Automated Unit Test Generation during Software Development: A Controlled Experiment and Think-Aloud Observations

José Miguel Rojas[1] Gordon Fraser[1] Andrea Arcuri[2]

[1]Department of Computer Science, The University of Sheffield, United Kingdom
[2]Scienta, Norway and SnT Centre, University of Luxembourg, Luxembourg

ABSTRACT

Automated unit test generation tools can produce tests that are superior to manually written ones in terms of code coverage, but are these tests helpful to developers *while they are writing code*? A developer would first need to know when and how to apply such a tool, and would then need to understand the resulting tests in order to provide test oracles and to diagnose and fix any faults that the tests reveal. Considering all this, does automatically generating unit tests provide any benefit over simply writing unit tests manually?

We empirically investigated the effects of using an automated unit test generation tool (EvoSuite) during development. A controlled experiment with 41 students shows that using EvoSuite leads to an average branch coverage increase of +13%, and 36% less time is spent on testing compared to writing unit tests manually. However, there is no clear effect on the quality of the implementations, as it depends on how the test generation tool and the generated tests are used. In-depth analysis, using five think-aloud observations with professional programmers, confirms the necessity to increase the *usability* of automated unit test generation tools, to *integrate* them better during software development, and to *educate* software developers on how to best use those tools.

Categories and Subject Descriptors. D.2.5 [Software Engineering]: Testing and Debugging – *Testing Tools*;

Keywords. Automated unit test generation, controlled experiment, think-aloud observations, unit testing

1. INTRODUCTION

Modern automated test generation tools and techniques can efficiently generate program inputs that lead to execution of almost any desired program location. In the context of automated unit test generation for object oriented software there are tools that exercise code contracts [1, 2] or parameterised unit tests [3], try to exhibit undeclared exceptions [1, 4], or simply aim to achieve high coverage [5–9]. There are even commercial tools like Agitar One [10] and Parasoft JTest [11] that generate unit tests that capture the current behaviour for automated regression testing. In previous work [12], we have shown that *testers* can achieve higher code coverage when using automatically generated tests than when testing manually. However, most previous investigations considered unit test generation independently of the *developers*: Does the use of an automated unit test generation tool support software developers in *writing code and unit tests*? And if not, how do automated unit test generation tools need to be improved in order to become useful?

In order to provide a better understanding of the effects automatic unit test generation has on software developers, we empirically studied a scenario where developers are given the task of implementing a Java class and an accompanying JUnit test suite. We studied this scenario using two empirical methods: First, we performed a controlled empirical study using 41 human subjects, who were asked to complete two coding and testing tasks, one manually, and one assisted by the EvoSuite unit test generation tool [13]. Second, we conducted five think-aloud observations with professional programmers to understand in detail how developers interact with the testing tool during implementation and testing. Our experiments yielded the following key results:

Effectiveness: Generated unit tests tend to have higher code coverage, and they can support developers in improving the coverage of their manually written tests (on average +7% instruction coverage, +13% branch coverage, +7% mutation score), confirming previous results [12]. However, the influence of automated unit test generation on the quality of the implemented software depends on how a tool and its tests are used.

Efficiency: When using automated unit test generation, developers spend less time on unit testing (36% less time in our experiments). Although participants commented positively on this time saving, it comes at the price of higher uncertainty about the correctness of the implementation, and we observe that spending more time with the generated tests leads to better implementations.

Usability: Readability of automatically generated unit tests is a key aspect that needs to be optimised (63% of participants commented that readability is the most difficult aspect of generated tests). If generated unit tests do not represent understandable, realistic scenarios, they may even have detrimental effects on software.

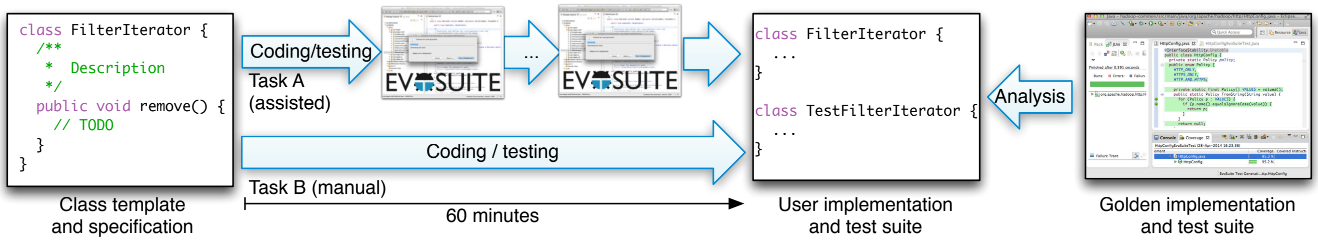
Integration: Test generation tools typically optimise for code coverage, but this is not what developers do. We observed three different approaches to unit testing, each posing different requirements on how testing tools should interact with developers and existing tests.

Education: Simply providing a test generation tool to a developer will not magically lead to better software—developers need education and experience on when and how to use such tools, and we need to establish best practices. For example, we found a moderate correlation between the frequency of use of our test generation tool and the number of implementation errors committed by developers.

The main contributions of this paper are the controlled empirical study (Section 3) and the think-aloud observations (Section 4),

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

ISSTA'15, July 12–17, 2015, Baltimore, MD, USA
ACM. 978-1-4503-3620-8/15/07
http://dx.doi.org/10.1145/2771783.2771801



```
class FilterIterator {
  /**
   *  Description
   */
  public void remove() {
    // TODO
  }
}
```

```
class FilterIterator {
  ...
}

class TestFilterIterator {
  ...
}
```

Figure 1: Experimental setup overview. There are two tasks: To implement a class and test suite assisted by EVOSUITE (Task A), or entirely manually (Task B). Participants performed both tasks on different classes (A followed by B or vice versa). The resulting implementation and test suite were evaluated for code coverage, mutation analysis, and conformance with a *golden* test suite.

whose results align with and extend those of our previous study focused on a *testing-only* scenario [12]. Both empirical methods are applied in the experimental scenario described in detail in Section 2 and summarised in Figure 1: Participants are given the task of implementing a Java class *and* a JUnit test suite, either manually or supported by EVOSUITE; analysis is then performed on the resulting implementations and test suites.

2. EXPERIMENT PLANNING

The goal of our controlled experiment is to investigate the usefulness of automated unit test generation during the software development process, and to understand how this usefulness can be improved. This section describes the experimental setup and procedure for the controlled experiment in detail, following existing reporting guidelines for empirical software engineering research [14].

2.1 The EvoSuite Unit Test Generation Tool

There are many different unit test generation tools to choose from (e.g., [1–9]). For our experiment infrastructure we require a tool that runs on Linux (e.g., [1,4,6–9]). The tool should be usable through an IDE rather than requiring developers to switch to the command-line (e.g., [1,8]). Finally, we consider a scenario where developers are intended to make use of the generated unit tests to support their own implementation as well as test suite, without being required to provide specifications or parameterised unit tests (although this would be an interesting follow-up experiment). The number of generated tests should remain low to enable manual processing, which excludes random testing tools, and leaves EVOSUITE [8].

EVOSUITE generates test suites with the aim of maximising code coverage (e.g., branch coverage), minimising the number of unit tests and optimising their readability. The generated tests include assertions that capture the current behaviour of the implementation; i.e., all tests pass initially, and developers need to provide their own test oracles, possibly making use of the suggested assertions. The EVOSUITE Eclipse plug-in allows developers to generate a JUnit test suite for any Java class by right-clicking the class name and selecting the “Generate JUnit tests” option. We configured EVOSUITE to run for 60s on each invocation using its default configuration. At each invocation, EVOSUITE creates a new test suite. As we need *one* JUnit test suite at the end to evaluate (and, e.g., to use for regression testing), we asked participants to copy and paste each generated unit test that should be retained into a dedicated (initially empty) JUnit class. That is, participants are asked to populate this JUnit class with the automatically generated unit tests they consider useful from each run of the tool, tests which they can further refine or refactor.

2.2 Research Questions

RQ1: Does using EVOSUITE during software development lead to test suites with higher code coverage?

