Developer-Centric Test Amplification: User-Guided Test Amplification

Version of June 19, 2022

Danyao Wang

Developer-Centric Test Amplification: User-Guided Test Amplification

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Danyao Wang
born in Yunnan, China

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Developer-Centric Test Amplification: User-Guided Test Amplification

Author: Danyao Wang
Student id: 5274788
Email: d.wang-9@student.tudelft.nl

Abstract

Automated test generation techniques improve the efficiency of software testing. However, the opacity of the test generation process and concerns about the readability of generated tests make it difficult for software developers to accept them. Developer-centric test amplification creates easy-to-understand test cases by amplifying existing test cases that developers are familiar with and assists developers in integrating them into their test suite. We propose user-guided test amplification to allow developers to guide the test amplification to generate new test cases based on their branch coverage expectations. We create a user-guided test amplification prototype that starts with the method developers want to test, aids developers in communicating which branch should be covered, and assists developers in inspecting and selecting the amplified test cases. We conduct a technical case study with two Java projects and show that our approach cannot always produce a test case to cover a given branch because objects are not initialized with the right parameter values to fulfill the target branch condition. We also perform a user study with 12 software developers to investigate developers' opinions on our approach. The evaluation result shows that the user-guided test amplification generates amplified test cases that developers are satisfied with and is especially useful when developers want to generate tests to cover a specific branch. Connecting the developers' coverage goal and the amplified test cases enables developers to understand and select the test cases more easily.

Preface

When I write this preface, I am about to complete my final assignment in my master's phase, my master's thesis. Looking back, the day I just arrived in the Netherlands seems like yesterday. Feeling the time flies by has made me realize how fulfilling the past two years have been, especially the last year of my master thesis project. After studying data science for many years, I switched to software engineering. Designing and developing projects that can be applied in real life is so fascinating. The much help I received along the way made the journey even more memorable.

First, I would like to thank my supervisors, Andy Zaidman and Carolin Brandt. Andy's initial introduction to the research topic motivated me to start this master's thesis project. His grasp of the thesis procedure keeps my project progress on the right track. I am very grateful to Andy for his valuable feedback and suggestions at crucial points in the project, which were essential for me to complete this thesis. As my daily supervisor, Caro always gives me detailed feedback and advice at our weekly meetings. She always gave great answers to every question I had, which made me admire her erudition and strong scientific research ability. Her detailed guidance and assistance have always helped me to remove obstacles in the progress of the project.

Second, I would like to thank the 12 software developers who participated in my user study. They actively participated in my user study with interest in my research topic and provided valuable material for my thesis. Several developers even volunteered more time than expected to share their views. Their contributions are an indispensable part of my thesis.

Finally, I want to thank my parents and friends. Although I haven't seen my parents for two years, their financial support and concern from afar have always encouraged me to finish my thesis. My dear friends, whether thousands of miles away or close at hand, always cared and encouraged me and also helped me recruit participants for my user study. Their help was so important for me to complete this thesis.

Danyao Wang
Delft, the Netherlands
June 19, 2022

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

Testing is important for software development [41] but labour-intensive and expensive [2]. Earlier studies estimated that testing often accounts for more than 50% of total development costs [7]. Many automated test generation techniques have been developed to help developers write tests, reduce the cost and enhance the effectiveness of software testing [2]. Despite the benefits of automated test generation, some recent studies have revealed some issues that prevent software developers from incorporating automated test generation into their daily practice, including the opacity of the test generation process, the lack of collaboration with developers, and developers' skepticism about the readability of the generated tests [4, 35, 38, 34].

Brandt and Zaidman proposed developer-centric test amplification to help developers generate test cases that are easy to understand and integrate into their test suite [8]. Test amplification is a technique that generates new test cases by adapting existing, manually written test cases and can improve the code coverage of the existing test suite [9]. Developer-centric test amplification provides amplified test cases that developers can take over into their manually maintained test suite, with the developer accepting the test case being central [8]. Brandt and Zaidman's approach generates new test cases by amplifying test cases that developers have manually written. Developers' familiarity with the existing test cases helps make the amplified test cases easy to understand. Specifically, they designed a tool that generates amplified test cases that contribute additional instruction coverage to the test suite. Moreover, developers can interact with the tool to explore and inspect the amplified test cases and add them to their test suite. We use the term open test amplification to describe their design as it provides new test cases that can bring any additional instruction coverage for the entire project. Developers mainly participate in the final stage of inspection and acceptance in the open test amplification.

Inspired by the developer-centric test amplification, we propose user-guided test amplification to involve developers more directly in test amplification and make the test amplification process more transparent to them. Besides, we expect to improve the understandability and relevance of the test cases to developers by connecting developers' coverage requirements with the amplified test cases. The user-guided test amplification includes developers in the test amplification process, with developers providing their branch coverage requirement to guide the test amplification in generating the tests they want.

We created a prototype to evaluate the value of the user-guided test amplification approach to software developers and compare it with the open test amplification approach. The prototype consists of an interaction layer and a supportive test amplification tool. The interaction layer allows developers to convey their branch coverage expectations for the amplified test cases and inspect the test amplification result. The test amplification tool receives the branch coverage requirement and generates amplified test cases according to the requirement. We develop the prototype based on the open test amplification tool, which is implemented with a test exploration IntelliJ plugin TestCube [8] and a test amplification tool DSpot [10]. We implement the user-guided test amplification interaction layer in TestCube. We add a directed amplification method in DSpot to support amplifying test cases in the direction of covering specific branches. Based on DSpot's result, TestCube filters amplified test cases that meet developers' requirements and displays easy-to-understand results to assist developers in building a test suite.

