An Empirical Study about the Effectiveness of Debugging When Random Test Cases Are Used

Mariano Ceccato, Alessandro Marchetto
Fondazione Bruno Kessler
Trento, Italy
{*ceccato,marchetto*}*@fbk.eu*

Leonardo Mariani
University of Milano Bicocca
Milano, Italy
mariani@disco.unimib.it

Cu D. Nguyen, Paolo Tonella
Fondazione Bruno Kessler
Trento, Italy
{*cunduy,tonella*}*@fbk.eu*

***Abstract*—Automatically generated test cases are usually evaluated in terms of their fault revealing or coverage capability. Beside these two aspects, test cases are also the major source of information for fault localization and fixing. The impact of automatically generated test cases on the debugging activity, compared to the use of manually written test cases, has never been studied before.**

In this paper we report the results obtained from two controlled experiments with human subjects performing debugging tasks using automatically generated or manually written test cases. We investigate whether the features of the former type of test cases, which make them less readable and understandable (e.g., unclear test scenarios, meaningless identifiers), have an impact on accuracy and efficiency of debugging. The empirical study is aimed at investigating whether, despite the lack of readability in automatically generated test cases, subjects can still take advantage of them during debugging.

***Keywords*-Empirical Software Engineering; Debugging; Automatic Test Case Generation**

I. Introduction

Automatic test case generation is an important area of software testing. The scientific community has reserved a great attention to test generation techniques, which are now available for many popular frameworks and programming languages. For instance, Randoop can automatically generate test cases for Java classes by randomly combining method invocations [13]; PEX can generate test cases that cover all the statements in a .NET class by combining concrete and symbolic execution [17]; μTest uses mutation analysis to generate test suites for Java programs [8].

Empirical evidence has indicated that those automated solutions are effective in discovering programming faults [13], [2], [17], [10], [8]. However, the data and examples in these studies also show that automatically generated (autogen, for brevity) test cases are generally less readable and intuitive than manually designed test cases. In fact manually defined test cases usually address a well-defined scenario identified by a tester, while autogen test cases result from a random or coverage-oriented process, and do not address a clear scenario. As a consequence, testers might find it difficult to interpret a failure exposed by an autogen test case, and the debugging of such a failure might be expensive compared to debugging from a manually designed test case.

Even if a certain lack of readability is intrinsic of autogen test cases, there is no empirical evidence that autogen test cases impact debugging negatively. Understanding the costs of test automation, including indirect costs such as the cost of debugging, is of critical importance for testers who have the responsibility of designing effective testing strategies for their software.

In this paper, we present an empirical study that aims at qualitatively and quantitatively evaluating the impact of autogen test cases on debugging. Our study considers debugging of 8 faults in the `JTopas`[1] and `XML-security`[2] applications. We evaluate the effectiveness of debugging when using a manually designed test suite and a test suite generated by Randoop. The study is based on the activity of 14 BSc (Bachelor of Science) students and 15 MSc (Master of Science) students involved in debugging tasks during two replications of a controlled experiment. In this study, we used Randoop [13] because among many different automatic test case generation techniques, we find that random test cases produced by Randoop are particularly difficult to understand. Thus, if autogen test cases introduce inefficiencies in debugging, random test cases are likely to expose the issue.

Autogen test cases are often short and composed of simple, but sometimes unrelated, sequences of method calls, while manually designed tests address rather complicated, but meaningful, scenarios. Our study investigates whether debugging tasks driven by autogen test cases are as reasonably accurate and efficient as those driven by manually designed test cases.

The paper is organized as follow. Section II presents the design of the experiment. Section III reports the experimental results. Section IV discusses the results. Section V compares our achievements with related work. Section VI provides final remarks and discusses future work.

[1] http://jtopas.sourceforge.net/jtopas/
[2] http://santuario.apache.org/

II. Experiment Definition and Planning

This section reports the definition, design and settings of the experiments in a structured way, following the template and guidelines by Wohlin et al. [20].

The *goal* of the study is to investigate the differences between manually written and autogen test cases, with the purpose of evaluating how well they support debugging tasks. The *quality focus* regards how manually written and autogen test cases affect the capability of developers to correctly and efficiently debug the code. The results of this experiment are interpreted regarding two *perspectives*: (1) a researcher interested in empirically validating autogen test cases and (2) a quality manager who wants to understand whether the time spent in writing test cases pays off when facing actual debugging tasks.

The *context* of the study is defined as follows: the *subjects* of the study are developers facing debugging tasks, while the *objects* are applications that contain the faults to be fixed. We collected empirical data from two replications: the first one involved 7 MSc students of the University of Trento, attending the course of "Software analysis and testing". The second replication involved 14 BSc students of the University of Milano Bicocca, attending the course of "Software analysis and testing", and 8 MSc students of the University of Milano Bicocca, attending the course of "Software quality control". Subjects from both studies have skills in Java programming, debugging, and use of the Eclipse IDE[3]. The first replication, with less subjects, provided feedback used to fine tune the experimental procedure before running the second, larger replication. For instance, we improved the training material and the instructions provided to the subjects before carrying out the second replication.

The applications used in the experiment are two real-world Java applications, *Jtopas* and *XML-security*. *Jtopas* is a simple tokenizer that can tokenize text files as inputs and allows users to customize the grammar of the input files, by specifying the structure of keywords, compounds and comments, and the case sensitivity. Jtopas consists of 15 classes and 2,171 MeLOC (Method Lines of Code). *XML-security* is a library that provides functionalities to sign and verify signatures in XML documents. It supports many mature digital signature and encryption algorithms on standard XML formats, such as XHTML and SOAP. It consists of 228 classes, for a total of 14,754 MeLOC. Both applications are available with a full suite of manually written tests.

