The impact of test case summaries on bug fixing performance: An empirical investigation

Automated test generation tools have been widely investigated with the goal of reducing the cost of testing activities. However, generated tests have been shown not to help developers in detecting and finding more bugs even though they reach higher structural coverage compared to manual testing. The main reason is that generated tests are difficult to understand and maintain. Our paper proposes an approach, coined TestScribe, which automatically generates test case summaries of the portion of code exercised by each individual test, thereby improving understandability. We argue that this approach can complement the current techniques around automated unit test generation or search-based techniques designed to generate a possibly minimal set of test cases. In evaluating our approach we found that (1) developers find twice as many bugs, and (2) test case summaries significantly improve the comprehensibility of test cases, which is considered particularly useful by developers.

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1467v3 | CC-BY 4.0 Open Access | rec: 18 Feb 2016, publ: 18 Feb 2016

The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation

Sebastiano Panichella,[1] Annibale Panichella,[2] Moritz Beller,[2]
Andy Zaidman,[2] Harald C. Gall[1]
[1]University of Zurich, Switzerland
[2]Delft University of Technology, The Netherlands
panichella@ifi.uzh.ch {a.panichella,m.m.beller,a.e.zaidman}@tudelft.nl gall@ifi.uzh.ch

ABSTRACT

Automated test generation tools have been widely investigated with the goal of reducing the cost of testing activities. However, generated tests have been shown not to help developers in detecting and finding more bugs even though they reach higher structural coverage compared to manual testing. The main reason is that generated tests are difficult to understand and maintain. Our paper proposes an approach, coined TestDescriber, which automatically generates test case summaries of the portion of code exercised by each individual test, thereby improving understandability. We argue that this approach can complement the current techniques around automated unit test generation or search-based techniques designed to generate a possibly minimal set of test cases. In evaluating our approach we found that (1) developers find twice as many bugs, and (2) test case summaries significantly improve the comprehensibility of test cases, which is considered particularly useful by developers.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code Inspections and Walk-throughs, Testing Tools*;
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation, Enhancement*

Keywords

Software testing, Test Case Summarization, Empirical Study

1. INTRODUCTION

Software testing is a key activity of software development and software quality assurance in particular. However, it is also expensive, with overall testing consuming as much as 50% of overall project effort [8, 36], and programmers spending a quarter of their work time on developer testing [6].

Several search-based techniques and tools [16, 21, 40] have been proposed to reduce the time developers need to spend on testing by automatically generating a (possibly minimal) set of test cases with respect to a specific test coverage criterion [11, 21, 25, 28, 41, 43, 51, 54]. These research efforts produced important results: automatic test case generation allows developers to (i) reduce the time and cost of the testing process [5, 11, 13, 54]; to (ii) achieve higher code coverage when compared to the coverage obtained through manual testing [10, 22, 41, 43, 51]; to (iii) find violations of automated oracles (e.g. undeclared exceptions) [16, 22, 35, 40].

Despite these undisputed advances, creating test cases manually is still prevalent in software development. This is partially due to the fact that professional developers perceive generated test cases as hard to understand and difficult to maintain [18, 44]. Indeed, a recent study [23, 24] reported that developers spend up to 50% of their time in understanding and analyzing the output of automatic tools. As a consequence, automatically generated tests *do not improve the ability of developers to detect faults* when compared to manual testing [12, 23, 24]. Recent research has challenged the assumption that structural coverage is the only goal to optimize [1, 56], showing that when systematically improving the readability of the code composing the generated tests, developers tend to prefer the improved tests and were able to perform maintenance tasks in less time (about 14%) and at the same level of accuracy [18]. However, there is no empirical evidence that such readability improvements produce tangible results in terms of the number of bugs actually found by developers.

This paper builds on the finding that readability of test cases is a key factor to optimize in the context of automated test generation. However, we conjecture that the quality of the code composing the generated test cases (e.g., input parameters, assertions, etc.) is not the only factor affecting their comprehensibility. For example, consider the unit test `test0`[1] in Figure 1, which was automatically generated for the target class `Option`[2]. From a bird's-eye view, the code of the test is pretty short and simple: it contains a constructor and two assertions calling `get` methods. However, it is difficult to tell, without reading the contents of the target class, (i) what is the behavior under test, (ii) whether the generated assertions are correct, (iii) which if-conditions are eventually traversed when executing the test (coverage). Thus, we need a solution that helps developers to quickly understand both tests and code covered.

Paper contribution. To handle this problem, our paper proposes an approach, coined TestDescriber, which is designed to automatically generate summaries of the portion of code exercised by each individual test case to pro-

This paper was first submitted to the International Conference on Software Engineering (ICSE) for peer review in August 2015. It got accepted at ICSE'16. This is a pre-print copyx.

[1]The test case has been generated using Evosuite [21].
[2]The class *Option* has been extracted from the *apache commons* library

```
1| public class TestOption {
2|
3| @Test
4| public void test0() throws Throwable {
5|    Option option0 = new Option("", "1W|^");
6|    assertEquals("1W|^", option0.getDescription());
7|    assertEquals("", option0.getKey());
8|  }
9| }
```

Figure 1: Motivating example

vide a dynamic view of the class under test (CUT). We argue that applying summarization techniques to test cases does not only help developers to have a better understanding of the code under test, but it can also be highly beneficial to support developers during bug fixing tasks, improving their bug fixing performance. This leads us to the first research question:

> **RQ1:** *How do test case summaries impact the number of bugs fixed by developers?*

Automatically generated tests are not immediately consumable since the assertions might reflect an incorrect behavior if the target class is faulty. Hence, developers should manually check the assertions for correctness and possibly add new tests if they think that some parts of the target classes are not tested. This leads us to our second research question:

> **RQ2:** *How do test case summaries impact developers to change test cases in terms of structural and mutation coverage?*

The contributions of our paper are summarized as follows:

- we introduced *TestDescriber* a novel approach to automatically generate natural language summaries of JUnit test cases and the portion of the target classes they are going to test;
- we conducted an empirical study involving 30 human participants from both industry and academia to investigate the impact of test summaries on the number of bugs that can be fixed by developers when assisted by automated test generation tools;
- we make publicly available a replication package[3] with (i) material and working data sets of our study, (ii) complete results of the survey; and (iii) rawdata for replication purposes and to support future studies.

2. THE TESTDESCRIBER APPROACH

This section details the TestDescriber approach.

