A Large Scale Study On the Effectiveness of Manual and Automatic Unit Test Generation

Beatriz Souza
Federal University of Campina Grande
beatriz.souza@ccc.ufcg.edu.br

Patrícia Machado
Federal University of Campina Grande
patricia@computacao.ufcg.edu.br

ABSTRACT

Recently, an increasingly large amount of effort has been devoted to implementing tools to generate unit test suites automatically. Previous studies have investigated the effectiveness of these tools by comparing automatically generated test suites (ATSs) to manually written test suites (MTSs). Most of these studies report that ATSs can achieve higher code coverage, or even mutation coverage, than MTSs, particularly when suites are generated from defective code. However, these studies usually consider a limited amount of classes or subject programs, while the adoption of such tools in the industry is still low. This work aims to compare the effectiveness of ATSs and MTSs when applied as regression test suites. We conduct an empirical study, using ten programs (1368 classes), written in Java, that already have MTSs and apply two sophisticated tools that automatically generate test cases: Randoop and EvoSuite. To evaluate the test suites' effectiveness, we use line and mutation coverage. Our results indicate that MTSs are, in general, more effective than ATSs regarding the investigated metrics. Moreover, the number of generated test cases may not indicate test suites' effectiveness. Furthermore, there are situations when ATSs are more effective, and even when ATSs and MTSs can be complementary.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

automatic test generation, empirical studies, mutation testing

ACM Reference Format:
Beatriz Souza and Patrícia Machado. 2020. A Large Scale Study On the Effectiveness of Manual and Automatic Unit Test Generation. In *34th Brazilian Symposium on Software Engineering (SBES '20), October 21–23, 2020, Natal, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422392.3422407

1 INTRODUCTION

Software testing is staple in any software development process [44] because it helps in identifying the presence of faults [19, 44, 45]. However, writing good tests, particularly unit tests, is often considered difficult [19, 37], tedious [45], and time-consuming [44]. For instance, a field study with software engineers presented by Beller et al. [20] shows that half of the developers do not test in their IDEs. Also, approaches such as Test-Driven Development (TDD), which highly depend on writing test cases, have not been largely practiced.

Tools and techniques for automatic generation of unit test cases have been proposed to support developers in unit testing [15, 44], particularly, following a white-box strategy where the code under test is analyzed to select inputs and capture observable behavior in assertions. Evosuite [25] and Randoop [41] are examples of such tools. While Evosuite implements a search-based generation technique, Randoop implements a feedback-directed random technique. Both tools have been part of most of the editions of the Java Unit Testing Tool Competition, with Evosuite often ranking first, even though they have achieved a comparable performance in the 2019 edition [36]. On the one hand, generation tools can be appealing, as they promise to produce satisfactory test cases in a short time with minimal effort [19]. On the other hand, the industry's adoption of such tools is still low [13, 28, 44].

To evaluate the effectiveness of the test suites generated by the different unit test generation techniques, we can take some metrics into account. Test suite quality is frequently measured based on the amount of code covered and on the fault detection ability of the test suite [38].

Previous studies investigated the effectiveness of manually written test suites (MTSs) against automatically generated test suites (ATSs). Most of these studies report that ATSs can achieve higher code coverage [19, 28, 44], or even mutation coverage [19, 44], than MTSs. Moreover, studies indicate that ATSs are still limited to detect faults [13, 44]. Other studies suggest that ATSs and MTSs may detect different faults [43, 48]. However, these studies only used a limited amount of classes or subject programs to conduct their analysis. Since both MTSs and ATSs face challenges to adoption in the industry, *we need further investigation to identify their current limitations and contributions* to the unit testing practice.

In this work, we empirically evaluate MTSs and ATSs using ten open source projects from Apache Commons [5] (with a total of 1368 classes). We apply two state-of-the-art unit test generation tools for Java, Randoop and EvoSuite, and investigate the test suites in terms of line coverage and strong mutation coverage with mutants generated by the PITest tool [22]. On top of that, we also explore the non-determinism of the tools and whether there is a correlation between the test suite size and the ability of test suites to detect faults. It is important to remark that we focus on regression test cases only. Thus, we generate ATSs from the clean program and consider only test cases that pass on the clean program. While

SBES '20, October 21–23, 2020, Natal, Brazil

ACM ISBN 978-1-4503-8753-8/20/09…$15.00
https://doi.org/10.1145/3422392.3422407

this choice might raise the Clean Program Assumption problem [21], it can prevent that mutations get encoded in the assertions [32].

Therefore, the main contributions of this paper are as follows:

- A large-scale study, applying two state-of-the-art automated unit test generators for Java to ten open source projects with MTSs;
- A detailed analysis of how non-deterministic the ATSs are;
- A detailed comparison between the effectiveness of the test suites generated with the different unit test generation techniques.

We organize this article as follows. We explain the test generation techniques in Section 2. Section 3 describes the approach to conduct our study. Section 4 presents and discusses the results of our study. In Section 5, we present possible threats to validity and limitations. Finally, we relate our work to others (Section 6) and present concluding remarks (Section 7).

2 TEST CASE GENERATION

This section briefly discusses the approaches of manually writing and automatically generating test suites and also describe the Randoop and EvoSuite tools.

2.1 Manually Writing Test Cases

Test suites are most often written manually, either by the developers themselves or through a quality assurance team [38]. Writing good tests, particularly unit tests, can be challenging [37, 44]. Currently, there is no formal method to help standardize the test writing practice [19, 31, 37, 38]. However, there are some widely-accepted best practices such as boundary-value analysis, data structures analysis, control-flow path execution, error management paths execution (i.e. exception handling checking), mock objects, and environments generation [19, 37]. Additionally, anti-patterns, named test smells, have been investigated [29]. In general, the methods and styles of writing individual tests, the fulfillment of coverage and fault-finding goals, and the ordering of test suites are left to industry requirements or personal preference [38].

2.2 Automatically Generating Test Cases

Due to the high cost and inconsistencies introduced when developing test suites by hand, automatic test suite generation research is on the rise [38]. To support developers in unit testing, researchers have explored different approaches to generate unit tests automatically, thus relieving the developers of part of their hard work [45].