A. Hypotheses Formulation and Variable Selection

Based on the study definition reported above, we can formulate the following null hypotheses to be tested:

- H_{01} There is no difference in the **accuracy** of debugging, when debugging is supported either by manually written or autogen test cases.
- H_{02} There is no difference in the **efficiency** of debugging, when debugging is supported either by manually written or autogen test cases.

These two hypotheses are *two-tailed*, because there is no a-priori knowledge on the expected trend that should favor either manually written or autogen test cases. On the one hand, manual test cases are meaningful for a developer who is determining the position of a fault, while automatic tests may contain meaningless statements and code identifiers, that may confuse developers. On the other hand, manual tests could be difficult to understand because they may require understanding of complex parts of the application logic, while automatic tests are simpler, since they are generated without a clear knowledge of the application business logic.

The null hypotheses suggest that we have two *dependent variables*: *debugging accuracy* and *debugging efficiency*. To measure debugging accuracy and efficiency, we asked subjects to fix eight faults in the object application source code. We defined faults that satisfy the following requirements: (1) faults are located in different parts of the applications; (2) each fault is revealed by a manual and an autogen test case; (3) faults do not interact: different faults are revealed by disjoint (manual or autogen) test cases, that is each test case reveals at most one fault; and (4) faults are based on the bugs available in the Software-artifact Infrastructure Repository (SIR)[4]. Since Randoop was not able to reveal some of the SIR faults as provided in the repository, in order to meet the four requirements we slightly changed some faults to simplify their detection. However, such changes do not modify the nature of the faults; they just make the used test suites able to detect them. For instance, according to the SIR repository, the fault #3 of JTopas is injected into the application by removing the `getMessage(..)` method from the `ExtIndexOutOfBoundsException` class. We changed the location of fault #3 to the `TokenException` class, so that it can be revealed by both autogen and manual tests, since random tests do not expose this fault if left inside the former class.

When a fault was revealed by multiple test cases, we randomly selected one test case to use in the study. In reality, developers often have a larger suite of test cases, that reveal the fault or can be used for regression testing. However, providing a full test suite to subjects could represent a too wide source of variability for a controlled in-lab experiment, as different starting points could be used by different developers, based on their own experience and style. Instead, we wanted all the subjects to start from the same test and work under exactly the same initial conditions.

[3] http://www.eclipse.org

[4] http://sir.unl.edu

In our experiments, the *accuracy* of debugging is measured as the number of correctly fixed faults. We objectively evaluated the correctness of the fixes by running a predefined set of test cases (not provided to subjects) and checking if they all pass. The *efficiency* of debugging is evaluated as the number of correct tasks (i.e., the number of correctly fixed faults) divided by the total amount of time spent for these tasks (measured in minutes): $eff = \frac{\sum Corr_i}{\sum Time_i}$; where $Corr_i$ is equal to 1 if the i-th task is performed correctly, 0 otherwise, while $Time_i$ is the time spent to perform the i-th task. In other words, efficiency is measured as the number of correctly performed tasks per minute.

The *independent variable* (the main factor of the experiment) is the treatment during the execution of debugging tasks. The two alternative treatments are: (1) manually written test cases, those distributed as unit tests for the object applications (i.e., we obtained them from the SIR repository); and, (2) test cases automatically generated by Randoop [13].

The *understandability* of the test cases that reveal faults might affect the debugging performance and may vary substantially between manual and autogen test cases. Since we cannot control this factor in the experiments, because it depends on how manual test cases have been defined and how the test case generation algorithm works, we measure whether any difference occurs in our experimental setting. The test case understandability might, in fact, represent one of the key features in which manual and autogen test cases differ, which could possibly explain some of the observed performance differences.

Unfortunately, there is no easy, widely-accepted way of measuring the understandability of test cases. We approximate such measurement by considering two specific factors of understandability, namely identifier meaningfulness and complexity of the test code. For the former factor we manually classify each identifier in a test case as either *Artificial* (automatically generated) or *UserDef* (user-defined) and count the respective numbers.

In order to measure the test case complexity, we consider both static and dynamic metrics, which provide an approximation of how complex a test case is from the developer's perspective[5]. Metrics are computed using the Eclipse plugin Metrics (http://metrics.sourceforge.net). As static metrics we measure:

- *MeLOC*, non-blank and non-comment lines of code inside each method body;
- *McCabe* cyclomatic complexity of each test method.

As dynamic metrics, we consider the amount of code executed by each test case. We count it at two granularity levels:

- *Exec. methods*, the number of methods executed by a test case;
- *Exec. LOCs*, the number of statements executed by a test case.

To determine whether manual and autogen test cases differ according to the identifier meaningfulness and code complexity metrics, we introduce two additional, derived null hypotheses:

DH_{03} There is no difference in the number of **artificial / user-defined identifiers** of the manually written and autogen test cases used in the experiment.

DH_{04} There is no difference in the **static / dynamic complexity** of the manually written and autogen test cases used in the experiment.

Experimental support for the alternative hypotheses associated with DH_{03} and DH_{04} would provide useful information for the interpretation of the results about the two dependent variables (considered in H_{01} and H_{02}).

B. Co-factors Identification

We measured the following co-factors that could influence the dependent variables:
(1) *The subjects' ability:* the ability of subjects in performing debugging tasks was measured using a pre-test questionnaire and a training session. The pre-test questionnaire included questions about their programming and debugging ability, their experience with the development of large applications, and their scores in academic courses related to development and testing. In the training session, subjects were asked to complete tasks that consist of answering some code-understanding questions and fixing faults in each of the two object applications. According to the answers given to the pre-questionnaire and the results of the training lab we classified the subjects into three categories: *high ability* subjects (4 in the first experiment and 5 in the second one) are those who had experience with the development of large applications, have an academic score of at least 27/30[6] and completed correctly at least 50% of the tasks in the training lab; *medium ability* subjects (respectively 2 and 13) are those who correctly completed at least 25% of the tasks in the training lab and either had experience with the development of large applications or have an academic score higher than 27/30; the rest of the subjects are classified as *low ability* subjects (respectively 2 and 4).
(2) *The subjects' experience:* we classified BSc students as *low experience* subjects and MSc students as *high experience* subjects.
(3) *The object system (aka application):* since we adopted a balanced design with two systems, subjects could show different performance on different systems. Hence the system is also a co-factor.