2.1 Approach Overview

Figure 2 depicts the proposed TestDescriber approach, which is designed to generate automatically summaries for test cases leveraging (i) structural coverage information and (ii) existing approaches on code summarization. In particular, TestDescriber generates summaries for the portion of code exercised by each individual test case, thus, providing a dynamic view of the code under test. We notice that unlike TestDescriber, existing approaches on code summarization [19, 20, 34, 37, 48] generate static summaries of source code without taking into account which part of the code is exercised during test case execution. Our approach consists of four steps: ① *Test Case Generation*, ② *Test Coverage Analysis*, ③ *Summary Generation*, and ④ *Summary Aggregation*. In the first step, namely *Test Case Generation*, we generate test cases using Evosuite [21]. In the second step *Test Coverage Analysis*, TestDescriber identifies the code exercised by each individual test case generated in the previous

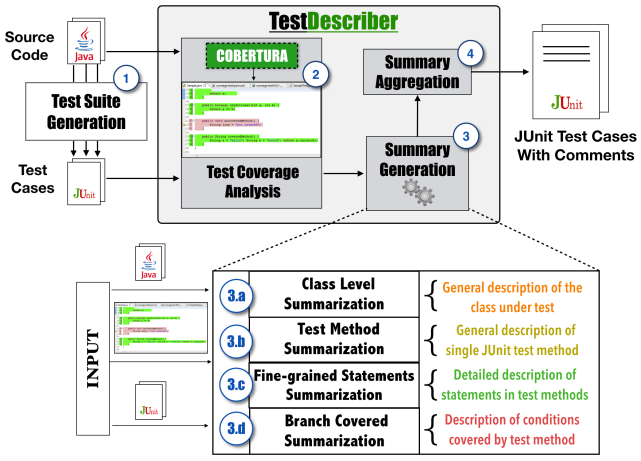

Figure 2: Overview TestDescriber

step. To detect the executed lines of code we rely on *Cobertura*[4] a tool based on *jcoverage*[5]. The goal of this step is to collect the information that will be summarized in the next steps, such as the list of statements tested by each test case, the used class attributes, the used parameters and the covered conditional statements etc. During the step *Summary Generation*, TestDescriber takes the collected information and generates a set of summaries at different levels of granularity: (i) a global description of the class under test, (ii) a global description of each test case, (iii) a set of fine-grained descriptions of each test case (describing for example statements and/or branch executed by the test case). Finally, during the *Summary Aggregation* step the extracted information and/or descriptions are added to the original test suite. An example of tests summaries generated by TestDescriber, for the test case showed in Figure 1, which tests the Java Class *Option* of the system *Apache Commons CLI*[6], can be found in Figure 3. The complete example of generated test suite for such class is available online [7].

2.2 Test Suite Generation

Researchers have proposed several methods capable of automatically generating test input based on the source code of the program under test based on different search strategies, such as genetic algorithms [21, 41], symbolic execution [10], etc. Among them, we have selected Evosuite [21], a tool that automatically generates JUnit test cases with JUnit assertions for classes written in Java code. Internally, Evosuite uses a genetic algorithm to evolve candidate test suites (individuals) according to the chosen coverage criterion where the search is guided by a fitness function [21], which considers all the test targets (e.g., branches, statements, etc.) at the same time. In order to make the test cases produced more concise and understandable, at the end of the search process the best test suite is post-processed to reduce its size while preserving the maximum coverage achieved. The final step of this post-processing consists of adding test assertions, i.e., statements that check the outcome of the test code. These assertions are generated using a mutation-based heuristic [25], which adds all possible assertions and then selects the minimal subset of those able to reveal mutants injected in the code. Consequently, the final test suite serves

[3] http://dx.doi.org/10.5281/zenodo.45120

[4] http://cobertura.github.io/cobertura/

[5] http://java-source.net/open-source/code-coverage/jcoverage-gpl

[6] https://commons.apache.org/proper/commons-cli/

[7] http://www.ifi.uzh.ch/seal/people/panichella/TestOption.txt

as starting point for a tester, who has to manually revise the assertions. It is important to note that *the use of Evosuite is not mandatory* in this phase of the TestDescriber, indeed, it is possible to rely on other existing tools such as *Randoop* [8] to generate test cases. However, we select Evosuite since (1) it generates minimal test cases with the minimal set of test assertions reaching high structural coverage [23, 24] and (2) it reached top-2 in last 3 SBST tool competitions.

2.3 Test Coverage Analysis

Once the test cases are generated, TestDescriber relies on *Cobertura*, to find out which statements and branches are tested by each individual test case. However, with the aim at generating tests summaries for the covered information we need more fine-grained information regarding the code elements composing each covered statement, such as attributes, method calls, the conditions delimiting the traversed branches, etc. In the next step TestDescriber extracts keywords from the identifier names of such code elements, to build the main *textual* corpus required for generating the *coverage summaries*. Therefore, on top of Cobertura we built a parser based on JavaParser[9] to collect the following information after the execution of each test case: (i) the list of *attributes* and *methods* of the CUT directly or indirectly invoked by the test case; (ii) for *each invoked method* our parser collects all the *statements* executed, the *attributes/variables* used and *calls to other methods* of the CUT; (iii) the Boolean values of branch decisions in the if-statements to derive which conditions are verified when covering a specific true/false branch of the CUT. The *output* of this phase is represented by the list of fine-grained *code elements* and the *lines of code* covered by each test case.

2.4 Summary Generation

The goal of this step is to provide to the software developer a higher-level view of which portion of the CUT each test case is going to test. To generate this view, TestDescriber extracts natural language phrases from the underlying covered statements by implementing the well known Software Word Usage Model (SWUM) proposed by Hill *et al.* [30]. The basic idea of SWUM is that *actions*, *themes*, and any *secondary arguments* can be derived from an arbitrary portion of code by making assumptions about different Java naming conventions, and using these assumptions to link linguistic information to programming language structure and semantics. Indeed, method signatures (including class name, method name, type, and formal parameters) and field signatures (including class name, type, and field name) usually contain *verbs*, *nouns*, and *prepositional phrases* that can be expanded in order to generate readable natural language sentences. For example, *verbs* in method names are considered by SWUM as the *actions* while the *theme* (i.e., subjects and objects) can be found in the rest of the name, the formal parameters, and then the class name.

Pre-processing. Before identifying the linguistic elements composing the covered statements of the CUT, we split the identifier names into component terms using the Java camel case convention [30, 48], which splits words based on capital letters, underscores, and numbers. Then, we expand abbreviations in identifiers and type names using both (i) an external dictionary of common short forms for English words [45] and (ii) a more sophisticated technique called *contextual-based expansion* [29], that searches the most appropriate expansion for a given abbreviation (contained in class and method identifiers).

Part-of-speech tagging. Once the main terms are extracted from the identifier names, TestDescriber uses *LanguageTool* [10], a Part-of-speech (POS) tagger to derive which terms are *verbs* (actions), *nouns* (themes) and *adjectives*. Specifically, *LanguageTool* is an open-source Java library that provides a plethora of linguistic tools (e.g., spell checker, POS tagger, translator, etc.) for more than 20 different languages. The output of the POS tagging is then used to determine whether the names (of method or attribute) should be treated as Noun Phrases (NP), Verb Phrases (VP), and Prepositional Phrases (PP) [30]. According to the type of phrase, we used a set of heuristics similar to the ones used by Hill *et al.* [30] and *Sridhara et al.* [48] to generate natural language sentences using the pre-processed and POS tagged variables, attributes and signature methods.

Summary Generation. Starting from the noun, verb and prepositional phrases, TestDescriber applies a template-based strategy [34, 48] to generate summaries. This strategy consists of using pre-defined templates of natural language sentences that are filled with the output of SWUM, i.e., the pre-processed and tagged source code elements in covered statements. TestDescriber creates three different types of summaries at different levels of abstractions: (i) a general description of the CUT, which is generated during a specific sub-step of the Summary Generation called *Class Level Summarization*; (ii) a brief summary of the structural code coverage scores achieved by each individual JUnit test method; (iii) a fine grained description of the statement composing each JUnit test method in order to describe the flow of operations performed to test the CUT. These fine-grained descriptions are generated during two different sub-steps of the Summary Generation: the *Fine-grained Statements Summarization* and the *Branch Covered Summarization*. The first sub-step provides a summary for the statements in the JUnit test methods, while the latter describes the if-statements traversed in the executed path of the CUT.

