakes. That is, we looked for: (1) faulty implementations written by participants, differing from the golden implementation by *exactly* one failing golden test; and (2) faulty test suites that contain exactly one test which fails on the golden implementation but passes on the participant's implementation.

As a result of our search in this dataset, we selected the failing tests and corresponding implementations for classes `FixedOrderComparator` (referred to as `comparator` in the rest of the paper) and `ListPopulation`

```
1  public int compare(Object obj1, Object obj2) {
2        isLocked = true;
3  +     if(map.get(obj1) == null && map.get(obj2) == null){
4  +          return 0;
5  +     }
6        if (map.get(obj1) == null) { // returns 1 or -1
```

Fig. 1: Example of a codefix for the `comparator` class. When two unknown objects are passed to the `compare` function, the result should be 0.

```
1 public Chromosome getFittestChromosome() {
2 -    Chromosome fitter = null;
3 +    Chromosome fitter = this.chromosomes.get(0);
4      for (Chromosome c : this.chromosomes)
5         if (c.compareTo(fitter) > 0)
6             fitter = c;
7      return fitter;
8 }
```

Fig. 2: Example of a codefix for the `listpopulation` class. A `NullPointerException` is thrown by `compareTo` due to the `fitter` Chromosome being `null`.

(`listpopulation`), which complied with our selection criteria (see Table I for more details). For each golden and resp. minimally-faulty version, we then used EvoSuite to generate exactly one unit test that failed on the minimally-faulty and resp. the golden version. To conceal the origin of the source code and tests, we then unified Java package names, and used uniform class and test names.

We manually investigated the faults and ensured that their fix is challenging but achievable in a short controlled session. To illustrate the type of artifacts selected, Figure 1 and Figure 2 show the faulty implementations of the selected classes, including examples of correct fixes. As examples for test-fixing artifacts, Figure 3 and Figure 4 respectively show the failing manually written and automatically generated tests used for `comparator`, along with the required fix. When selecting faults, we tried to balance the difficulty of the required fixes, aiming for bugs that could be fixed with a few lines of code.

C. Tasks

The failure of a test can generally be traced back to a regression in the source code or a defect in the test, e.g., as a result of a feature change. We build these two scenarios into our experiment setup by considering two types of maintenance tasks: *code-fixing* and *test-fixing*. Since we are interested in the effects of generated tests on these types of tasks, we consider two treatments: *manual* and *generated*. For each of the two selected classes (`comparator` and `listpopulation`), we have the following experiment tasks:

- ⟨codefix, manual⟩: Faulty implementation, manually written golden test.
- ⟨codefix, generated⟩: Faulty implementation, test automatically generated for golden implementation.
- ⟨testfix, manual⟩: Golden implementation, faulty manually written test.
- ⟨testfix, generated⟩: Golden implementation, faulty test automatically generated for faulty implementation.

By construction, the tests in all tasks are in failure, indicating a fault in either the code (codefix tasks) or in the test itself

```
public void test() {
    try {
-     Object[] emptyArray = {};
+     Object[] emptyArray = null;
      FixedOrderComparator comparator = new
          FixedOrderComparator(emptyArray);
      fail("Exception was supposed to be thrown!");
    } catch (IllegalArgumentException e) {
      assertTrue(true);
    }
}
```

Fig. 3: A manually written test for class `comparator` requiring fixing. The argument should be null instead of the empty array to trigger the exceptional behavior.

```
public void test() throws Throwable {
    FixedOrderComparator fixedOrderComparator0 = new
        FixedOrderComparator();
    fixedOrderComparator0.setUnknownObjectBehavior(1);
    int int0 = fixedOrderComparator0.compare((Object) null,
        (Object) null);
    assertEquals(1, fixedOrderComparator0.
        getUnknownObjectBehavior());
    assertTrue(fixedOrderComparator0.isLocked());
-   assertEquals(1, int0);
+   assertEquals(0, int0);
}
```

Fig. 4: An automatically generated test for class `comparator` requiring fixing. The comparison of two null references should return 0 (equal) instead of 1 (greater than).

(testfix tasks). Given the source code of the class under test, the specification of the expected code behavior in JavaDoc form, and a JUnit class consisting of only the failing test, the participants had to identify the required type of fix and record their decisions. The correctness of their decision was then revealed, and they had to accordingly produce and submit either a codefix or a testfix.

D. Experiment Procedure

The experiments started with a 20-minute refresher tutorial on the use of JUnit and a walk-through of the tasks involved in the study. Then, two main sessions followed, each with a maximum length of 60 minutes and a 5-minute break in between. Each participant was assigned two maintenance tasks, one per session. To prevent any learning effects, no participant was assigned the same fix type or class across sessions (e.g., if the participant's first assignment was codefix on `comparator`, then the second assignment was testfix on `listpopulation`).

Participants started each session by launching a fully-featured Eclipse Kepler IDE instance, with a pre-configured workspace according to their assigned tasks (a Java project with the class implementation, its dependencies and the failing test). The rest of the environment consisted of the Ubuntu Linux OS and Java SE 7u55. Participants were encouraged to use any technique they considered appropriate to approach their maintenance tasks, e.g., running the failing test, checking coverage, using the IDE's debugging feature, making temporary changes, etc.