We evaluate the effectiveness of the directed test amplification method added in DSpot by conducting a technical case study to estimate the probability that it can generate tests that meet a given branch coverage requirement. We recruited 12 software developers and conducted user study interviews with them to investigate the value of the user-guided test amplification interaction in TestCube. We also propose further suggestions for creating an effective user-guided test amplification interface by analyzing the evaluation result.

1.2 Research Questions

This thesis investigates how user-guided test amplification helps developers build and maintain their test suite and compare it with open test amplification. Specifically, we propose the research questions below.

Research Question 1

Does the directed test amplification method generate test cases that satisfy the developers' branch coverage expectations?

This project aims at allowing developers to guide test amplification to generate test cases that meet their branch coverage expectations. Therefore, it is crucial to evaluate if the amplified test cases generated by the directed test amplification method can meet the branch coverage expectations of developers.

We answer this research question from both a technical and a user perspective. Technically, we execute the directed test amplification using a set of branch coverage requirements as input and assess how frequently the directed test amplification can generate new test cases that satisfy the branch coverage requirements. In addition, we invite developers to create test cases with the directed test amplification method and provide feedback on the result.

Research Question 2

Does the directed test amplification method generate more test cases that fulfill developers' branch coverage expectations than the open test amplification method?

We add a new directed test amplification method to complement DSpot so that it can lead test amplification towards the direction of meeting developers' branch coverage expectations. We want to verify that the new directed test amplification method can generate more test cases that fulfill developers' branch coverage expectations than Brandt and Zaidman's open test amplification method.

Research Question 3

How do developers perceive the user-guided test amplification?

We want to learn what developers think about the user-guided test amplification interaction in TestCube and determine if it can effectively assist developers in guiding test amplification and generating test cases they want.

Research Question 4

What different value do the user-guided test amplification and the open test amplification bring to developers?

We want to explore the new value that the user-guided test amplification brings to developers compared with Brandt and Zaidman's open test amplification and see if it can help developers better leverage test amplification to enhance their test suites.

Research Question 5

What are the key facets to creating an effective interface for user-guided test amplification?

We want to figure out what factors are crucial to creating an effective user-guided test amplification interface and make recommendations for future relevant research.

1.3 Research Method

This section introduces the research methods we use to answer the research questions in Section 1.2.

1.3.1 Design and Implement User-Guided Test Amplification

To explore the user-guided test amplification, we design and implement a prototype by extending TestCube and DSpot.

We implement a new interaction in TestCube to allow developers to guide the test amplification. Developers start the test amplification by selecting a method they want to test. Then TestCube visualizes the selected method by a Control Flow Graph (CFG) [1]. Developers convey their branch coverage expectations for amplified test cases by selecting a branch they want the new test cases to cover in the CFG. Moreover, we add a result window to display the amplified test cases and visualize the new coverage they bring to the target method in the CFG. Developers can add the amplified test cases to the existing test suite.

We extend DSpot to support user-guided test amplification in the back-end. First, we extend DSpot to be able to receive the guidance information provided by developers interacting with TestCube. As the guidance is specific branch coverage expectation, we extend DSpot to be able to compute the branch coverage of amplified test cases, which is the criterion to determine if an amplified test case satisfies the developers' requirement. Furthermore, we add a directed amplifier, D-amplifier, to support directed test amplification better as all the amplifiers used in the open test amplification amplify test cases without direction.

1.3.2 Technical Case Study

To answer RQ1 and RQ2, we conduct a case study [43] to evaluate the new directed test amplification method in DSpot. Specifically, we prepare several Java projects as the objects to perform the directed test amplification and assess the result.

We add d D-amplifier in DSpot to support the directed test amplification, amplifying manually written test cases and generating amplified test cases that satisfy specific branch coverage requirements. Therefore, we prepare several Java projects and identify all the branches in the source code to build up a case study dataset. The branches are what developers hope to have covered. Then, we sample a sequence of branches in the dataset as a coverage goal to execute directed test amplification with DSpot. In actual development scenarios, developers need to write a test method and then use test amplification to amplify it. In the experiment, we use existing test methods in the project as the initial test case to amplify. Eventually, we check how many sampled branches can be covered by the amplified test cases, indicating how well the test amplification results can meet the developers' branch coverage expectations. We use the result of the case study to answer RQ1.

We compare the directed amplification method enabled by the D-amplifier to the open test amplification approach to validate its effect. We use the two test amplification methods to amplify the same initial test cases with the same branch coverage requirement. Finally, we evaluate the efficiency of these two approaches by calculating the ratio of branches covered and the frequency with which they can generate test cases that satisfy the branch coverage requirement, which is how we answer RQ2.

1.3.3 User Study

As users play a crucial role in the developer-centric test amplification, we conduct a user study to evaluate the user-guided test amplification designed and implemented in TestCube and DSpot and compare it with the open test amplification. We recruit 12 software developers, invite them to use TestCube, and interview them about their experience.

Developers are invited to generate test cases in the user-guided and open test amplification methods and compare them to explore the differences. After developers try out the two test amplification approaches, we conduct a semi-structured interview to investigate their opinions on them. The semi-structured interview is based on a questionnaire prepared in advance, containing a series of closed-ended questions. We ask the participants to fill in answers to each question and explain why they select a particular option. We also flexibly ask additional questions based on the interviewee's responses, allowing us to interview them in-depth. During the interview, we encourage the participants to propose suggestions for our tool.