Class Level Summarization. The focus of this step is to give to a tester a quick idea of the responsibility of the class under test. The generated summary is especially useful when the class under test is not well commented/documented. To this end we implemented an approach similar to the one proposed by Moreno *et al.* in [37] for summarizing Java classes. Specifically, Moreno *et al.* defined a heuristics based approach for describing the class behavior based on the most relevant methods, the superclass and class interfaces, and the role of the class within the system. Differently, during the *Class Level Summarization* we focus on the single CUT by considering only its interface and its attributes, while a more detailed description of its methods and its behaviour is constructed later during the sub-step *Fine-grained Statements Summarization*. Specifically, during this sub-step are considered only the lines *executed by each test case* using the coverage information as base data to describe the CUT behavior. Figure 3 shows an example of summary (in orange) generated during the *Class Level Summarization* phase for the class `Option.java`. With this summary the developer has the possibility to have a quick understanding of the CUT

[8] https://github.com/randoop/randoop
[9] https://github.com/javaparser/javaparser
[10] https://github.com/languagetool-org/languagetool

```
1| /** The main class under test is Option. It describes
2|  * a single option and maintains information regarding:
3|  *  - the option;
4|  *  - the long option;
5|  *  - the argument name;
6|  *  - the description of the option;  3.a
7|  *  - whether it has required;
8|  *  - whether it has optional argument;
9|  *  - the number of arguments;
10|  *  - the type, the values and the separator of the option;**/
11| public class TestOption {
12| /** OVERVIEW: The test case "test0" covers around 3.0% 3.b
13|  * (low percentage) of statements in "Option" **/
14| @Test
15| public void test0() throws Throwable {
16| // The test case instantiates an "Option" with option 3.c
17| // equal to "", and description equal to "1W|^".
18| Option option0 = new Option("", "1W|^");
19| // Then, it tests:
20| // 1)  whether the description of option0 is equal to 3.c
21| // "1W|^";
22|  assertEquals("1W|^", option0.getDescription());
23| // 2) whether the key of option0 is equal to ""; 3.c
24| // The execution of the method call used in the assertion 3.d
25| // implicitly covers the following 1 conditions:
26| // - the condition "option equal to null" is FALSE;
27|  assertEquals("", option0.getKey());
28|  }
29| }
```

Figure 3: Example of summary generated by TestDescriber for a JUnit test method exercising the class *Options.java*

without reading all of its lines of code.

Test Method Summarization. This step is responsible for generating a general description of the statement coverage scores achieved by each JUnit test method. This description is extracted by leveraging the coverage information provided by Cobertura to fill a pre-defined *template*. An example of summary generated by TestDescriber for describing the coverage score is depicted in Figure 3 (in yellow): before each JUnit test method (`test0` in the example) TestDescriber adds a comment regarding the percentage of statements covered by the given test method independently from all the other test methods in `TestOption`. This type of description allows to identify the contribution of each test method to the final structural coverage score. In the future we plan to complement the statement coverage describing further coverage criteria (e.g. branch or mutation coverage).

Fine-grained Statement Summarization. As described in Section 2.3 TestDescriber extracts the fine-grained list of *code elements* (e.g. methods, attributes, local variables) composing each statement of the CUT covered by each JUnit test method. This information is provided as input to the *Fine-grained Statements Summarization* phase, thus, TestDescriber performs the following three steps: (i) parses all the instructions contained in a test method; (ii) it uses the SWUM methodology for each instruction and determines which kind of operation the considered statement is performing (e.g. if it declares a variable, it uses a constructor/method of the class, it uses specific assertions etc.) and which part of the code is executed; and (iii) it generates a set of customized natural-language sentences depending on the selected kind of instructions. To perform the first two steps, it assigns each statement to one of the following categories:

- *Constructor of the class*. A constructor typically implies the instantiation of an object, which is the implicit *action*/*verb*, with some properties (parameters). In this case, our descriptor links the constructor call to its corresponding declaration in the CUT to map *formal* and *actual* parameters. Therefore, pre-processing and POS tagger are performed to identify the verb, noun phrase and adjectives from the constructor signature. These linguistic elements are then used to fill specific natural language templates for constructors. Figure 3 contains an example of a summary generated to describe the constructor `Option(String, String)`, i.e., the lines 16 and 17 (highlighted in green).
- *Method calls*. A method implements an operation and typically begins with a verb [30] which defines the main *action* while the method caller and the parameters determine *theme* and *secondary arguments*. Again, the linguistic elements identified after pre-processing and POS tagging are used to fill natural language templates specific for method calls. More precisely, the summarizer is able to notice if the result of a method call is assigned as value to a local variable (assignment statement), thus, it adapts the description depending on the specific context. For particular methods, such as getters and setters, it uses ad-hoc templates that differ from the templates used for more general methods.
- *Assertion statements*. This step defines the test oracle and enables to test whether the CUT behaves as intended. In this case the name of an assertion method (e.g. *assertEquals*, *assertFalse*, *notEquals* etc) defines the type of test, while the input parameters represent respectively (i) the expected and (ii) the actual behavior. Therefore, the template for an assertion statement is defined by the (pre-processed) assertion name itself and the value(s) passed (and verified) as parameter(s) to the assertion. Figure 3 reports two examples of descriptions generated for assertion methods where one of the input parameters is a method call, e.g., , `getKey()` (the summary is reported in line 23 and highlighted in green).

Branch Coverage Summarization. When a test method contains method/constructor calls, it is common that the test execution covers some if-conditions (branches) in the body of the called method/constructor. Thus, TestDescriber, after the *Fine-grained Statements Summarization* step, enriches the standard method call description with a summary describing the Boolean expressions of the `if` condition. Therefore, during the *Branch Coverage Summarization* step TestDescriber generates a natural language description for the tested `if` condition. When an `if` condition is composed of multiple Boolean expressions combined via Boolean operators, we generate natural language sentences for the individual expressions and combine them. Thus, during the *Branch Coverage Summarization*, we adapt the descriptions when an if-condition contains calls to other methods of the CUT. In the previous example reported in Figure 3, when executing the method call `getKey()` (line 27) for the object `option0`, the test method `test0` covers the false branch of the if-condition `if (opt == null)`, i.e., it verifies that `option0` is not `null`. In Figure 3 the lines 24, 25 and 26, (highlighted in red) represent the summary generated during the *Branch Coverage Summarization* for the method call `getKey()`.

2.5 Summary Aggregation

The Information Aggregator is in charge of enriching the original JUnit test class with all the natural language summaries and descriptions provided by the summary generator. The summaries are presented as different block and inline comments: (i) the general description of the CUT is added as a block comment before the declaration of the test class; (ii) the brief summaries of the statement coverage scores achieved by each individual JUnit test method is added as

Table 1: Java classes used as objects of our study

Project	Class	eLOC	Methods	Branches
Commons Primitives	ArrayIntList	65	12	28
Math4J	Rational	61	19	36

block comments before the corresponding test method body; (iii) the fine-grained descriptions are inserted inside each test method as inline comments to the corresponding statements they are summarizing.

3. STUDY DESIGN AND PLANNING

3.1 Study Definition

The *goal* of our study is to investigate to what extent the summaries generated by TestDescriber improve the *comprehensibility* of automatically generated JUnit test cases and *impact* the ability of developers to fix bugs. We measure such an impact in the context of a testing scenario in which a Java class has been developed and must be tested using generated test cases with the purpose of identifying and fixing bugs (if any) in the code. The *quality* focus concerns the understandability of automatically generated test cases when enriched with summaries compared to test cases without summaries. The *perspective* is of researchers interested in evaluating the *effectiveness* of automatic approaches for the test case summarization when applied in a practical testing and bug fixing scenario. We therefore designed our study to answer the following research questions (RQs):

RQ1 *How do test case summaries impact the number of bugs fixed by developers?* Our first objective is to verify whether developers are able to identify and fixing more faults when relying on automatically test cases enriched with summaries.