RQ2: Does using EVOSUITE during software development lead to developers spending more or less time on testing?

RQ3: Does using EVOSUITE during software development lead to software with fewer bugs?

RQ4: Does spending more time with EVOSUITE and its tests lead to better implementations?

2.3 Object Selection

The task given to the participants of the experiment was to implement a Java class with its test suite. We selected the classes to implement from open source projects to ensure they represent realistic, non-artificial tasks. As a source for these classes we considered the 41 Java libraries contained in the Apache Commons project [15]. We manually selected four Java classes based on the following criteria. Classes should...

- contain 40-70 non-commenting source statements (NCSS).
- be testable by EVOSUITE with at least 80% code coverage.
- not exhibit features currently not handled by EVOSUITE’s default configuration, e.g., I/O or GUI.
- challenge the participants’ programming skills but be simple enough to be coded *and* tested in one hour.
- have a clear and apparent purpose and be well documented.
- have no inner classes and few dependencies.

The first criterion (size) was estimated appropriate for the planned duration of the task (60 minutes) based on past experience; size was measured using the JavaNCSS tool [16]. By applying the second criterion (coverage) we narrowed the search to 52 classes, on which we manually applied the remaining criteria to finally select the classes listed in Table 1.

For each selected class, we created a minimal Eclipse project consisting of only that class and its dependencies. We removed package prefixes (e.g., `org.apache.commons.collections`) and copyright information to prevent participants from finding source code or test cases on the Internet during the experiment. We converted the classes to skeleton versions consisting only of the public API (i.e., we removed all non-public members, and left only stubs of the method implementations). Furthermore, we refined the JavaDoc documentation to ensure that it was unambiguous and could serve as specification. In order to confirm the suitability of these classes, we conducted a pilot study with Computer Science PhD students as subjects, which particularly helped to refine and clarify the original JavaDoc specifications.

2.4 Participant Selection

We recruited participants for this experiment by sending email invitations to second to fourth year computer science undergraduate students, as well as software engineering master students, at the University of Sheffield. Students of all these levels have at least basic understanding and experience in programming Java, testing with JUnit, and using Eclipse. A total of 41 students were recruited to take part in the experiment (32 second to fourth-year undergraduate and nine master students).

Table 1: Selected Java classes

Class	NCSS	Methods	Instruction Coverage	Branch Coverage	Description
FilterIterator	49	11	85.0%	91.7%	Iterator that only shows elements that satisfy a Predicate validation
FixedOrderComparator	68	10	81.5%	77.5%	Comparator which imposes a specific order on a specific set of objects
ListPopulation	54	13	80.0%	77.3%	Genetic population of chromosomes, represented as a List
PredicatedMap	44	9	100.0%	100.0%	Decorates a Map object to only allow elements satisfying a Predicate validation

According to a background survey, all participants had previously used the Eclipse IDE and the JUnit Testing Framework, and had at least two years of programming experience. 76% declared to have between two and four years of programming experience in Java. 83% had previous experience with automated test case generation tools (e.g., we use Randoop and EVOSUITE in some classes). The survey also included a 10-item Java programming test, where the average score was 6.93/10. 68% declared they understood the concept of code coverage *well* or *very well*. One participant affirmed to *always* write unit tests, whereas two said they *never* do; 27 (66%) *rarely* or *occasionally* write unit tests, and 11 (27%) said they *often* do.

2.5 Experiment Procedure

We started with a 30 minute tutorial session, which included a live demo on a small example of how the experiment would proceed. Participants then practised on a simplified version of the main task: they implemented a class with a single method that is supposed to return the integer division of its arguments. We interacted with the participants to make sure they all had a good understanding of Java, JUnit, EVOSUITE, and their task.

Each participant performed two tasks, one with EVOSUITE and one without, each on a different class, with the goal to implement as much of the behaviour described in the JavaDoc specification as possible, and to write unit tests that achieve the highest possible branch coverage. Each task consisted in implementing the set of public methods of the class and thoroughly testing them, and participants were allowed to add private methods or fields to the class. For the *manual* task all code and unit tests have to be written manually; for the *assisted* task participants must write code by hand, but can use EVOSUITE to generate unit tests at any moment during the session.

The duration of each task was 60 minutes. A fixed assignment of two classes per participant ID, each with different treatment, was created prior to the experiment, in order to reduce the time between tutorial and main experiment. Our assignment ensured similar sample sizes for all classes and treatments, and each participant ID was assigned to a PC in the computer lab. Participants chose PCs freely upon arrival to the facilities, without receiving any details about their assignment. We made sure that no two neighbouring participants were working on the same class or treatment at the same time. Each participant was paid 30 GBP, and was asked to fill in an exit questionnaire before leaving.

2.6 Data Collection

The experiment took place on May 21, 2014 in the computer lab of the Computer Science Department at the University of Sheffield. The pre-configured environment consisted of Ubuntu Linux, Java SE 7u55, Eclipse Kepler 4.3.2 and the EVOSUITE, EclEmma [17] and Rabbit [18] Eclipse plug-ins.

The EclEmma plug-in allowed participants to check coverage of their test suites during the experiment (e.g., illustrated with highlighting of covered code). Furthermore, the Rabbit plug-in was set up to run in the background and collect usage data such as number of times tests were executed, number of times coverage was measured, time spent writing code or debugging, etc. In order to have the entire implementation history for each participant, we extended the EVOSUITE plug-in, such that every time a file was saved, EVOSUITE was invoked, or a JUnit test suite was executed, the current status of the project was copied to a local directory and to a remote server.

2.7 Data Analysis

The original implementation of the four selected Java classes, along with tests suites manually written by their developers, constitutes our *golden implementations* and *golden test suites* (see Table 1 for coverage statistics). Our data analysis consists of evaluating *a)* the participants' test suites when executed on both participants' implementations and golden implementations; and *b)* the golden test suites when applied to the participants' implementations.

Coverage analysis was performed using EclEmma, which measures instruction and branch coverage; instruction coverage refers to Java bytecode instructions and thus is similar to statement coverage on source code. EclEmma's definition of branch coverage counts only branches of conditional statements, not edges in the control flow graph [19]. Therefore, we use both instruction and branch coverage for data analysis. The Major mutation framework [20] was used to perform mutation analysis on the resulting test suites. Empirical evidence supports the use of mutation analysis as a trustable technique to measure test suites quality in testing experiments [21,22].

2.8 Statistical Analysis

To rigorously analyse the data resulting from the experiment, we use different statistical tests [23,24], effect sizes [25] and power analysis [26]. This was necessary due to the limited sample size (i.e., the number of participants) and the fact that EVOSUITE is based on a randomised algorithm. With small sample sizes, on one hand non-parametric tests might not have enough power to detect statistical difference. On the other hand, there might not be enough data for a parametric test to be robust to deviations from a normal distribution [23,24] (note, due to the Central Limit theorem, there is no need for the data to be normally distributed, as long as there are enough data points [23]).

As parametric test to check differences between average values of two distributions, we used the Student T-test (with Welch correction). Effect size is measured with the Cohen d statistics [25,26], which is the difference in the averages of two distributions divided by the pooled standard deviation. Given a non-null effect size, to study the probability of accepting the null hypothesis when it is false (i.e., claiming no statistical difference when there is actually a difference), we performed power analyses to find the minimal sample size N for which the null hypothesis could be rejected (confidence level for Type I error of $\alpha = 0.05$ and power $\beta = 0.8$ for Type II error, where α and β have typical values taken from the scientific literature).

We also employed the non-parametric Mann-Whitney U-test to check for statistical difference among the stochastic rankings of two compared distributions. As effect size, we used the Vargha-Delaney $\hat{A}_{12}$ statistic [25,27]. If, for example, we compare the results of EVOSUITE-supported testing with manual testing, then an effect size of $\hat{A}_{12} = 0.5$ would mean that manual testing and EVOSUITE-supported testing resulted in the same coverage; $\hat{A}_{12} < 0.5$ would mean that EVOSUITE-supported participants produced lower coverage, and $\hat{A}_{12} > 0.5$ would mean that EVOSUITE-supported participants produced higher coverage.