A typical maintenance scenario upon a test failure involves (a) identifying whether the code or the test needs fixing (*decision*), and (b) actually performing the fix (*fixing*). Although these activities generally intertwine [36], a natural dependency exists between them: a correct fix is only possible if the cause of the failure is correctly identified first. This gives rise to the potential problem that if a participant makes the wrong decision (contradicting their task), his/her fix would be invalid. To minimize this risk, we required participants to record their decision as soon as they made it. If their decision was correct, they simply continued with their assigned task, but if they made the wrong decision (and were on course to perform the wrong maintenance task), we revealed their assigned task (codefix or testfix) in an effort to steer them towards completion. We encouraged the participants to reach a decision 30 minutes into their task, reminding them that declaring "I don't know" was a valid choice in case they were not sure of their answers; 30 minutes was determined adequate after the pilot studies (see Section III-F). Although somewhat extrinsic to the maintenance process, this "decision checkpoint" was necessary to prevent the loss of valuable data-points in the study. (See discussion of threats to validity in Section III-I.)

E. Data Collection

For each performed task, participants produced three outcomes: their decisions, their fixes, and their answers to an exit survey. Decisions and fixes were collected via a dedicated website. Decisions were submitted by simply clicking one of three choices (codefix, testfix, or don't know). Fixes were submitted by uploading either the class under test or the JUnit test suite containing a single test. Survey answers were collected using Google Forms. The survey questionnaire consisted of the Likert-scale questions shown in Figure 7, free-form questions asking participants to describe their approach to the maintenance task, and questions on the challenges faced.

F. Pilots and Replications

Two pilot studies were conducted to finalize the experimental setup described in this section. The first pilot was conducted at the National University of Quilmes (AR) with 13 participants on June 2, 2016. The second one, with an improved setup, was conducted at the University of Buenos Aires (AR) with 7 students on June 26, 2016.

The main study comprises a baseline experiment and two exact replications where only the subject pool changed [37]. The baseline experiment took place at the University of Sheffield (UK) on December 2, 2016 (24 participants). Two *exact* replications were conducted so that the results could be combined, to achieve a sufficiently large number of participants and improve confidence in our findings. The first replication took place on December 6, 2016 at University of Leicester (UK) (27 participants), and the second replication took place at the University of Sheffield (UK) on December 9, 2016 (24 participants).

G. Data Analysis

We first evaluate the time it took participants to decide the type of fix needed and the correctness of these decisions. We then evaluate the time taken by the participants to perform the actual fixes. Finally, we measure the quality of their submitted solutions. The measurement of the quality of the solution differs

TABLE I: Selected Java classes. NCSS and Methods respectively refer to the # of Non Commenting Source Statements, and methods in the classes. The Instruction and Branch coverage values refer to the coverage levels obtained by executing the golden tests on the classes.

Class Name	Reference	NCSS	Methods	Instruction Cov.	Branch Cov.	Golden Test CLOC	Description
FixedOrderComparator	comparator	68	10	81.5%	77.5%	137	Comparator which imposes a specific order on a specific set of objects
ListPopulation	listpopulation	54	13	80.0%	77.3%	149	Genetic population of chromosomes, represented as a List

according to the maintenance task. For code-fixing tasks, a codefix solution is classified as correct if it does not break any additional tests from the golden test suite and satisfies a manual inspection. For test-fixing tasks, we created a reference solution (fix) for each faulty test-case, based on our understanding of the original test purpose. Then, three authors of the paper independently inspected each submitted test and classified them as either "correct", "incorrect", or "no-clear-decision" with respect to the reference solution.

Any test-fixes with disagreement on the categorization that could not be resolved by discussion were then evaluated using mutation-analysis. We used the Major mutation framework [38] to determine whether all mutants killed by the golden tests were also killed by the fixed tests submitted by the participants. If the fixed test killed all mutants killed by the golden test, then we marked the solution as *correct*, as it satisfies the same test purpose. If some of the mutants killed by the golden test survived the fixed test, then this indicates that the original test purpose is no longer satisfied, and such a testfix was marked as *incorrect*. For example, this happens when assertions or method calls are removed from the test to make it pass. Overall, out of 75 testfix submissions, a clear decision could not be made only for 11, out of which seven were classified as "correct" and four as "incorrect" using this methodology.

H. Statistical Analysis

To measure statistical significance when comparing treatments, the following tests were used: (1) for comparisons of duration values, we used the non-parametric Mann-Whitney U test [39], (2) for comparisons of correctness, we used the non-parametric Fisher's exact test [40]. Non-parametric tests were used as the Shapiro–Wilk test did not confirm a normal distribution. Besides significance at $\alpha < 0.05$, we also report all p-values to allow readers to better interpret the results.

I. Threats to Validity

Empirical studies are essential to validate and understand the merits of new software engineering technologies. However, a number of challenges arise when involving human participants [41], ranging from difficulties in recruiting participants, to task design, and risk of inconclusive results due to limited statistical power. Although we followed a rigorous methodology based on existing guidelines and best practices [42], like any other empirical study [43], ours has threats to validity.

a) Participants: All participants in our study have a software engineering or computer science background. A study of this magnitude (75 participants) would be hard to carry out without the help of a student population. Whereas there are some contrasting views on whether students can be representatives of real world professionals [44]–[46], more recent work suggests that when empirical studies are carefully scoped, similar performance is observed between students and professionals [47]. To reduce this threat, we selected only participants that passed our competency test (see Section III-A).

b) Objects: The artifacts used in our experiment were created by human developers in a previous unit testing empirical study [34]. The authors of that study selected them from a collection of open-source projects using a well-defined systematic protocol. We borrowed these artifacts due to their manageable size, availability, and amenability for our purposes. A possible threat to validity is that the defects produced in that study may not be representative of defects introduced in larger industrial projects. However, while the changes are mostly small, in practice a large number of maintenance tasks may involve small fixes. For instance, nearly half (47%) of the bug-fixes of the Defects4J (v1.1) collection of bugs [33] require two or less lines of code to be added and/or removed.