RQ2 *How do test case summaries impact developers to change test cases in terms of structural and mutation coverage?* The aim is assessing whether developers are more prone to change test cases to improve their structural coverage when the summaries are available.

3.2 Study Context

The *context* of our study consists of (i) *objects*, i.e., Java classes extracted from two Java open-source projects, and (ii) *participants* testing the selected objects, i.e., professional developers, researchers and students from the University of Zurich and the Delft University of Technology. Specifically, the object systems are *Apache Commons Primitives* and *Math4J* that have been used in previous studies on search-based software testing [23, 24, 44]. From these projects, we selected two Java classes: (i) `Rational` that implements a rational number, and (ii) `ArrayIntList`, which implements a list of primitive int values using an array. Table 1 details characteristics of the classes used in the experiment. *eLOC* counts the effective lines of source code, *i.e.* source lines without purely comments, braces and blanks [33]. For each class we consider a faulty version with five injected faults available from previous studies [23, 24]. These faults were generated using a mutation analysis tool, which selected the five mutants (faults) more difficult to kill, i.e., the ones that can be detected by the lowest number of test cases [23, 24]. These classes are non-trivial, yet feasible to test within an hour; they do not require (i) to learn complex algorithms and (ii) to examine other classes in the same library [23]. To recruit participants we sent email invitations to our contacts from industrial partners as well as to students and researchers from the Department of Computer Science at the University of Zurich and at Delft University of Technology. In total we sent out 44 invitations (12 developers and 32 researchers). In the end, 30 subjects (67%) performed the experiment and sent their data back, see Table 2. Of them, 7 were professional developers from industry and 23 were students or senior researchers from the authors' Computer Science Departments. All of the 7 professional developers have more than seven years of programming experience in Java (one of them more than 15 years). Among the 23 subjects from our departments, 2 were Bachelor's students, 5 were Master's students, 14 PhD students, and 2 senior researchers. Each participant had at least three years of prior experience with Java and the JUnit testing framework.

Table 2: Experience of Participants

Programming Experience	Absolute #	Frequency
1-2 years	1	3.3%
3-6 years	20	66.6%
7-10 years	8	26.6%
>10 years	1	3.3%
Σ	30	100%

3.3 Experimental Procedure

The experiment was executed offline, i.e., participants received the experimental material via an online Survey platform[11] that we use to collect and to monitor time and activities. An example of survey sent to the participants can be found online[12]. Each participant received an experiment package, consisting of (i) a statement of consent, (ii) a pre-test questionnaire, (iii) instructions and materials to perform the experiment, and (iv) a post-test questionnaire. Before the study, we explained to participants what we expected them to do during the experiment: they were asked to perform two testing sessions, one for each faulty Java class. They could use the test suite (i.e., JUnit test cases) generated by Evosuite to test the given classes and to fix the injected bugs. Each participant received two tasks: (i) one task included one Java class to test plus the corresponding generated JUnit test cases enriched WITH the summaries generated by *TestDescriber*; (ii) the second task consisted of a second Java class to test together with the corresponding generated JUnit test cases WITHOUT summaries.

The experimental material was prepared to avoid learning effects: each participant received two different Java classes for the two testing tasks; each participant received for the first task test cases enriched with corresponding summaries, while for the second task they received the cases without the summaries. We assigned the tasks to the participants in order to have a balanced number of participants which test (i) the first class with summaries followed by the second class without summaries; and (ii) the first class without summary followed by the second class with summaries. Since Evosuite uses randomized search algorithms (i.e., each run generates a different set of test cases with different input parameters), we provided to each participant different starting test cases.

Before starting the experiment, each participant was asked to fill in the pre-study questionnaire reporting their programming and testing experience. After filling in the questionnaire, they could start the first testing task by opening

[11] http://www.esurveyspro.com

[12] http://www.ifi.uzh.ch/seal/people/panichella/tools/TestDescriber/Survey.pdf

the provided workspace in the Eclipse IDE. The stated goals were (i) *to test the target class* as much as possible, and (ii) *to fix the bugs*. Clearly, we did not reveal to the participants where the bugs were injected, nor the number of bugs injected in each class. In the instructions we accurately explain that the generated JUnit test cases are green since Evosuite, as well as other modern test generation tools [16, 40], generate assertions that reflect the current behavior of the class [21]. Consequently, if the current behavior is faulty, the assertions reflect the incorrect behavior and, thus, must be checked and eventually corrected [23].

Therefore, participants were asked to start reading the available test suite, and to edit the test cases to (eventually) correct the assertions. They were also instructed to add new tests if they think that some parts of the target classes are not tested, as well as to delete tests they did not understand or like. In each testing session, participants were instructed to spend no more than 45 minutes for completing each task and to finish earlier if and only if (i) they believe that their test cases cover all the code and (ii) the found and fixed all the bugs. Following the experiment, subjects were required to fill in an exit survey we used for qualitative analysis and to collect feedback. In total, the duration of the experiment was two hours including completing the two tasks and filling in the pre-test and post-test questionnaires.

We want to highlight that we did not reveal to the participants the real goal of our study, which is to *measure the impact of test case summaries on their ability to fix bugs*. As well as we did not explain them that they received two different tasks one with and the other one without summaries. Even in the email invitations we use to recruit participants, we did not provide any detail to our goal but we used a more general motivation, which was to *better understand the bug fixing practice of developers during their testing activities when relying on generated test cases*.

3.4 Research Method

At the end of the experiment, each participant produced two artifacts for each task: (i) the test suite automatically generated by Evosuite, with possible fixes or edits by the participants, e.g., adding assertions to reveal faults; and (ii) the original (fixed) target class, i.e., without (some of) the injected bugs. We analyze the target classes provided by the participants in order to address **RQ1**: for each class we inspect the modifications applied by each participant in order to verify whether the modifications are correct (true bug fixing) or not. Thus, we counted the exact number of seeded bugs fixed by each participant to determine to what extent test summaries impact their bug fixing ability.

For **RQ2** we computed several structural coverage metrics for each test suite produced when executed on the original classes, i.e., on the target classes without bugs [23, 24]. Specifically, we use *Cobertura* to collect statement, branch, and method coverage scores achieved. The mutation score was computed by executing the JUnit test suite using PIT[13], a popular command line tool that automatically seeds a Java code generating mutants. Then, it runs the available tests and computes the resulting mutation score, i.e., the percentage of mutants detected by the test suites. As typical in mutation testing, a mutant is killed (covered) if the tests fail, otherwise if the tests pass then the mutation is not covered.