The questions in the questionnaire are divided into four parts. The first two parts are about the participants' impression of the user-guided test amplification and the open test amplification individually. The third part asks developers to compare the two approaches' differences explicitly, and the last part looks into developers' overall impression of the test amplification result and TestCube. The answer for the first part is analyzed to answer RQ3. We answer RQ4 by comparing the result of the first two parts and analyzing the third part's result. The last part contributes to answering RQ1 from the developers' perspective. We answer RQ5 by analyzing the evaluation result of our user-guided test amplification prototype and developers' suggestions.

1.4 Contributions

This thesis makes the following contributions:

1. Extended TestCube: A new user-guided test amplification interaction in the TestCube plugin, which visualizes code coverage and assists users in conveying their branch coverage expectations for amplified test cases and displays the corresponding result for user selection.
2. New amplifier in DSpot: A directed amplifier that supports directed test amplification that aims at generating new test cases meeting specific branch coverage requirements.
3. New selector in DSpot: A selector that can compute branch and line coverage of tests and select amplified test cases based on coverage requirements.
4. An evaluation of the effect of the new directed amplification method in test amplification aiming at specific branch coverage requirements.
5. Evaluate the value of user-guided test amplification and differences with the open test amplification.
6. Four suggestions for creating an effective user-guided test amplification interface.

1.5 Thesis Outline

Chapter 2 sketches the background and related work of the thesis topic. In Chapter 3, we introduce the design and implementation of the user-guided test amplification in TestCube and DSpot. Chapter 4 describes the technical case study and user study we conducted to evaluate the user-guided test amplification. Chapter 5 displays the evaluation results and discusses them to answer RQ1-RQ5. Finally, we summarize the thesis and discuss some potential future work opportunities.

Chapter 2

Background and Related Work

In this chapter, we introduce the background and related work. User-guided test amplification is motivated by the developer-centric test amplification [8] and developed based on Brandt and Zaidman's implementation, which we term open test amplification. We introduce open test amplification in detail in Section 2.1. Section 2.2 introduces previous work related to our topic, including interactive and directed test generation.

2.1 Background

Open test amplification starts from amplifying existing test cases and provides amplified test cases that bring additional instruction coverage to the test suite. It is implemented with TestCube and DSpot. TestCube provides an interface allowing developers to start the test amplification and inspect the amplification result. DSpot is the back-end that conducts the test amplification. Developers start test amplification in TestCube by selecting an existing test method and asking TestCube to amplify it. TestCube then runs DSpot to amplify the selected test case. After DSpot finishes the test amplification, TestCube displays the result, and developers can choose to keep or discard the amplified test cases interactively.

2.1.1 Developer-Centric Test Amplification with DSpot

Brandt and Zaidman's developer-centric test amplification revises the test amplification process of DSpot to generate shorter, easier-to-understand test cases and select to keep those that bring additional instruction coverage [8]. Figure 2.1 provides an overview of the developer-centric test amplification approach.

At the start of the amplification process, the developer-centric test amplification removes all the assertions and the method calls in the original test case as they will not match the new amplified test case. Then the developer-centric test amplification uses input mutation to explore a test case's input space. Literals like integers, booleans, and strings are slightly modified or replaced by random values. It also removes, duplicates, or adds new method calls on existing objects, which means it is necessary to have an existing object for the method call mutation. Specifically, DSpot only supports calling public, non-static, and non-abstract methods. It can also create new objects or literals as parameters for method

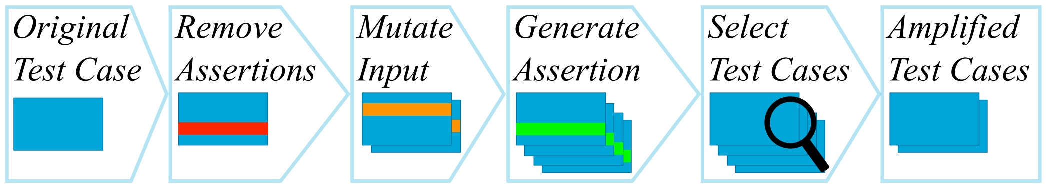


Figure 2.1: Overview of the open test amplification process within DSpot [8].

calls. With the input mutation, a series of new test cases are generated. Then the developer-centric test amplification adds one assertion to each test case. The developer-centric test amplification only adds the assertion that the value it asserts changed through the mutation to reflect the changes brought by the preceding input mutation.

After the assertion generation, we receive a broad range of test cases and can select them according to a selection criterion. Brandt and Zaidman's open test amplification design selects test cases based on the instruction coverage that DSpot supports for calculating. Specifically, the open test amplification keeps the amplified test cases contributing additional instruction coverage to the test suite.