Table 2: Comparisons of average instruction/branch coverage and mutation scores of the test suites developed using EvoSuite (Assisted) and manually without (Manual). Results are calculated by running these test suites on the participants' implementations. We also report the pooled standard deviation (sd), non-parametric effect size $\hat{A}_{12}$, p-value of U-test (U pv), parametric Cohen d effect size, p-value of T-test (T pv), and its minimal required sample size N resulted from a power analysis at $\alpha = 0.05$ and $\beta = 0.8$.

Class	Measure	Assisted	Manual	sd	$\hat{A}_{12}$	U pv	d	T pv	N
FilterIterator	Instructions	70.6%	62.0%	30.5%	0.62	0.36	0.28	0.52	198
	Branches	63.0%	39.1%	37.7%	0.67	0.18	0.63	0.15	40
	Mutation	43.2%	28.9%	33.3%	0.62	0.33	0.43	0.33	86
FixedOrderComparator	Instructions	45.7%	60.5%	32.2%	0.38	0.38	-0.46	0.33	76
	Branches	38.2%	57.1%	32.1%	0.32	0.20	-0.59	0.21	47
	Mutation	28.6%	29.6%	25.0%	0.53	0.88	-0.04	0.93	11327
ListPopulation	Instructions	67.3%	28.4%	29.8%	0.79	0.03	1.31	0.01	10
	Branches	83.5%	26.2%	24.7%	0.92	0.00	2.32	0.00	4
	Mutation	33.2%	15.1%	21.1%	0.68	0.17	0.86	0.07	22
PredicatedMap	Instructions	45.4%	50.8%	41.6%	0.44	0.71	-0.13	0.78	944
	Branches	41.5%	50.4%	43.5%	0.42	0.58	-0.21	0.66	371
	Mutation	32.5%	35.7%	35.4%	0.44	0.71	-0.09	0.85	1945

2.9 Threats to Validity

Construct: We use the number of failing tests of the golden test suite as a proxy for the quality of the implementation; the number of failures may be different from the number of faults (e.g., many tests may fail due to the same fault). We used coverage and mutation scores to estimate the quality of test suites. While evidence supports that real faults are correlated with mutants [21,22], it is possible that the use of faults created by developers may yield different results.

Internal: Extensive automation is used to prepare the study and process the results. It is possible that faults in this automation could lead to incorrect conclusions.

To avoid bias, we assigned participants to objects randomly, but manually tweaked the assignment to guarantee balanced sample sizes and to ensure no two neighbouring participants would work on the same class or treatment at the same time. Participants without sufficient knowledge of Java and JUnit may affect the results; therefore, we only accepted participants with past experience, and we provided the tutorial before the experiment. In the background questionnaire, we included five JUnit and five Java quiz questions. On average, 6.93 out of these 10 questions were answered correctly, which strengthens our belief that the existing knowledge was sufficient. A blocking assignment strategy may have resulted in a better assignment based on the participants' knowledge, but we pre-generated our assignment to avoid splitting the experiment into two sessions in order to have the time to evaluating the answers to the background questionnaire and designing the blocking assignment.

Experiment objectives may have been unclear to participants; to counter this threat we tested and revised all our material on a pilot study, and interacted with the participants during the tutorial exercise and experiment to ensure they understood the objectives. As each participant performed two tasks, it is possible that those with an *assisted* task in the first session could grasp insight on how tests should be written out of inspection of the automatically generated tests. To lessen the impact of this learning effect, our assignment of objects to participants ensures that each pair of classes/treatments occurs in all possible orders. Our training in the use of EvoSuite may not be sufficient, and indeed our results suggest that developer education is important. However, there are no established best practices on how to use automated unit test generation tools during development, so we can only provide basic training, and hope to establish these best practices as a *result* of our experiment.

To counter fatigue effects we included 15-minutes breaks after the tutorial session and between the two main sessions, and provided soft drinks and snacks throughout the experiment. In order to minimise participants' communication, we imposed exam conditions, and explicitly asked participants not to exchange information or discuss experiment details during the breaks.

External: For pragmatic reasons, the participants of our study are all students, which is a much discussed topic in the literature (e.g., [28,29]). However, we see no reason why automatic unit test generation should be useful only to developers with many years of experience, and argue that students are thus at least close to the population of interest [30].

The set of target classes used in the experiment is the result of a manual but systematic selection process. The classes are small to allow implementation within the short duration of the experiment; however, classes in object oriented programs are often small, but may have more external dependencies. It may be that classes with more dependencies make generated unit tests more difficult to read, or, on the other hand, more helpful in understanding these dependencies. Thus, to which extent our findings can be generalised to arbitrary programming and testing tasks remains an open question.

We used EvoSuite, and other tools may lead to different results. However, the output of EvoSuite is similar to that of other tools aiming at code coverage. Random or specification driven tools would represent different use cases, likely requiring a different experimental setup.

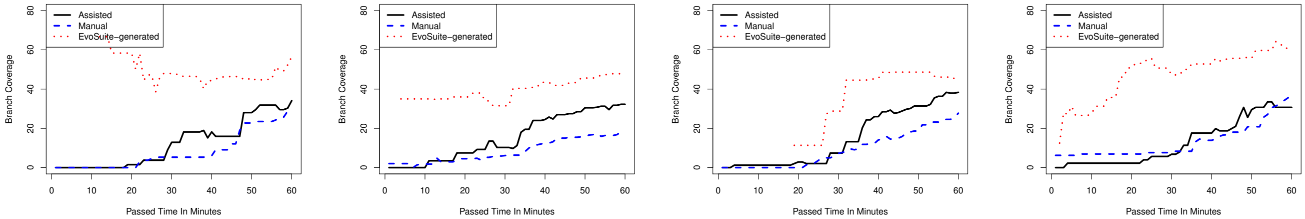
Conclusion: Our study involved 41 human participants and four Java classes. Each participant was assigned two tasks, one manual and one assisted by EvoSuite. Hence, nine to twelve participants performed each of the combinations of testing approach and Java class. This small number of participants can lead to statistically non-significant results. However, as results may differ a lot between different classes, we decided to aim for data on more classes, rather than statistically stronger data on fewer classes.

3. EXPERIMENT RESULTS

3.1 RQ1: Effects on Resulting Test Suites

To determine the effects on the test suites, we evaluate them with respect to the participants' implementations as well as the golden implementation. A test suite that covers the implemented behaviour well intuitively should be a good regression test suite, whereas during development one would hope that the test suite covers the specified behaviour (golden implementation) well.

Table 2 summarises the results for instruction/branch coverage and mutation scores measured on the participants' implementations. For FilterIterator and ListPopulation coverage and mutation scores are higher when using EvoSuite, with medium to large effect sizes – for ListPopulation the large increase is statistically significant. For PredicatedMap instruction and branch coverage are lower when



(a) FilterIterator (b) FixedOrderComparator (c) ListPopulation (d) PredicatedMap

Figure 2: Time analysis, per minute, for each class, of the average branch coverage on golden implementation.

Table 3: Obtained average instruction and branch coverage on the golden implementations, of the test cases developed supported by EVOSUITE (Assisted) and manually without (Manual). We also report the pooled standard deviation (sd), non-parametric effect size $\hat{A}_{12}$, p-value of U-test (U pv), parametric Cohen d effect size, p-value of T-test (T pv), and its minimal required sample size N resulted from a power analysis at $\alpha = 0.05$ and $\beta = 0.8$.

Class	Instruction Coverage								Branch Coverage							
	Assisted	Manual	sd	$\hat{A}_{12}$	U pv	d	T pv	N	Assisted	Manual	sd	$\hat{A}_{12}$	U pv	d	T pv	N
FilterIterator	56.8%	53.6%	30.8%	0.56	0.65	0.10	0.81	1441	40.9%	30.3%	32.6%	0.63	0.29	0.33	0.45	149
FixedOrderComparator	51.2%	31.4%	29.8%	0.70	0.15	0.66	0.16	36	34.7%	20.9%	24.8%	0.68	0.19	0.56	0.24	52
ListPopulation	37.8%	28.7%	32.0%	0.49	0.97	0.29	0.53	194	37.2%	28.2%	29.6%	0.52	0.92	0.30	0.51	171
PredicatedMap	43.3%	46.6%	39.3%	0.47	0.83	-0.09	0.85	2163	41.9%	50.0%	39.6%	0.43	0.61	-0.20	0.66	375

using EVOSUITE. For FixedOrderComparator there is also a decrease in coverage; although the mutation score is slightly lower on average, the effect size shows a small increase with EVOSUITE.