Once we have collected all the data, we used statistical tests to verify whether there is a statistical significant difference between the scores (e.g., the number of fixed bugs) achieved by participants when relying on tests with and without summaries. We employed non-parametric tests since the Shapiro-Wilk test revealed that neither the number of detected bugs, nor the coverage or mutation measures follow a normal distribution ($p \ll 0.01$). Hence, we used the non-parametric Wilcoxon Rank Sum test with a p-value threshold of 0.05. Significant p-values indicate that there is a statistical significant difference between the scores (e.g., number of fixed bugs) achieved by the two groups, i.e., by participants using test cases with and without summaries. In addition, we computed the effect-size of the observed differences using the Vargha-Delaney ($\hat{A}_{12}$) statistic [52]. The Vargha-Delaney ($\hat{A}_{12}$) statistic also classifies the obtained effect size values into four different levels (*negligible*, *small*, *medium* and *large*) that are easier to interpret. We also checked whether other *co-factors*, such as the programming experience, interact with the main treatment (test summaries) on the dependent variable (number of bugs fixed). This was done using a two-way permutation test [4], which is a non-parametric equivalent of the two-way Analysis of Variance (ANOVA). We set the number of iterations of the permutation test procedure to 1,000,000 to ensure that results did not vary over multiple executions of the procedure [4].

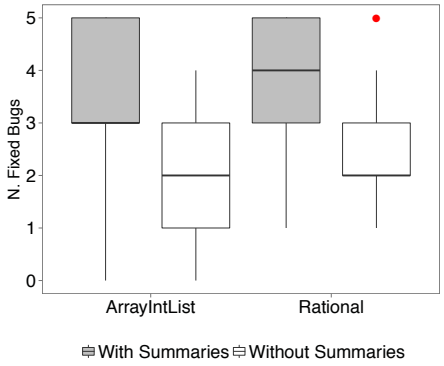

Figure 4: RQ1: Bugs fixed with and without summaries.

Parameter Configuration. There are several parameters that control the performance in terms of structural coverage for Evosuite; in addition, there are different coverage criteria to optimize when generating test cases. We adopted the default parameter settings used by Evosuite [21], since a previous empirical study demonstrated [2] that the default values widely used in the literature give reasonably acceptable results. For the coverage criterion, we consider the default criterion, which is *branch coverage*, again similar to previous experiments [23, 24]. The only parameter that we changed is the running time: we run Evosuite for ten minutes in order to achieve the maximum branch coverage.

4. RESULTS

In the following, we report results of our study, with the aim of answering the research questions formulated in Section 3.

4.1 RQ1: Bug Fixing

Figure 4 depicts the box-plots of the number of bugs fixed by the participants, divided into the (i) target classes to fix and (ii) the availability of TestDescriber-generated summaries. The results indicate that for both tasks the number

[13] http://pitest.org/

of bugs fixed is substantially higher when to the participants *had test summaries* at their disposal. Specifically, from Figure 4 we can observe that for the class `ArrayIntList` participants without TestDescriber summaries were able to correctly identify and fix 2 out of 5 bugs (median value; 40% of injected bugs) and no participant was able to fix all the injected bugs. Vice versa, when we provided to the participants the TestDescriber summaries, the median number of bugs fixed is 3 bugs and about 30% of the the participants were able to fix all the bugs. This result represents an important improvement (+50% of bugs were fixed by participants) if we consider that in both the scenarios, WITH and WITHOUT summaries, the amount of time given to the participants was the same. Similarly, for `Rational`, when relying on test cases with summaries, the median number of bugs fixed is 4 out of 5 (80%) and 31% of participants were able to fix all the bugs. Vice versa, using test cases without summaries the participants fixed 2 bugs (median value). Hence, when using the summaries the participants were able to fix twice as many number of bugs (+100%) with respect to the scenario in which they were provided test cases without comments.

The results of the Wilcoxon test highlight that the use of TestDescriber summaries significantly improved the bug fixing performance of the participants in each target class achieving p-values of 0.014 and < 0.01 for `ArrayIntList` and `Rational` respectively (which are smaller than the significance level of 0.05). The Vargha-Delaney $\hat{A}_{12}$ statistic also reveals that the magnitude of the improvements is *large* for both target classes: the effect size is 0.76 and 0.78 for `ArrayIntList` and `Rational` respectively. Finally, we used the two-way permutation test to check whether the number of fixed bugs between the two groups (test cases with and without summaries) depends on and interacts with the participants' programming experience, which can be a potential co-factor. The two-way permutation test reveals that (i) the number of bugs fixed is not significantly influenced by the programming experience (p-values $\in \{0.5736, 0.1372\}$) and (ii) there is no significant interaction between the programming experience and the presence of test case summaries (p-values $\in \{0.3865, 0.1351\}$). This means that all participants benefit from using the TestDescriber summaries, independent of their programming experience.

This finding is particularly interesting if we consider that Fraser *et al.* [23, 24] reported that there is no statistical difference between the number of bugs detected by developers when performing manual testing or using automatically generated test cases to this aim. Specifically, in our study we included (i) two of the classes Fraser *et al.* used in their experiments (`ArrayIntList` and `Rational`), and for them we (ii) considered the same set of injected bugs and (iii) we generated the test cases using the same tool. In this paper we show that the summaries generated by TestDescriber can significantly help developers in detecting and fixing bugs. However, a larger sample size (i.e., more participants) would be needed to compare the performances of participants when performing manual testing, i.e., when they are not assisted by automatic tools like Evosuite and TestDescriber at all. In summary, we can conclude that

RQ1 *Using automatically generated test case summaries significantly helps developers to identify and fix more bugs.*

Table 3: Statistics for the test suites edited by the participants for *ArrayIntList*

Variable	Factor	Min	Mean	Max	p-value	A_{12}
Method Cov.	With	0.36	0.63	0.86	0.83	-
	Without	0.50	0.65	0.86		
Statement Cov.	With	0.52	0.68	0.85	0.83	-
	Without	0.61	0.68	0.85		
Branch Cov.	With	0.55	0.68	0.82	0.87	-
	Without	0.59	0.67	0.82		
Mutation Score	With	0.13	0.29	0.45	0.45	-
	Without	0.13	0.30	0.52		

Table 4: Statistics for the test suites edited by the participants for *Rational*

Variable	Factor	Min	Mean	Max	p-value	A_{12}
Method Cov.	With	0.89	0.95	1.00	0.80	-
	Without	0.89	0.95	1.00		
Statement Cov.	With	0.92	0.97	1.00	1.00	-
	Without	0.92	0.97	1.00		
Branch Cov.	With	0.85	0.86	0.90	0.89	-
	Without	0.85	0.86	0.90		
Mutation Score	With	0.52	0.71	0.93	0.08	0.69 (M)
	Without	0.31	0.61	0.89		

4.2 RQ2: Test Case Management

To answer RQ2, we verify whether there are other measurable features instead of the test case summaries the might have influenced the results of RQ1. To this aim, Tables 3 and 4 summarise the structural coverage scores achieved by the test suite produced by human participants during the experiment. As we can see from Table 3 there is no substantial difference in terms of structural coverage achieved by the test suites produced by participants with and without test case summaries for `ArrayIntList`. Specifically, method, branch and statement coverage are almost identical. Similar results are achieved for `Rational` as shown in Table 4: for method, branch and statement coverage there is no difference for the tests produced by participants with and without test summaries. Consequently for both the two classes the p-values provided by the Wilcoxon test are not statistically significant and the effect size is always *negligible*. We hypothesize that these results can be due to the fact that the original test suite generated by Evosuite, that were used by the participants as starting point to test the target classes, already achieved a very high structural coverage ($> 70\%$ in all the cases). Therefore, even if the participants were asked to manage (when needed) the test cases to correct wrong assertions, at the end of the experiment the final coverage was only slightly impacted by these changes.