2.1.2 Open Test Amplification Interaction in TestCube

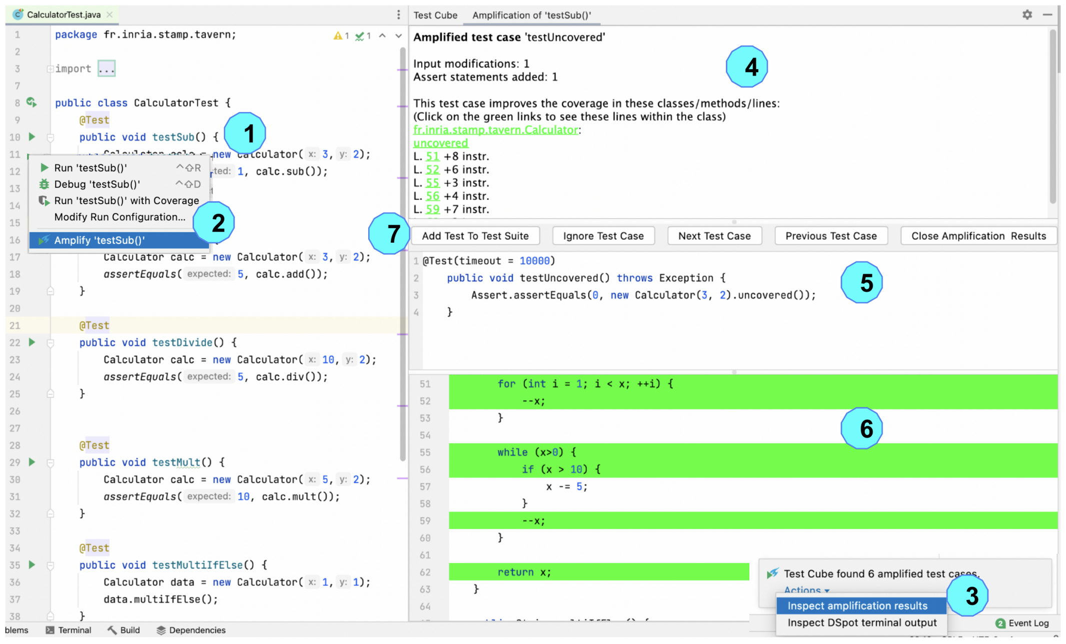
The test exploration plugin TestCube was designed to assist developers in generating and exploring test cases using the open test amplification approach. Figure 2.2 shows the procedure of the open test amplification interaction.

Developers begin open test amplification by selecting one existing test method (① in Figure 2.2) in the test suite and asking TestCube to amplify it (②). When the amplification finishes, TestCube notifies the developer with a pop-up, and developers can click to inspect the amplification result (③).

The result window consists of one amplified test case (⑤), an information box (④), five navigation buttons (⑦), and a coverage inspection editor (⑥). The information box contains the information of the amplified test case, including the additional instructions it covers and the number of modifications applied. The navigation buttons allow developers to inspect different amplified test cases, add them to the test suite, ignore them and close the result window. The coverage inspection editor highlights the additional covered lines in the source code when developers click on the additional instruction coverage line information in the information box.

2.2 Related Work

In this section, we discuss past research related to two aspects of our work: interactive test generation, which allows humans to guide the test generation, and directed test generation, which has a specific goal for test generation.



```
package fr.inria.stamp.tavern;

import ...

public class CalculatorTest {
    @Test
    public void testSub() {
        Calculator calc = new Calculator( x: 3, y: 2);
        assert(expected: 1, calc.sub());

        Calculator calc = new Calculator( x: 3, y: 2);
        assertEquals( expected: 5, calc.add());
    }

    @Test
    public void testDivide() {
        Calculator calc = new Calculator( x: 10, y: 2);
        assertEquals( expected: 5, calc.div());
    }


    @Test
    public void testMult() {
        Calculator calc = new Calculator( x: 5, y: 2);
        assertEquals( expected: 10, calc.mult());
    }

    @Test
    public void testMultiIfElse() {
        Calculator data = new Calculator( x: 1, y: 1);
        data.multiIfElse();
    }
```

Amplified test case 'testUncovered'

Input modifications: 1
Assert statements added: 1

This test case improves the coverage in these classes/methods/lines:
(Click on the green links to see these lines within the class)
fr.inria.stamp.tavern.Calculator:
uncovered
L. 51 +8 instr.
L. 52 +6 instr.
L. 55 +3 instr.
L. 56 +4 instr.
L. 59 +7 instr.

```
@Test(timeout = 10000)
    public void testUncovered() throws Exception {
        Assert.assertEquals(0, new Calculator(3, 2).uncovered());
    }
```

```
51          for (int i = 1; i < x; ++i) {
52              --x;
53          }
54
55          while (x>0) {
56              if (x > 10) {
57                  x -= 5;
58              }
59              --x;
60          }
61
62          return x;
63      }
64
```

Test Cube found 6 amplified test cases.

Figure 2.2: Overview of the open test amplification interaction in TestCube.

2.2.1 Interactive Test Generation

Several techniques incorporate humans and use the information humans provide to guide the test generation. Marculescu et al. proposed Interactive Search-Based Software Testing (ISBST) to involve domain specialists in test generation [25, 26]. Search-Based Software Testing (SBST) uses meta-heuristic optimizing search techniques, such as genetic algorithms, to automate or partially automate testing processes, such as the automatic generation of test data [32]. ISBST is designed to use a dynamically adapted fitness function during the search process, and domain-specialist users are allowed to adjust the fitness function. The fitness function is composed of attributes relevant to system quality that leads the optimization of the test cases. By changing the relative importance of these attributes, the domain specialist can change the fitness function and influence the search. Marculescu et al. carried out a series of implementations and validations for the concept of ISBST in industry [27, 28] and academic [29] and concluded that the ISBST system develops test cases that are not found by manual techniques [29] and the interaction with domain specialists makes the system more usable and more readily accepted in an industrial setting [28]. Seeing the potential of ISBST, Marculescu et al. further transfer ISBST to industry [30] using the model of technology transfer to industry proposed by Gorschek et al. [16] and discuss the lessons they learned along the way.