c) Generated tests: EvoSuite was the only tool used to generate the automated tests used in this study. EvoSuite was selected for two reasons: (a) it is representative of the state of the art in unit test generation; and (b) adding a tool producing different types of tests, e.g., Randoop [8], would have either required 50% more participants to achieve the same high number of data-points per treatment, or dropping one of the classes. Faced with these less desirable alternatives, the most reasonable choice is to rely on future replications, as common with empirical studies [43].

d) Procedure: All participants performed a codefix and a testfix task, in random order and with random manual/generated treatment. To avoid participants inferring the second task based on their first one, we told participants that they may have to perform the same type of task twice due to the random assignment. To avoid an intrinsic human bias against automatically generated tests, participants were not made aware of the origin of the test they worked with, neither explicitly (they were not told which task they were assigned) nor implicitly (we used uniform class and test names).

Identifying and performing fixes are highly complementary activities. As mentioned in Section III-D, a decision checkpoint was used to steer participants who were struggling to identify their assigned tasks. This allowed us to elicit fixes matching assigned tasks from all participants and within our time constraints. As a trade-off, our efficiency results may not generalize to contexts where developers have unlimited time to complete the maintenance tasks. To increase confidence in our findings, we can confirm that all trends observed in Table III and Figure 6 persist when looking exclusively at participants who made the right maintenance decisions.

Participants were unfamiliar with the tests and classes under test used in our study. While this is a common scenario in practice, e.g., when maintaining legacy code, our findings may not generalize to contexts where developers are more familiar

TABLE II: Comparison of correct decisions given Manual or Generated tests. Columns Manual and Generated indicate the number of correct decisions made out of the total number of decisions, along with the ratio in parentheses. The number of "don't know" answers is shown in brackets. Ratios shown in bold text indicate higher ratio than the alternative treatment.

Task type	Task class	Manual	Generated	p-val.
All	All	43/75 (57%) [6]	44/75 **(59%)** [6]	1.00
codefix	All	21/38 **(55%)** [1]	17/37 (46%) [2]	0.49
testfix	All	22/37 (59%) [5]	27/38 **(71%)** [4]	0.34
All	comparator	18/38 (47%) [1]	22/37 **(59%)** [1]	0.36
codefix	comparator	5/19 (26%) [0]	8/18 **(44%)** [0]	0.31
testfix	comparator	13/19 (68%) [1]	14/19 **(74%)** [1]	1.00
All	listpopulation	25/37 **(68%)** [5]	22/38 (58%) [5]	0.48
codefix	listpopulation	16/19 **(84%)** [1]	9/19 (47%) [2]	0.04
testfix	listpopulation	9/18 (50%) [4]	13/19 **(68%)** [3]	0.32

TABLE III: Comparison of correct fixes when given Manual or Generated tests. Columns Manual and Generated indicate the number of correct fixes made out of the total number of fixes, along with the ratio in parentheses. Ratios denoted with bold text indicate higher ratio than the alternative treatment.

Task type	Task class	Manual	Generated	p-val.
All	All	46/75 **(61%)**	43/75 (57%)	0.74
codefix	All	25/38 **(66%)**	20/37 (54%)	0.35
testfix	All	21/37 (57%)	23/38 **(61%)**	0.82
All	comparator	26/38 (68%)	26/37 **(70%)**	1.00
codefix	comparator	14/19 **(74%)**	9/18 (50%)	0.18
testfix	comparator	12/19 (63%)	17/19 **(89%)**	0.12
All	listpopulation	20/37 **(54%)**	17/38 (45%)	0.49
codefix	listpopulation	11/19 (58%)	11/19 (58%)	1.00
testfix	listpopulation	9/18 **(50%)**	6/19 (32%)	0.32

with the code under maintenance. As usual, further replications are needed to investigate alternative contexts.

e) Data collection: There is a risk that our data collection framework could contain bugs that affect our analysis. This was mitigated by conducting preliminary pilot studies where we tested the reliability of all the functionalities of the framework (e.g., recording responses and submitting artifacts).

f) Data Analysis: We used test execution results as well as mutation analysis to help assess the correctness of the submissions (both for test and code fixes). As such, bugs in our analysis scripts as well as the quality of the golden test suites may result in submissions being misclassified. To mitigate this threat, we manually evaluated all submissions as detailed in Section III-G, and make all our experiment material publicly available for scrutiny and replication.

g) Statistical Significance: Due to the intrinsic limitations of empirical studies involving humans (e.g., recruiting participants and assigning tasks), their results tend to offer limited statistical power and inconclusive answers [43]. While our study is already large, further replications [37] will be necessary to confirm the generalizability of our conclusions.

IV. Results

Our study (baseline experiment and two exact replications) resulted in a total of 150 data-points based on 75 participants. In this section we present the results of the analysis, as described in the previous section, for each of the research questions.

A. RQ1 (Effectiveness): How do automatically generated tests influence the effectiveness of developers at identifying maintenance tasks?

Table II summarizes the effectiveness of participants at identifying whether the fault lies in the code or in the test. Overall, there is almost no difference between the use of generated and manually written tests: respectively 43/75 and 44/75 decisions based on manually written and generated tests tests were correct. However, different trends are observed between the codefix and testfix tasks. When the fault lies in the *test*, the number of correct decisions is substantially higher for generated tests (27/38 vs. 22/37 with manually written tests). In contrast, recognizing that the code needs to be fixed appears to be easier using manually written tests, as particularly shown by the `listpopulation` class (16/19 vs. 9/19 with generated tests). Our conjecture is that developers might be more likely to blame tests that they find harder to understand—and, arguably, in this case the generated tests are less readable.