For the mutation analysis, the mutation scores achieved with the tests produced by the participants seem to be slightly lower when using test summaries (-1% on average) for `Array-IntList`. However, the Wilcoxon test reveals that this difference is not statistically significant and the Vargha-Delaney $\hat{A}_{12}$ measure is *negligible*. For `Rational` we can notice an improvements in terms of mutation score (+10%) for the tests produced by participants who were provided with test summaries. The Wilcoxon test reveals a *marginal* statistical significant p-value (0.08) and the Vargha-Delaney $\hat{A}_{12}$ measures an effect size *medium* and positive for our test summaries, i.e., participants provided test cases able to kill more mutants when using the test summaries. A replication study with more participants would be need to further investigate whether the mutation score can be positively influenced when using tests summaries.

RQ2 *Test case summaries do not influence how the developers manage the test cases in terms of structural coverage.*

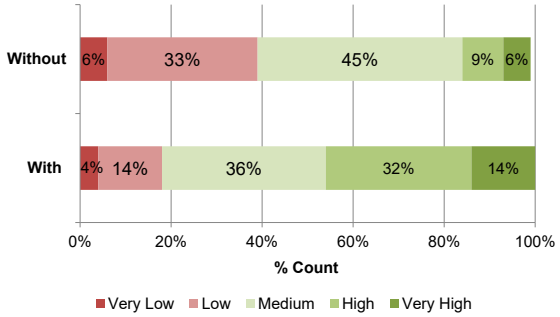

Figure 5: Perceived test comprehensibility WITH and WITHOUT TestDescriber summaries.

5. DISCUSSION AND LESSONS LEARNT

In the following, we provide additional, qualitative insights to the quantitative study reported in Section 4.

Summaries and comprehension. At the end of each task we asked each participant to evaluate the comprehensibility of the test cases (either with or without summary) using a Likert scale intensity from very-low to very-high (involving all the 30 participants). When posing this question we did not explicitly mention terms like *"test summaries"* but instead *"test comments"* to avoid biased answers by the participants. Figure 5 compares the scores given by participants to the provided test cases (i.e., generated by Evosuite) according to whether the tests were enriched (WITH) or not (WITHOUT) with summaries. We can notice that when the test cases were commented with summaries (WITH) 46% of participants labled the test cases as easy to understand (high and very high comprehensibility) with only 18% of participants that considered the test cases as incomprehensible. Vice versa, when the test cases were not enriched with summaries (WITHOUT) only 15% of participants judged the test cases as easy to understand, while a substantial percentage of participants (40%) labeled the test case as difficult to understand. The Wilcoxon test also reveals that this difference is statistical significant (p-value = 0.0050) with a positive and *medium* effect size (0.71) according to the Vargha-Delaney $\hat{A}_{12}$ statistic. Therefore, we can argue that

> *Test summaries statistically improve the comprehensibility of automatically generated test case according to human judgments.*

Post-test Questionnaire. Table 5 reports the results to questions from the exit survey. The results demonstrate that in most of the cases the participants considered the test summaries (when available) as the most important source of information to perform the tasks after the source code itself, i.e., the code of the target classes to fix. Indeed, when answering Q1 and Q2 the most common opinion is that the source code is the primary source of information (47% in Q1 and 43% of the opinions in Q2), followed by the test summaries (20% in Q1 and 53% in Q2). In contrast, participants deem the actual test cases generated by Evosuite to be less important than (i) the test summaries and (ii) the test cases they created themselves during the experiment. As confirmation of this finding, we received positive feedback from both junior and more experienced participants, such as *"the generated test cases with comments are quite useful"* and *"comments give me [a] better (and more clear) picture of the goal of a test."*

Table 5: Raw data for exit questionnaire (SC=Source Code, TCS=TC Summaries, TC=Test Cases, and MTC=Manually written TC).

Questions		SC	TCS	TC	MTC	Other
Q1: What is the best source of information?		**47%**	20%	20%	13%	0
Q2: Can you rank the specified sources of information in order of importance from 1 (high) to 5 (low)?	(rank 1)	**43%**	27%	27%	3%	0%
	(rank 2)	17%	**53%**	30%	0%	0%
	(rank 3)	27%	23%	**33%**	10%	7%
	(rank 4)	17%	17%	10%	**57%**	0%
	(rank 5)	0%	3%	13%	7%	**77%**

Questions	Disagree		Agree	
	Fully	Partial	Partially	Fully
Q3: Adding or changing the tests leads to better tests?	0%	14%	**45%**	**41%**
Q4: Without comments, tests are difficult to read and understand?	0%	33%	**23%**	**43%**
Q5: Adding assertions to tests WITH comments is prohibitively difficult?	**13%**	**60%**	27%	0%
Q6: Adding assertions to tests WITHOUT comments is prohibitively difficult?	**10%**	**47%**	43%	0%
Q7: I had enough time to finish my task	7%	24%	**52%**	**17%**
Q8: Automatically generated unit tests exercise the *easy* parts of the program.	0%	20%	**73%**	**7%**

Table 6: Raw data of the questionnaire concerning the evaluation of TestDescriber summaries.

Content adequacy	
Response category	**Percentage of Ratings**
Is not missing any information.	**50%**
Missing some information.	**37%**
Missing some very important information.	13%
Conciseness	
Response category	**Percentage of Ratings**
Has no unnecessary information.	**38%**
Has some unnecessary information.	**52%**
Has a lot of unnecessary information.	10%
Expressiveness	
Response category	**Percentage of Ratings**
Is easy to read and understand.	**70%**
Is somewhat readable and understandable.	**30%**
Is hard to read and understand.	0%

From Table 5 we can also observe that participants mainly considered the tests generated by Evosuite as a starting point to test the target classes. Indeed, these tests must be updated (e.g., checking the assertions) and enriched with further manually written tests (Q3), since in most of the cases they test the easier part of the program under tests (according to 80% of opinions for Q8). Automatically generated tests are in most of cases (66% of participants) considered difficult to read and understand (Q4), especially if not enriched with summaries describing what they are going to test (Q5 and Q6).

Quality of the Summaries. Finally, we ask the participants to evaluate the overall quality of the provided test summaries, similarly as done in traditional work on source code summarization [37, 48]. We evaluate the quality according to three widely known dimensions [37, 48]:

- *Content adequacy*: considering only the content of the comments of JUnit test cases, is the important information about the class under test reflected in the summary?
- *Conciseness*: considering only the content of the comments in the JUnit test cases, is there extraneous or irrelevant information included in the comments?

- *Expressiveness*: considering only the way the comments of JUnit test cases are presented, how readable and understandable are the comments?

The analysis is summarized in Table 6. The results highlight that (i) 87% of the participants consider the TestDescriber comments adequate (they do not miss very important information); (ii) 90% of them perceive the summaries sufficiently concise as they contain no (38%) or only some unnecessary information (52%); (iii) 100% of participants consider the comments easy to read and/or somewhat readable. In summary, the majority of the participants consider the comments generated by TestDescriber very concise and easy to understand.

Feedback. Comments collected from the survey participants mentioned interesting feedback to improve TestDescriber summaries:

- *Redundant information from test to test*: developers of our study were concerned by the fact that for similar test cases TestDescriber generates the same comments and, as solution, they suggested to generate, for each assertion already tested in previous test methods, a new inline comment which specifies that the assertion was already tested in a previous test method.
- *Useless naming of test methods*: for several participants the name of the test does not give any hint about the method under test. They suggest to (i) *"...rename the method names to useful names... so that it is possible to see at a glance what is actually being tested by that test case"* or (ii) *"...describe in the javadoc of a test method which methods of the class are tested."*.