[5]Although some of the used metrics are actually size metrics, we regard them as test case complexity indicators, since they reflect the perceived complexity associated with using the test case during debugging.

[6]In the Italian academic grade system, a score of 27/30 corresponds to a B in the ECTS grade system; to an A- in the US system.

(4) *The experiment session (aka Lab):* subjects could spend some effort to familiarize with the experimental environment during the first session. We measured whether any learning effect occurred between the two labs.
(5) *The fault to be fixed:* as faults are all different, the specific features of the fault to be fixed may interact with the main factor.

For each co-factor, we test if there is any effect on the debugging accuracy and debugging efficiency and we check their interaction with the main factor. We formulate the following null hypotheses on the co-factors:

H_{0c_i} the co-factor i, $i = \overline{1..5}$, does not significantly interact with the kind of test cases to influence accuracy and efficiency in performing debugging tasks.

These null hypotheses are also two-tailed, because we do not have any a-priori knowledge about the direction in which a co-factor could influence accuracy and efficiency.

C. Experimental Design

We adopted a counter-balanced design: each replication of the experiment consisted of two experimental sessions (*Lab 1* and *Lab 2*), with 2-hours allocated for each lab. Subjects have been split into four groups, balancing the level of ability and experience in each group. This design ensures that each subject works on the two applications (*Jtopas* and *XML-security*) and with the two different treatments (*manually* written and *randomly* generated test cases), as shown in Table I. Moreover, this design allowed us to study the effect of each co-factor, using statistical tests (i.e., analysis of variance).

Table I
EXPERIMENTAL DESIGN. R = RANDOOP TEST CASES, M = MANUALLY WRITTEN TEST CASES.

	Group1	Group2	Group3	Group4
Lab 1	Jtopas M	XmlSecurity M	Jtopas R	XmlSecurity R
Lab 2	XmlSecurity R	Jtopas R	XmlSecurity M	Jtopas M

D. Experimental Procedure and Material

Before the experiment, we asked the subjects to fill a pre-questionnaire in which we collected information about their ability and experience in programming and testing. Subjects have also been trained with lectures on testing and debugging. Moreover, subjects participated in a training laboratory where they were asked to cope with debugging tasks very similar to the experiment on the object applications. This made us confident that subjects were quite familiar with both the development/debugging environment and the source code of the applications to debug. We wanted to make sure that subjects spend enough time to familiarize with the applications during training, so that the time measured during the actual experiment was required mostly to understand the test case and to identify and fix faults on already familiar applications. The accuracy in the training tasks has been recorded and used to assess the subjects' level of ability.

To perform the experiment, subjects used a personal computer with the Eclipse development environment (already used in the training), equipped with a standard Java debugger. We distributed the following material:

- The application code: depending on the group either Jtopas or XML-security. The code contains four faults;
- Four test cases: either manually written or autogen, depending on the group the subjects belong to, as shown in Table I. Each test case reveals exactly one fault; test cases are supposed to be addressed in order and they are sorted according the their difficulty, from easier to harder to fix;
- Slides describing the experimental procedure.

Before the experiment, we gave subjects a clear description of the experimental procedure, but no reference was made to the study hypotheses. The experiment has been carried out according to the following procedure:

1) Import the application code in Eclipse;
2) For each of the four test cases, (i) mark the start time, (ii) run the test case and use it to debug the application and fix the fault (iii) mark the stop time;
3) Create an archive containing the modified source code and send it to the experimenter by email;
4) Fill a post-experiment survey questionnaire.

During the experiment, teaching assistants were present in the laboratory to prevent collaboration among subjects and to verify that the experimental procedure was respected – in particular that faults were addressed in the right order and time was correctly marked.

After the experiment, subjects have been asked to fill a post-experiment survey questionnaire, devoted to gaining insight about the subjects' behavior during the experiment and finding justification for the quantitative observations. The questionnaire consists of 17 questions, expressed in a Likert scale [12], related to: Q1: Whether the time given to complete the tasks was enough; Q2: Clarity of tasks; Q3-4: Difficulties experienced in understanding the source code of the application and the source code of the test cases; Q5: Difficulties in understanding the features under test; Q6: Difficulties in identifying the portion of code to change; Q7-8: Use and usefulness of the Eclipse debugging environment; Q9: Number of executions of the test case; Q10-11: Percentage of total time spent looking at the code of the test cases and of the application; Q12: Difficulties in using the test cases for debugging; Q13: Whether the bugs have been fixed without fully understanding them, relying just on test cases; Q14: Need for inspecting the application code to understand bugs; Q15: Perceived level of redundancy in test cases; Q16: Usefulness of local variables in test cases

to understand the test; Q17: Whether they found the test cases misleading.

E. Analysis Method

We used a non-parametric statistical test to check the two hypotheses related to the accuracy and efficiency of subjects in performing debugging tasks (H_{01} and H_{02}). The use of non-parametric tests does not impose any constraint on the normal distribution of the population. Since the empirical data is intrinsically paired (the same subjects attended both labs, thus worked with both randomly generated and manually written test cases), we used the Wilcoxon two-tailed paired test [20] to check the hypotheses. In order to use this test, however, we had to make the experimental data paired, by removing data points of subjects who did not participate in both experimental sessions (few students could not attend both labs). Such a test allows to check whether differences exhibited by the same subjects with different treatments (manual and random tests) over the two labs are significant. We assume significance at a 95% confidence level (α=0.05), so we reject the null-hypothesis when p-value$<$0.05. The same statistical test was used to address the derived null hypotheses DH_{03} and DH_{04}.