> *RQ1: Overall, developers were equally accurate at identifying maintenance tasks when using manually written and automatically generated tests.*

B. RQ2 (Effectiveness): How do automatically generated tests influence the effectiveness of developers at performing maintenance tasks?

When looking at the effectiveness at correctly performing their maintenance tasks (Table III), we observe that participants were overall similarly effective at producing correct fixes given generated or manually written tests. A small (statistically insignificant) increase in successful task completion of 4% can be observed when developers were guided by manually written tests (46/75) compared to generated tests (43/75). In particular, participants submitted slightly more correct code fixes when guided by manually written tests (25/38) than when guided by generated tests (20/37). The difference is even smaller (arguably negligible) for testfix tasks, where generated tests led to only two more correct test fixes than manually written tests (23/38 vs 21/37). This similar effectiveness also holds when looking only at participants who correctly identified the maintenance task (all data can be found in the replication package).

Looking at individual classes, the general pattern is reflected well in the `comparator` class, where participants produced correct fixes more often when using manually written tests (14/19) than generated tests (9/18); fixing the generated tests for this class seems to be easier (17/19) than fixing the manually written tests (12/19). For the `listpopulation` class, the participants were equally effective at codefix tasks given either manually written or generated tests (11/19). However, in this case the generated test was slightly more difficult to fix (6/19) than the manually written test (9/18). Our conjecture is, again, that readability is a contributing factor. For example, for both testfix tasks the test with slightly less correct fixes (`comparator`/manual, `listpopulation`/generated) contains

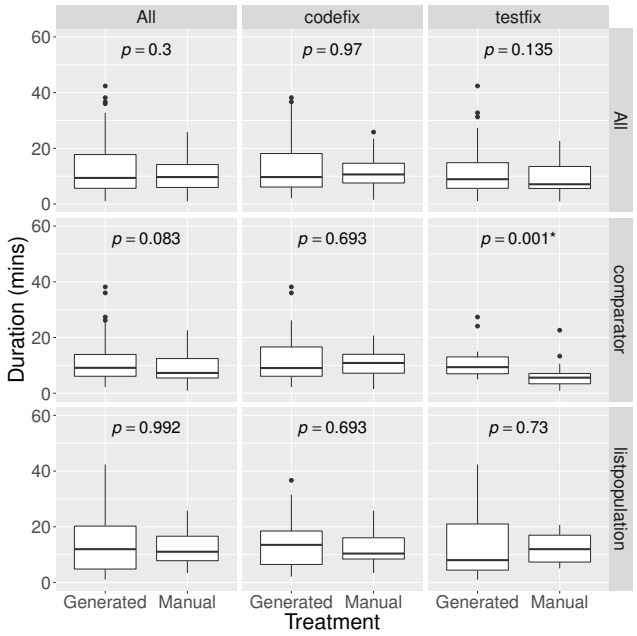


Fig. 5: Box-plots comparing the time developers took to identify their maintenance tasks across treatments, grouped by task type and class.

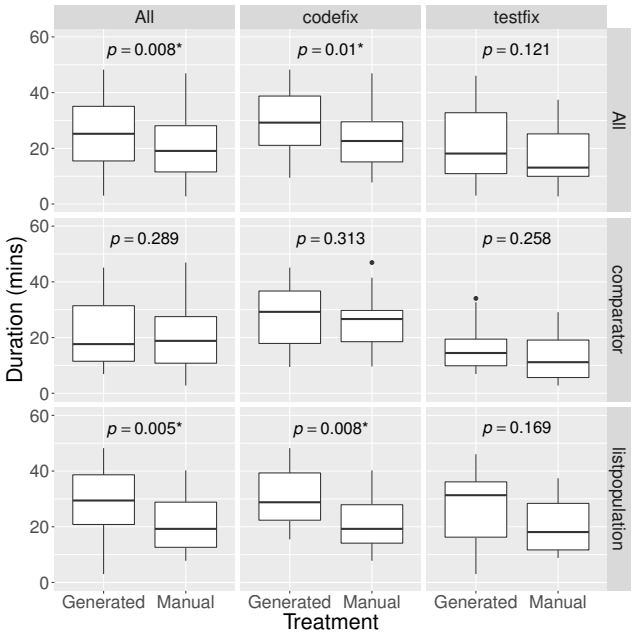


Fig. 6: Box-plots comparing the overall time developers took to complete their maintenance tasks across treatments, grouped by task type and class.

a try/catch construct to expect exceptions, which may impact readability (even though the try/catch construct is not involved in the fix)—we discuss this further in Section V-C.

RQ2: Developers were similarly effective at producing correct fixes with generated or manually written tests.

C. RQ3 (Efficiency): How do automatically generated tests influence the efficiency of developers when identifying and performing maintenance tasks?

Having established that developers were similarly effective at maintaining code and tests when supported by generated or manually written tests, we now analyze developers' maintenance efficiency. We first consider the time developers needed to identify whether the class under test or the test code needed maintenance (Figure 5). We then take a look at how this identification time grows into overall efficiency (Figure 6).

Figure 5 compares the time spent for the decision tasks between generated and manually written tests. Overall, these efficiency results resemble the effectiveness results discussed in RQ1 and RQ2, in that there is mostly no significant difference between generated and manually written tests. The only exception is the testfix task of the `comparator` class, where the time for the generated test is significantly longer than for the manually written counterpart. This might be related to the higher number of method calls and assertions in the generated test, and thus its overall readability.

