How far are we from testing a program in a completely automated way, considering the mutation testing criterion at unit level?

Filipe Santos Araujo
Computing Department – UFSCar
São Carlos, São Paulo – Brazil
filipe.santos.araujo@gmail.com

Auri Marcelo Rizzo Vincenzi
Computing Department – UFSCar
São Carlos, São Paulo – Brazil
auri@ufscar.br

ABSTRACT

Testing is a mandatory activity to guarantee software quality. Not only knowledge about the software under testing is required to generate high-quality test cases, but also knowledge about the business rules implemented in software product to cover more than 80% of its source code. Therefore, we investigate in this study the adequacy, effectiveness, and cost of smart and random automated generated test sets for Java programs. We observed that the smart generated test sets, in general, are more adequate and less expensive than random generated tests, but regarding effectiveness, random generated test are more efficient. Moreover, we observed that smart automated test sets are complementary between them, and we explored if random generated test sets could be complementary to smart automated test sets as well. When we combined smart generated test sets, we observed an increase of more than 8% in statement coverage and more than 15% in mutation score when compared to random generated test sets. However, when we added random generated test sets to previous combination of smart generated test sets, results show a lower increase of statement coverage and mutation score, while increasing considerably the test set generation cost. Therefore, we advocate that the use of random testing should be integrated with smart generated tests only with a minimization strategy to avoid redundant test sets, keeping the cost reasonable.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**; **Software verification and validation**.

KEYWORDS

Software Testing, Automated Test Data Generator, Coverage Testing Mutation Testing, Test Set Combination

ACM Reference Format:
Filipe Santos Araujo and Auri Marcelo Rizzo Vincenzi. 2020. How far are we from testing a program in a completely automated way, considering the mutation testing criterion at unit level?. In *19th Brazilian Symposium on Software Quality (SBQS'20), December 1–4, 2020, São Luís, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3439961.3439977

1 INTRODUCTION

Software quality is achieved by extensive testing activity. The cost spent to correct faults responsible for a failure tend to decrease significantly when it is detected during early stages of software development [5]. Usually, the process of testing begins with unit testing, followed by integration and then system testing. Therefore, the focus of this paper is unit testing.

The initial step of unit testing activity is the selection of test cases to evaluate each unit under testing. Such test set can be manually generated by the tester or automatically generated by automated test data generators (ATDG). In the latter, usually, the effort is spent on creating only the test data to be provided as input to software under testing. The reason is because ATDG has no knowledge about the business rules the software product is inserted and, consequently, ATDG is unlikely to generate the the expected output for each input.

Our research group submitted a project to FAPESP, aiming at providing a full automated mutation-based testing process and, based on previous experience/experiment [21], we need to combine ATDG to improve mutation score.

Therefore, this study discusses the results of an evaluation of four different ATDG and their effectiveness with respect to mutation testing. We carried out the experiment on a set of 33 software products which implement data structures and a piece of a compiler.

We used four different ATDG– EvoSuite [10], Palus [23], Randoop [15], and JTExpert [17] to generate test sets automatically. We use EvoSuite because it represents the state of the art of test data generator [10]. JTExpert [17] is another ATDG which uses search-based techniques for test data generation. We used Palus because it represents the so called Concolic Testing [23], providing one more feature for our ATDG set. Finally, we used Randoop [15] because it is a random ATDG that serves as baseline. Moreover, these ATDG generate test sets in JUnit format, making data collection automation easier.

The primary purpose of this paper is to investigate the quality of the ATDG test sets isolated and combined concerning:

- adequacy, based on the statement coverage criterion;
- effectiveness, based on mutation testing;
- cost, based on the number of generated test cases.

It is important to state that there are several ways to measure the cost, such as the time spent to generate test sets, the number of generated test sets or the time taken to set up ATDG. Since we are dealing with low complexity programs in this study, we decided to use the number of generated test sets as the cost metric. In future experiments we intend to explore the other cost metrics mentioned above.

We organized the paper as follows. In Section 2, we present the necessary background to understand this article. In Section 3 we discuss related works. In Section 4.1, we use the Goal Question Metric template to define the experiment we performed. In Section 4.2, we present the preparation for the experiment execution. In Section 5, we describe the experiment operational aspects. In Section 6, we introduce the data and analyze the obtained results. Finally, in Section 8, we draw the conclusion and point out future work.

2 BACKGROUND

A test case is defined as a tuple $\langle I, E_O \rangle$, where I is the input, also called test data, and E_O is the expected output concerning the unit specification. When we execute the unit with I it produces the resulting output, R_O. When $R_O = E_O$, the unit under testing behaves as specified. On the other hand, when $R_O \neq E_O$, a failure is detected by the test case.

However, it is difficult to identify E_O, in general, because this task demands human intervention. This fact makes the task unfeasible as the number of tests increases, because the tester have to decide if $R_O = E_O$ is true or false for each test execution.

Regarding ATDG, they generate input data I, but not the expected output data E_O. In general, E_O is defined later by the tester, who knows the unit specification and can infer E_O for each automatically generated I.

Therefore, in this study, we adopted the following strategy for the chosen ATDG: i) ATDG read the source or bytecode of a given unit under testing (UUT); ii) ATDG generate a set of input data SI for UUT; iii) ATDG run UUT with each $I \in SI$ and collect the resulting output R_O and; iv) ATDG assume R_O is the expected output E_O and create the test case $\langle I, R_O \rangle$.

Observe that, in this situation, all generated test case will pass in the current implementation of UUT, i.e., no test case will fail. These test cases, which assume the expected output as the current resulting output, are called by the authors of such tools as "regression test cases" [15].

Automatic test data generation is useful when it can generate a few test cases which improve the coverage of a given testing criterion. Moreover, the main advantages of using ATDG are that they can make the testing process faster, cheaper and reproducible because they can generate the same test set as many times we need very quickly. The primary disadvantage is that, in general, they are not able to generate test data to cover specific business rule, demanding human intervention in this case.

Among the many testing techniques and criteria existing in literature, mutation testing is widely used to validate the quality of generated test data. The idea behind mutation testing is the fact the programmers are aware of the common mistakes they make when developing software [7]. Mutation testing consists in/on creating faulty versions of the software under testing – called mutants – and then applying the generated test sets on these mutants. If all test sets passes on the original program, it is expected that they do not pass on the mutants. Therefore, we calculate the mutation score as show in Equation 1:

$$ms = \frac{dm}{am} \tag{1}$$

where ms is the mutation score, dm the dead mutants and am the total of generated mutants, excluding the equivalent ones.