Lessons Learnt. As indicated in Section 4.2 test suites having high structural coverage are not necessarily more effective to help developers in detecting and fixing more bugs. Most automatic testing tools consider structural coverage as the main goal to optimize for, with the underlying assumption that higher coverage is strongly related to a test's effectiveness [3]. However, our results seem to provide a clear evidence that this is not always true as also confirmed by the non-parametric Spearman ρ correlation test: the correlation between the number of bugs fixed and the structural coverage metrics is always lower than 0.30 for `ArrayIntList` and 0.10 for `Rational`. Only the mutation score has a correlation coefficient larger than 0.30 in both the two classes. On the other hand, the results of RQ1 provide clear evidence that the summaries generated by TestDescriber play a significant role even if they do not change the code and the structural coverage of the original test cases generated by Evosuite. Therefore, we can argue that *comprehensibility* or *readability* are two further dimensions that should be considered (together with structural coverage) when systematically evaluating automatic test generation tools.

6. THREATS TO VALIDITY

In this section, we outline possible threats to the validity of our study and show how we mitigated them.

Construct Validity. Threats to construct validity concern the way in which we set up our study. Due to the fact that our study was performed in a remote setting in which participants could work on the tasks at their own discretion, we could not oversee their behaviour. The metadata sent to us could be affected by imprecisions as the experiment was conducted offline. However, we share the experimental data with the participants using an online survey platform, which forces the participants (1) to perform tasks in the desired order and (2) to fill in the questionnaires. Therefore, participants only got access to the final questionnaire after they had handed in their tasks, as well as they could not perform the second task without finishing the first one. Furthermore, the online platform allows us to monitor the total time each participant spent on the experiment. We also made sure participants were not aware of the actual aim of the study.

Internal Validity. Threats to internal validity concern factors which might affect the causal relationship. To avoid bias in the task assignment, we randomly assigned the tasks to the participants in order to have the same number of data points for all classes/treatments. To ensure that a sufficient number of data points are collected for statistical significance tests, each participant performed two bug fixing tasks —one with test summaries and one without, on different classes— rather than one single task, to produce 60 data points in this study. The two Java classes used as objects for the two tasks have similar difficulty and can easily be tested in 45 minutes, even for intermediate programmers [23, 24]. Another factor that can influence our results is the order of assignments, i.e., first with summaries and then without summaries or vice versa. However, the two-way permutation test reveals that there is no significant interaction between the order of assignments and the two tasks on the final outcome, i.e., the number of bugs fixed (p-value = 0.7189).

External Threats. External threats concern the generalizability of our findings. We considered two Java classes already used in two previous controlled experiments investigating the effectiveness of automated test case generation tools compared to manual testing [23, 24]. We also use the same set of bugs injected using a mutation analysis tool, which is common practice to evaluate the effectiveness of testing techniques in literature [23, 24, 25]. We plan to evaluate TestDescriber with a bigger set of classes, investigating its usefulness in the presence of more complex branches. Future work also needs to address which aspects of the generated summaries are useful. Is the coverage summary useful to developers and if so, in what way?

Another threat can be that the majority of our study participants have an academic background. Recent studies have shown that students perform similarly to industrial subjects, so long as they are familiar with the task at hand [31, 38]. All our student participants had at least 3 years of experience with the technologies used in the study, see Section 3.2. Moreover, our population included a substantial part of professional developers and the median programming experience of our participants is 3-6 years. Nevertheless, we plan to replicate this study with more participants in the future in order to increase the confidence in the generalizability of our results.

Conclusion Threats. In our study we use TestDescriber to generated tests summaries for JUnit test cases generated by Evosuite. It might be possible that using different automatic test generation tools may lead to different results in terms of test case comprehensibility. However, we notice that (i) coverage, (ii) structure and (iii) size of test cases generated with Evosuite are comparable to the output produced by other modern test generation tools, such as Randoop [40], JCrasher [16], etc.

We support our findings by using appropriate statistical

tests, i.e. the non-parametric Wilcoxon test and the two-way permutation test to exclude that other co-factors (such as the programming experience) can affect our conclusion. We also used the Wilk-Shapiro normality test to verify whether the non-parametric test could be applied to our data. Finally, we used the Vargha and Delaney $\hat{A}_{12}$ statistical test to measure the magnitude of the differences between the different treatments.

7. RELATED WORK

In this section, we discuss the related literature on source code summarization and readability of test cases.

Source Code Summarization. Murphy's dissertation [39] is the earliest work which proposes an approach to generate summaries by analysing structural information of the source code. More recently, Sridhara *et al.* [47] suggested to use pre-defined *templates* of natural language sentences that are filled with linguistic elements (verbs, nouns, etc.) extracted from important method signature [19, 20]. Other studies used the same strategy to summarize Java methods [26, 34, 48], parameters [50], groups of statements [49], Java classes [37], services of Java packages [27] or generating commit messages [15]. Other reported applications are the generation of source code documentation/summary by mining text data from other sources of information, such as bug reports [42], e-mails [42], forum posts [17] and question and answer site (Q&A) discussions [53, 55].

However, Binkley *et al.* [7] and Jones *et al.* [46] pointed out that the evaluation of the generated summaries should not be done by just answering the general question *"is this a good summary?"* but evaluated *"through the lens of a particular task"*. Stemming from these considerations, in this paper we evaluated the impact of automatically generated test summaries in the context of two bug fixing tasks. In contrast, most previous studies on source code summarization have been evaluated by simply surveying human participants about the quality of the provided summaries [7, 26, 34, 37, 47, 48].

Test Comprehension. The problem of improving test understandability is well known in literature [14], especially in the case of test failures [9, 57]. For example, Zhang *et al.* [57] focused on failing tests and proposed a technique based on static slicing to generate code comments describing the failure and its causes. Buse *et al.* [9] proposed a technique to generate human-readable documentation for unexpected thrown exceptions [9]. However, both these two approaches require that tests fail [57] or throw unexpected Java exceptions [9]. This never happens for automatically generated test cases, since the automatically generated assertions reflect the current behaviour of the class [24]. Consequently, if the current behaviour is faulty the generated assertions do not fail because they reflect the incorrect behavior.

Kamimura *et al.* [32] argued that developers might benefit from a consumable and understandable textual summary of a test case. Therefore, they proposed an initial step towards generating such summaries based on static analysis of the code composing the test cases [32]. From an engineering point of view, our work resumes this line of research; however, it is novel for two main reasons. First of all our approach generates summaries combining three different levels of granularity: (i) a summary of the main responsibilities of the class under test (class level); (ii) a fine-grained description of each statement composing the test case as done in the past [32] (test level); (iii) a description of the branch conditions traversed in the executed path of the class under test (coverage level). As such, our approach *combines* code coverage and summarization to address the problem of describing the effect of test case execution in terms of structural coverage. Finally, we evaluate the impact of the generated tests summaries in a real scenario where developers were asked to test and fix faulty classes.

Understandability is also widely related with the test size and number of assertions [3]. For these reasons previous work on automatic test generation focused on (i) reducing the number of generated tests by applying a post-process minimization [21], and (ii) reducing the number of assertions by using mutation analysis [25], or splitting tests with multiple assertions [56]. To improve the readability of the code composing the generated tests, Daka *et al.* [18] proposed a mutation-based post-processing technique that uses a domain-specific model for unit test readability based on human judgement. Afshan *et al.* [1] investigates the use of a linguistic model to generate more readable input strings. Our paper shows that summaries represent an important element for complementing and improving the readability of automatically generated test cases.