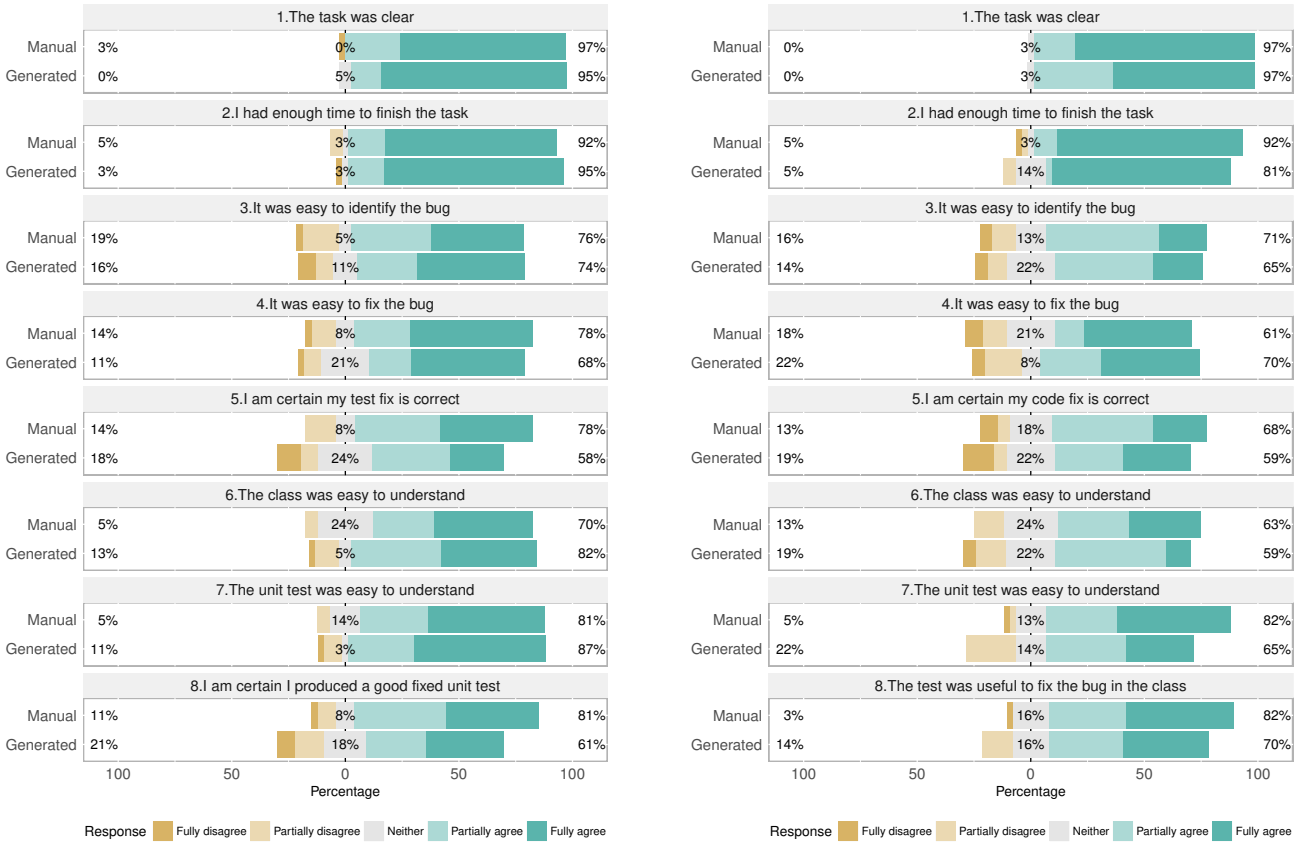
Looking at the *total* time developers spent on identifying their tasks and effectively performing the necessary fixes reveals a clear trend in favor of manually written tests. As Figure 6 shows, it took participants *significantly* longer ($p = 0.008$) to complete their tasks when using generated tests. This trend holds across task types and classes—although the distinction is particularly evident with `listpopulation`—and suggests that automatically generated tests are less helpful for code fixing and harder to fix when they break than manually written tests. These trends also hold when looking only at the overall efficiency of participants who submitted correct fixes (data for this can be found in our accompanied artifact package).

Comparing the duration times shown in Figure 5 and Figure 6 shows that participants invested 50% of their time in identifying the source of a problem. We also observe that participants who failed to correctly identify their tasks took as long to implement fixes as those who accurately identified their tasks. This is surprising, considering that the former group had to review and understand the failing test and code before proceeding to perform their fixes. Ultimately, this suggests that, irrespective of the fix type and origin of the failing tests, participants struggled to actually implement satisfactory fixes.

RQ3: Developers were significantly more efficient at maintenance tasks when using manually written failing tests.

D. RQ4 (Perception): How do automatically generated tests influence the developers' perception of maintenance tasks?

Figure 7 summarizes answers to the survey questions, for test-fixing and code-fixing tasks. Overall, the participants found the task clear (Question 1), and the allocated time sufficient (Question 2). The participants also reported little difficulty with the task of identifying the fault (Question 3). For the remaining questions, notice that while small differences can be observed between manually written and generated tests, the majority of participants had a similar opinion about the questions.



(a) Test-fixing tasks

(b) Code-fixing tasks

Fig. 7: Overview of the survey responses relating to code and test maintenance tasks.