There are many automatic test case generation tools available, which can be classified into two categories: Deterministic and Learning-Based [38]. Deterministic automatic test case generators normally analyze method parameters and basic paths to create unit tests. Tools such as JUnitDoclet [10], CoView [1], and CodePro [6], are classified as deterministic. Learning-Based automatic test case generation tools, on the other hand, use learning algorithms to improve the overall quality of the generated test suites. The two top-ranked tools in this area are Randoop [4] and EvoSuite [2] [38], which are the tools that we empirically study in this paper.

2.2.1 Randoop. A unit test generator for Java [4, 41] that automatically creates JUnit tests for classes. Randoop generates tests using the feedback-directed random test technique, which is a pseudo-random technique that create sequences of invocations of methods for the class under test. To create assertions, the tool executes the sequences and capture the behavior from the results. The tool can be applied to find defects in the class under test, but also to create regression test suites.

Effectiveness of Randoop has been confirmed through empirical studies. For instance, the tool uncovered unknown defects in widely-used libraries, such as Sun's and IBM's JDKs and a core .NET component. Also, it has been used in industry, for example, at ABB corporation [4]

2.2.2 EvoSuite. A search-based tool [26] that uses a genetic algorithm to generate test suites for Java classes automatically [2, 25]. For each class, the tool automatically produces a JUnit Test Suite by maximizing coverage. As input the tool receives the name of the class under test and its full classpath (where the bytecode of the class and its dependencies can be found).

EvoSuite can be used from the command line and through plugins for common software development infrastructure in industrial Java projects [17] such as IntelliJ IDEA [8], Jenkins CI [9], or Maven [12].

The effectiveness of EvoSuite has been evaluated on open source as well as industrial software in terms of code coverage, fault detection effectiveness, and effects on developer productivity [27]. In the first two, the fourth and fifth editions of the unit testing tool competition at the International Workshop on Search-Based Software Testing (SBST), EvoSuite ranked first, for achieving the highest overall score among the competing unit test generation tools [27].

3 EMPIRICAL STUDY SETUP

The unit testing practice still demands effective ways of generating test suites that, among other desirable features, are capable of detecting as many faults as possible. On the one hand, MTSs are more common in practice despite the costs and challenges to build them. On the other hand, ATSs are less costly to produce but so far have achieved low adoption, mainly due to current limitations. These bring us to our focus question: *how effective are MTSs and ATSs to cover code and detect faults?*

This section describes an empirical study that investigates this question. In the sequel, we present the detailed research questions, the subject programs, the metrics, the mutation testing tool and mutation operators, and an overview of the experimental procedure.

3.1 Research Questions

The following questions guide this study:

RQ1: How non-deterministic are the test case generation tools?
RQ2: Which test suite is more effective in terms of line coverage?
RQ3: Which test suite is more effective in terms of strong mutation coverage?
RQ4: Which test suite is more effective in terms of detecting certain mutations?
RQ5: Is there a correlation between the quantity of test cases in a test suite and its capability of mutant detection?

3.2 Subject Programs

We selected the programs to compose our experiment based on the following requirements:

(1) The project is built with Maven [12];
(2) The project has a manually written test suite;
(3) It is possible to generate test cases for the project using both Randoop [4] and EvoSuite [2];
(4) It is possible to generate mutants of the project using PITest [3].

We evaluated the above requirements for all 43 projects from the Apache Commons repository [5]. The 10 projects that suit the requirements are described in Table 1.

Table 1: Subject programs used in our study.

Project	Version	# of Classes	LOC
BCEL	6.3	426	30812
CLI	1.4	25	2790
Codec	1.12	86	8325
CSV	1.6	14	1742
Email	1.5	21	2815
FileUpload	1.4	47	2425
Imaging	1.0	403	32607
Lang	3.8.1	238	27646
Statistics	1.0	35	2665
Validator	1.6	73	7409
Total	—	1368	119236

3.3 Evaluation Metrics

One of the key challenges of developers when testing code is determining a test suite's quality [30]. The most popular method to predict fault detection capability based only on the test suite itself and the current version of the software under test is the use of code coverage criteria [14]. Code coverage describes structural aspects of the executions of a software under test performed by a test suite. For example, statement coverage indicates which statements in a program's source code were executed, branch coverage indicates which branches were taken, and path coverage describes the paths explored in a program's control flow graph [30].

Mutation analysis is another popular technique to assess the quality of test suites [24, 33]. The idea behind mutation testing is that small syntactic changes are inserted in the original program by applying mutation operators to create faulty programs called mutants. Mutants caught by a test suite are said to be killed. The intuition is that a test suite that kills more mutants is adequate to detect defects when they occur [35].

In this paper, we evaluate the test suites in terms of line coverage and strong mutation coverage.

3.4 Mutation Testing: Tool and Operators

We use PITest [3, 22], a mutation testing tool for Java, to conduct our study. PITest is considerably fast as it manipulates bytecode and runs only the tests that have chances to kill the used mutants (i.e., the tests that execute the instruction where the mutant is located). PITest's major advantage is that it is robust, easy to use, and well-integrated with development tools [22]. PITest has a set of mutation operators available. By default, there are seven mutation operators enabled (See Table 2). These operators are designed to generate hard to kill mutants, and generating a minimal amount of equivalent mutants [16]. In our study, we use the seven mutation operators enabled by default in PITest.

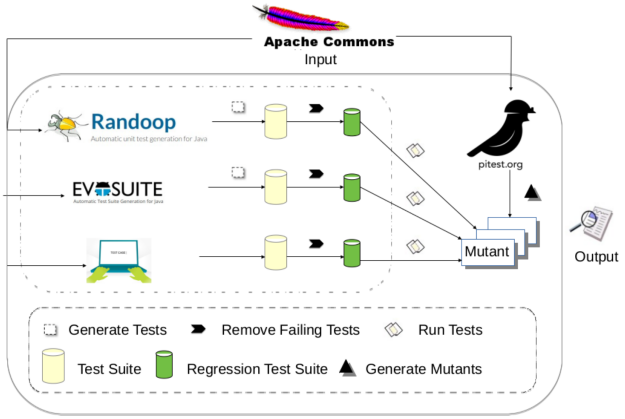

Figure 1: Overview of the experimental procedure.