According to the obtained results, ATDG test sets are complementary and should be used together to improve the test set quality.

3 RELATED WORK

Other researchers have already evaluated the relationship between test sets generated by ATDG in different contexts, using different set of tools and programs.

Leitner et al. [13] developed a tool, called AutoTest, which tries to combine the best practices of manual and automated tests. AutoTest assumes developers adopt the concept of design by contract during software development. Contracts are executable preconditions, postconditions, and invariants embedded in the software source code. Therefore, the tool uses the contract information to automatically generate test cases and also allows the tester to manually specify test cases for specific purposes or business rules, while automatically testing other software parts. The downside is that developers must include the contracts in the software source code to get all benefits from AutoTest tool.

Smeets and Simons [19] evaluated the quality of JUnit test set manually generated and test sets generated by two ATDG– Randoop and JWalk [18] – against mutants generated by MuJava [14]. They argued automatic test generators, in general, ensure completeness even in covering software parts where require more elaborated manual test sets. Also, both manually and automated test sets, achieved a mutation score below 70%, being considered incomplete on reaching satisfactory metrics.

Kracht, Petrovic, and Walcott-Justice [12] performed an empirical evaluation of manually versus ATDG generated test suites, aiming at reducing the cost developers have during testing activity. For that, they selected 10 programs [9] based on their size and manual JUnit tests availability. Regarding ATDG, they selected EvoSuite and CodePro and they evaluated the test sets based on branch coverage and mutation testing criteria. Results showed EvoSuite achieved higher branch coverage and mutation score compared to CodePro, but manually generated test sets obtained a higher strong correlation between branch coverage [16] and mutation score. Although they used MAJOR [11] as a mutation testing tool for supporting mutation testing, the results obtained are similar to the ones reported by Vincenzi et al. [21] (describe below) and in this paper.

Kracht, Petrovic, and Walcott-Justice [12] selected the set of program based on their size, using the lines of code (LOC) criteria/metric. Their large program with 18K LOC has an average Cyclomatic Complexity (CC) of 2.82, and the smallest program with 783 LOC has an average CC of 2.05. In our study, although we are using small programs regarding LOC, the average CC is 3.3.

Vincenzi et al. [21] compared the adequacy, effectiveness and cost of manually generated test sets to automatically generated test sets for Java programs. They combined test sets generated by three ATDG, EvoSuite, CodePro and Randoop, with manual generated test sets. As result they achieved, on average, more than 10%, statement coverage and mutation score, regarding adequacy and effectiveness, when compared to the rates of manual test set, keeping a reasonable cost.

The major difference between the work from Vincenzi et al. [21] and this one is that, in this work, we remove the manual generated test set aiming at investigating whether similar results of adequacy, effectiveness, and cost can be obtained without human intervention.

4 EXPERIMENT STUDY

In this section we present the steps adopted in the experiment: Experiment Definition, Experiment Planning, Selection of Variables and Experiment Design.

4.1 Experiment Definition

In this study we used the Goal Question Metric template [4] to define our experimental study. We summarized in the topics below:

- Object of Study: The objects of study are automatically generated test sets.
- Purpose 1: The purpose is to evaluate the complementary aspects of automatically generated test sets generated by four different ATDG.
- Purpose 2: The purpose is to evaluate the complementary aspects of automatically generated test sets generated by smart and randomized ATDG.
- Quality Focus: The quality focus is the adequacy, effectiveness, and cost of test sets, evaluated against statement coverage criterion, mutation testing criterion [1, 2], and number of test cases in the test set.
- Perspective: The perspective of the experimental study is from the researcher's point of view.
- Context: The researcher defined the experiment, considering a set of programs in the data structure domain and also performed the study. The study involves a participant (the first author) working on a set of objects, i.e., it is a multi-object variation study.

4.2 Experiment Planning

In the experiment planning, we set out the hypotheses and variables of the study. Then, we created our experiment plan to guide both the conduction and analysis of the obtained data. In the following, we present an overview of the main activities composing the experiment plan.

4.2.1 Context Selection. As we mentioned in the Section 1, the primary goal of our experiment is to investigate the complementary aspects of test sets automatically generated by smart and random ATDG. In the investigation, we measure the statement coverage of each test set, as well as the mutation score. Also, we evaluate the size of each test set.

In this experiment, we take a set of 33 Java programs, which 32 of it are from [20, 21] and we additionally included the program Identifier which is part of a compiler. These programs in general, implement simple data structures that are well known in literature [24].

Thus, we classified our experiment as an offline study, performed by a master student, addressing a real problem – identification and comparison of adequacy, effectiveness, and cost of automatically generated test sets in a particular context.

4.2.2 Formulation of Hypotheses. Our experimental study aims at evaluating the complementary aspect of test sets generated by smart and random ATDG, looking for evidence that may define new hypotheses for future work.

In attempt to define the hypotheses we measured: i) the adequacy of a test set regarding statement coverage test criterion, ii) the effectiveness of a test set regarding the mutation score it determines, and iii) the cost considering the number of test cases in the test set. All these measures can be influenced by: (i) the testing tools adopted and (ii) the size and complexity of the programs under testing.

In this way, we defined the formal hypothesis under investigation as we present in Table 1.

Table 1: Hypothesis formalized – Adequacy, Effectiveness and Cost

Null Hypothesis	**Alternative Hypothesis**
There is no difference of adequacy among the smart ATGD ($AllS$) and random ATGD automated test sets (R). $H1_0$: $Adequacy(AllS) = Adequacy(R)$	There is a difference of adequacy among the smart ATGD ($AllS$) and random ATGD automated test sets (R). $H1_1$: $Adequacy(AllS) \neq Adequacy(R)$
There is no difference of effectiveness among the smart ATGD ($AllS$) and random ATGD automated test sets (R). $H2_0$: $Effectiveness(AllS) = Effectiveness(R)$	There is a difference of effectiveness among the smart ATGD ($AllS$) and random ATGD automated test sets (R). $H2_1$: $Effectiveness(AllS) \neq Effectiveness(R)$
There is no difference of cost among the smart ATGD ($AllS$) and random ATGD automated test sets (R). $H3_0 : Cost(AllS) = Cost(R)$	There is a difference of cost among the smart ATGD ($AllS$) and random ATGD automated test sets (R). $H3_1 : Cost(AllS) \neq Cost(R)$

4.3 Selection of Variables

In the step, we defined the independent and dependent variables for the experiment, based on the context and the established hypotheses.