In order to understand whether the test case complexity is a property which characterizes the main treatment (manual vs. autogen test cases), we measured the performance of the test case complexity metrics as predictors of the treatment. Specifically, we computed the confusion matrix where each test case complexity metrics is a possible actual factor, and the binary classification between manual and autogen test cases is the predicted factor. Standard classification metrics (number of true/false positives/negatives) and derived metrics (precision, recall, accuracy and F-measure) are then used to assess the degree to which manual and autogen test cases can be separated using the test case complexity as the distinguishing feature. Specifically, correct classifications are indicated as TP (true positives, i.e., correctly classified as Autogen) and TN (true negatives, i.e., correctly classified as non-Autogen), while errors are of two types: FP (false positives, i.e., manual test cases classified as Autogen) and FN (false negatives, i.e., autogen test cases classified as non-Autogen). Among the several indicators that can be used to assess the quality of a classifier [18], we consider four of the most widely used indicators: precision, recall, accuracy and F-measure. They are defined as follows: *precision* = TP / (TP + FP); *recall* = TP / (TP + FN); *accuracy* = (TP + TN) / (TP + FP + TN + FN); *F-measure* = 2 *precision* * *recall* / (*precision* + *recall*).

The analysis of co-factors, that is the test of hypotheses H_{0c_1}, H_{0c_2}, H_{0c_3}, H_{0c_4}, H_{0c_5}, is performed using a two-way Analysis of Variance (Anova) and, when detected, interactions are visualized using interaction plots.

Regarding the analysis of survey questionnaires, we evaluate questions related to availability of enough time, general difficulties found by subjects and the use of the debugging environment (Q1-Q8) by verifying that the answers are either "Strongly agree" (2) or "Agree" (1). We test medians, using a one-tailed Mann-Whitney test for the null hypothesis $\widetilde{Qx} \leq 0$, where 0 corresponds to "Undecided", and $\widetilde{Qx}$ is the median for question Qx.

Among these questions, for those specific to test cases (Q4-Q8), answers of subjects using manually written tests were compared with answers of subjects using randomly generated tests. In this case a two-tailed Mann-Whitey test is used for the null hypothesis $\widetilde{Q}_{Random} = \widetilde{Q}_{Manual}$. The same comparison is also performed for questions Q9-Q17.

III. Results

A. Debugging Accuracy

Figure 1 (top) shows the box-plot of the accuracy of fault fixing. The figure compares the number of correct answers given by subjects in the first experiment (left-hand side), second experiment (middle), and overall results (right-hand side), when faults are debugged using either manually written or randomly generated test cases.

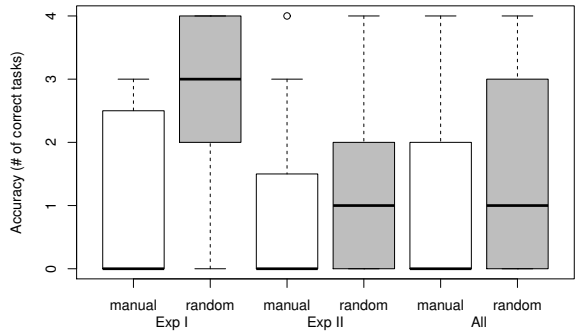
Figure 1 (bottom) reports descriptive statistics including the number of subjects who participated in both Labs (paired analysis), mean, median and standard deviation, together with the p-value for the Wilcoxon's test. We can notice that subjects who used autogen tests showed better accuracy (i.e., correctly fixed more faults) than subjects who used manually written tests, in both experiments. Even if the trend is clearly evident, data from the first experiment are not statistically significant. This could be due to the small number of subjects involved in the first experiment. Data from the second experiment and overall data confirm this trend with significance at 95% confidence level, although showing a smaller gap between random and manually written tests. Thus we can reject H_{01} and conclude that subjects perform significantly better when using autogen tests than manually written tests.

B. Debugging Efficiency

The same procedure used with accuracy was also applied to efficiency. Figure 2 (top) shows the box-plot for efficiency with the two treatments. It compares the efficiency of the subjects in the first (left-hand side) and second experiment (middle) and overall result (right-hand side), when faults are debugged using either manually written or randomly generated test cases.

Figure 2 (bottom) reports descriptive statistics of the paired data and the p-value for the Wilcoxon test. The trend shown for accuracy is confirmed here: the efficiency of subjects working with autogen tests is higher than when working with manually written tests.

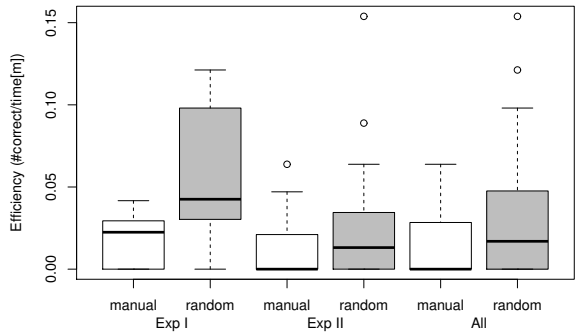
While data from the first experiment are not significant (though the trend is evident), data from the second experiment and from the two experiments together confirm the



Exp	N	diff.mean	diff.median	diff.sd	p.value
I	4	1.25	1.00	2.22	0.42
II	16	0.38	0.00	0.62	**0.04**
All	20	0.55	0.00	1.10	**0.03**

Figure 1. Analysis of accuracy

trend observed in the first experiment, with a significance at 95% confidence level, although showing a smaller gap between random and manually written tests. Thus we can reject H_{02} and conclude that subjects perform significantly faster when using random tests than manually written tests.