8. CONCLUSION AND FUTURE WORK

Recent research has challenged the assumption that structural coverage is the only goal to optimize [1, 54], suggesting that understandability of test cases is a key factor to optimize in the contest of automated test generation. In this paper we handle the problem of *usability* of automatic generated test cases making the following main contributions:

- We present *TestDescriber* a novel approach to generate natural language summaries of JUnit tests. TestDescriber is designed to automatically generate summaries of the portion of code exercised by each individual test case to provide a dynamic view of the CUT.

- To evaluate TestDescriber, we have set up an empirical study involving 30 human participants from both industry and academia. Specifically, we investigated the impact of the generated test summaries on the number of bugs actually fixed by developers when assisted by automated test generation tools.

Results of the study indicate that (RQ1) TestDescriber substantially helps developers to find more bugs (twice as many) reducing testing effort and (RQ2) test case summaries do not influence how developers manage test cases in terms of structural coverage. Additionally, TestDescriber could be used to automatically document tests, improving their readability and understandability. Results of our post-test questionnaire reveal that test summaries significantly improve the comprehensibility of test cases. Future work is directed towards different directions. We plan to further improve TestDescriber summaries by (i) considering the feedback received by the participants of our study, (ii) combining our approach with recent work that improves the readability of the code composing the generated test [18], (iii) complementing the generated summaries including further coverage criteria, such as branch or mutation coverage. Also, we aim to replicate the study involving additional developers.

References

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

[2] A. Arcuri and G. Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.

[3] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. Test code quality and its relation to issue handling performance. *IEEE Trans. Software Eng.*, 40(11):1100–1125, 2014.

[4] R. D. Baker. Modern permutation test software. In E. Edgington, editor, *Randomization Tests*. Marcel Decker, 1995.

[5] L. Baresi and M. Miraz. Testful: Automatic unit-test generation for java classes. In *Proceedings of the International Conference on Software Engineering - Volume 2 (ICSE)*, pages 281–284. ACM, 2010.

[6] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015. To appear.

[7] D. Binkley, D. Lawrie, E. Hill, J. Burge, I. Harris, R. Hebig, O. Keszocze, K. Reed, and J. Slankas. Task-driven software summarization. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 432–435. IEEE, 2013.

[8] F. P. J. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[9] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 273–282. ACM, 2008.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 322–335. ACM, 2006.

[11] A. Cavarra, C. Crichton, J. Davies, A. Hartman, and L. Mounier. Using uml for automatic test generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*. Springer-Verlag, 2002.

[12] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella. Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. *ACM Trans. Softw. Eng. Methodol.*, 25(1):5:1–5:38, 2015.

[13] S. Chen, H. Miao, and Z. Qian. Automatic generating test cases for testing web applications. In *Proc. of the International Conference on Computational Intelligence and Security Workshops (CISW)*, pages 881–885, 2007.

[14] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing testsuites to aid in software understanding. In *Proc. European Conference on Software Maintenance and Reengineering (CSMR)*, pages 213–222. IEEE, 2007.

[15] L. F. Cortes-Coy, M. L. Vásquez, J. Aponte, and D. Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 275–284. IEEE, 2014.

[16] C. Csallner and Y. Smaragdakis. Jcrasher: An automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004.

[17] B. Dagenais and M. P. Robillard. Recovering traceability links between an api and its learning resources. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 47–57. IEEE, 2012.

[18] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015. To appear.

[19] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Using IR methods for labeling source code artifacts: Is it worthwhile? In *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, pages 193–202, 2012.

[20] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Labeling source code with information retrieval methods: an empirical study. *Empirical Software Engineering*, 19(5):1383–1420, 2014.

[21] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Trans. Software Eng.*, 39(2):276–291, 2013.

[22] G. Fraser and A. Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2015.

[23] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Trans. Softw. Eng. Methodol.* To Appear.

[24] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 291–301. ACM, 2013.

[25] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 147–158. ACM, 2010.

[26] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 35–44. IEEE, 2010.

[27] M. Hammad, A. Abuljadayel, and M. Khalaf. Automatic summarising: The state of the art. *Lecture Notes on Software Engineering*, 4(2):129–132, 2016.

[28] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Trans. Softw. Eng.*, 36(2):226–247, 2010.

[29] E. Hill, Z. P. Fry, H. Boyd, G. Sridhara, Y. Novikova, L. Pollock, and K. Vijay-Shanker. Amap: Automat-

ically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the International Working Conference on Mining Software Repositories (MSR)*, pages 79–88. ACM, 2008.

[30] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 232–242. IEEE, 2009.

[31] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects - comparative study ofstudents and professionals in lead-time impact assessment. *Empirical Softw. Engg.*, 5(3):201–214, Nov. 2000.

[32] M. Kamimura and G. Murphy. Towards generating human-oriented summaries of unit test cases. In *Proc. of the International Conference on Program Comprehension (ICPC)*, pages 215–218. IEEE, May 2013.

[33] S. MacDonell. Reliance on correlation data for complexity metric use and validation. *ACM Sigplan Notices*, 26(8):137–144, 1991.

[34] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 279–290. ACM, 2014.

[35] B. Meyer, I. Ciupa, A. Leitner, and L. Liu. Automatic testing of object-oriented software. In *SOFSEM 2007: Theory and Practice of Computer Science*, volume 4362 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin Heidelberg, 2007.

[36] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In *Software Evolution*, pages 173–202. Springer, 2008.

[37] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 23–32. IEEE, May 2013.

[38] T. Mortensen, R. Fisher, and G. Wines. Students as surrogates for practicing accountants: Further evidence. In *Accounting Forum*, volume 36, pages 251–265. Elsevier, 2012.

[39] G. C. Murphy. *Lightweight Structural Summarization As an Aid to Software Evolution*. PhD thesis, 1996. AAI9704521.

[40] 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 (OOPSLA)*, pages 815–816. ACM, 2007.

[41] A. Panichella, F. Kifetew, and P. Tonella. Reformulating branch coverage as a many-objective optimization problem. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[42] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora. Mining source code descriptions from developer communications. In *Proceedings of the International Conference on Program Comprehension, ICPC*, pages 63–72. IEEE, 2012.

[43] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 25–34. IEEE, 2001.

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

[45] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proc. Int'l Conference on Software Engineering (ICSE)*, pages 499–510. IEEE, 2007.

[46] K. Spärck Jones. Automatic summarising: The state of the art. *Inf. Process. Manage.*, 43(6):1449–1481, 2007.

[47] G. Sridhara. *Automatic Generation of Descriptive Summary Comments for Methods in Object-oriented Programs*. PhD thesis, Newark, DE, USA, 2012. AAI3499878.

[48] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 43–52. ACM, 2010.

[49] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 101–110. IEEE, 2011.

[50] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Proceedigs of the International Conference on Program Comprehension (ICPC)*, pages 71–80. IEEE, 2011.

[51] P. Tonella. Evolutionary testing of classes. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128. ACM, 2004.

[52] 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*, 25(2):101–132, 2000.

[53] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora. Codes: Mining source code descriptions from developers discussions. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 106–109. ACM, 2014.

[54] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal. Automatic generation of system test cases from use case specifications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 385–396. ACM, 2015.

[55] E. Wong, J. Yang, and L. Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 562–567. IEEE, 2013.

[56] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 52–63. ACM, 2014.

[57] S. Zhang, C. Zhang, and M. Ernst. Automated documentation inference to explain failed tests. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 63–72. IEEE, 2011.