Independent Variables. A variable that can be manipulated or controlled during an experiment is called an independent variable [22]. In this work, we defined the following independent variables: i) the testing tools adopted; ii) the testing criteria used; iii) the size and complexity of the programs under test; and iv) the test sets. The latter variable is the only factor of interest to the experiment. Therefore, we have two treatments for the test sets in this work: smart and random automatically generated. We set the other variables in the experiment to not interfere with the obtained results.

Dependent Variables. On the other hand, the dependent variables are those which allow us to observe the manipulation effects of the independent variables [22]. In this study, we defined the statement coverage as dependent variable to describe the adequacy of the test sets.

Another dependent variable we defined is the mutation score to measure the effectiveness of the test set. Therefore, we use mutation testing as a fault model to evaluate the ability of the test sets on detecting well-known faults represented by the mutants [1]. As the number of mutants killed by a test set increases, the effectiveness of these test sets in detecting faults also increases.

The last variable we defined is the number of test cases to describe the costs of the test set. Also, we collected other metrics, detailed in Section 5.2.

4.4 Experiment Design

The analysis of the results is directly impacted by the experiment design. Moreover, it minimizes the influence of adverse factors on the results. As established earlier, the quality of test sets is the only factor of interest addressed by our study.

In Figure 1 we illustrate the test sets evaluation strategy we follow. From a program *Prog* we use the automatic test data generators, EvoSuite (*E*), JTExpert (*J*), Palus (*P*) and Randoop (*R*) to generate four automated test sets for each *Prog*.

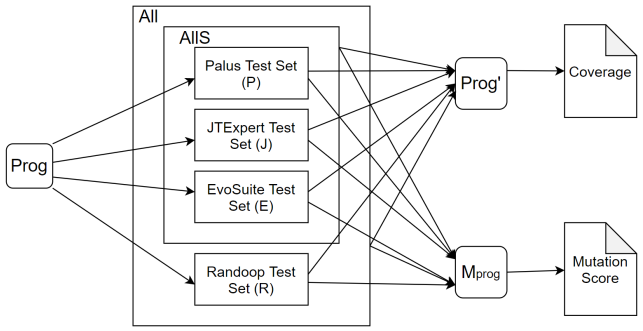

Figure 1: Test sets evaluation strategy.

We followed a similar approach to Vincenzi et. al. [21], combining the test sets to evaluate the complementary aspect between them. For instance, *AllS* represents the union of all ATDG test sets which use heuristics to generate test cases ($E \cup J \cup P$). In another combination of test set, *All*, we included Randoop to the previous combination ($E \cup J \cup P \cup R$), representing the union of all ATDG.

For each combination of test set mentioned above, we repeated the test set generation using 10 different seeds in the ATDG, then we measured the average statement coverage and mutation score. We used Pitest (PIT) [6] to calculate the these previous metrics after each test set execution.

Although our experiment is similar to Vincenzi et. al. [21] approach, we did not fix the maximum number of test cases for each program as the maximum number of test cases generated by the other tools for Randoop. Instead, we decided to limit Randoop generation time to 30 seconds, which is half of the default time limit.

5 EXPERIMENT OPERATION

The experiment operation includes the preparation of artifacts and tools, the execution of activities defined in the experiment plan and the validation of the data collected during the execution.

5.1 Preparation

As we mentioned in subsection 4.2.1, we used 33 Java programs to run the test sets generated by the ATDG and then we performed the test sets combinations as explained in subsection 4.4.

Before we executed the experiment, we performed some adjustments in the programs set. Firstly, we created a Maven Project for each program using Eclipse [8], since the ATDG can be called by Maven [3] scripts. Secondly, we created a pattern for the test set names in a way they are related to the ATDG that generated them. By doing these adjustments, we facilitate the execution of Python scripts, mentioned in the following paragraph.

Therefore, we wrote Python scripts to call Maven to: i) compile program and test sets, ii) run test sets, iii) compute statement coverage and mutation score and iv) generate test reports. In Table 2 we summarize the tools and versions we used and their purpose in our experiment.

To perform the experiment we used a laptop with the following specifications:

- Model: Dell Inspiron;
- OS: Linux Ubuntu 20.04 LTS 64 bits;
- Processor: Core i7;
- RAM: 32 GB;
- Hard disk: 1 TB.

5.2 Execution

The first step in program execution is the computation of static metrics on each program (Table 3). For each program we compute the following metrics:

- Non Commenting Source Statements (NCSS);
- Cyclomatic Complexity Number (CCN);
- Cyclomatic Complexity Average (CCA);
- Number of test cases on each test set: EvoSuite (*E*), JTExpert (*J*), Palus (*P*) and Randoop (*R*);
- Number of requirements demanded to cover statement coverage; and
- Number of generated mutants considering all mutation operators available in PIT.

Table 2: Tools version and purpose

Tool	Version	Purpose
JavaNCSS	32.53	Static Metric Computation
EvoSuite	1.0.6	Test Generator
JTExpert	1.4	Test Generator
Randoop	4.2.4	Test Generator
Pitest	1.3.2	Mutation and Coverage Testing
Eclipse	4.14.0	Integrate Development Environment
Maven	3.6.3	Application Builder
JUnit	4.12	Framework for Unit Testing
Python	2.7.18	Script language

An important information here is that we run the test generators using their default configuration, except for Randoop, as we mentioned in subsection 4.4.

5.3 Data Validation

To make sure we will get the desired data from the scripts, we first ran all ATDG manually in several programs. Secondly, we compared the manually collected data to the one gathered by the scripts. After these steps, as we got no difference in both collected data, we increased our confidence in our scripts and then started the data collection for all 33 programs.

6 DATA ANALYSIS

First, we would like to characterize the programs we used in our experiment. In Table 3 we can see that they are simple programs, implemented by 1 to 3 classes (1.5 on average), 6.1 methods on average, summing up around 40.2 LOC. Column CCM corresponds to the maximum cyclomatic complexity found in the methods of a given program, and CCA is the average cyclomatic complexity. CCM varies from 2 (minimum) to 9 (maximum) cyclomatic complexity, 4.9 on average. CCA ranges from 1.3 to 9 (maximum) average cyclomatic complexity, 3.3 on average.

The last two columns present the number of test requirements demanded by statement coverage criterion (#Req) and mutation testing criterion (#Mut). In the case of mutation testing, we used all mutation operators implemented on PIT.

We present in Table 4 the collected data about the statement coverage and the mutation score obtained by each test set. From Table 4, considering individual test sets, from better to worst, we have J, E, R and P, regarding statement coverage. With mutation score, R achieved the best score, followed by J, E and P.