3.5 Experiment Procedure

Figure 1 presents an overview of our study procedure. For each project in Table 1, we generated 10 test suites with Randoop [4] (version 4.1.1) and 10 test suites with Evosuite [2] (version 1.0.6), using their default configuration, except for the following argument changes: Evosuite's `separateClassLoader = false` to avoid conflicts with PITest's bytecode instrumentations; Randoop's `flaky-test-behavior = DISCARD` to remove flaky tests, which are tests that can intermittently pass or fail even for the same code version [39]. We also changed Randoop's `randomseed` in every execution, using integers generated by a pseudo random integer generator, to produce different test suites, as Randoop is deterministic by default [4].

The complete ATSs, and also the MTSs present in the projects, have tests that fail in the production code version considered in the study. Since our study focus on regression test suites as well as this is also a requirement of PITest, we manually removed all the failing tests from all the test suites.

To ensure that we had reliable tests, we ran PITest (version 1.4.5), which is deterministic, for each regression test suite, 10 times to look for flaky tests. We could only observe negligible differences for the ATSs and a slight difference for the MTS of *Statistics* and the fourth test suite generated by EvoSuite for the *CLI* project. According to our analysis, this difference has not impacted general conclusions reported for the MTSs compared to the ATSs. In any case, we used the mean of the 10 executions of PITest's outputs, including line coverage, to answer from RQ2 onward.

4 EMPIRICAL STUDY: RESULTS AND ANALYSIS

In this section, we present the answer to each research question, in turn, indicating how the results answer each. Furthermore, we discuss the results focusing on examples collected from data. All

Table 2: PITest's mutation operators enabled by default.

Operator	Description	Example
Conditionals Boundary	Replaces one relational operator instance with another operation	$< \rightarrow \leq$
Increments	Replaces increments with decrements and vice versa	$++ \rightarrow --$
Invert Negatives	Removes the negative from a variable	$-a \rightarrow a$
Math	Replaces binary arithmetic operations with another operation	$+ \rightarrow -$
Negate Conditionals	Negates one relational operator	$== \rightarrow !=$
Return Values	Replaces the return values of method calls	return 0 → return 1
Void Method Calls	Removes method calls to void methods	void m() →

Table 3: Standard deviation of the line coverages achieved by ten test suites generated by Randoop and EvoSuite for each subject program.

Project	Randoop	EvoSuite
BCEL	**0.054268**	0.038281
CLI	**0.073278**	0.052409
Codec	0.006871	**0.011868**
CSV	0.018792	**0.047030**
Email	0.015281	**0.033940**
FileUpload	0.006565	**0.050808**
Imaging	0.022082	**0.087673**
Lang	0.033009	**0.033074**
Statistics	**0.064588**	0.033372
Validator	0.022523	**0.045844**

Table 4: Standard deviation of the mutation coverages achieved by ten test suites generated by Randoop and EvoSuite for each subject program.

Project	Randoop	EvoSuite
BCEL	0.029402	**0.031350**
CLI	0.047546	**0.144000**
Codec	0.017999	**0.045306**
CSV	0.025489	**0.065018**
Email	0.011073	**0.038507**
FileUpload	0.006270	**0.054695**
Imaging	0.011942	**0.073979**
Lang	**0.032860**	0.028388
Statistics	**0.098861**	0.018868
Validator	0.010764	**0.052549**

data from our study, the statistical analysis, and reproducibility instructions can be found in our notebook [46].

4.1 RQ1: Tools Non-Determinism

To investigate the non-determinism in the generation techniques of Randoop and Evosuite, we calculate the mean, median, and standard deviation of the line coverage and mutation coverage, considering one execution of PITest for each test suite, of the ten test suites by each project. Tables 3 and 4 show the computed standard deviations. We have that, for line coverage, the test suites generated by EvoSuite have a greater standard deviation than those test suites generated by Randoop for seven out of the ten subject programs. The three projects whose test suites generated by Randoop have a greater standard deviation than those generated by EvoSuite, in terms of line coverage, are *BCEL*, *CLI*, and *Statistics* (values are highlighted in Table 3). Moreover, for mutation coverage, the test suites generated by EvoSuite have a greater standard deviation than those generated by Randoop for eight out of the ten subject programs. The two subject programs whose test suites generated by Randoop have a greater standard deviation than those generated by EvoSuite, in terms of mutation coverage, are *Lang* and *Statistics* (values are highlighted in Table 4).

> **RQ1**: *Both tools generate test suites for each project with different line and mutation coverage. EvoSuite tends to show more non-deterministic behavior than Randoop.*

4.2 RQ2: Line Coverage

Figure 2 presents the line coverages achieved by ATSs generated by Randoop, EvoSuite, and MTSs. We compared them using the Wilcoxon test, and we found that, with 95% confidence, the MTSs achieve greater line coverage than the ATSs (p-value = 0.000976). However, we did not find a statistically significant difference between the line coverages achieved by the test suites generated by Randoop and EvoSuite, with 95% confidence (p-value = 0.3477).

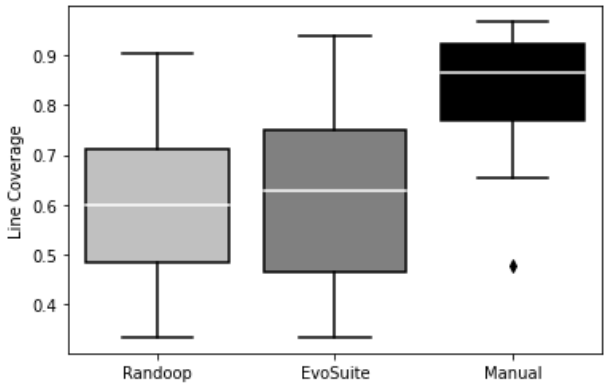

Figure 2: Line Coverages achieved by the ATSs generated by Randoop, EvoSuite, and MTSs.

Figure 3 shows the line coverages obtained by all ATSs generated by EvoSuite and Randoop for each of the projects. EvoSuite is more effective in six out of 10 projects with no overlaps between the boxes, whereas Randoop is more effective in three of them. Moreover, the *CSV* and *Email* projects are the most challenging ones for EvoSuite when compared to Randoop, whereas the *Imaging* and *CLI* projects

are the most challenging ones to Randoop. Despite all of the recent efforts to improve the test case generation techniques implemented by the tools, the classes in these projects may present constructions that are still hard to explore [13].