Exp	N	diff.mean	diff.median	diff.sd	p.value
I	4	0.05	0.05	0.06	0.12
II	16	0.01	0.01	0.03	**0.03**
All	20	0.02	0.01	0.04	**0.01**

Figure 2. Box-plots of fixing efficiency.

C. Test Case Understandability

Comparing the number of artificial and user-defined identifiers in the fault revealing test cases of JTopas and XML-security (see Table II), one can see that no artificially generated identifier is present in any of the manual test cases. User-defined identifiers are of course present also in autogen test cases. For instance, names of classes instantiated or methods called in the test cases. Artificial identifiers may be present in manual test cases as well, for instance when code generation tools (e.g., tools for parser generation from grammars) are used. This never happens in our two case studies.

Each random test case has on average 28 artificial identifiers more than the corresponding manual test case and such a difference is statistically significant at level 0.05 according to the Wilcoxon paired test. Random tests have on average 15 non-artificial identifiers less than the corresponding manual tests. This difference is statistically significant at level 0.1 (not at level 0.05).

In summary, the number of meaningless identifiers (Artificial) in random test cases is substantially greater than in manual test cases and the number of meaningful identifiers (UserDef) substantially smaller. Hence, we can reject the null hypothesis DH_{03}, but the alternative hypothesis does not explain the observed difference in accuracy and efficiency, which goes in the opposite direction, with random tests associated to superior performance.

Table II
OCCURRENCES OF ARTIFICIAL/USER-DEFINED IDENTIFIERS IN THE TEST CASES

	Autogen tests		Manual tests	
	Artificial IDs	UserDef IDs	Artificial IDs	UserDef IDs
JTopas				
T1	20	4	0	36
T2	18	9	0	59
T3	19	8	0	26
T4	61	22	0	16
XML-security				
T1	7	3	0	9
T2	63	27	0	18
T3	13	5	0	20
T4	23	7	0	21

Table III
PAIRED ANALYSIS OF STATIC (TOP) AND DYNAMIC (BOTTOM) TEST CASE METRICS (WILCOXON'S TEST).

Metric	N	diff.mean	diff.median	diff.sd	p.value
MeLoc	8	0.62	0.50	21.12	1.00
McCabe	8	-0.62	0.00	2.07	0.59
Methods	8	-91.38	-65.00	120.77	**0.04**
LOCs	8	-559.38	-416.50	725.33	**0.04**

Comparing the values of the complexity metrics for random and manual test cases per fault, one can observe (see Table III) that while static metrics (MeLOC and McCabe) exhibit no substantial difference between manual and random test cases, dynamic metrics (Executed methods and executed LOCs) show a major difference when comparing manual and random test cases. In both applications, the number of methods and statements executed by manual test cases is substantially higher than by random tests. The ratio is the order of two for Jtopas, while it is even bigger for

XML-security (reaching an order of magnitude when LOCs are considered).

As we can see from the results in Table III, for dynamic metrics the difference between manual and random test cases is statistically significant at 95% confidence level, so we can reject the null hypothesis DH_{04} (with respect to dynamic metrics).

In summary, manual test cases are dynamically more complex and this might explain the observed performance degradation exhibited by subjects working with manual test cases, despite the presence of more meaningful identifiers in these test cases.

We computed the confusion matrix (not shown for lack of space) associated with a nearest neighbor classifier that predicts the test case type based on one of the two dynamic complexity metrics (either the executed methods or LOCs). The predictor classifies each test case by determining the test case having the closest dynamic complexity metrics value and assigning it to the class (Autogen or Manual) of such closest test case [5].

Table IV
PREDICTION PERFORMANCE METRICS FOR THE TEST CASE BASED NEAREST NEIGHBOR CLASSIFIER ON EXECUTED METHODS OR LOCS.

Metric	Precision	Recall	Accuracy	F.measure
Exec. Methods	0.75	0.75	0.75	0.75
Exec. LOCs	0.80	1.00	0.88	0.89

Executed LOCs is a better predictor than executed methods. The values reported in Table IV (bottom) for this indicator are quite close to 1, showing that in our experiment it is possible to predict the type of a test case from complexity metrics (specifically, executed LOCs) with high accuracy. This means that the two types of test cases used in the experiment (autogen vs. manual) can be characterized with high accuracy as low dynamic complexity vs. high dynamic complexity.

D. Analysis of Co-factors

For lack of space we do not report all the Anova tables, but just the p-values. However, all detailed analysis results can be found in the technical report [3].

For each co-factor, Table V shows the result of the two-way Anova of efficiency by treatment and co-factor, for the first and second experiment, and for the overall results.

We first analyze *Ability* (*high*, *medium* or *low*) and *Experience* (*BSc*, or *MSc* student). For the first experiment, we cannot consider *Experience* as a co-factor, because all subjects involved in the first experiment share the same level (i.e, MSc). We can notice that both Ability and Experience have a significant effect. While Ability does not, Experience does interact with the main factor treatment in a (marginally) statistically significant way when we consider overall data (p-value = 0.07). The interaction plot in Figure 3 (top) shows that experienced subjects are definitely more efficient when working with random tests rather than manual tests, while less experienced subjects only marginally improve their efficiency when working with random tests.

We can hence see that while for non-skilled/non-experienced subjects the difference between manual and autogen tests is small, skilled/experienced subjects performed better when they worked with random tests than with manual tests.