The primary difference between their work and ours is that they involve domain specialists in the test generation while we let software developers guide the test generation. They pointed out the importance of perfecting how automated test systems communicate with users and ensuring that their findings are understandable to the users when transferring

ISBST to industry [30]. We address this by designing an interface in a test exploration tool and visualizing all the information and test amplification results for the users' interaction.

Besides ISBST, some recent studies also extend the interactivity of test generation. Murphy et al. propose to apply Grammatical Evolution into SBST and incorporates human expertise into the search [33]. Grammatical Evolution is a grammar-based evolutionary algorithm. It uses a grammar, often a context free grammar, to create syntactically correct objects in any arbitrary language [36]. They proposed that users can define the search space they want their tests to be created from by specifying a grammar. However, a further empirical study is needed to validate the effectiveness of their approach.

Ramírez et al. observed two key issues that stymie the acceptance of automated test cases by analyzing various studies that evaluated the effectiveness and acceptance of test generation tools, which are the opacity of the automatic test generation process and the lack of cooperation with the tester [34]. To solve the issues, they make the test case's readability become their work's goal. They let testers guide the test generation by interacting with the testers for readability assessment when generating test cases using search-based algorithms. Testers' subjective assessment of test cases' readability through scores will be used to compare candidate test cases with the same fitness (aggregation of coverage criteria). Our work also addresses the concerns they raised. We cooperate with testers and make the process transparent by letting testers express their branch coverage goal and guide the test generation. We also improve the readability of test cases by connecting the amplified test cases with testers' coverage goals.

2.2.2 Directed Test Generation

Search-Based Software Testing (SBST) uses search algorithms to automatically find test cases that optimize test goals [2]. SBST uses a fitness function to capture the test objective, which is the test generation direction. SBST has been used to automate test generation for various test goals, such as maximizing structural coverage [15, 6, 21, 18], crash reproducing [39, 12, 11], and optimizing software function [42].

SBST for structural coverage is the most well studied within SBST [17]. Many approaches aim at maximizing structural coverage. The widely known tool EvoSuite generates tests for Java towards satisfying a coverage criterion [15]. AUSTIN is an open source SBST system for the C language that maximizes branch coverage [21]. TestFul generates tests for Java and aims to reach high statement and branch coverage [6]. Holmes et al. propose to use the relative line of code of software components to guide test generation as code coverage is expensive to compute; methods with more lines will be called with a higher probability [18]. Holmes et al.'s approach still aims at reaching high branch and statement coverage even though they do not use it as an explicit test generation goal. All of these approaches aim at the overall structural coverage of the software without targeting coverage of specific parts of the code structure while our work generates test cases for a specific branch.

Test suite augmentation techniques are used in regression testing to identify changed behaviors of a program and to generate test cases targeting the changes if the existing test suite is insufficient to handle the changes [40]. Most augmentation techniques are based on specific code coverage criteria [40]. Some of the test augmentation approaches only focus on identifying changed code and provide guidance for test generation but do not actually generate test cases [3, 37]; while Xu et al. devoted a series of work to the directed test generation aiming at new coverage requirements arising from new versions of a program as it evolves [44, 45, 47].

Xu et al. first proposed a concolic testing algorithm to address test augmentation [44]. Specifically, they locate the branches of the evolved program that are not covered by the existing test suite and generate test cases to cover them one by one. For each uncovered branch, they find its source node and the existing test cases whose execution traces reach the source node. Then they explore the different directions of the path conditions of the existing test cases with a concolic testing method to find new test cases that can cover the target branch. They applied their technique to an original version and 41 revised versions of one of the Siemens programs, Tcas, from the SIR [14] repository and achieved branch coverage rates between 95% and 100%.

Xu et al. address the same problem with a genetic algorithm in a subsequent work [45]. The algorithm targets an uncovered branch of the evolved program each time. The fitness function measures a test case's distance from the target branch. Minimizing the fitness function generates test cases that can cover the target branch. Their experiment shows that the branch coverage result reached the best when they used all existing test cases to compose the initial population for the genetic algorithm; 35% to 46% of the target branches are covered.

The concolic algorithm requires many constrain solver calls, and the generic algorithm needs to use all the test cases. Both of them require much computational effort. Xu et al. explore the two approaches and several factors' influence on the cost and effectiveness of test generation techniques [46]. Based on their findings, Xu et al. proposed a hybrid approach by combing the concolic and genetic algorithms to get a more cost-effective approach [47]. This new approach runs both the concolic and genetic methods for multiple rounds until no new branches are covered. They first apply the concolic testing in each round and pass the output to the genetic algorithm as the initial population. They conducted an empirical study and concluded that the hybrid algorithm is more effective than the two individual approaches but less efficient than the concolic test case augmentation technique. Xu et al. further investigated different factors' influence on their approach through an empirical investigation [48] and proposed a revised version of the hybrid approach [20]. The revised hybrid approach is an interleaving framework that interleaves test case generation algorithms dynamically and can adjust other factors potentially affecting the success of test generation flexibly, such as the initial test case they use. Their experiments show that a technique in which two test case generation algorithms are fully dynamic interleaved outperforms their previous techniques.

Although their work also uses existing test cases to generate new test cases for specific branches, they focus on test augmentation, which aims to cover the branches of a new version of a program that are not covered by the existing test cases when the program evolves. However, our test amplification method focuses on all uncovered branches of software. Besides, our approach is more cost-effective as we only use a small number of initial test cases and amplify them for one round. In contrast, their approach needs to use a large number of initial test cases and perform multiple rounds of iterations and calculations.