When attempting to fix the fault in the test, more participants found fixing manually written tests easier than generated tests. Conversely, when fixing the fault in the code, participants found the code-fixing task easier using generated tests, than when given manually written tests (Question 4). This, however, is in contrast with the actual success outcome of the participants (cf. Table III), which is likely due to confusion caused by the randomness and structurally-simple nature of the generated tests (i.e., while it may be hard to determine what the test intends to do, ultimately these tests are comprised of simple sequences of test statements). Given that it is hard to understand the intent of an automatically generated test, it is understandable that fixing a test is easier when it is manually written. However, when fixing code, even if the purpose of the test is difficult to understand, the test might still be simple enough to help the developer at least *localize* the fault in the code. In line with observations from previous studies [29], a useful application may be to use the failing test mostly as an oracle for the bugfix instead of investing time in trying to fully understand it.

Participants reported higher confidence in the correctness of their fix for both code and test fixing tasks when using manually written tests (Question 5). This is to be expected as automatically generated tests contain relatively random sequences of statements. As such it can be challenging for a developer to surmise the expected behavior of the test.

For understanding the class under test, there was little difference between participants who were trying to fix manually written tests versus those who were trying to fix generated ones. However, when given faulty code, participants found manually written tests more helpful to understand the class under test (Question 6). One plausible explanation for this is the fact that manually written tests can often work as documentation of the intended behavior of the program.

Participants found understanding generated tests slightly easier than manually written tests when they needed to fix the tests (Question 7). This suggests that when participants know that a test is broken, they find generated tests less confusing. However, when required to use the tests to fix a fault in the code, participants found manually written tests easier to understand compared to automatically generated tests. This supports our previous conjectures that participants find automated tests more difficult to understand. Also, supporting our previous observations, we found that participants had higher confidence in the quality of their fix when a manually written test was involved (Question 8, test-fixing). Moreover, they found manually written tests more useful for undertaking code-fixing tasks (Question 8, code-fixing).

RQ4: Developers were less confident about their actions when guided by generated tests and perceived them as less helpful for understanding the maintenance tasks.

V. Detailed Discussion

Our results suggest that automatically generated tests affect maintenance efficiency and confidence, but not effectiveness. In this section, we discuss root-causes and implications.

A. Do generated tests hinder maintenance?

The observation that the accuracy of maintenance decisions and tasks was hardly affected by automatically generated tests is an important result, as a reduction in accuracy would be a potential obstacle for the adoption of automated test generation. However, the increase of 29% in the time taken for maintenance given automatically generated tests indicates that there is a trade-off: Although automated test generation can save time during development (Rojas et al. [34] reported a reduction of 36%), this could end up being repaid during maintenance.

Determining whether this is indeed the case requires further research and depends on a host of factors. There is the amount of maintenance that needs to be performed; if the project is immature and unstable (requiring frequent maintenance activities), the cost of using automatically generated tests might be higher than if the project is mature and stable. There is also the question of how automated test generation is applied, where it fits into the development process, and how it supports existing manual tests. Automatically generated tests can be discarded and re-generated at any point with little or no effort (as opposed to manually written tests), which provides the opportunity to tailor processes to minimize the maintenance overhead of generated tests, whilst maximizing their impact.

It might also be that an increase in maintenance time is perfectly acceptable, if this is the price of better software quality. Our experiment implicitly assumes that manually written tests are replaced with generated tests — maybe we need to more fundamentally change the testing process with automated test generators. What if developers are more likely to test, write more testable code, or write more assertions and code contracts, given automated test generation tools? Whether better software quality is actually something that automated test generation can achieve is, once again, a matter of further research.

B. What makes generated tests difficult to understand?

Given that the tests in our experiment were not accompanied by comments or descriptive names, participants had to read the test code and infer the original intention of the test. Intuitively, this can be challenging for generated tests, where the statements are based on random generation and variables have artificial names—which would explain the increase of 29% in the time taken for maintenance given automatically generated tests.

Responses to the free-form questions in the exit survey seem to corroborate this intuition. For example, one participant stated: *"As the purpose of the test case was not clear from the beginning, I just followed my intuition in order to fix it [...]. But I felt that I could have deleted the test case and I would still have completed the task"*. Another participant felt even more frustrated about the testing purpose by stating: *"I knew what didn't work but I didn't understand the purpose of the test and so could not fix it."*

Even if a test is not understandable, it was still possible to infer the intended behavior from the JavaDoc documentation in our experimental setup. This is corroborated by more frequent mentioning of JavaDocs when asked what helped to perform the codefix in the free text responses, e.g.: *"[...] read the java doc to get the general idea of what the Class's purpose was"*, or *"[...] read the javadoc at the beginning of the classes to be tested."*. Indeed, participants spent the majority of their time (56.8%) on source code rather than test code (measured with the Rabbit [48] Eclipse plug-in); even more so when given generated tests (61.0%). Considering that documentation is perceived helpful, this provides further incentive for research on generating test documentation (e.g., [4], [49]). The general problem of understandability also suggests that more work on improving the representation [5], [30] would be useful. Ideally, however, tests should be generated that are more *realistic* to begin with, instead of trying to explain unrealistic tests.

C. Types of test fixes and their influence

Our data also provides insights into how types of test fix tasks influence efficiency. In particular, our data shows that when the problems with a test were restricted to its assertions, then developers were quickly able to identify and fix the error (for both manually and automatically generated tests). An example can be seen in Figure 4, Lines 7-8, where participants only had to change the expected value in the assertion.