Observing the pattern, E generates more test sets with 100% of coverage followed by J, R and P. Only for programs 9, 10, 24 and 31, the other ATDG obtained better results, regarding statement coverage. With exception of program 9, the latter three are among the most complex programs, and the other ATDG performed better than E. Regarding mutation score, E did not have any program that achieved 100%, while the other ATDG achieved 100% in two programs (6 and 7).

As can be observed on Table 5, all the tools generated, on average, at least one test case for each program. Nevertheless, due to some incompatibility, EvoSuite (E) fails to get coverage and mutation score on program 10. The same happens to Palus (P) on program 30. P also fails to get mutation score on programs 1, 20 and 30, as can be observed on Table 4. We could not identify the reason for this until the time of written this paper. Further, we intend to investigate this fact in deep aiming at improving such a tools integration.

In Table 5 we present data regarding the number of generated test sets. Colums S, I, F and To refer to test cases successfully generated, ignored, failed and the total of them. As expected, R generated the highest number of test cases on average, followed by P, J and E. We can observe that R generated the maximum number of 3283.7 test cases for program 32 and the minimum of 34 test cases for programs 6 and 7. P generated 316.4 tests as maximum and 1, on average. J generated, on average, 16.1 tests and 1.4. Lastly, E generated the average amount of maximum 12.9 test cases and 1 minimum.

Considering the three hypotheses we have established based on statement coverage, mutation score (Table 4) and number of test sets (Table 5), we applied the Shapiro-Wilk normality. Then, we concluded our collected data do not have a normal distribution with a confidence level of 95%, e.g. p-value ≤ 0.05. Therefore, we used a non-parametric test, the Wilcoxon rank sum test, to verify the difference among groups (smart and random automated test sets), considering a level of confidence of 95% ($\alpha = 0.05$). In Tables 6 to 8 we present these normality results for each pair of test sets. Since we are testing multiple hypotheses, we applied the Holm-Bonferroni correction method which resulted in the column "corrected p-value".

6.1 Analysis of Adequacy

On average, J obtained a coverage of 94.6% with standard deviation (SD) of 9.2%. E obtained a coverage of 93.6% and SD of 21.7. R obtained a coverage of 91.0% and SD of 16.3; and P obtained a coverage of 82.4 and SD of 26.0%. But there are specific situations where one test set performs better than other. We highlight in grayscale the cells where each test set performs better than the others. When the coverage is the same, we highlight all test sets cells with the same value.

In Table 6 we present the Wilcoxon test on statement coverage criterion of ATDG, comparing random to smart test sets, individually and combined, for all 33 programs. The statistics suggest that concerning Randoop (R) and EvoSuite (E) test sets, R and JTExpert (J) test sets there is no statistical difference in terms of adequacy because the corrected p-value is above 0.05. Therefore, considering these test sets we have to accept the null hypothesis $H1_0$. On the other hand, there is a statistical difference between (R) and Palus (P) test sets because the corrected p-value is below 0.05 and, therefore, we rejected the null hypothesis, in this case, accepting the alternative hypothesis $H1_1$.

6.2 Analysis of Effectiveness

In Table 7 we present the Wilcoxon test on mutation score of random and smart ATDG generated test sets, individually and combined, for all 33 programs. In this case, observe that the pairwise comparison of Randoop (R) with the ATDG's tests, only regarding to Palus P, the test suggest there is significant statistical difference, and then we rejected the null hypothesis, $H2_0$. Regarding the other ATDG's test sets, EvoSuite (E) and JTExpert (J), corrected p-value is above 0.05, meaning that we should accept the null hypothesis $H2_0$ because there is no significant statistical difference regarding effectiveness.

6.3 Analysis of Cost

As we expected, Randoop (R) generated, on average, 1132.3 test cases, which is a considerable number of tests; Palus (P) generates 73.5 test cases on average and JTExpert (J) generates on average 7.1 and EvoSuite (E) generates 5.8 test cases on average. Particularly for E and J, we consider these numbers of test cases manageable in case the tester wants to check if they are correct. Even though P generates more than 10 times sets comparing to J, it is still way less than R, which generated more than 13 times test cases in average. Moreover, the standard deviation from E and J test sets is smaller than the ones presented by P and R.

Finally, in Table 8 we present the Wilcoxon test on number of test cases generated by smart and random ATDG for all 33 programs. We can observe that for all cases, corrected p-value is considerably below 0.05, which means that the alternative hypothesis $H3_1$ should be accepted, rejecting $H3_0$. This result is not surprising because, as mentioned earlier, we did not fix the number of test cases generated by R close to the other ATDG.

6.4 Complementary Aspect of Test Sets

After analyzing each ATDG with Randoop R and their grayscale patterns of individual test sets, we then performed an incremental

Table 3: Static information of the Java programs

ID	Program	#Classes	#Methods	NCSS	CCM	CCA	#Req	#Mut
1	Max	1	1	8	3	3,0	4	14
2	MaxMin1	1	1	13	4	4,0	8	21
3	MaxMin2	1	1	14	4	4,0	8	21
4	MaxMin3	1	1	32	9	9,0	16	61
5	Sort1	1	1	11	4	4,0	10	21
6	FibRec	1	1	8	2	2,0	6	12
7	FibIte	1	1	8	2	2,0	6	12
8	MaxMinRec	1	1	26	5	5,0	13	41
9	Mergesort	1	2	22	6	4,0	16	56
10	MultMatrixCost	1	1	18	6	6,0	14	75
11	ListArray	1	4	20	3	1,8	12	29
12	ListAutoRef	2	4	23	2	1,3	12	21
13	StackArray	1	5	20	3	1,8	12	27
14	StackAutoRef	2	5	27	3	1,4	17	27
15	QueueArray	1	5	24	3	2,0	19	40
16	QueueAutoRef	2	5	32	3	1,6	23	32
17	Sort2	2	7	74	6	3,4	49	141
18	HeapSort	1	9	59	5	2,7	40	116
19	PartialSorting	1	10	62	5	2,5	42	120
20	BinarySearch	1	4	32	8	3,5	21	55
21	BinaryTree	2	11	85	7	3,0	48	145
22	Hashing1	2	10	61	5	2,1	35	88
23	Hashing2	2	12	88	7	3,2	51	162
24	GraphMatAdj	1	9	60	5	2,9	42	134
25	GraphListAdj1	3	16	66	4	1,6	34	95
26	GraphListAdj2	2	14	88	6	2,2	51	113
27	DepthFirstSearch	3	16	65	4	1,6	33	94
28	BreadthFirstSearch	3	16	65	4	1,6	33	94
29	Graph	3	16	65	4	1,6	33	94
30	PrimAlg	1	5	40	7	2,6	31	71
31	ExactMatch	1	4	55	8	6,3	40	205
32	AproximateMatch	1	1	24	7	7,0	19	88
33	Identifier	1	3	30	9	7,7	22	114
Avg		1,5	6,1	40,2	4,9	3,3	24,8	73,9
SD		0,7	5,2	25,4	2,0	2,0	14,7	50,3

combinations of them in attempt to observe if there is a complementary aspect between the test sets to check how much they can improve adequacy and effectiveness. Therefore, we created two more test sets: i) *AllS*, representing the combination of EvoSuite (*E*), JTExpert (*J*) and Palus test sets (*P*); and ii) *All*, representing the addition of Randoop tests (*R*) to the previous combination *AllS*.