Table V
ANOVA OF EFFICIENCY BY TREATMENT & CO-FACTOR C_i

Co-factor	Exp I	Exp II	All
Treatment	0.01	0.05	0.01
Ability	**0.01**	**0.00**	**0.00**
Treatment:Ability	0.15	0.36	0.55
Treatment		0.08	0.01
Experience		**0.01**	**0.00**
Treatment:Experience		0.17	**0.07**
Treatment	0.09	0.11	0.02
System	0.61	0.30	0.59
Treatment:System	0.55	0.17	0.19
Treatment	0.09	0.09	0.02
Lab	0.41	0.13	0.10
Treatment:Lab	0.71	**0.02**	**0.04**

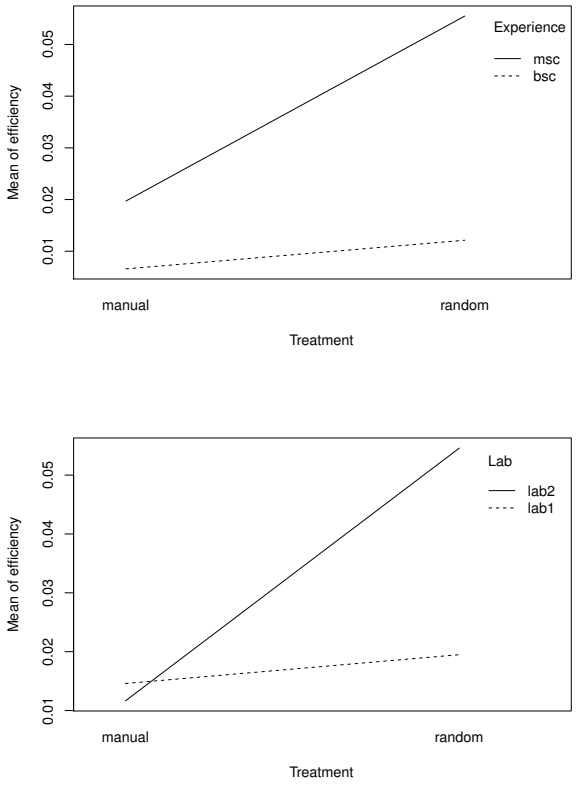


Figure 3. Interaction plots between Treatment & Experience (top) and Treatment & Lab (bottom), for efficiency (both experiments)

We also analyze whether the particular *System* used in the experiment (Jtopas or XML-security) influenced the results. The two-way Anova by Treatment and System (Table V, 3rd sector) indicates no significant effect of the applications and no interaction with the main factor.

Let us now analyze the *Lab* as co-factor (Table V, bottom). We can notice that Lab is a significant co-factor in the second experiment and overall, thus there is a learning effect between the two experimental sessions. Figure 3 (bottom) shows the interaction plot between the efficiency and the labs. Subjects who used random tests first do not show any learning effect, while there is a significant learning effect for subjects who used manually written test cases first.

Last, we analyze the role of *faults* as a co-factor to see if the faults influenced the result and/or interacted with the main factor (manual vs. autogen test) to influence the result. We cannot study the impact of the co-factor on accuracy and efficiency, as they are metrics over all faults, and we are interested in each fault individually. So we resort to $Corr_i$ and $Time_i$ as the dependent variable available for each (i-th) fault.

The analysis is performed separately for the two systems (Jtopas and XML-Security) because faults are different. Results of the two-way Anova by Treatment and Fault on *Correctness* are significant only for JTopas overall (see Table VI), but they never interact with the main factor. Results of the two-way Anova by Treatment and Fault on *Time* are significant for JTopas on the first and second experiment, while for XML-Security they are significant on the second experiment and overall (see Table VI), but again they never interact with the main factor.

Table VI
ANOVA OF CORRECTNESS/TIME BY TREATMENT & FAULT

	Correctness			Time		
Co-factor	Exp I	Exp II	All	Exp I	Exp II	All
Treatment	0.87	0.77	0.34	0.26	0.04	0.90
Fault(JT)	0.07	0.45	**0.00**	**0.00**	**0.00**	0.08
Treatm:Flt(JT)	0.95	0.70	0.15	0.38	0.11	0.12
Treatm	0.00	0.52	0.02	0.03	0.92	0.01
Fault(XS)	1.00	0.90	0.71	0.33	**0.00**	**0.01**
Treatm:Flt(XS)	1.00	0.07	0.64	0.43	0.60	0.59

E. Analysis the Post Questionnaire Survey

We used answers to questions from Q1 to Q8 to gain insights on the subjects' activity. Results are summarized in Table VII. Considering data over all the experiments, answers to questions Q1, Q2, Q4, Q7 and Q8 produced statistically significant results (p-value < 0.05), while answers to the other questions are not significant. Subjects found the time slightly insufficient to accomplish all the tasks ($\widetilde{Q1} = 0$, i.e. "not certain"). Subjects considered the tasks to be clear (Q2) and, overall, they had no difficulty in understanding the source code of the test cases (Q4). Finally, the debugger was used (Q7) only in the second experiment but it was judged useful (Q8) in all the experiments. During the first experiment even if judged useful, subjects did not significantly use the debugger (p-value$_{Q7} = 0.85$).

Table VII
ANALYSIS OF POST QUEST Q1-Q8. MANN-WHITNEY TEST FOR THE NULL HYPOTHESIS $median(Qx) <= 0$

Quest.	Exp I		Exp II		All	
	median	p.value	median	p.value	median	p.value
Q1	0.00	0.81	0.50	**0.01**	0.00	**0.05**
Q2	2.00	**0.00**	1.00	**0.00**	1.00	**0.00**
Q3	0.00	0.74	0.00	0.36	0.00	0.52
Q4	1.00	**0.03**	0.50	0.07	1.00	**0.02**
Q5	0.50	**0.04**	0.00	0.98	0.00	0.86
Q6	0.00	0.84	0.00	0.76	0.00	0.85
Q7	-1.00	0.85	1.00	**0.00**	1.00	**0.00**
Q8	1.00	**0.03**	1.00	**0.00**	1.00	**0.00**

Then we compared the answers for questions specific to test cases (Q4 to Q17), to understand if any statistical difference can be observed between subjects who worked with manually written test cases and those who used autogen ones. The unpaired Mann-Whitney's test never reported statistical significance, so we omit the table for lack of space (it can be found in the technical report [3]).