These differences in coverage are potentially related to how the generated test suites were used: Table 4 shows that for FilterIterator and ListPopulation the number of times participants using EVOSUITE measured coverage is also higher, whereas for the other two classes the number of times coverage was measured was higher for participants testing manually; e.g., for FixedOrderComparator EVOSUITE users measured coverage less than twice, whereas participants doing manual testing measured coverage on average 10 times. Thus it seems that if developers focus on coverage, they can increase coverage with generated tests.

To analyse the effects of the test suites with respect to the specified behaviour, we use the golden implementations: The intended behaviour is specified in elaborate JavaDoc comments, but it is also embodied by the golden implementations. Hence, as a proxy measurement for how well the test suites cover the *specified* behaviour, we measure the instruction and branch coverage the participants' test suites achieve on the golden implementations. Table 3 summarises these results, and shows the outcomes of the statistical analysis. For FilterIterator and FixedOrderComparator both instruction coverage and branch coverage of the golden implementations are higher with medium effect sizes. For ListPopulation the branch coverage is higher, and for instruction coverage the effect size suggests a marginal decrease despite higher average coverage. For PredicatedMap the decrease in coverage observed on the participants' implementations is also visible on the golden implementation.

The increase in coverage for FixedOrderComparator is particularly interesting, considering that we saw a decrease of coverage of the implemented behaviour (Table 2). Our conjecture is that this is an effect of how the generated test suites are used: Although participants did not measure coverage often (Table 4) they did spend time on the generated tests (9.3 minutes on average), and in that time apparently improved the tests to reflect specified behaviour.

Figure 2 shows how coverage of specified behaviour evolves over time. The solid and segmented lines correspond to test suites produced on *Assisted* and *Manual* tasks, respectively. The dotted lines correspond to the automatically generated test suites *as produced* by EVOSUITE. We can see that the EVOSUITE-generated tests achieve the highest coverage: If correct behaviour is implemented, EVOSUITE will cover it. However, participants did not fully use (e.g., copy and edit) all generated tests in their own test suites, as shown by the lower coverage on the *Assisted* tasks.

For FixedOrderComparator, ListPopulation and FilterIterator (to a lesser degree towards the end in the latter), we observe how the participants assisted by EVOSUITE were able to achieve higher coverage with their own test suites. For PredicatedMap it seems that EVOSUITE was not used very often until around half an hour into the task. From then to about 55 minutes there seems to be a small benefit to using EVOSUITE's tests, but in general the gap between generated tests and the tests actually used by the participants is very large. In the end, participants testing only manually achieved slightly higher coverage. We conjecture that for PredicatedMap, automatically generated tests for more complex behaviour are less readable, as our think-aloud study will confirm (Section 4). However, overall this data suggests that using EVOSUITE can lead to test suites with higher code coverage, depending on how it is used.

RQ1*: Our experiment shows that coverage can be increased with* EVOSUITE*, depending on how the generated tests are used.*

3.2 RQ2: Effects on Time Spent on Testing

To see how the use of EVOSUITE influences the testing behaviour, we consider the actions performed on the unit tests (running, running in debug mode, measuring coverage), and we consider the time spent on the test suite. All these values are provided by the Rabbit plug-in.

Table 4 summarises the behaviour of users of EVOSUITE and manual testers: EVOSUITE users measured the coverage of their tests more often than manual testers for all classes but FixedOrderComparator. This suggests that a common way to investigate generated tests is to see what code they cover. EVOSUITE users ran their tests in debug mode slightly more often, but overall had fewer test runs.

The time spent on test classes is measured by calculating the total time spent on any file with name ending in `*Test.java`. This time is significantly lower for users of EVOSUITE for all classes except FilterIterator. This is expected behaviour: Using an automated testing tool is supposed to reduce the workload of the developer, and our experiment shows that by using EVOSUITE our participants spent less time, sometimes even achieving higher coverage.

RQ2*: In our experiment, using automated unit test generation reduced the time spent on testing in all four classes.*

Table 5: Comparisons of average number of failures, errors, and their sum, obtained by running the golden test suites on the participants' implementations developed using EvoSuite (Assisted) and without, manually (Manual). We also report the pooled standard deviation (sd), non-parametric effect size $\hat{A}_{12}$, p-value of U-test (U pv), parametric Cohen d effect size, p-value of T-test (T pv), and its minimal required sample size N resulted from a power analysis at $\alpha = 0.05$ and $\beta = 0.8$.

Class	Measure	Assisted	Manual	sd	$\hat{A}_{12}$	U pv	d	T pv	N
FilterIterator	Failures	1.91	2.45	1.62	0.33	0.18	-0.34	0.44	139
	Errors	4.18	3.82	2.67	0.62	0.35	0.14	0.76	847
	Fa.+Er.	6.09	6.27	2.20	0.51	0.95	-0.08	0.85	2307
FixedOrderComparator	Failures	1.33	1.73	2.13	0.38	0.37	-0.19	0.68	459
	Errors	2.89	2.55	2.80	0.52	0.91	0.12	0.79	1044
	Fa.+Er.	4.22	4.27	2.26	0.45	0.74	-0.02	0.96	31438
ListPopulation	Failures	4.91	3.10	3.14	0.67	0.20	0.58	0.20	48
	Errors	1.45	2.20	1.53	0.38	0.36	-0.49	0.28	67
	Fa.+Er.	6.36	5.30	2.73	0.65	0.27	0.39	0.39	104
PredicatedMap	Failures	4.90	5.67	7.55	0.41	0.43	-0.10	0.83	1523
	Errors	10.70	8.67	12.52	0.54	0.78	0.16	0.73	596
	Fa.+Er.	15.60	14.33	15.30	0.50	1.00	0.08	0.86	2290

Table 4: Comparisons between participants using EvoSuite (Assisted) and without (Manual) in terms of how many times on average (per participant) the test suites were run, executed with a debugger and with a code coverage tool, as well as the average number of minutes spent on the tests in the Eclipse code editor.

Class	Property	Assisted	Manual	d	T pv	N
FilterIterator	JUnit coverage	9.00	6.00	0.33	0.47	143
	JUnit debug	0.50	0.09	0.43	0.44	84
	JUnit run	7.80	13.73	-0.66	0.15	37
	Minutes spent on tests	18.52	20.03	-0.22	0.62	318
FixedOrderComparator	JUnit coverage	1.89	9.56	-1.35	0.05	10
	JUnit debug	0.22	0.11	0.22	0.66	319
	JUnit run	7.22	12.89	-0.55	0.27	53
	Minutes spent on tests	9.31	15.80	-0.86	0.07	22
ListPopulation	JUnit coverage	5.89	4.00	0.35	0.49	131
	JUnit debug	0.56	0.00	0.67	0.35	36
	JUnit run	8.22	11.00	-0.36	0.48	125
	Minutes spent on tests	12.61	24.96	-1.33	0.01	10
PredicatedMap	JUnit coverage	5.27	6.38	-0.19	0.72	441
	JUnit debug	0.00	0.88	-0.97	0.21	18
	JUnit run	4.36	5.62	-0.20	0.68	395
	Minutes spent on tests	7.70	14.35	-1.07	0.03	15

3.3 RQ3: Effects on Implementation Quality

To measure the quality of the resulting implementations, we executed the golden test suite on each participant's implementation and measured the number of test failures and errors. In the JUnit framework, a "failure" means that a JUnit assertion (e.g. `assertTrue`) in the test failed, while an "error" means that an unexpected exception in the test execution propagated to the JUnit test. The more tests of the golden test suite fail on a participant's implementation, the more faults there likely are in that implementation. However, note that the number of failures and errors does not necessarily equal the number of faults in the implementation, as several tests may fail due to the same fault.