Besides targeting the changed code when software evolves and generating tests for regression testing, some researchers realize it is useful to generate test cases covering a particular code element for debugging [24, 13]. Ma et al. proposed directed symbolic execution to generate test cases covering a specific line [24]. Specifically, they proposed two types of directed symbolic execution, shortest-distance symbolic execution, and call-chain-backward symbolic execution. Shortest-distance symbolic execution prioritizes the path with the shortest distance to the target line [24]. Call-chain-backward symbolic execution follows the call-chain backward from the target method containing the target line until it finds a realizable path to the target [24]. The two directed symbolic execution approaches use the distance to the target line as information to guide symbolic execution. Dinges et al. proposed symcretic execution to generate test data covering a specific branch or statement in a program [13]. Their approach combines symbolic backward execution and concrete forward execution. They first use symbolic backward execution to find an execution path from the target to any program's entry points but 'skip' over problematic constraints for the symbolic decision procedure. Then concrete forward execution uses a heuristic search to find inputs that satisfy the constraints skipped by the symbolic backward execution. Both their and our work specify the direction for test generation using the target code elements. The directed symbolic execution and the symcretic execution need to analyze the path to the target code element and generate tests from scratch. The constraint solver and the heuristic search require extensive computation. However, our approach uses existing tests to generate new ones and does not use symbolic execution to reduce the computational cost.

Chapter 3

Design and Implementation

This chapter introduces the design and implementation of the user-guided test amplification prototype in detail, including the user-guided test amplification interaction in TestCube and the directed test amplification method in DSpot. In Section 3.1, we provide an overview of the implementation. Then we discuss more design details and motivation of TestCube and DSpot in Section 3.2 and Section 3.3.

3.1 Overview of the Implementation

We realized the user-guided test amplification by implementing a new interaction process in TestCube and a corresponding supportive directed test amplification method in DSpot. Figure 3.1 shows an overview of the user-guided test amplification prototype. A developer first conveys expectations for the new test cases in TestCube by selecting one method they want to test and one branch of the method they want to have covered. Then TestCube looks for existing test cases as the input of the test amplification. TestCube will remind the developer to write an initial test case if it cannot find an existing one. Then DSpot receives the initial test cases with the user's guidance from TestCube and amplifies them based on the direction, the target method and the target branch. Finally, TestCube analyzes DSpot's results and displays the amplified test cases satisfying the developer's coverage expectations. The developer can add or ignore the amplified test cases when inspecting the result.

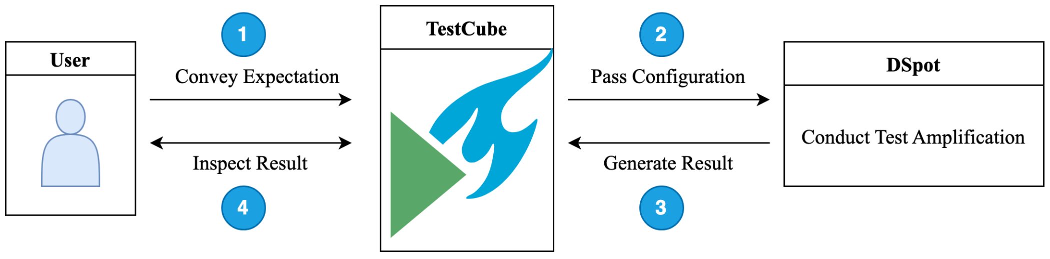


Figure 3.1: Overview of the user-guided test amplification implementation.

3.2 TestCube Extension

TestCube provides the interfaces where developers convey their branch coverage expectations for the new tests and accept the amplified test cases by inspecting the amplification results. Specifically, we help developers convey their branch coverage expectations by showing a Control Flow Graph (CFG) of the method they want to test and letting them select one branch they hope to have covered. We show the amplified test cases with their branch coverage highlighted in the CFG, and then developers can choose to add them to the test suite.

3.2.1 Developers' Expectation

The user-guided test amplification interaction in TestCube is designed to assist developers in communicating their expectations for the new amplified test cases. The most commonly used engineering goal of test amplification is to improve coverage according to a coverage criterion [9]. Therefore, we use code coverage to express users' expectations of an amplified test case. TestCube already adopted instruction coverage as a criterion to select amplified test cases in the open test amplification [8]. However, it is a very fine-grained evaluation metric, more suited to demonstrating the final result of amplified tests than helping developers express their expectations of the amplified tests.