This was not the case when the behavior of the test itself had to be changed. When this was the case, participants struggled more with manually written tests (63% correct fixes in comparison to 89% when given automatically generated tests, for the `comparator` class, see Table III). An example of this can be found in the `comparator` task, when the behavior of the class under test had to change from checking for empty arrays to checking for `null` parameters (Figure 3, Lines 3-4).

A similar trend was noticed when participants had to fix tests with exceptional behavior. In this case the participants seemed more confused about whether the test intended to test a) exceptional behavior (i.e., the unit test checks whether a method throws an exception), b) the non-exceptional behavior (i.e., the unit test checks that a method does not throw a certain exception), or c) some aspect of the method itself, regardless of whether it throws an exception or not. For example, one participant reported: *"The test seemed to want to test the exceptional behavior of the method. It had an exception that the method did not throw. I looked at what the method did throw and then created a test that would test that behavior."*

Indeed, this confusion was reflected in our results: when test-fixing involved a test-case with exceptional behavior, participants took slightly longer (3'12" longer on average) and made more mistakes (8 fewer correct fixes), for both generated and manually-written tests. Interestingly, this was not the case when the code needed to be fixed. However, generated tests are intuitively more likely to reflect exceptional behavior, as they use nonsensical inputs and unrealistic sequences of calls. Thus, future studies are necessary to investigate how exceptional behavior in tests affects the overall maintenance effort.

VI. Related Work

While the most common way to evaluate automated test generation tools is in terms of coverage or fault detection (e.g., [50]), there are several studies that involved human participants. Although the studies by Fraser et al. [51] and Rojas et al. [34] are related as they are based on the same test generation tool (EvoSuite), these studies compare the behavior of participants when writing their own tests compared to the use of test generators, rather than comparing how participants work with existing tests. However, the studies by Ceccato et al. [29], Panichella et al. [4], and Daka et al. [5] involved participants performing tasks related to test understanding or maintenance.

Ceccato et al. [29] compared the effectiveness and efficiency of human participants at *debugging*, given manually written and automatically generated test cases. Debugging of test failures caused by faults in the code is also one aspect of our experiments, and our results for RQ2 corroborate the results of one of their experiments (based on EvoSuite and 15 experienced developers), in that the accuracy at fixing a bug in the code seems to be independent of the origin of the failing tests. However, our study goes beyond the debugging task considered by Ceccato et al.: We investigate the more holistic task of diagnosing a test failure with respect to the specification, identifying the causing fault, and performing a fix, which can be applied not only on the code, but also on the tests. This is not only a fundamentally different experiment design (using real code/test faults, specifications, etc.), but also requires participants to apply a completely different approach to scrutinizing and understanding the test and program code. Indeed our experiments have shown that the necessity to understand the test cases has clear effects on *identifying and fixing* application code (RQ3), whereas Ceccato et al. note that "[e]xperienced subjects did not spend time understanding the purpose of the test cases", but rather focused on understanding the buggy code. Contrasting these two alternative perspectives on the problem of using automatically generated tests suggests that a particularly suitable application of automatically generated tests might be to trigger violations of code assertions or code contracts, such that developer interactions with the tests are restricted to plain debugging.

The studies by Panichella et al. [4] and Daka et al. [5] focused specifically on the role of the understandability of generated tests. The study by Panichella et al. [4] provided participants with two types of tests (conventional tests, and tests accompanied by textual test summaries) to debug the program under test. While this scenario does not include the possibility for tests requiring fixes (as is the case in our experiment), the results did show that providing natural text summaries of tests, and thus supporting test understanding, makes participants more effective at deciding when a test reveals a fault. Similarly, Daka et al. [5] investigated the effect of test readability on the time it takes developers to predict whether a generated test would pass or fail. Participants were given two types of tests (conventional generated tests, and generated tests that had been optimized for readability). They established that readability did indeed have a significant impact on the time taken by developers to reach a decision at a similar level of accuracy; our experiment results corroborate these findings.

VII. Conclusions

Automated test generation is commonly evaluated in relatively narrow terms—usually in terms of the code coverage the generated test suites achieve, or their capacity to expose faults. In this paper we have sought to evaluate automatically generated tests in terms of a more applied software-engineering context: to see how helpful they are with respect to software maintenance tasks. To this end, we created an elaborate study setup, and used it in a controlled experiment and two replications, resulting in a total of 150 data-points based on 75 participants.

A primary finding of our experiment is that the effectiveness of developers at performing maintenance tasks is not affected by the use of automatically generated tests; our participants were similarly effective at fixing generated and manually written tests, and they were similarly effective at fixing code in order to make generated or manually written tests pass. This is an important reinforcement for automated test generation: Although practitioners are often still skeptical of automatically generated tests, we show that these tests are as supportive as manually written tests during software maintenance.

However, our results show maintenance tasks take longer with generated tests. Developers perceive automatically generated tests to be less helpful for understanding maintenance tasks, and generated tests induce less confidence in developers carrying out maintenance tasks. This calls for research on improving or explaining generated tests (e.g., [4]–[6]). It would be even better, though possibly even more challenging than explaining unrealistic or confusing tests, to generate tests that represent realistic, understandable scenarios in the first place.

Finally, our results provide insights into how automatic test generators might best be integrated into the software development workflow—a question which, to date, has seen little research (e.g., [52]), and may be affecting adoption of automated test generators in practice. Maintaining generated tests can take more time, and using automated test generation tools may at the same time lead to more tests. Thus, it might be best to integrate automatically generated tests into the code base only at later development stages (see Section V-A), when the rate of change reduces, whereas it might be more useful to simply re-generate tests during more active development phases. However, it might also be that the use of automated test generation requires a more fundamental change of the development and testing process, for example by moving the manual effort of writing tests to writing assertions.