Table 5 summarises these results, and shows the statistical comparison between users of EvoSuite and manual testers for each class. For all classes except ListPopulation, the number of failures is smaller and the number of errors larger for users of EvoSuite. While the best performing participants had the same number of errors/failures for FixedOrderComparator (1), PredicatedMap (1), and FilterIterator (3) with and without using EvoSuite, for ListPopulation the best performing participant (1) used EvoSuite (vs. 3 for the best manually testing participant). However, the values are generally very close (considering sum of failures and errors, there is no/negligible effect on FilterIterator and PredicatedMap, a small negative effect on ListPopulation, and a small positive effect on FixedOrderComparator). Consequently, our experiment produced no clear evidence for either better or worse performance resulting from the use of EvoSuite.

***RQ3**: Our experiment provided no evidence that software quality changes with automated unit test generation.*

3.4 RQ4: Effects of Time Spent with EvoSuite

Using EvoSuite reduces the time spent on testing, but is this a good thing? If tests are generated automatically, there is the danger that testing is done in a more sloppy way — if developers just take generated tests for granted and invest less time in using them to find errors, this may lead to overall worse results. To see whether this is the case, we look at the correlation between the (lack of) correctness of the resulting participants' implementations and *a)* the number of EvoSuite runs, and *b)* the time spent on the generated tests.

Table 6 summarises the Pearson's correlations, together with the 95% confidence intervals and size N for the power analysis. For FilterIterator, ListPopulation, and PredicatedMap there is a moderate relationship, only for FixedOrderComparator there is no correlation. This suggests that the more often participants invoked EvoSuite, the more errors and failures their resulting implementation had. Conversely, the correlation between the time spent on the generated tests has a weak negative relationship with the errors and failures for FilterIterator and FixedOrderComparator, and a strong negative relationship for PredicatedMap. Only ListPopulation shows no correlation here. Thus, the more time participants spent working with the tests generated by EvoSuite, the better the implementation. Running EvoSuite often and working only superficially with the generated tests has a negative effect.

***RQ4**: Our experiment suggests that the implementation improves the more time developers spend with generated tests.*

3.5 Participants Survey

After the experiment, all participants filled in an exit survey consisting of 42 items (39 agreement questions and three open text questions), which provided us feedback on the difficulties they faced during the experiment. Due to space reasons, we only describe the most important aspects of this survey here.

Figure 3 summarises the responses to eight questions on the difficulty of the tasks, using a 5-point Likert scale. Note that each participant performed one experiment with manual testing and one supported by EvoSuite, thus all participants answered these questions. In general, participants agree more that they had enough time

Table 6: For participants using EVOSUITE, correlation analysis between lack of correctness (measured with the sum of failures and errors, cf. Table 5) and *a)* the number of times EVOSUITE is run, and *b)* the time spent on the generated test suites.

Class	Property	Correlation	CI	N
FilterIterator	EVOSUITE runs	0.35	[-0.32, 0.78]	63
	Time spent on tests	-0.29	[-0.76, 0.38]	93
FixedOrderComparator	EVOSUITE runs	-0.03	[-0.65, 0.61]	8063
	Time spent on tests	-0.22	[-0.75, 0.48]	163
ListPopulation	EVOSUITE runs	0.32	[-0.35, 0.77]	76
	Time spent on tests	0.02	[-0.59, 0.61]	24938
PredicatedMap	EVOSUITE runs	0.35	[-0.32, 0.78]	63
	Time spent on tests	-0.49	[-0.84, 0.15]	30

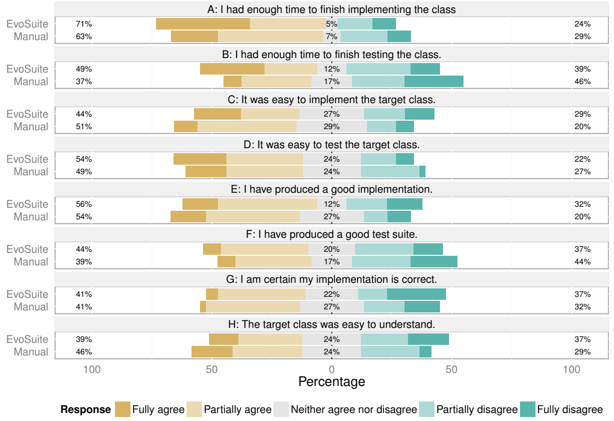


Figure 3: Agree/disagree Likert chart for the exit questionnaire, where the same question is asked for both the cases in which EVOSUITE was and was not used. (Each participant worked on two classes, one with and one without EVOSUITE.)

to finish the implementation and the test suite when using EVOSUITE. They also agree more strongly that testing was easy, and that they had produced a good test suite when using EVOSUITE. On the other hand, they claimed to have had more difficulties in implementing the target class, and to be less certain about the correctness of their implementation when using EVOSUITE. This view is supported by our data (cf. **RQ1** and **RQ3**): Using automatically generated unit tests may lead to better tests, but not necessarily to better implementations.

Interestingly, when asked about their experience with EVOSUITE, most participants seemed quite happy with the tool: There is strong agreement that it helped testing, but also slight agreement that it helped with the implementation. The assertions selected by the tool do not show up as a problem, but general readability of the tests does. There is strong agreement that the difficulty of the class under test is a deciding factor, and although one might think that using automatically generated tests may help in understanding the class, participants tended to agree that the generated tests only covered the "easy" behaviour.

When queried on how EVOSUITE could be improved, out of eight options readability and coverage received the highest agreement, while the user interface and the number and complexity of tests received the smallest agreement. A free text question on suggestions for improvement received 19 responses about readability and 11 suggestions to add comments to tests.

A free text question on what is perceived as most difficult when manually writing tests resulted in most responses related to the difficulty of understanding the class under test (14), followed by the difficulty of covering all cases during testing — a problem for which automated unit test generation is well suited. On the other hand, the question on what is most difficult when writing unit tests supported by EVOSUITE received 26 responses related to the readability of the generated unit tests, whereas understanding the class under test and other problems were only mentioned a few times.

3.6 Statistical Implications for Future Studies

The empirical study conducted in this paper is the first that aims at studying the effects of an automated test case generation tool during software development. We analysed different properties of interest, but in most cases the results of the statistical analyses were not significant: there was not enough power to reject many of the null hypotheses, although non-negligible effect sizes were present.

In statistics, this usually would happen when there is large variance in the compared distributions. The solution would be to employ larger sample sizes, although that can be a major challenge in software engineering research. But how many participants to employ? After a first study, one can calculate effect sizes and perform power analysis to find the required minimal sample size N. Unfortunately, in our context, the N values are so high (cf. Table 2–6) that suitable sample sizes are hardly feasible in constrained academic environments. However, we plan and encourage further studies in this domain, such that, over time, meta-analyses based on several studies can lead to a reliable body of knowledge, from which to draw reliable conclusions.

One possible explanation for our result is that other factors influencing software development, such as programming skills and experience of the developers, may have larger effects than an automated unit test generation tool. This suggests that future studies should control better for such factors, but it also suggests that the usefulness of automated unit test generation tools and the best way to apply them will vary between programmers with different degrees of experience and skills. As a first step towards a better understanding of this problem, the next section takes a closer look at how developers interact with EVOSUITE.

4. THINK-ALOUD STUDY

The controlled experiment described in the previous section evaluated the effects of providing an automated test generation tool to developers while coding. This is not a well-established use-case, but rather a foray into a new territory of automated testing. Hence, our usage scenario (developers applying the tool based on their intuition, copying tests they want to keep) may not be perfectly aligned with developers' needs. Although our data already allowed us to see correlations between tool usage and effectiveness, the question remains what constitutes a better strategy to make the most out of the automated test generation tool. To address this, we conducted one-on-one think-aloud observations with professional developers.

Think-aloud observations [31,32] are a popular qualitative method in usability testing [33], where they are regarded among the most valuable methods to elicit reports of thought sequences. In contrast to other qualitative methods such as interviews, which rely on the participant's ability to memorise and recollect thoughts, think-aloud observations allow to educe information about behaviour *during* the completion of a given task.