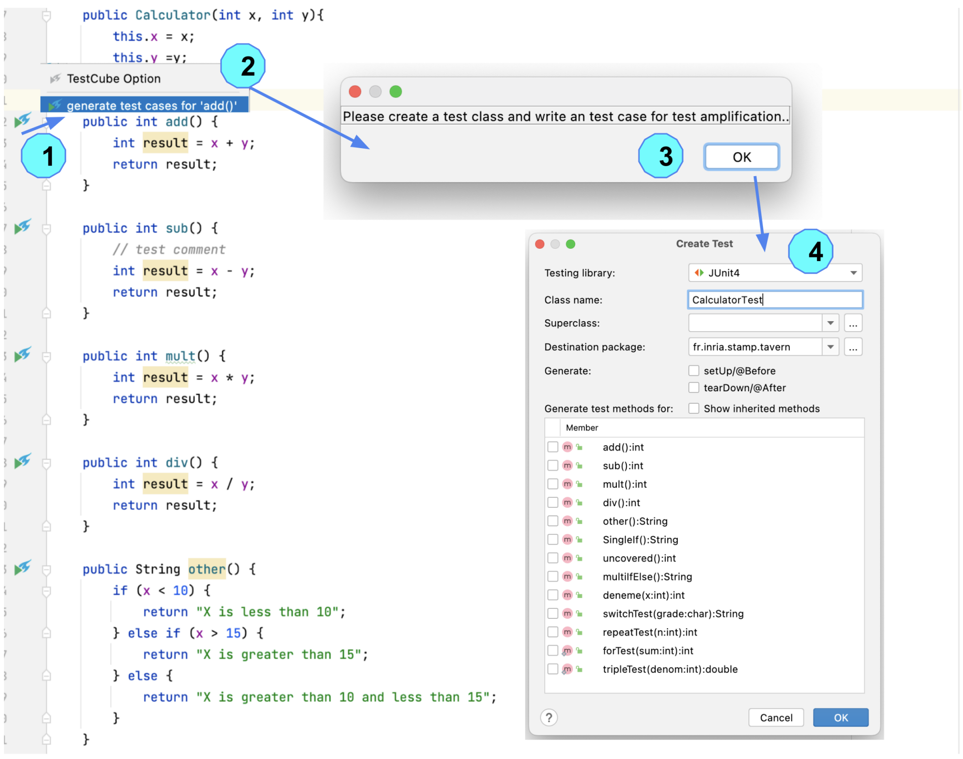
We use branch coverage to represent developers' expectations. Branch coverage indicates how many branches in a codebase tests execute. In our design, developers specify the branch they want to test instead of setting a branch coverage goal, as branch coverage is a quantitative metric that is difficult to define accurately as good or bad, while the branch is an easy code structure for developers to understand.

First, developers select one method they want to test. Branches are the possible execution paths the code can take after a decision statement is evaluated, representing different scenarios the method will face. When developers write tests for a specific method, they usually would like to test if the method is doing what it is supposed to do in different scenarios. By specifying the branch of one method they hope amplified test cases to cover, developers indicate the scenario they would like to test. In addition, it is often the case that covering a specific branch also means covering a series of corresponding lines of code. By specifying the target code branch, the developer is also considering the line coverage expectation of the amplified test cases but avoiding potentially complicated operations of selecting multiple lines of code.

3.2.2 Initial Test Cases

TestCube generates new test cases by amplifying existing test cases; hence having an existing manually written test case is a prerequisite for test generation. In our implementation, TestCube uses all test methods in the test class of the class containing the method the user wants to test as input for directed test amplification. Developers may use TestCube to generate new test cases at any stage of development when they are writing tests from scratch for a class or have already written some tests. To ensure that developers can use TestCube in both conditions, TestCube should remind and help developers write an initial test case for the class they want to test when there are no existing test cases.

When a developer selects a method and asks TestCube to generate tests for it, TestCube will look for the existing test class for the class containing the selected method. If no est class exists for it, TestCube will ask the developer to create a test class and write an initial test case. Also, TestCube will display a window to assist the developer in writing a Junit test class, shown in Figure 3.2. TestCube will ask the developer to write one test method if there is an empty test class without test methods, which is shown in Figure 3.3.



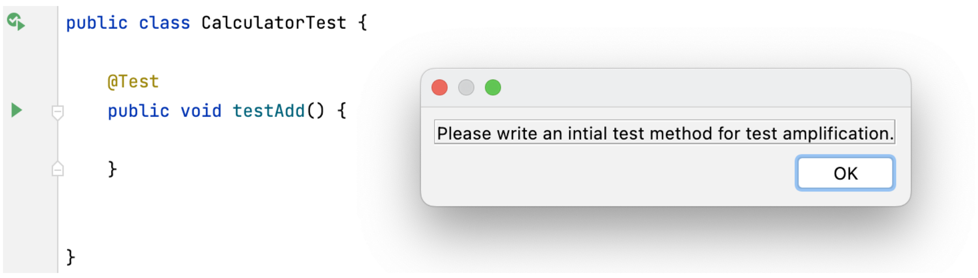
```
public Calculator(int x, int y){
    this.x = x;
    this.y =y;

public int add() {
    int result = x + y;
    return result;
}

public int sub() {
    // test comment
    int result = x - y;
    return result;
}

public int mult() {
    int result = x * y;
    return result;
}

public int div() {
    int result = x / y;
    return result;
}

public String other() {
    if (x < 10) {
        return "X is less than 10";
    } else if (x > 15) {
        return "X is greater than 15";
    } else {
        return "X is greater than 10 and less than 15";
    }
}
```

Please create a test class and write an test case for test amplification..

Testing library: JUnit4
Class name: CalculatorTest
Superclass:
Destination package: fr.inria.stamp.tavern
Generate: ☐ setUp/@Before ☐ tearDown/@After
Generate test methods for: ☐ Show inherited methods

	Member
☐	add():int
☐	sub():int
☐	mult():int
☐	div():int
☐	other():String
☐	SingleIf():String
☐	uncovered():int
☐	multiIfElse():String
☐	deneme(x:int):int
☐	switchTest(grade:char):String
☐	repeatTest(n:int):int
☐	forTest(sum:int):int
☐	tripleTest(denom:int):double

Figure 3.2: Reminder of creating an initial class.