> **RQ2**: *MTSs achieve better line coverage. There is no statistical difference between the ATSs generated by Randoop and EvoSuite.*

4.3 RQ3: Mutation Coverage

Figure 4 presents the mutation coverages achieved by ATSs generated by Randoop, EvoSuite, and MTSs. We compared the mutation coverages achieved by the test suites using the Wilcoxon test, and we found that, with 95% confidence, the MTSs achieve greater mutation coverage than the ATSs (p-value = 0.000976). Moreover, we found that, with 95% confidence, test suites generated by EvoSuite achieve higher mutation coverage than those generated by Randoop (p-value = 0.00293).

Figure 5 shows the mutation coverage obtained by all ATSs generated by EvoSuite and Randoop for each of the projects. We can see that EvoSuite is more effective in five out of 10 projects with no overlaps between the boxes, whereas Randoop is more effective in three of them. Moreover, the *CSV* and *Email* projects are still the most challenging ones for EvoSuite when compared to Randoop, but the difference is not as high as when considering line coverage. Furthermore, for the *Validator* project, EvoSuite presents a lower mutation coverage, whereas EvoSuite's line coverage surpasses Randoop. These facts can be explained by whether key assertions are present in test cases: code coverage is not always correlated with fault detection capability [21, 28].

> **RQ3**: *MTSs achieve better mutation coverage, followed by the ATSs generated by EvoSuite.*

4.4 RQ4: Mutation Operator Coverage

Figure 6 shows the coverages achieved by the test suites, generated with the different test generation techniques, for each mutation operator applied. For each operator, we compared, using the Wilcoxon test, the coverages achieved by the test suites. We found that, with 95% confidence, the MTSs are more effective than those generated by Randoop and EvoSuite in detecting six out of the seven mutation types applied (p-value < 0.05). We only found that there is no difference statistically significant between the MTSs and the ATSs in detecting the *Invert Negatives* mutation (p-value > 0.05). It is important to remark that PITest could only apply this operator in three projects: *Lang*, *Imaging* and *Statistics*. Together, these projects account for 676 of the 1368 classes considered in the study.

Using the Wilcoxon test with 95% confidence, we found that test suites generated by EvoSuite are more effective than those generated by Randoop in detecting three mutation types (p-value < 0.05): *Conditionals Boundary*, *Negate Conditionals*, and *Math*. Moreover, we found that there is no statistically significant difference between the test suites generated by EvoSuite and Randoop in detecting the other four mutation types applied (p-value > 0.05): *Return Values*, *Void Method Calls*, *Increments*, and *Invert Negatives*.

> **RQ4**: *Except for the* Invert Negatives *operator, MTSs are more effective than ATSs when considering individual mutation operators. EvoSuite ATSs are more effective than Randoop ones when considering the* Conditionals Boundary, Negate Conditionals, *and* Math *operators.*

4.5 RQ5: Test Suite Size and Mutation Coverage

Figure 7 presents the correlations that we found between the number of test cases and the mutation coverage achieved by the test suites. We used the *Spearman rank correlation* coefficient to verify whether there is a correlation between the number of test cases and the mutation coverage of each test suite. Table 5 presents the correlation coefficients (ρ) computed. There is a large (ρ is over 0.5) and very large (ρ is over 0.7) positive correlation between MTSs' size and mutation coverage and between size and line coverage, respectively. Moreover, there is a small (ρ is over 0.1) positive correlation between EvoSuite ATSs size and mutation coverage and between size and line coverage. However, no correlation can be found for Randoop ATSs since the *p-value* obtained is over 0.05. In this case, the correlation coefficient has no statistical significance.

	Mutation		**Line Coverage**	
	p-value	**Spearmen ρ**	**p-value**	**Spearmen ρ**
EvoSuite	0.02955	0.2177306	0.0374	0.2084624
Randoop	0.5403	-0.6195054	0.9788	-0.002691374
Manual	0.0288	0.6848485	0.0158	0.7333333

Table 5: Spearman correlation value (ρ) between test suite size and mutation score, and between test suite size and line coverage.

Table 6 presents the total number of test cases of the test suites for each program. The test suites generated by Randoop have the largest number of test cases, whereas the MTSs have the smallest amount of test cases. These numbers may explain the correlations found. The size of the test suite does not necessarily imply effectiveness. As MTSs' sizes have relevant positive correlations with their effectiveness, each test case may add considerable value to the test suite, whereas ATSs' test cases tend to be more redundant.

Table 6: Test suites' size (number of test cases) generated for each project by each unit test generation technique.

Project	Randoop	EvoSuite	Manual
BCEL	5023	3435	118
CLI	257	455	409
Codec	3200	1254	901
CSV	2209	102	321
Email	3211	134	190
FileUpload	3676	332	82
Imaging	2444	3501	91
Lang	2328	3101	4118
Statistics	2331	749	386
Validator	2716	1092	536
Total	27399	14159	7152

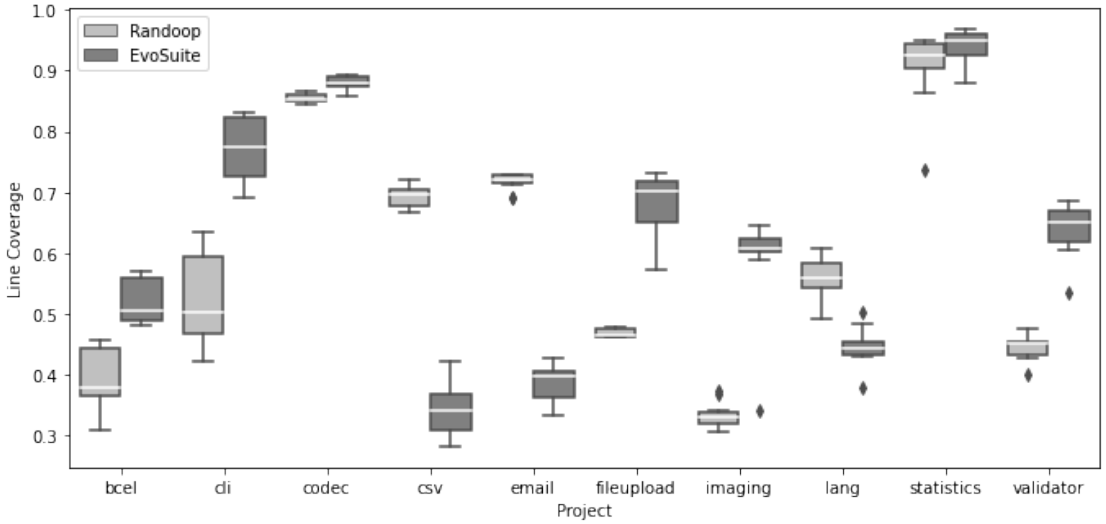

Figure 3: Line Coverage variation from 10 ATSs generated with Randoop and EvoSuite for each one of the 10 subject projects.