4.1 Method

We recruited five professional Java software developers from industry and academia to participate in the think-aloud observations. Recruiting consisted in sending personal emails and advertising on the IRC channel of a local software developers' community from Sheffield, UK. All selected participants have solid experience in Java programming and testing and are familiar with the JUnit Testing

Framework. They have no or little experience with EVOSUITE, and none of them has participated in a think-aloud observation before.

Each participant is assigned one of the classes described in Table 1 (one participant per class, and the randomly chosen PredicatedMap class for the fifth participant). All participants are given the *assisted* task of implementing their assigned class and an accompanying test suite with the help of EVOSUITE. As in the controlled experiment, their starting point is the skeleton of their target class and an empty test suite. They are free to write their own unit tests manually, to use EVOSUITE to automatically generate unit tests and utilise those, or a combination of both approaches. For practicality, and considering the cognitive overhead that thinking aloud represents, we imposed a time limit of two hours to complete the task. We carried out a pilot observation with a PhD student which helped assess the viability of our setup and anticipate problems.

The observations were conducted by the first author of this paper. The data collection mechanism was similar to that used in our main experiment (Section 2.6). In addition, qualitative data was collected by taking notes, and by recording the participant's speech and the computer screen during the observations. The notes taken during the session were later augmented with analysis of the speech and screen recordings.

Before the observations, participants received a coaching talk, where the think-aloud method was explained, unit testing and the EVOSUITE tool were described, and their assigned task was introduced. Participants were asked to permanently verbalise their thoughts as they worked on the task. Interaction between the observer and participants was limited to the minimum possible during the observations. Participants were only prompted to continue verbalising when they had stopped doing so. In order to elicit information that could not be expressed verbally during the observation, we introduced a retrospective phase upon completion of the task where participants were invited to reflect on the experience, recollect their thoughts and articulate some final remarks.

4.2 Threats to Validity

The think-aloud observations share large parts of the setup from the controlled experiment, and thus also share the main threats to validity not related to participant selection or assignment (Section 2.9). In addition, there are well-known limitations to think-aloud observations. First, their outcome is influenced by the participant's degree of articulation. Our study does not escape this intrinsically human caveat, but fortunately only one participant showed some discomfort in this regard at the beginning of his session. Another limitation is that participants tend to describe their actions instead of providing insight on their thinking process. In our observations, participants were often reading aloud the provided material or narrating their actions rather than vocalising their thoughts. We tried to alleviate this with careful scrutiny of participants' behaviour and by prompting them to elaborate more whenever we thought necessary. Finally, as in most empirical methods, there is the threat of participants being too conscious about the experimental setup. In particular, three of the five participants admitted they tested their classes more thoroughly than they would normally do for work. We can only embrace this behaviour as an opportunity to gain more insight of their testing habits and their expectations from test generation tools.

4.3 Results and Interpretation

The observations took place between 21 July and 13 August 2014 and lasted 1:20 to 2:00 hours. Four of the five participants completed the task in the given time frame. Only in a few occasions it was necessary to provide technical guidance to participants about their tasks. In exceptional cases, when we observed interesting or unusual behaviour or to stimulate the less articulate participants, we also prompted questions such as "What are you thinking about?" and "Why did you do that?". Table 7 summarises the results of the five observations, and this section describes their most relevant aspects.

Participant 1 behaved according to our expectations from someone who has not used a test generation tool before. He first focused on shaping up the implementation, and then wrote some tests manually. He ran EVOSUITE for the first time 56 minutes into the task, at which point he had already implemented most of the specified behaviour and a sensible test suite. When deciding to use EVOSUITE, he claimed that testing thoroughly even the small target class was getting tedious and that he could have been more efficient had he used EVOSUITE earlier. He reckoned he could compare an EVOSUITE-generated test suite with his own manual test suite in terms of coverage, and "maybe" combine them together.

On the first run, EVOSUITE generated six tests. Unfortunately, most of them were testing code that the participant had already written tests for, failing to exercise relevant yet uncovered branches, which confused and disappointed him. After reflecting about the non-deterministic nature of EVOSUITE, he decided to run it again. This second time he got 97% branch coverage, whereas his manual test suite had achieved 85%. While comparing coverage of the two tests suites, he realised that one of his tests was now failing. Thus, instead of inspecting the generated tests to try to utilise those that could improve the coverage of his test suite, he switched focus to try to discover why that test was failing.

Later, after 30 minutes refining his implementation and test suite, he ran EVOSUITE again. The newly generated test suite achieved 100% coverage. Although his test suite also reached 100% coverage, he decided to enhance it with generated tests, since they were exercising interesting scenarios that he had not thought of before. For instance, EVOSUITE generated a test case in which the method `remove` was invoked right after the initialisation of a FilterIterator object, which he had not considered in his manual tests. *"Those are the tests that are nice to have automatically generated"*, he said.

Participant 2 applied a testing approach that benefited highly from automated test generation tools support. At the beginning of the task, he focused on coding his target class, even making some preliminary assumptions in order to complete a first running version. It took him 19 minutes to do so, and then he decided to run EVOSUITE for the first time. Running EVOSUITE as soon as there is some behaviour seems to be a good practice since time spent on writing unit tests for the most trivial behaviour can be drastically reduced. The tool's output helped him detect mis-implemented behaviour. Specifically, one test case contained an invocation of a method in the target class with null arguments and produced normal output, whereas the specification said it should raise an exception. He implemented this behaviour and re-ran EVOSUITE. Re-generating test suites after important changes in the implementation is also a good practice: it shows that the participant is aware that the tool will generate a test suite that reflects implemented behaviour only.

After three cycles of automatically generating unit tests, inspecting, refactoring, and using them to refine his target class and test suite, the participant decided to take a more coverage-oriented approach to utilising the generated tests. He ran EVOSUITE for the sixth time, copied *all* the generated tests into his own test suite, and systematically commented tests out one by one, removing those that did not have an impact on the coverage of his test suite. In the end, his test suite consisted entirely of unit tests either refactored or copied verbatim from the EVOSUITE output. Importantly, this observation gives prominence to the need of more usable integration of the test generation tool: *"What would be really useful here is if EvoSuite could tell me which of the test that is generated cover*

Table 7: Results of Think-aloud Observations

Participant	Target Class	Duration	EvoSuite	Number	Cov. Own Impl.		Mutation	Cov. Golden Impl.		Failure+Errors	
ID			times run	of tests	Instruction	Branch	Score	Instruction	Branch	Failure	Error
1	FilterIterator	01:50	3	15	100.0%	93.8%	63.0%	98.2%	91.7%	1	0
2	FixedOrderComparator	01:20	6	20	100.0%	97.3%	78.7%	85.7%	72.5%	1	0
3	ListPopulation	02:00	2	23	97.6%	94.4%	63.3%	89.3%	95.5%	1	1
4	PredicatedMap	01:38	2	7	100.0%	100.0%	60.0%	95.9%	93.8%	1	0
5	PredicatedMap	01:40	5	10	100.0%	100.0%	100.0%	82.8%	81.2%	0	0

new behaviour that my test suite doesn't, because I am doing that manually at this stage."

Participant 3 applied a test-driven approach to the assigned task where EvoSuite, and probably any code-based automated test generation tool, has less opportunity to assist. Test-driven developers are used to writing (failing) tests *before* implementing the specified behaviour. Hence, once the target class is completed, a big portion of the test suite has also been produced, although the resulting test suite could still be extended with automatically generated unit tests. However, we did not observe this use case with this participant, likely because he did not manage to finish the implementation within the time constraint imposed in our experimental setup.

The participant ran EvoSuite for the first time at minute 52, after implementing the constructors for his target class, and before implementing any other method. When prompted to articulate why he had decided to run EvoSuite, he said he expected to get some tests exercising exceptional behaviour. However, while articulating his answer, he realised that the tool would only produce tests for observed behaviour, which he had not yet implemented. Unsurprisingly, he decided not to use any of the generated test cases. Twenty minutes later, with more behaviour implemented, he ran EvoSuite again. This time he found some useful tests. However, instead of importing them to his test suite, he decided to imitate their intention: *"I didn't like that generated code, so I'm going to write those tests manually."* After that, he continued the task entirely manually and did not run EvoSuite or inspect generated tests again.