In Table 4 we can see that *AllS*, regarding statement coverage, had 3 out of 33 programs below 100% and achieved an average of 99.3%, which is slightly higher than *R*, besides having lower standard deviation. Moreover, by observing the respective corrected p-value in Table 6, we can see that it is below 0.05%, thus we accepted the alternative hypothesis $H1_1$.

Regarding mutation score, we observe in Table 4 *AllS* achieved 100% on average only in 2 out of 33 programs, which was also achieved by *P*, *J* and *R*. On the other hand, *AllS* achieved a general average of 74.2%, considerably higher than *R*, without mentioning a lower standard deviation. By observing Table 7 we state that the respective corrected p-value is below 0.05%, e.g., there is significant statistical difference and thus we should accept the alternative hypothesis $H2_1$.

About the number of generated test sets, *AllS* generated 86.4 test sets on average while *R* generated 1132.3, more than 13 times higher, as we can observe in Table 5. Moreover, in Table 8 we see the corresponding corrected p-value is below 0.05%, indicating we should accept the alternative hypothesis $H3_1$, because the is significant statistical difference of cost between *R* and *AllS*.

Therefore, regarding adequacy, effectiveness and cost, we can see in Tables 6 to 8 there is statistical difference between *R* and *AllS* test sets because the corrected p-value is below 0.05, making us to accept the alternative respective hypotheses $H1_1$, $H2_1$ and $H3_1$.

The last step is analyzing the union of all combinations of ATDG including *R*, e.g. *All*. In Table 4 we see only 3 out of 33 programs are below 100% regarding statement coverage, which also occurred with *AllS*, but still *All* got higher results in 2 of them (24 and 26). Besides, *All* achieved 99.5% statement coverage on average, slightly higher than *AllS* mentioned above, and also got a lower standard deviation; thus we accepted the alternative hypothesis $H1_1$. Regarding mutation score, *All* also had 2 out of 33 programs achieving 100%, which was the same as the other ATDG did, but the average was slightly higher than *AllS*, which suggests that we should accept the alternative hypothesis $H2_1$.

By analyzing the number of generated test sets, in Table 5, we can see clearly that just by the fact that we added *R* to our previous combination, generating *All*, consequently we have even more test sets than *R* by itself.

7 THREATS TO VALIDITY

It is crucial to be aware of the potential threats do validity of a research. In the following, we point to these threats concerning internal, external, construction and conclusion of our work.

Table 4: Average Statement Coverage and Average Mutation Score per Test Set

ID	Statement Coverage per Test Set						Mutation Score per Test Set					
	E	*P*	*J*	*R*	*AllS*	*All*	*E*	*P*	*J*	*R*	*AllS*	*All*
1	100.0	75.0	100.0	100.0	100.0	100.0	44.3	0.0	27.9	50.0	49.3	64.3
2	100.0	98.8	100.0	100.0	100.0	100.0	29.0	28.1	83.8	52.4	83.8	83.8
3	100.0	96.3	100.0	100.0	100.0	100.0	29.0	26.7	84.3	52.4	84.3	84.3
4	100.0	68.8	100.0	93.8	100.0	100.0	21.3	10.6	79.7	35.5	79.7	79.8
5	100.0	61.0	100.0	70.0	100.0	100.0	56.0	28.0	50.5	30.0	77.5	78.5
6	100.0	100.0	100.0	100.0	100.0	100.0	92.5	100.0	100.0	100.0	100.0	100.0
7	100.0	100.0	100.0	100.0	100.0	100.0	92.5	100.0	100.0	100.0	100.0	100.0
8	100.0	96.2	100.0	100.0	100.0	100.0	23.1	26.4	76.0	48.1	82.4	83.1
9	91.9	95.0	100.0	100.0	100.0	100.0	19.8	32.5	52.9	95.5	64.5	95.5
10	0.0	92.9	100.0	92.9	100.0	100.0	0.0	26.6	34.9	30.4	45.3	45.6
11	100.0	91.7	91.7	98.3	100.0	100.0	68.1	61.3	57.4	76.8	73.5	78.1
12	100.0	100.0	92.5	100.0	100.0	100.0	69.6	73.9	70.0	82.6	74.8	83.9
13	100.0	91.7	91.7	96.7	100.0	100.0	63.5	77.1	67.4	77.4	81.3	81.3
14	100.0	100.0	100.0	100.0	100.0	100.0	55.2	81.6	69.7	83.9	84.2	85.5
15	100.0	94.7	93.2	99.5	100.0	100.0	79.8	72.1	65.6	82.8	87.9	90.5
16	100.0	100.0	83.5	100.0	100.0	100.0	66.2	67.6	53.0	73.0	78.1	82.4
17	100.0	59.8	100.0	66.7	100.0	100.0	44.6	21.7	52.4	24.5	68.2	68.8
18	100.0	57.5	81.0	73.8	100.0	100.0	67.1	27.6	31.0	31.0	73.5	73.9
19	100.0	57.6	100.0	73.3	100.0	100.0	75.7	23.1	54.3	30.7	84.4	84.4
20	100.0	23.8	60.5	76.2	100.0	100.0	64.7	0.0	23.0	33.7	65.4	70.4
21	88.8	12.5	81.3	54.2	88.8	88.8	93.8	1.3	38.6	17.5	93.8	93.8
22	100.0	99.1	96.9	100.0	100.0	100.0	65.1	71.3	48.2	82.5	84.3	93.9
23	100.0	96.0	94.6	98.1	100.0	100.0	61.2	51.0	48.1	79.6	74.8	86.6
24	19.0	86.0	89.5	96.2	89.5	96.2	7.0	51.5	44.2	68.4	53.1	69.0
25	100.0	95.9	98.8	100.0	100.0	100.0	59.3	58.1	57.8	76.8	74.5	81.9
26	98.4	90.2	93.3	95.3	99.0	99.2	60.1	55.7	54.7	68.3	74.6	78.1
27	100.0	96.4	100.0	100.0	100.0	100.0	64.4	54.8	56.5	76.7	76.7	81.0
28	100.0	93.3	100.0	100.0	100.0	100.0	64.4	55.4	53.3	76.7	76.3	82.1
29	100.0	93.9	100.0	100.0	100.0	100.0	64.4	55.3	59.2	75.6	76.2	81.4
30	100.0	0.0	71.9	28.1	100.0	100.0	30.3	0.0	31.8	1.3	42.4	42.4
31	90.0	100.0	100.0	100.0	100.0	100.0	31.3	55.3	34.0	57.6	56.8	58.6
32	100.0	100.0	100.0	100.0	100.0	100.0	44.2	39.5	29.1	39.5	51.6	51.6
33	100.0	95.9	100.0	90.9	100.0	100.0	46.3	64.6	41.8	51.8	75.1	75.8
Avg	93.6	82.4	94.6	91.0	99.3	99.5	53.1	45.4	55.5	59.5	74.2	78.2
SD	21.7	26.0	9.2	16.3	2.6	2.0	23.5	27.5	19.5	25.3	14.1	14.1