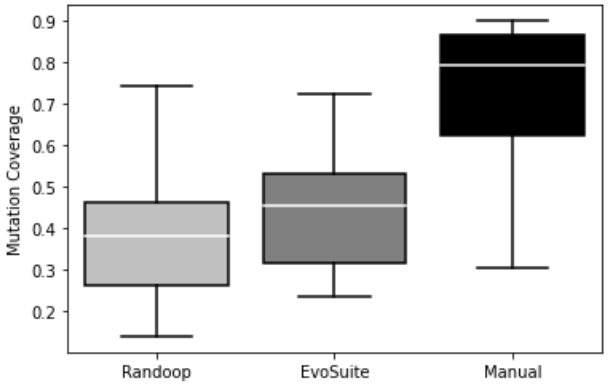

Figure 4: Mutation Coverages achieved by the ATSs generated by Randoop, EvoSuite, and MTSs.

RQ5: *There is a large positive correlation between MTSs' size and mutation coverage. Also, there is a small positive correlation for EvoSuite ATSs', but no correlation can be found for Randood ATSs'.*

4.6 Discussion

According to other studies presented in the literature that corroborate with our findings, current test case generation tools are limited to detect faults [13, 45]. The main challenges are difficulty to cover code or produce assertions that require the creation of complex objects, dealing with external dependencies, and private methods/-fields.

As an example from our study, consider the code excerpt in Listing 1 from the *AlreadySelectedException* class of the *CLI* project. The following mutation could not be even covered by either EvoSuite or Randoop test suites. Note that *option* is a private field. On the other hand, the MTS killed the mutant.

```
getOption : mutated return of Object value for
org/apache/commons/cli/AlreadySelectedException::getOption to
( if (x != null) null
  else throw new RuntimeException )
```

Listing 1: When the MTSs are more effective and ATSs do not cover the mutant

```
1  public Option getOption() {
2      return option;
3  }
```

Another interesting example is the code excerpt from the *AmbiguousOptionException* class of the *CLI* project presented in Listing 2. For the *createMessage* method, the PITest tool created three mutants: one *Negated Conditional* for the **while** and **if** command conditions (lines 11 and 15) and a mutated *Return Value* similar to the one presented before (line 20). Randoop ATSs killed the first negated conditional (at line 11), did not cover the second (at line 15), and covered but did not kill the third (line 20). On the other hand, Evosuite ATSs did not cover any of the mutants, whereas the MTS covered but did not kill the mutants. Note that *createMessage* is a private method, and *matchingOptions* can be a tricky object to handle for data generation.

Listing 2: When Randoop ATSs are more effective

```
1
2  private static String createMessage (
3      final String option,
4      final Collection<String> matchingOptions) {
5      final StringBuilder buf =
6          new StringBuilder("Ambiguous option: '");
7      buf.append(option);
8      buf.append("'  (could be: ");
9      final Iterator<String> it =
10         matchingOptions.iterator();
11     while (it.hasNext()) {
12         buf.append("'");
13         buf.append(it.next());
```

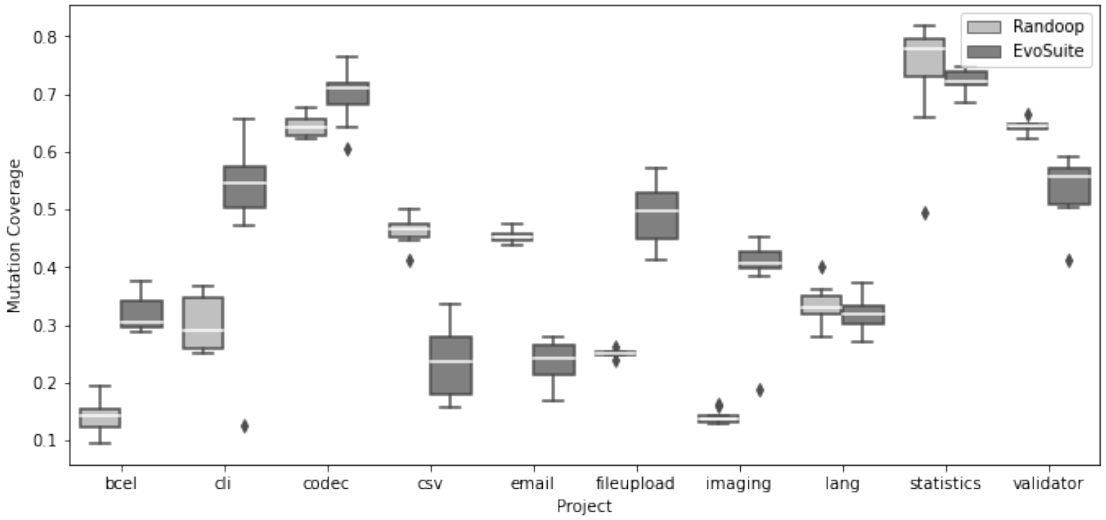

Figure 5: Mutation Coverage variation from 10 ATSs generated with Randoop and EvoSuite for each one of the 10 subject projects.