The participant did not complete the implementation of the target class in the allocated time. Whereas we have no reason to believe that he could have performed better with a different use of the tool, we do believe that usability must be addressed to help test-driven developers. In fact, he pointed out a feature that could have helped him: *"It would be nice if I could generate tests for a particular method, or involving a particular method."* In the retrospective phase, he elaborated more on this thought, implying that a method-level test generation approach would have helped him develop and maintain incremental confidence in the code he was writing.

Participant 4 was the least comfortable with the format of the think-aloud observation, and we needed to prompt him several times to keep verbalising.

Once he fleshed out the implementation of his target class, he ran EvoSuite. Interestingly, after generating tests and refactoring them, he did not *run* them. The first time he actually ran tests was 87 minutes into the task. He got several failures, fixed some assertions, then debugged one test and fixed a serious bug. Unlike participants who focused more on coverage, he showed more interest in fully understanding each generated test. Whereas this is a good practice in general (see results of **RQ4**), it seemed to be detrimental for this participant, since he resented several readability aspects of the generated tests and therefore spent much effort refactoring tests before importing them to his test suite: *"I used generated tests as starting point; I changed them to look more like how I would have written them."* Indeed, it seemed that EvoSuite generates lengthy and hard to read tests for PredicatedMap, which likely also explains the results for **RQ1** for this class.

Interestingly, once his implementation and test suite had grown considerably, he ran EvoSuite again and tried a split-window view to compare his and EvoSuite's test suite. As with Participant 2, this suggests the need for better integration of test generation tools. Finally, he also acknowledged being conscious about the experimental setup: *"I'd probably be a lot stingier in the number of tests that I'd implement. I wouldn't normally in real life be aiming for 100% coverage. I'd probably end up with fewer tests without this tool but I couldn't tell you if they would be all the right tests."*

Participant 5 ran EvoSuite before writing a single line of code, out of curiosity about what EvoSuite would produce for one particular method, which was not yet implemented. After looking at the resulting tests, he was comprehensibly not satisfied with the output. He then decided to flesh out the target class; once he had implemented one of its public methods, he went back to the previously generated test suite to look for a suitable test. A better use would have consisted of re-generating the test suite and then searching for a test exercising that specific method. He realised this after some minutes inspecting the generated tests and thus ran EvoSuite again. He then acknowledged that generated tests had changed to something more meaningful, but reckoned he still needed to add more behaviour to his target class. After 35 minutes working on the target class and writing manual tests, he ran EvoSuite again. This time, and after inspecting the generated tests for less than two minutes, he claimed that generated tests were too complex and decided to proceed manually. Again, this readability problem for PredicatedMap is also reflected in the results for **RQ1** of our controlled experiment.

Upon completion of the task, the participant ran EvoSuite one last time to compare code coverage. Whereas he had achieved 100% manually, the generated test suite achieved 87.5%. When asked to reflect about his experience, the participant made a case on how important readability is: *"Coverage is easy to assess because it is a number, while readability of the tests is a very non-tangible property. What is readable to me may not be readable to you. It is readable to me just because I spent the last hour and a half doing this."*

4.4 Lessons Learned

Software developers use different strategies for coding and testing tasks. Thus, automated unit test generation tools, and EvoSuite in particular, should also provide different mechanisms to assist each testing strategy. Based on the think-aloud observations we identified three main strategies: *traditional*, where developers first complete implementation and then write unit tests; *test-driven*, where unit tests are produced before the actual implementation; and *hybrid*, where unit tests are written incrementally to exercise behaviour as it is implemented. Our observations suggest that a code-based tool like EvoSuite can prove more useful for the traditional or hybrid strategy. Nevertheless, test-driven developers could also benefit from EvoSuite by complementing their test suites once the implementation is complete.

As already suggested by the results of our main experiment, readability of unit tests is paramount. Although a coverage-oriented unit testing approach would allow developers to utilise generated tests without entirely understanding them, the quality of a test case seems to highly depend on how easy eliciting its intended behaviour is.

We also learned that code coverage does not seem to be the driving force for professional developers on a coding and testing task. For instance, Participant 3 did not check coverage at all while working on his task, while Participant 4 did so only after 45 minutes of work. Moreover, two of our participants admitted to have tested their target class more thoroughly than they would normally do at work, only because the goal of their task was to achieve high code coverage.

The education of professional developers on the use of automated test generation tools is important: Although all participants were instructed and practised the use of EVOSUITE, *efficient* use of the tool requires more experience and establishment of best practices. For instance, Participant 1 realised that he could have used generated tests for trivial behaviour only after having written a number of manual tests. Another interesting example is Participant 5, who first ran EVOSUITE on an almost empty class, and later, ran it again on a partially implemented class, but invested less than two minutes inspecting the generated test suite before deciding to ignore it and proceed manually. These observations suggest that, independently of their testing strategy, software developers need to be educated on how and when to use automated unit testing tools, and our experiments are a step towards identifying the best practices.

The observations also unveiled potential for improvement of EVOSUITE in terms of usability. For instance, providing an automated mechanism to optimally combine existing test suites with automatically generated ones could play an important role in the successful integration of EVOSUITE during software development. As Participant 2 suggested, such a feature could save much time and effort in realistic scenarios. We might be looking at a level of integration of EVOSUITE into the Eclipse IDE similar to that of the Pex white-box unit testing tool for .NET into the MS Visual Studio IDE [3].

5. RELATED WORK

Empirical studies with human subjects are still rare in software engineering research; for example, Sjøberg et al. [34] found controlled studies with human subjects in only 1.9% of 5,453 analysed software engineering articles, a trend that was confirmed by Buse et al. in their survey of 3,110 software engineering articles [35]. Indeed, a recent survey [36] acknowledges the need for empirical studies involving human subjects as an important step towards transferring software testing research into industrial practice.

In previous work [12], we studied the effect of using EVOSUITE for testing only. Participants were presented with the implementation of a class and asked to produce a unit test suite for it. Results showed that automated unit test generation can support testers in producing better test suites but does not help in finding more faults. In this work, we consider the application of EVOSUITE by developers writing their own code, rather than testers writing tests for existing code. Participants only received specifications in JavaDoc form and were asked to produce both implementation and test suites.

This is also the main difference with respect to related studies: Ceccato et al. [37] conducted two controlled experiments to study the impact of randomly generated tests on debugging tasks and found a positive influence. Ramler et al. [38] empirically compared manual testing with testing assisted by automated test generation tools, and found that fault detection rates were similar, although each method revealed different kinds of faults. In these two studies, the implementations of the classes under test were provided.

Saff and Ernst [39] found statistically significant evidence that student developers were more productive on programming tasks if they were provided with the extra feedback of regression tests continuously running in the background and notifications popping up as soon as a new error was discovered. Although this work relied on existing test suites, it suggests an interesting pathway to integrate test generation more seamlessly into programming activities.

Hughes and Parkes [40] surveyed studies that applied think-aloud observations in software engineering topics such as program understanding for maintenance tasks [41] and debugging [42]. More recently, Roehm et al. [43] used think-aloud observations and interviews to assess program comprehension by 28 professional developers. Owen et al. [44] and Whalley and Kasto [45] used the think-aloud method to study the way in which developers approached and solved programming tasks. Ostberg et al. [46] combined qualitative methods, including think-aloud observations, to study how useful was automatic static program analysis for bug detection.

6. CONCLUSIONS

Since early work on automated test data generation a common line of reasoning is that, as testing is time consuming and difficult, automating any of its steps will be beneficial. It is plausible that generating tests to exercise specifications or assertions improves software quality by demonstrating violations. On the other hand, many testing tools make no assumptions on the availability of specifications, and just aim to maximise code coverage. Whether this is actually of benefit for developers writing code is something that, to the best of our knowledge, has not been investigated before.