Internal: we have identified the Context Selection (Section 4.2.1) as a potential threat once we did not perform a random selection of the programs set, but these are programs traditionally used on other experiments. Another internal threat is related to the tool. All of them may contain faults, which can lead to incorrect conclusions. To minimize these, we run each test generator ten times with different seeds.

External: this study used EvoSuite, Randoop, JTExpert, and Palus for automatic test data generation. It is possible that using different ATDG may yield different results. Nevertheless, these tools are considered state of the art on ATDG for Java.

Construct: we used mutations at the unit level as a fault model to measure ATDG effectiveness. Although Andrews et al. [1] confirmed the relation between mutations with real faults, using a different mutation testing tool may yield different results on effectiveness.

Conclusion: our program set, composed of 33 simple Java programs, might be relatively small, which can lead to different results if we increase both the number and size of the programs. Nevertheless, we are working at the unit level, and some units have enough complexity for a Java method.

8 CONCLUSION

In this study, we performed an investigation of the complementary aspects of smart and random automated generated test sets over 33 data structure programs. For that, we used four different automatic test data generators (ATDG). We used the default setting for each ATDG, except for Randoop in which we decreased the generation time limit in half because of the elevated number of generated test cases in default time.

When we evaluate the ATDG individually regarding adequacy, EvoSuite test sets (E) achieved both the highest statement coverage average and number of programs with 100% coverage on average, followed by JTExpert test sets (J). Randoop test sets (R) got higher statement coverage than Palus (P). Although E was highlighted above, statistical analysis suggested that there is no difference on adequacy between R, E and J test sets.

Concerning effectiveness, R obtained higher mutation score in the majority of programs, totaling 59,5% on average. The other ATDG achieved higher mutation score in 15 programs, being J the second best test sets which achieved 55.5% statement coverage on average. However, statistical analysis shows that there is no significant difference between R-E and R-J, once the respective mutation scores are close to each other.

Regarding the cost of random and smart ATDG test sets, was the only hypothesis in which statistical analysis suggests the alternative hypothesis $H3_1$ for all comparisons. The reason is R generates a high amount of test cases even when the default time limit is decreased.

We performed an investigation of the complementary aspects of smart and random ATDG test sets. The idea is to combine all test