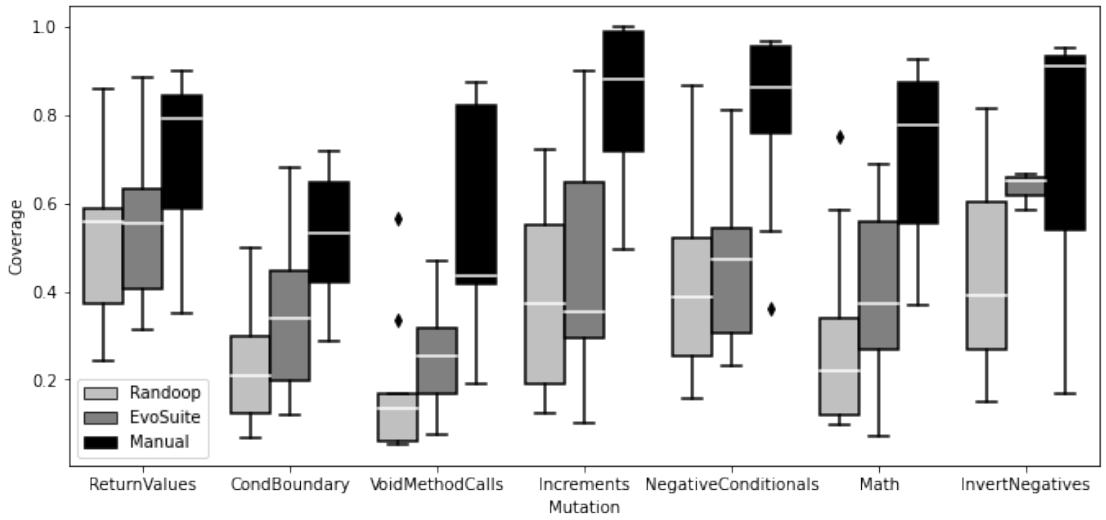

Figure 6: Coverages achieved by the ATSs generated by Randoop, EvoSuite, and MTSs, for each mutation operator applied.

```
14          buf.append("'");
15          if (it.hasNext()) {
16              buf.append(", ");
17          }
18      }
19      buf.append(")");
20      return buf.toString();
21  }
```

Listing 3 presents a code excerpt from the *ArrayElementValue* class of the *BCEL* project. Only the EvoSuite ATSs killed the six mutants created by the PITest tool for the *toString* method (at lines 4, 6, and 11). The Randoop ATSs and the MTS did not even cover the mutants.

Listing 3: When EvoSuite ATSs are more effective

```
1  public String toString() {}
2      final StringBuilder sb = new StringBuilder();
3      sb.append("{");
4      for (int i = 0; i < evalues.length; i++)    {
5              sb.append(evalues[i]);
6              if ((i + 1) < evalues.length) {
7                  sb.append(",");
8              }
9          }
10         sb.append("}");
11         return sb.toString();
12     }
```

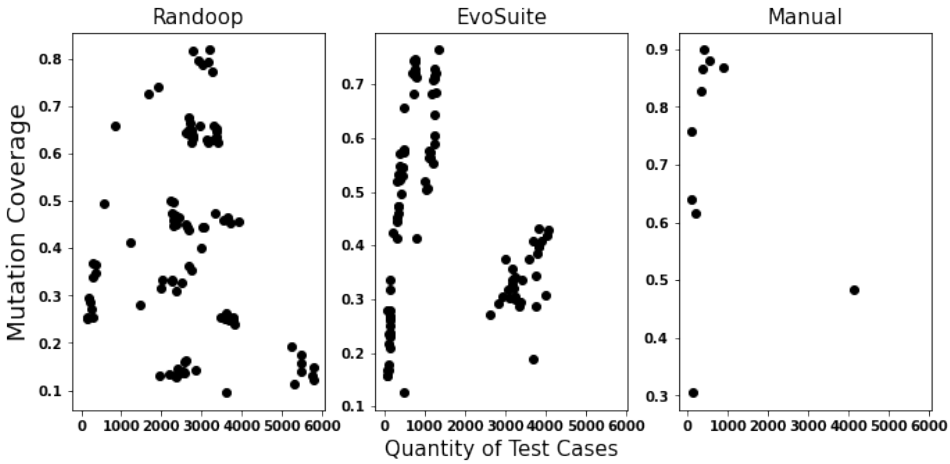

Figure 7: Correlation between the number of test cases and the mutation coverage achieved by the ATSs, generated by Randoop, EvoSuite, and MTSs.

As we illustrate through the above examples, despite the most common situation where MTSs achieved higher coverage and mutation score in our study, MTSs are not always more effective than ATSs [19]. Also, they are hard and error-prone to construct and may not be as effective as ATSs to detect tricky faults that require a more elaborate code analysis [19].

Moreover, evolution can improve test suites coverage whereas also degrade them. From the number of years and the number of releases between the oldest available release and the release considered in our study (for 9 out of 10 projects), we cannot conclude whether evolution is a determinant factor to the MTSs effectiveness. We cannot observe a correlation. For instance, the oldest projects are *Codec* and *Lang* (16 years of development). Their test suites achieve 92% and 95% line coverage, respectively, and 87% mutation coverage. The youngest project is *CSV* (4 years of development). Its test suites achieve 92% line coverage and 83% mutation coverage.

Despite the limitations presented by both ATSs and MTSs, it is important to remark that practices to construct such suites have improved in the past years. Efforts have been pursued to make writing test suites more cost-effective, for instance, by investigating strategies and standards to enhance the quality in MTSs along with productivity [29, 37]. Likewise, ATSs have achieved steady progress in improving their effectiveness as annually reported by the Java Unit Testing Competition [36].

Furthermore, strategies for creating MTS and ATSs have a crucial difference that suggests they might be complementary [44, 48]. While ATSs creation relies primarily on white-box techniques, MTSs creation usually relies on testers' intuition and target expected behavior in a black-box fashion. Therefore, both efforts by following potentially different points of view may be necessary to achieve high effectiveness in unit testing. From the examples we show in this section, there are cases where MTSs and ATSs are complementary. Considering the 18 mutants created for the *ArrayElementValue* class of the *BCEL* project, EvoSuite ATSs killed 12, the MTS killed 5, and Randoop ATSs killed 1. Together EvoSuite ATSs and the MTS killed 15 mutants. Moreover, for the *AmbiguousOptionException*, together the Randoop ATSs, and the MTS killed two out of the four mutants created.

5 THREATS TO VALIDITY AND LIMITATIONS

There are several threats to the validity of this work. First, to compose our subject projects set, we selected programs written in Java, built with Maven, with manually written test suites, and also projects that both Randoop and EvoSuite are able to generate test cases. Although the projects that we chose range from 1742 to 32607 lines of code, and vary on their purpose, those projects may represent a specific set of applications (e.g. well structured and tested Java programs).

Second, we used almost all the default configuration values of the tools that automatically generate test cases. While tuning can have an impact on the performance of a search algorithm, in the context of test data generation, it is difficult to find good settings that significantly outperform the "default" values suggested in the literature [18, 38].