In this paper, we investigated this scenario using two empirical methods: First, we conducted a controlled experiment to measure any effects observable when providing developers with the EVOSUITE unit test generation tool. Second, we observed professional programmers while implementing code assisted by EVOSUITE. Our experiments provide reasonable evidence that using EVOSUITE can have a beneficial effect on resulting test suites. On the other hand, the effects on the quality of the implemented code depend on how the unit test generation tool and its unit tests are used.

Our experiments suggest concrete next steps to improve the effect automated unit test generation can have on developers:

Readable unit tests. Automatically generated unit tests must be easy to read and understand (cf. [47,48]), otherwise developers may be distracted or stop using the tool.

Integration into development environments. In the spirit of continuous testing, test generation tools should evolve test suites (cf. [49–52]) and integrate with existing tests.

Best practices. As automated unit test generation is not commonly used in practice currently, developers need to be educated on the best practices, which we are trying to establish; e.g., a good approach is to apply test generation whenever a new significant part of behaviour has been implemented.

Our experiments are naturally limited, but an important first step towards understanding how automated unit test generation can support developers. It will be important to complement these results with further empirical studies, such as replications with (1) different (larger) population samples, (2) other test generation tools, and (3) enhanced integration of the unit test generation tools into the development environment.

To support replicability, all the experimental material is made available at `http://www.evosuite.org/study-2014/`.

7. ACKNOWLEDGEMENTS

Supported by the National Research Fund, Luxembourg (FNR/P10/03) and the EPSRC project “EXOGEN” (EP/K030353/1). Thanks to P. McMinn, M. Gheorghe, N. Walkinshaw, G. Badkobeh, R. Roller, and R. Lefticaru for comments on earlier versions of this paper.

8. REFERENCES

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

[2] B. Meyer, I. Ciupa, A. Leitner, and L. L. Liu, "Automatic testing of object-oriented software," in *SOFSEM'07: Theory and Practice of Computer Science*, ser. LNCS, vol. 4362. Springer-Verlag, 2007, pp. 114–129.

[3] N. Tillmann and J. N. de Halleux, "Pex — white box test generation for .NET," in *Int. Conference on Tests And Proofs (TAP)*, ser. LNCS, vol. 4966. Springer, 2008, pp. 134 – 253.

[4] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Softw. Pract. Exper.*, vol. 34, pp. 1025–1050, 2004.

[5] P. Tonella, "Evolutionary testing of classes," in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 119–128.

[6] J. H. Andrews, F. C. H. Li, and T. Menzies, "Nighthawk: a two-level genetic-random unit test data generator," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2007, pp. 144–153.

[7] L. Baresi, P. L. Lanzi, and M. Miraz, "TestFul: an evolutionary test approach for Java," in *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, 2010, pp. 185–194.

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

[9] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "CarFast: achieving higher statement coverage faster," in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2012, pp. 35:1–35:11.

[10] (2014) Agitar One. [Online]. Available: http://www.agitar.com

[11] (2014) Parasoft JTest. [Online]. Available: http://www.parasoft.com/jtest

[12] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated white-box test generation really help software testers?" in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2013, pp. 291–301.

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

[14] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. Sjøberg, Eds. Springer London, 2008, pp. 201–228.

[15] (2014) Apache Commons Libraries. [Online]. Available: http://commons.apache.org/

[16] (2014) JavaNCSS - a source measurement suite for Java. Version 32.53. [Online]. Available: http://www.kclee.de/clemens/java/javancss

[17] (2014) EclEmma - Java code coverage for Eclipse. Version 2.3.1. [Online]. Available: http://www.eclemma.org/

[18] (2014) Rabbit - Eclipse statistics tracking plugin. Version 1.2.1. [Online]. Available: https://code.google.com/p/rabbit-eclipse/

[19] N. Li, X. Meng, J. Offutt, and L. Deng, "Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report)," in *IEEE Int. Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 380–389.

[20] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 433–436.

[21] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, 2005, pp. 402–411.

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

[23] M. Fay and M. Proschan, "Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules," *Statistics Surveys*, vol. 4, pp. 1–39, 2010.

[24] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability (STVR)*, vol. 24, no. 3, pp. 219–250, 2012.

[25] R. Grissom and J. Kim, *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum, 2005.

[26] J. Cohen, "A power primer," *Psychological bulletin*, vol. 112, no. 1, pp. 155–159, 1992.

[27] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

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

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

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

[31] K. A. Ericsson and H. A. Simon, *Protocol Analysis: Verbal Reports as Data (revised edition).* MIT Press, Cambridge, MA, 1993.

[32] K. A. Ericsson, "Valid and non-reactive verbalization of thoughts during performance of tasks - towards a solution to the central problems of introspection as a source of scientific data," *Consciousness Studies*, vol. 10, no. 9-10, pp. 1–18, 2003.

[33] S. McDonald, H. Edwards, and T. Zhao, "Exploring think-alouds in usability testing: An international survey," *IEEE Transactions on Professional Communication*, vol. 55, no. 1, pp. 2–19, March 2012.

[34] D. Sjoberg, J. Hannay, O. Hansen, V. By Kampenes, A. Karahasanovic, N. Liborg, and A. C Rekdal, "A survey of controlled experiments in software engineering," *IEEE Transactions on Software Engineering (TSE)*, vol. 31, no. 9, pp. 733–753, 2005.

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

[36] A. Orso and G. Rothermel, “Software Testing: A Research Travelogue (2000–2014),” in *ACM Future of Software Engineering (FOSE)*, 2014, pp. 117–132.

[37] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella, “An empirical study about the effectiveness of debugging when random test cases are used,” in *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, 2012, pp. 452–462.

[38] R. Ramler, D. Winkler, and M. Schmidt, “Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code?” in *IEEE Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2012, pp. 286–293.

[39] D. Saff and M. D. Ernst, “An experimental evaluation of continuous testing during development,” in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 76–85.

[40] J. Hughes and S. Parkes, “Trends in the use of verbal protocol analysis in software engineering research,” *Behaviour and Information Technology*, vol. 22, no. 2, pp. 127–140, 2003.

[41] A. M. Vans, A. von Mayrhauser, and G. Somlo, “Program understanding behavior during corrective maintenance of large-scale software,” *Int. Journal of Human-Computer Studies*, vol. 51, no. 1, pp. 31–70, 1999.

[42] J. E. Hale, S. Sharpe, and D. P. Hale, “An evaluation of the cognitive processes of programmers engaged in software debugging,” *Software Maintenance: Research and Practice*, vol. 11, no. 2, pp. 73–91, 1999.

[43] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, “How do professional developers comprehend software?” in *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, 2012, pp. 255–265.

[44] S. Owen, P. Brereton, and D. Budgen, “Protocol analysis: A neglected practice,” *Commun. ACM*, vol. 49, no. 2, pp. 117–122, 2006.

[45] J. Whalley and N. Kasto, “A qualitative think-aloud study of novice programmers’ code writing strategies,” in *ACM Conf. on Innovation and Technology in Computer Science Education (ITiCSE)*, 2014, pp. 279–284.

[46] J.-P. Ostberg, J. Ramadani, and S. Wagner, “A novel approach for discovering barriers in using automatic static analysis,” in *ACM Int. Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2013, pp. 78–81.

[47] G. Fraser and A. Zeller, “Exploiting common object usage in test case generation,” in *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, 2011, pp. 80–89.

[48] S. Afshan, P. McMinn, and M. Stevenson, “Evolving readable string test inputs using a natural language model to reduce human oracle cost,” in *Int. Conference on Software Testing, Verification and Validation (ICST)*, 2013, pp. 352–361.

[49] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, “Test-suite augmentation for evolving software,” in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2008, pp. 218–227.

[50] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, “Directed test suite augmentation: techniques and tradeoffs,” in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2010, pp. 257–266.

[51] M. Mirzaaghaei, F. Pastore, and M. Pezze, “Supporting test suite evolution through test case adaptation,” in *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, 2012, pp. 231–240.

[52] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, “ReAssert: Suggesting repairs for broken unit tests,” in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2009, pp. 433–444.