To validate and generalize the results of our empirical study and our conclusions, further replications are important [37] (e.g., using larger code-bases, larger fixes, expert developers, other test-generation tools, complete test suites instead of exactly one failing test, etc.). To this extent, we make all experimental material available as a comprehensive artifact package at **http://evosuite.org/maintenance**.

REFERENCES

[1] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*. ACM, 2000, pp. 73–87.

[2] R. L. Glass, *Facts and fallacies of software engineering*. Addison-Wesley Professional, 2002.

[3] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2011, pp. 416–419.

[4] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 547–558.

[5] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 107–118.

[6] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: what if test code quality matters?" in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, pp. 130–141.

[7] C. Csallner and Y. Smaragdakis, "Jcrasher: an automatic robustness tester for java," *Software: Practice and Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[8] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2007, pp. 75–84.

[9] B. Meyer, I. Ciupa, A. Leitner, and L. L. Liu, "Automatic testing of object-oriented software," in *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 2007, pp. 114–129.

[10] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *European Conference on Object-Oriented Programming*. Springer, 2006, pp. 380–403.

[11] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 2, pp. 278–292, 2012.

[12] J. H. Andrews, F. C. Li, and T. Menzies, "Nighthawk: A two-level genetic-random unit test data generator," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 144–153.

[13] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, "Instance generator and problem representation to improve object oriented code coverage," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 3, pp. 294–313, 2015.

[14] P. Tonella, "Evolutionary testing of classes," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 119–128, 2004.

[15] A. Arcuri and X. Yao, "Search based software testing of object-oriented containers," *Information Sciences*, vol. 178, no. 15, pp. 3075–3095, 2008.

[16] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering (TSE)*, vol. 39, no. 2, pp. 276–291, 2013.

[17] J. H. Andrews, T. Menzies, and F. C. Li, "Genetic algorithms for randomized unit testing," *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 1, pp. 80–94, 2011.

[18] L. Baresi, P. L. Lanzi, and M. Miraz, "Testful: An evolutionary test approach for java," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2010, pp. 185–194.

[19] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and Z. Su, "Synthesizing method sequences for high-coverage testing," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 189–206, 2011.

[20] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution." in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 3440. Springer, 2005, pp. 365–381.

[21] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[22] G. Fraser, J. M. Rojas, J. Campos, and A. Arcuri, "EvoSuite at the SBST 2017 Tool Competition," in *International Workshop on Search-Based Software Testing (SBST)*. ACM, 2017, pp. 39–42.

[23] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: an exploratory study in industry," in *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012, p. 51.

[24] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "Reassert: Suggesting repairs for broken unit tests," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2009, pp. 433–444.

[25] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2010, pp. 207–218.

[26] M. Mirzaaghaei, F. Pastore, and M. Pezze, "Automatically repairing test cases for evolving method declarations," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–5.

[27] ——, "Supporting test suite evolution through test case adaptation," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2012, pp. 231–240.

[28] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang, "Is this a bug or an obsolete test?" in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 602–628.

[29] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella, "Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, pp. 5:1–5:38, Dec. 2015.

[30] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2013, pp. 352–361.

[31] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 654–665.

[32] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, Oct. 2005.

[33] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2014, pp. 437–440.

[34] J. M. Rojas, G. Fraser, and A. Arcuri, "Automated unit test generation during software development: A controlled experiment and think-aloud observations," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015, pp. 338–349.

[35] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2016, pp. 213–224.

[36] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 12, 2006.

[37] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "The role of replications in empirical software engineering," *Empirical Software Engineering*, vol. 13, no. 2, pp. 211–218, 2008.

[38] R. Just, F. Schweiggert, and G. M. Kapfhammer, "Major: An efficient and extensible tool for mutation analysis in a java compiler," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2011, pp. 612–615.

[39] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability (STVR)*, vol. 24, no. 3, pp. 219–250, 2014.

[40] R. A. Fisher, "On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P," *Journal of the Royal Statistical Society*, vol. 85, no. 1, pp. pp. 87–94, 1922.

[41] R. P. Buse, C. Sadowski, and W. Weimer, "Benefits and barriers of user evaluation in software engineering research," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 643–656, 2011.

[42] A. J. Ko, T. D. Latoza, and M. M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, 2015.

[43] W. F. Tichy, "Hints for reviewing empirical work in software engineering," *Empirical Software Engineering*, vol. 5, no. 4, pp. 309–312, 2000.

[44] J. Carver, L. Jaccheri, S. Morasca, and F. Shull, "Issues in using students in empirical studies in software engineering education," in *IEEE International Software Metrics Symposium*, 2003, pp. 239–249.

[45] M. Höst, B. Regnell, and C. Wohlin, "Using students as subjects—A comparative study of students and professionals in lead-time impact assessment," *Empirical Software Engineering*, vol. 5, no. 3, pp. 201–214, 2000.

[46] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 8, pp. 721–734, Aug. 2002.

[47] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Press, 2015, pp. 666–676.

[48] "*Rabbit - Eclipse Statistics Tracking plugin* URL: https://code.google.com/p/rabbit-eclipse/," 2014, version 1.2.1.

[49] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft, "Automatically documenting unit test cases," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 341–352.

[50] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[51] 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 Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, p. 23, 2015.

[52] N. Tillmann, J. De Halleux, and T. Xie, "Transferring an automated test generation tool to practice: From pex to fakes and code digger," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 385–396.