We can notice a remarkable difference in the use of the Eclipse debugger between the first and the second experiment (see Q7 in Table VII). In fact, subjects involved in the first experiment declared to have used the Eclipse debugger less than subjects involved in the second experiment (even though both recognize the potential usefulness of the debugger; see Q8 in Table VII). This might explain the larger gap between manual and random test cases observed in the first experiment, compared to the second experiment (see Figures 1 and 2 for a comparison of accuracy and efficiency, respectively). Without the debugger, subjects could take advantage only of simple test cases (i.e., those generated by Randoop), while they could not manage the complexity of most manual test cases, resulting in lower performance in the latter case. On the contrary, subjects who used the debugger more extensively were able to take advantage also of the complex test scenarios. However, they also had better performance with the simpler, random tests.

The extensive use of the debugger in the second experiment compared to the limited use of the debugger in the first experiment might also explain the learning effect mentioned in the analysis of the lab as co-factor. The learning effect has been observed only for the subjects who worked with manually written tests first. We can hypothesize that after analyzing manual tests with the debugger, subjects gain a knowledge of the debugging environment that can be effectively reused when analyzing random tests. On the contrary, subjects who analyzed random tests first did not gain knowledge that can be used in the analysis of manually written tests, due to the simplicity of the executions considered before.

IV. Discussion

A. Interpretation of Findings

The key result of the experiment is that (1) autogen test cases affect positively debugging, by improving the accuracy and efficiency of developers conducting fault localization and bug fixing tasks and that (2) in our experiment, manually written test cases are more complex than autogen test cases, but they contain more meaningful identifiers.

We can summarize our interpretations (*Int*) of the data collected in our experiment as follows:

Int1*: Meaningfulness of test case identifiers does not affect debugging accuracy and efficiency*. Manual test cases have more non-artificial identifiers and substantially less artificially generated identifiers (see analysis of identifiers in test cases). However, this does not result in superior debugging performance of subjects using manual test cases. Indeed, the opposite is true. We think the presence of meaningless identifiers in autogen tests is not an influential factor because such identifiers appear only when debugging the top-level methods in execution (i.e., the test methods). Below such top level, identifiers are perfectly understandable and meaningful. Moreover, any difficulty of interpretation of a test method due to its identifiers does not matter as long as the test reveals a fault. Subjects probably did not even attempt to attribute an intent to autogen tests.

Int2*: Test case complexity affects debugging accuracy and efficiency*. Manual test cases exercise complex, long execution scenarios that require substantial effort to be fully understood. Autogen test cases are simple, short linear sequences of method invocations that do not require any dedicated understanding activity (see analysis of dynamic test case complexity). In our experiment, this difference may (at least partially) explain the superior performance of autogen test cases. The difference in the dynamic metrics between manual and random test cases can be attributed to the different way in which the test cases have been constructed. Manual test cases have been designed to exercise a high level usage scenario, which involves application functionalities requiring a complex and extensive code execution. On the other hand, random test cases are focused on implementation functionalities (not user requirements) and consider a portion of the implementation quite locally, with a limited amount of dependencies from other code portions.

Int3*: Ability and experience are key factors affecting the debugging performance*. Even when provided with simple test cases, inexperienced programmers had a hard time fixing the bugs (see analysis of interaction between treatment and experience). Availability of focused, simple test cases empowers the fault finding capabilities of developers only if these have enough previous experience and related skills. Debugging is a hard task that requires training and discipline, and it cannot be learned in the short time provided by an empirical study like ours.

Int4*: Debugging performance improves over time, especially if developers are exposed to complex test scenarios*. We observed a learning effect, which was particularly relevant when manual (hence more complex) test cases were used (see analysis of interaction between treatment and lab). This seems to indicate that debugging tasks conducted in the past affect to a major extent the performance expected on a new debugging task. We think this has probably to do with an incrementally learned capability of fine tuning the debugging strategy, depending on the test scenario at hand. Such capability can be learned, but it requires training on complex scenarios.

Int5*: Usage of advanced debugging environments is fundamental with complex test scenarios*. Only subjects able to effectively use the Eclipse debugger could use the more complex, manual test cases to fix bugs (see analysis of feedback questionnaire). When the complexity of a test case becomes high, automation of the debugging activities is required in order for the tester to be able to effectively and efficiently investigate the execution, and locate and fix the fault. Still, simpler test cases amplify the benefits of the debugging environment.

We think the collected results have interesting implications on the use of tools for automated test case generation and on the testing process as a whole. The debugging capabilities of skilled and experienced testers can be amplified by providing them with focused and simple autogen test cases that reveal the fault to be fixed. Such a benefit is not compromised by the use of meaningless identifiers in the test cases. Hence, whenever the same bug can be revealed by complex, manual test cases, but also by simple, automated tests, the latter are preferable to maximize the debugging performance.

Based on the results obtained in our experiment, we reconsidered the whole testing process and the potential room for automated test case generation. We think that our results suggest the following strategy: (1) first, generate automated test cases and fix any bug possibly revealed by them; (2) write/consider manual test cases only later, since these are less effective for the bugs revealed by both classes of tests. However, since manual test cases tend to be more complex, we think they should not be replaced by the automated ones. They should be just considered later, for those bugs that require complex execution scenarios to be exposed.

B. Threats to Validity

The main threats to the validity of this experiment belong to the conclusion, internal, construct and external validity threat categories.

Conclusion validity threats concern the relationship between treatment and outcome. We used non parametric statistical tests (Mann-Whitney and Wilcoxon) [20] that do not require normal distribution of the experimental data.

The only parametric test we performed is Anova (used only to assess the interactions), which is however robust for deviations from normality. The survey questionnaire was designed using standard scales.