Table 5: Average Number of Generated Tests per Tool

	Evo				Palus				JTExpert				Randoop				AllS			
ID	S	I	F	To	S	I	F	To	S	I	F	To	S	I	F	To	S	I	F	To
1	3,3	0,0	0,0	3,3	6,0	0,0	0,0	6,0	3,0	0,0	0,0	3,0	2198,9	0,0	0,0	2198,9	12,3	0,0	0,0	12,3
2	4,0	0,0	0,0	4,0	21,7	0,0	0,0	21,7	2,7	0,0	0,0	2,7	1404,8	0,0	0,0	1404,8	28,4	0,0	0,0	28,4
3	4,0	0,0	0,0	4,0	28,8	0,0	0,0	28,8	3,9	0,0	0,0	3,9	1308,2	0,0	0,0	1308,2	36,7	0,0	0,0	36,7
4	7,4	0,0	0,0	7,4	40,0	0,0	0,0	40,0	8,2	0,0	0,0	8,2	1481,6	0,0	0,0	1481,6	55,6	0,0	0,0	55,6
5	2,4	0,0	0,0	2,4	29,8	0,0	0,0	29,8	3,0	0,0	0,0	3,0	1947,2	0,0	0,0	1947,2	35,2	0,0	0,0	35,2
6	2,0	0,0	0,0	2,0	24,8	0,0	0,0	24,8	1,6	0,0	0,0	1,6	34,0	0,0	0,0	34,0	28,4	0,0	0,0	28,4
7	2,0	0,0	0,0	2,0	28,5	0,0	0,0	28,5	1,4	0,0	0,0	1,4	34,0	0,0	0,0	34,0	31,9	0,0	0,0	31,9
8	3,0	0,0	0,0	3,0	29,5	0,0	0,0	29,5	4,7	0,0	0,0	4,7	2127,5	0,0	0,0	2127,5	37,2	0,0	0,0	37,2
9	2,1	0,0	0,0	2,1	16,8	0,0	0,0	16,8	2,8	0,0	0,0	2,8	1846,3	0,0	0,0	1846,3	21,7	0,0	0,0	21,7
10	2,0	0,0	0,0	2,0	28,9	0,0	0,0	28,9	3,2	0,0	0,0	3,2	1902,7	0,0	0,0	1902,7	34,1	0,0	0,0	34,1
11	4,4	0,0	0,0	4,4	59,9	0,0	0,0	59,9	2,9	0,0	0,2	3,1	853,5	0,0	0,0	853,5	67,2	0,0	0,2	67,4
12	3,0	0,0	0,0	3,0	68,2	0,0	0,0	68,2	2,7	0,0	0,1	2,8	668,4	0,0	0,0	668,4	73,9	0,0	0,1	74,0
13	3,9	0,0	0,0	3,9	77,1	0,0	0,0	77,1	4,0	0,0	0,1	4,1	768,1	0,0	0,0	768,1	85,0	0,0	0,1	85,1
14	3,0	0,0	0,0	3,0	72,9	0,0	0,0	72,9	2,9	0,2	0,4	3,5	677,0	0,0	0,0	677,0	78,8	0,2	0,4	79,4
15	4,1	0,0	0,4	4,5	67,3	0,0	0,0	67,3	4,2	0,0	0,1	4,3	374,6	0,0	0,0	374,6	75,6	0,0	0,5	76,1
16	2,9	0,0	0,2	3,1	78,6	0,0	0,0	78,6	3,9	0,0	0,1	4,0	382,6	0,0	0,0	382,6	85,4	0,0	0,3	85,7
17	11,9	0,0	0,0	11,9	38,0	0,0	0,0	38,0	16,1	0,0	0,0	16,1	2379,1	0,0	0,0	2379,1	66,0	0,0	0,0	66,0
18	10,1	0,0	0,3	10,4	136,6	0,0	0,0	136,6	10,3	0,1	0,0	10,4	564,7	0,0	0,0	564,7	157,0	0,1	0,3	157,4
19	11,8	0,0	0,1	11,9	95,5	0,0	0,0	95,5	12,9	0,0	0,1	13,0	370,9	0,0	0,0	370,9	120,2	0,0	0,2	120,4
20	5,4	0,0	0,0	5,4	1,0	0,0	0,0	1,0	2,6	0,1	0,1	2,8	573,2	0,0	0,0	573,2	9,0	0,1	0,1	9,2
21	11,1	0,0	1,8	12,9	1,4	0,0	0,0	1,4	12,2	0,1	0,0	12,3	1565,7	0,0	0,0	1565,7	24,7	0,1	1,8	26,6
22	4,2	0,0	0,8	5,0	96,4	0,0	0,0	96,4	6,5	0,2	0,3	7,0	562,9	12,9	0,5	576,3	106,8	0,2	1,4	108,4
23	7,9	0,0	0,3	8,2	83,3	0,0	0,0	83,3	9,6	0,1	0,3	10,0	378,2	1,9	0,1	380,2	100,8	0,1	0,6	101,5
24	1,0	0,0	0,0	1,0	35,5	0,0	0,0	35,5	8,4	0,1	0,1	8,6	406,4	0,0	0,0	406,4	44,9	0,1	0,1	45,1
25	8,8	0,0	0,0	8,8	83,5	0,0	0,0	83,5	14,9	0,1	0,0	15,0	1196,5	0,0	0,0	1196,5	107,2	0,1	0,0	107,3
26	9,5	0,7	0,0	10,2	79,7	0,0	0,0	79,7	13,2	0,0	0,2	13,4	369,9	0,0	0,0	369,9	102,3	0,7	0,3	103,3
27	8,9	0,0	0,0	8,9	111,8	0,0	0,0	111,8	13,9	0,0	0,0	13,9	1230,0	0,0	0,0	1230,0	134,6	0,0	0,0	134,6
28	8,9	0,0	0,0	8,9	121,3	0,0	0,0	121,3	14,0	0,0	0,2	14,2	1135,7	0,0	0,0	1135,7	144,2	0,0	0,2	144,4
29	8,9	0,0	0,0	8,9	128,2	0,0	0,0	128,2	14,0	0,0	0,0	14,0	974,6	0,0	0,0	974,6	151,1	0,0	0,0	151,1
30	4,1	0,0	0,0	4,1	1,0	0,0	0,0	1,0	2,8	0,0	0,0	2,8	60,0	0,0	0,0	60,0	7,9	0,0	0,0	7,9
31	11,0	0,0	0,0	11,0	313,4	0,0	0,0	313,4	15,4	0,0	0,0	15,4	3163,0	0,0	0,0	3163,0	339,8	0,0	0,0	339,8
32	3,6	0,0	0,0	3,6	316,4	0,0	0,0	316,4	3,7	0,0	0,0	3,7	3283,7	0,0	0,0	3283,7	323,7	0,0	0,0	323,7
33	6,5	0,0	0,0	6,5	102,8	0,0	0,0	102,8	5,1	0,0	0,0	5,1	1125,1	0,0	0,0	1125,1	114,4	0,0	0,0	114,4
Avg	5,7	0,0	0,1	5,8	73,5	0,0	0,0	73,5	7,0	0,0	0,1	7,1	1131,8	0,4	0,0	1132,3	86,1	0,1	0,2	86,4
Sd	3,4	0,1	0,3	3,5	73,3	0,0	0,0	73,3	4,9	0,1	0,1	4,9	851,5	2,3	0,1	851,2	76,6	0,1	0,4	76,6

Table 6: Adequacy: statement coverage

Test Set Pair	p-value	corrected p-value
R-E	0.1386833	0.2773666
R-P	4.299318e-05	0.0001719727
R-J	0.3603752	0.3603752
R-AllS	0.001757725	0.005273174

Table 7: Effectiveness: mutation score

Test Set Pair	p-value	corrected p-value
R-E	0.1077912	0.2155824
R-P	8.069576e-06	3.22783e-05
R-J	0.5371801	0.5371801
R-AllS	0.002673522	0.008020565

Table 8: Cost: number of test cases

Test Set Pair	p-value	corrected p-value
R-E	5.6418e-07	5.6418e-07
R-P	2.328306e-10	9.313226e-10
R-J	2.328306e-10	9.313226e-10
R-AllS	2.328306e-10	9.313226e-10

sets generated by the ATDG which use generation strategy other than random (referred as AllSmart, *AllS* for short), against random. In this case, *AllS* is better than random on all investigated aspect: adequacy, effectiveness, and cost.

Lastly, we compared *AllS* with the test sets generated by $AllS \cup R$, which we call *All* for short. In this case, we observed that, the combination of *AllS* with the test cases generated by Randoop *R* aggregate a little on adequacy and effectiveness but increase a lot in the cost of the strategy. In this way, we considered that the use of random testing needs to be integrated together with a minimization test set strategy to avoid redundant test sets, keeping the cost reasonable and manageable.

Therefore, answering the paper title question, we considered the ATDG evolved, but there are still around 26% of mutations (fault types) the ATDG we used were not able to detect. As mentioned below, we believe there is no general solution to this problem. We need to perform a qualitative study to understand if it is possible to kill some specific mutation operator's live mutants.