Third, the determination of the quality of software tests can be considered a subjective measurement. Although mutation score and coverage are two ways to measure test suite quality, that does not consider the readability of the test cases. If the developers who need to view tests in order to diagnose defects cannot understand what the tests do, then the human time and effort could be substantially increased [38]. Moreover, although mutation analysis can be used as a proxy for fault detection, it is restricted to the set of mutation operators available and it may not represent typical defects for the software at hand. Therefore, we could also have considered a set of real faults and its detectability by the test suites (ATS and MTS)

Fourth, the tool used for mutation analysis also leads to a potential internal threat to validity. PITest [3], a practical mutation testing tool for programs written in Java, was used. However, other options, such as MuJava [40], could be used in future comparisons.

Finally, MTSs may contain both unit and integration test cases since developers do not often distinguish precisely between them.

However, a recent study presented by Trautsch *et al.* [47] shows that there is no significant difference between the types of defects detected by the unit and integration test cases of a test suite, although the unit ones are more prone to detect hard to kill defects.

6 RELATED WORK

Bacchelli *et al.* [19] conducted a first exploratory study to empirically understand the differences between manual and automatic unit test generation. The authors considered test effectiveness in terms of code coverage, mutation score, as well as error finding. In particular, they investigated the FREENET project [7] as the subject of their study. Since FREENET used to lack a significant test suite, the authors first manually created tests for a subset of 15 classes. Afterward, the authors applied automatic test generation tools to the 15 classes and compared the automatically generated tests against the manually created ones. They selected three tools available at that time, namely Randoop [4], and JUnit Factory [11], designed for regression testing, and JCrasher [23], designed for defect revelation. The study reported that automatically generated tests could achieve a higher code and mutation coverage than the manually generated ones. The former could also generate unexpected scenarios that lead to the identification of faults, partially overlapping with those found manually.

Fraser *et al.* [28] conducted a human study aimed at understanding the practical value of automated testing tools. The experiment was organized in two studies: In the first one, they asked the participants to write tests for four different Java classes manually; in the second one, they asked them to generate test suites using EvoSuite for the same subjects. The authors showed that test case generation tools could achieve higher code coverage compared to manually created tests, while automatically generated tests resulted as less effective in detecting faults.

Similarly, Shamshiri *et al.* [45] investigated the fault detection ability of automatically generated tests compared to the manually written ones. The study involved three testing tools ran over the Defects4j [34] dataset (5 subject programs). In particular, they exercised the tools in a regression testing scenario. The test suites were generated on the fixed versions of the code and then executed against the buggy versions. Their findings were in line with Fraser *et al.* [28]: The tools were only able to find about half of the bugs.

Grano *et al.* [31] compared the readability of manually and generated tests. To conduct their study, Grano *et al.* [31] relied on EvoSuite to automatically generate test cases for three subject programs. They found that automatically generated test cases are significantly less readable than manually written ones. We do not investigate the readability of the tests used in our study.

Serra *et al.* [44] conducted a partial replication of the study of Bacchelli *et al.* [19], with the goal of evaluating the improvement achieved by automated test generation techniques over the course of the last ten years. In their study, Serra *et al.* [44] used Randoop, EvoSuite, and JTExpert to generate test cases automatically. Serra *et al.* [44] found that current automatic test case-generation tools can optimize coverage and mutation scores more than manually written tests.

Ramler *et al.* [42] carried out a study addressing how automated testing tools compare to manual testing. In their study, Ramler *et al.* [42] compared the fault detection of manually written test cases with randomly-generated test cases, using Randoop, for a Java collection class library containing 35 seeded defects. Fault detection rates were found to be similar, although the techniques revealed different kinds of faults.

Most close to our results, the study conducted by Vincenzi *et al.* [48] investigates the adequacy, effectiveness, and cost of manually generated test sets versus automatically generated test sets for Java programs. Vincenzi *et al.* [48] used 32 simple subject programs, implemented by 1 to 3 classes (1.5 on average), in their study. The authors found that, in general, manual test sets determine higher statement coverage and mutation score than automatically generated test sets. Moreover, Vincenzi *et al.* [48] recognized that the automatically generated test sets are complementary to the manual test set. Manual with automated test sets overcame more than 10%, on average, statement coverage and mutation score, compared to the rates of the manual test set while keeping a reasonable cost.

To the best of our knowledge, previous studies have considered a limited amount of classes or subject programs, whereas we used a set of 10 subject programs with 1368 classes in total. Also, we use the MTSs created by developers not involved in the study. Furthermore, we focus on regression test suites, and our results differ from previous studies [19, 28, 42, 44, 45]. Our study provides new evidence of the effectiveness of MTSs and ATSs, motivating further research in the area.

7 CONCLUSION

In this work, we investigate the effectiveness of automatically generated test suites (ATSs) and manually written test suites (MTSs) in terms of line coverage and mutation coverage. We focus on ATSs generated by the Randoop and EvoSuite tools and real MTSs from 10 open-source projects. Additionally, we empirically compare the randomness of the test suites generated by Randoop and EvoSuite. We also investigate whether there is a correlation between the test suite size and its ability to detect faults.

Our findings revealed that current automatic test case generation tools are not as effective as manually writing tests in terms of line coverage and mutation coverage. Moreover, the test suite size is not an indicator of effectiveness, especially for Randoop ATSs.

The results of our study differ from previous studies in the sense that MTSs are more effetive regarding both line and mutation coverage. However, they also present evidence that ATSs and MTSs may be complementary. To the best of our knowledge, our study is the largest so far handling manual test suites created by developers not involved in the study and focusing on regression test suites.

As future work, we intend to make our results more generalizable by considering more subject programs, unit test generation tools, evaluation metrics, and mutation testing tools. We also plan to check whether the results found also hold for other programming languages. Finally, results encourage further research on the complementary nature of ATSs and MTSs.

ACKNOWLEDGMENTS

This work was supported by the National Council for Scientific and Technological Development (CNPq)/Brazil (Process 437029/2018-2).

First author was supported by PIBIC/CNPq/Brazil. Second author was supported by CNPq/Brazil (Process 311239/2017-0).

REFERENCES

[1] 2014. CoView. http://www.codign.com Accessed: 2014-06-01.

[2] 2019. EvoSuite: Automatic Test Suite Generation for Java (2019). http://www.evosuite.org/evosuite/ Accessed: 2019-09-04.

[3] 2019. PITest Mutation Testing Tool for Java (2019). http://pitest.org/ Accessed: 2019-09-04.