Internal validity threats concern external factors that may affect the independent variable. Subjects were not aware of the experimental hypotheses. Subjects were not rewarded for the participation in the experiment and they were not evaluated on their performance in doing the experiment.

Construct validity threats concern the relationship between theory and observation. They are mainly due to how we measure the accuracy of debugging. We relied on previously defined test cases to objectively evaluate whether the fixes were correct. The ability of subjects was estimated during the training phase, on similar debugging tasks, and using the exam scores.

External validity concerns the generalization of the findings. We considered test cases generated by Randoop. Work-in-progress aims at experimenting with test cases generated using other techniques, such as concolic testing. Even if we considered two different real-world systems from different domains and with different complexity, more replications are desirable on other applications. The study was performed in an academic environment which may differ substantially from the industrial one. However, we mitigate this threat by considering ability and experience as a co-factor.

V. Related Work

Multiple strategies can be used to automatically generate unit test cases. For instance, it is possible to combine concrete and symbolic execution [17], [16], use dynamically detected assertions [10], apply mutation analysis [8] and randomly combine method invocations [13]. These techniques have been reported several times as effective solutions for revealing programming faults [1], [4], [6]. However, no studies are available about the impact of automatically generated test cases on debugging activities. In this paper, we report an experiment that, for the first time, measures the effectiveness of automatically generated test cases when used as part of debugging tasks. In particular we compare manually defined test cases with random test cases generated by Randoop. Our results indicate that automatically generated test cases represent a valid support for debugging. Even if in this study automatically generated test cases outperformed manually defined test cases in terms of accuracy and efficiency, we have to recall that automatically generated tests suffer well known limitations, such as the impossibility to generate domain-dependent oracles as well as the capability to generate tests of moderate length and complexity only [1].

Most of the literature on empirical studies related to testing and debugging presented experiments that do not involve human subjects. For instance, Frankl and Weiss [7] compared the capability of revealing faults when test cases satisfy different coverage criteria. However, their work did not consider the intensive manual activity spent to locate the fault, after a test case revealed it. A few empirical works on software testing and debugging involved human subjects, but they considered directions different from the work presented in this paper. For instance, the studies by Ricca et al [15] and Huang et al [11] focused on the testing process and strategy, evaluating the impact of the "test first" strategy either on the accuracy of change tasks or on the quality of the final code.

A few attempts have been made to investigate the relation between testing and debugging. Fry et al [9] presented an observational study on the accuracy of human subjects in locating faults. They discovered that certain types of faults are difficult to locate for humans. For instance, "extra statement" faults seem to be easier to detect than "missing statement" faults. However, the authors did not investigate the role of test cases. Weiser et al. [19] empirically evaluated the impact of a slicing tool on debugging. They did not observe any improvement when developers used a slicing tool to debug small, faulty programs. Parnin and Orso [14] performed an experiment to investigate how developers use debugging tools and whether these improve performance. Tools are proven to help complete debugging tasks faster. Still, this study does not consider the role of test cases.

VI. Conclusion

To the best of our knowledge, this is the first human-based study that evaluates the impact of different kinds of test cases on debugging. The data we collected from two experiments indicate that automatically generated test cases positively affect debugging, improving both the accuracy and the efficiency of developers working on fault localization and bug fixing tasks.

This result demonstrates that even if automatically generated test cases contain less understandable identifiers, the simplicity of these tests facilitates the implementation of correct fixes. In our experiments, subjects took advantage of the simple structure of the tests to debug problems faster and better, especially when subjects were well experienced.

To corroborate our findings, this study needs to be extended in several directions, such as considering more applications, more types of faults and more test case generation techniques. Nevertheless, the results reported in this paper provide useful insights for testers and software engineers, and clarify issues related to the yet unexplored interplay between test automation and human factors.

References

[1] J. Andrews, A. Groce, M. Weston, and R.-G. Xu. Random test run length and effectiveness. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 19–28, Washington, DC, USA, 2008. IEEE Computer Society.

[2] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. D. Tetali, and A. V. Thakur. Proofs from tests. *IEEE Transactions on Software Engineering*, 36:495–508, 2010.

[3] M. Ceccato, C. D. Nguyen, A. Marchetto, L. Mariani, and P. Tonella. Debugging tasks supported either by manual or automatic test cases, analysis of two replications: Trento and Milano. Technical report, FBK, TR-FBK-SE-2011-4_0, http://se.fbk.eu/en/techreps, September 2011.

[4] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 84–94, New York, NY, USA, 2007. ACM.

[5] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on on Information Theory*, 13(1):21–27, 1967.

[6] J. W. Duran. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 4:438 – 444, 1984.

[7] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the symposium on Testing, analysis, and verification*, TAV4, pages 154–164, New York, NY, USA, 1991. ACM.

[8] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA, pages 147–158, New York, NY, USA, 2010. ACM.

[9] Z. P. Fry and W. Weimer. A human study of fault localization accuracy. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[10] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, pages 213–223, New York, NY, USA, 2005. ACM.

[11] L. Huang and M. Holcombe. Empirical investigation towards the effectiveness of test first programming. *Inf. Softw. Technol.*, 51:182–194, January 2009.

[12] A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter, London, 1992.

[13] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA, pages 815–816, New York, NY, USA, 2007. ACM.

[14] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA, USA, 2011. ACM.

[15] F. Ricca, M. Torchiano, M. D. Penta, M. Ceccato, and P. Tonella. Using acceptance tests as a support for clarifying requirements: A series of experiments. *Information and Software Technology*, 51(2):270 – 283, 2009.

[16] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30:263–272, September 2005.

[17] N. Tillmann and J. D. Halleux. Pex: white box test generation for .NET. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] C. J. van Rijsbergen. *Information Retrieval (2nd ed.)*. Butterworths, London, UK, 1979.

[19] M. Weiser and J. Lyle. Experiments on slicing-based debugging aids. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 187–197, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[20] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.