As future work, we intend to extend the experimentation to large programs, to investigate quality aspects about each automatic test generators and the reasons they failed on generating test cases for some programs/methods. Different combinations ATDG must be investigated. We intend to establish an incremental testing strategy by combining different automatic test data generators aiming to reduce the number of faults previous to the software release. The idea is to keep a high statement/decision coverage on critical parts of the application on a full automated way.

It is also important to implement minimization strategies for reducing the overlapping between smart and random ATDG, contributing to speed up regression testing. In case of mutation testing, it is necessary to have a tool which allows the generation of a killing matrix (a matrix which related test case kills each mutant). Based on the killing matrix we can easily to implement a minimization strategy to keep the higher mutation score with the minimal number of test cases.

Finally, considering the data we have collected so far, by using only smart ATDG, approximately 0.7% of the source code remains uncovered, and 26% of the mutants remain alive. Aiming at finding specific test cases to touch this uncovered code or to kill these alive mutants, we intend to develop a selective instrumentation tool. This tool would receive as input specific source code parts, and generated another version of the program, instrumenting only those source code lines not yet covered or with alive mutants. In this way, even when using a random test generator, we only selected test cases which touch these very specific parts.

ACKNOWLEDGMENTS

The authors would like to thank the Brazilian funding agency CAPES – Grant Number 88882.426575/2019-01 – for its support for this research.

REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *XXVII International Conference on Software Engineering – ICSE'05*. ACM Press, 402–411. https://doi.org/10.1145/1062455.1062530 event-place: St. Louis, MO, USA.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering* 32, 8 (Aug. 2006), 608–624. https://doi.org/10.1109/TSE.2006.83

[3] Apache Software Foundation. 2016. Apache Maven Project. (June 2016). Disponível em: https://maven.apache.org/. Acesso em: 04/07/2016 bibtex*[howpublished=Página Web].

[4] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. Encyclopedia of Software Engineering. Vol. 2. John Wiley & Sons, Inc., 528–532. bibtex*[chapter=Goal Question Metric Paradigm].

[5] B. Boehm and V. R. Basili. 2001. Software Defect Reduction Top 10 List. *Computer* 34, 1 (2001), 135–137. https://doi.org/10.1109/2.962984 bibtex*[location=Los Alamitos, CA, USA;publisher=IEEE Computer Society Press].

[6] Henry Coles. 2015. PITest: real world mutation testing. (Jan. 2015). Disponível em: http://pitest.org/. Acesso em: 04/07/2016. bibtex*[howpublished=Página Web].

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11, 4 (April 1978), 34–43. https://doi.org/10.1109/C-M.1978.218136

[8] Eclipse Foundation. 2015. Eclipse IDE. (June 2015). Disponível em: https://eclipse.org/mars/. Acesso em: 04/07/2016 bibtex*[howpublished=Página Web].

[9] Gordon Fraser and Andrea Arcuri. 2012. Sound Empirical Evidence in Software Testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE Press, 178–188. bibtex*[acmid=2337245;numpages=11] event-place: Zurich, Switzerland.

[10] Gordon Fraser and Andrea Arcuri. 2016. EvoSuite at the SBST 2016 Tool Competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*. ACM, 33–36. https://doi.org/10.1145/2897010.2897020 bibtex*[acmid=2897020;numpages=4] event-place: Austin, Texas.

[11] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. 2011. Using Conditional Mutation to Increase the Efficiency of Mutation Analysis. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST'11)*. Association for Computing Machinery, New York, NY, USA, 50–56. https://doi.org/10.1145/1982595.1982606

[12] J. S. Kracht, J. Z. Petrovic, and K. R. Walcott-Justice. 2014. Empirically Evaluating the Quality of Automatically Generated and Manually Written Test Suites. In *2014 14th International Conference on Quality Software*. 256–265. https://doi.org/10.1109/QSIC.2014.33 ISSN: 1550-6002.

[13] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. 2007. Reconciling Manual and Automated Testing: The AutoTest Experience. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. 261a–261a. https://doi.org/10.1109/HICSS.2007.462 ISSN: 1530-1605.

[14] Y.-S. Ma, J. Offutt, and Y. R. Kwon. 2005. MuJava: an automated class mutation system: Research Articles. *STVR – Software Testing, Verification and Reliability* 15, 2 (2005), 97–133. https://doi.org/10.1002/stvr.v15:2 bibtex*[location=Chichester, UK, UK;publisher=John Wiley and Sons Ltd.].

[15] 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 (OOPSLA '07)*. ACM, 815–816. https://doi.org/10.1145/1297846.1297902 bibtex*[acmid=1297902;numpages=2] event-place: Montreal, Quebec, Canada.

[16] M. Roper. 1994. *Software Testing*. McGrall Hill.

[17] Abdelilah Sakti, Gilles Pesant, and Yann-Gaël Guéhéneuc. 2015. JTExpert at the Third Unit Testing Tool Competition. 52–55. https://doi.org/10.1109/SBST.2015.20

[18] Anthony J. Simons. 2007. JWalk: A Tool for Lazy, Systematic Testing of Java Classes by Design Introspection and User Interaction. *Automated Software Engg.* 14, 4 (Dec. 2007), 369–418. https://doi.org/10.1007/s10515-007-0015-3 bibtex*[location=Hingham, MA, USA;publisher=Kluwer Academic Publishers;acmid=1296046;issue_date=December 2007;numpages=50].

[19] N Smeets and A J H Simons. 2011. *Automated unit testing with Randoop, JWalk and MuJava versus manual JUnit testing*. Research Reports. Department of Computer Science, University of Sheffield/University of Antwerp, Sheffield, Antwerp.

[20] S. R. S. Souza, M. P. Prado, E. F. Barbosa, and J. C. Maldonado. 2012. An Experimental Study to Evaluate the Impact of the Programming Paradigm in the Testing Activity. *CLEI Electronic Journal* 15, 1 (April 2012), 1–13. Paper 3 bibtex*[publisher=scielouy].

[21] Auri MR Vincenzi, Tiago Bachiega, Daniel G de Oliveira, Simone RS de Souza, and Jose C Maldonado. 2016. The complementary aspect of automatically and manually generated test case sets. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. 23–30.

[22] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in software engineering*. Springer Heidelberg, New York, NY, USA. https://doi.org/10.1007/978-3-642-29044-2

[23] Sai Zhang. 2011. Palus: A Hybrid Automated Test Generation Tool for Java. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1182–1184. https://doi.org/10.1145/1985793.1986036

[24] Nivio Ziviani. 2011. *Project of Algorithms with Java and C++ Implementations*. Cengage Learning. (in Portuguese).