[4] 2019. Randoop: Automatic unit test generation for Java (2019). https://randoop.github.io/randoop/ Accessed: 2019-09-04.

[5] 2020. Apache Commons. https://commons.apache.org/ Accessed: 2020-04-28.

[6] 2020. CodePro Tutorial. https://self-learning-java-tutorial.blogspot.com/2015/06/codepro-tutorial.html Accessed: 2020-05-02.

[7] 2020. The Freenet Project: a peer-to-peer software platform for censorship-resitant communication. https://freenetproject.org/ Accessed: 2020-04-25.

[8] 2020. IntelliJ. https://www.jetbrains.com/idea/ Accessed: 2020-04-29.

[9] 2020. Jenkins. https://www.jenkins.io/ Accessed: 2020-04-29.

[10] 2020. JUnitDoclet. http://www.junitdoclet.org/ Accessed: 2020-05-02.

[11] 2020. JUnitFactory. http://www.junitfactory.com/ Accessed: 2020-04-25.

[12] 2020. Maven. https://maven.apache.org/ Accessed: 2020-04-28.

[13] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. 2017. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 263–272.

[14] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing* (1 ed.). Cambridge University Press, USA.

[15] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. [Jenny Li], and Hong Zhu. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978 – 2001. https://doi.org/10.1016/j.jss.2013.02.061

[16] Michael Andersson. 2017. *An Experimental Evaluation of PIT's Mutation Operators*. Bachelor thesis. Umeå University.

[17] Andrea Arcuri, José Campos, and Gordon Fraser. 2016. Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Chicago, IL, USA, 401–408.

[18] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18 (2013), 594–623.

[19] Alberto Bacchelli, Paolo Ciancarini, and Davide Rossi. 2008. On the Effectiveness of Manual and Automatic Unit Test Generation. In *2008 The Third International Conference on Software Engineering Advances*. IEEE, Sliema, Malta, 252–257.

[20] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman. 2019. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Transactions on Software Engineering* 45, 3 (2019), 261–284.

[21] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman. 2017. An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 597–608.

[22] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707

[23] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An Automatic Robustness Tester for Java. *Softw. Pract. Exper.* 34, 11 (Sept. 2004), 1025–1050. https://doi.org/10.1002/spe.602

[24] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41.

[25] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 416–419.

[26] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[27] Gordon Fraser, José M. Rojas, José Campos, and Andrea Arcuri. 2017. EvoSuite at the SBST 2017 Tool Competition. In *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*. IEEE, Buenos Aires, Argentina, 39–42.

[28] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 24, 4, Article 23 (Sept. 2015), 49 pages. https://doi.org/10.1145/2699688

[29] Vahid Garousi and Barış Küçük. 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software* 138 (2018), 52 – 81. https://doi.org/10.1016/j.jss.2017.12.013

[30] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code Coverage for Suite Evaluation by Developers. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 72–82. https://doi.org/10.1145/2568225.2568278

[31] Giovanni Grano, Simone Scalabrino, Harald C. Gall, and Rocco Oliveto. 2018. An Empirical Investigation on the Readability of Manual and Generated Test Cases. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 348–351. https://doi.org/10.1145/3196321.3196363

[32] David Honfi and Zoltán Micskei. 2019. Classifying generated white-box tests: an exploratory study. *Software Quality Journal* 27, 3 (2019), 1339–1380. https://doi.org/10.1007/s11219-019-09446-5

[33] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sep. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[34] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[35] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proc. of the 22nd ACM SIGSOFT Int. Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[36] F. Kifetew, X. Devroey, and U. Rueda. 2019. Java Unit Testing Tool Competition - Seventh Round. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*. 15–20.

[37] Pavneet Singh Kochhar, Xin Xia, and David Lo. 2019. Practitioners' views on good software testing practices. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 61–70. https://doi.org/10.1109/ICSE-SEIP.2019.00015

[38] Jeshua S. Kracht, Jacob Z. Petrovic, and Kristen R. Walcott-Justice. 2014. Empirically Evaluating the Quality of Automatically Generated and Manually Written Test Suites. In *2014 14th Int. Conf. on Quality Software*. IEEE, Dallas, USA, 256–265.

[39] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920

[40] Yu-Seung Ma, Jeff Offutt, and Yong Kwon. 2006. MuJava: a mutation system for java. 827–830. https://doi.org/10.1145/1134425

[41] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. ACM, New York, NY, USA, 815–816.

[42] Rudolf Ramler, Dietmar Winkler, and Martina Schmidt. 2012. Random Test Case Generation and Manual Unit Testing: Substitute or Complement in Retrofitting Tests for Legacy Code?. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, Cesme, Izmir, Turkey, 286–293.

[43] R. Ramler, K. Wolfmaier, and T. Kopetzky. 2013. A Replicated Study on Random Test Case Generation and Manual Unit Testing: How Many Bugs Do Professional Developers Find?. In *2013 IEEE 37th Annual Computer Software and Applications Conference*. 484–491.

[44] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall, and A. Bacchelli. 2019. On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, Montreal, QC, Canada, Canada, 121–125.

[45] Sina Shamshiri, René Just, José M. Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *2015 30th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*. IEEE, Lincoln, USA, 201–211.

[46] Beatriz Souza and Patrícia Machado. 2020. A Large Scale Study On the Effectiveness of Manual and Automatic Unit Test Generation (Artifact). https://colab.research.google.com/drive/1cjDv62j1aDAYvbeT0NbWSO1s_qlfLhub?usp=sharing Accessed: 2020-05-02.

[47] Fabian Trautsch, Steffen Herbold, and Jens Grabowski. 2020. Are unit and integration test definitions still valid for modern Java projects? An empirical study on open-source projects. *Journal of Systems and Software* 159 (2020), 110421. https://doi.org/10.1016/j.jss.2019.110421

[48] Auri M. R. Vincenzi, Tiago Bachiega, Daniel G. de Oliveira, Simone do Rocio Senger de Souza, and José Carlos Maldonado. 2016. The complementary aspect of automatically and manually generated test case sets. In *Proc. of the 7th Int. Workshop on Automating Test Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2016, Seattle, WA, USA, November 18, 2016*, Tanja E. J. Vos, Sigrid Eldh, and Wishnu Prasetya (Eds.). ACM, 23–30. http://dl.acm.org/citation.cfm?id=2994295