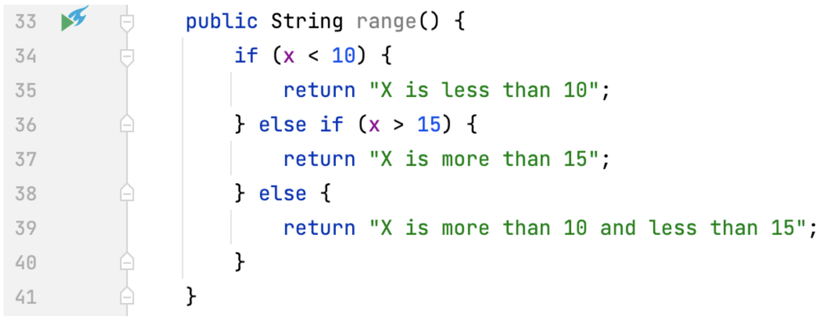
```
public class CalculatorTest {

    @Test
    public void testAdd() {

    }


}
```

Please write an intial test method for test amplification.

Figure 3.3: Reminder of writing an initial test.

3.2.3 Control Flow Graph

We show a CFG of the selected method based on the source code to better assist developers in identifying and pointing out the branch they want to cover. The example of the CFG is in Figure 3.4. The CFG in our implementation is based on PlantUML [5] and depicts a method's structure, including all branches and lines. The boxes represent code lines, and decision statements and arrows illustrate the code flow. The arrows out of each decision statement have "True" and "False" on them, denoting different branches. By inspecting the CFG, developers can see how the code runs following each true and false branch of each decision statement and quickly understand different scenarios that possibly need testing. Besides, each box of the CFG contains both the line number and corresponding source code, which can help developers locate the code in the source file.



```
public String range() {
    if (x < 10) {
        return "X is less than 10";
    } else if (x > 15) {
        return "X is more than 15";
    } else {
        return "X is more than 10 and less than 15";
    }
}
```

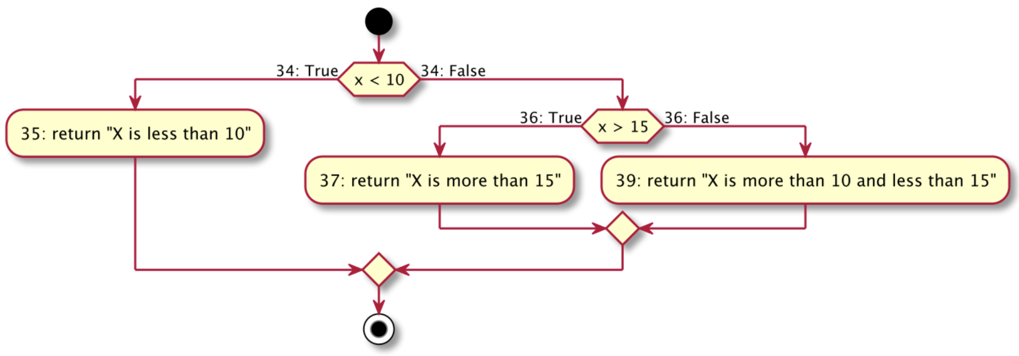

34: True
x < 10
34: False
35: return "X is less than 10"
36: True
x > 15
36: False
37: return "X is more than 15"
39: return "X is more than 10 and less than 15"

Figure 3.4: Example Control Flow Graph of method.

The selected method may have been partially covered as there are existing test methods that are the input of test amplification. Knowing which part of the method is uncovered is essential for developers. It allows the developer to learn about the current code coverage and identify branches that are not yet covered. Therefore, TestCube computes initial coverage for the selected method and first displays it in the CFG. We compute both branch coverage and line coverage as the CFG comprises branches and lines. Figure 3.5 shows the initial coverage identified in the CFG. The dark green border identifies the branches and lines covered by existing tests in the CFG. We use pink borders to identify the branches that are not covered and available for selection. Developers are allowed to select one branch each time. The pink border becomes red after being selected. Developers click the "Generate

Test Covering the Selected Branch" button below the CFG to start the test amplification. TestCube will amplify all the existing test cases and return test cases that can cover the selected branch.

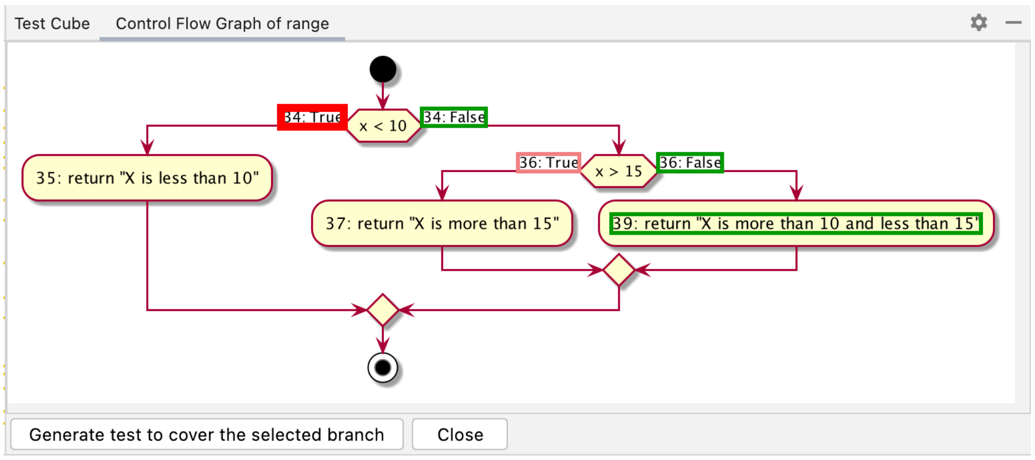
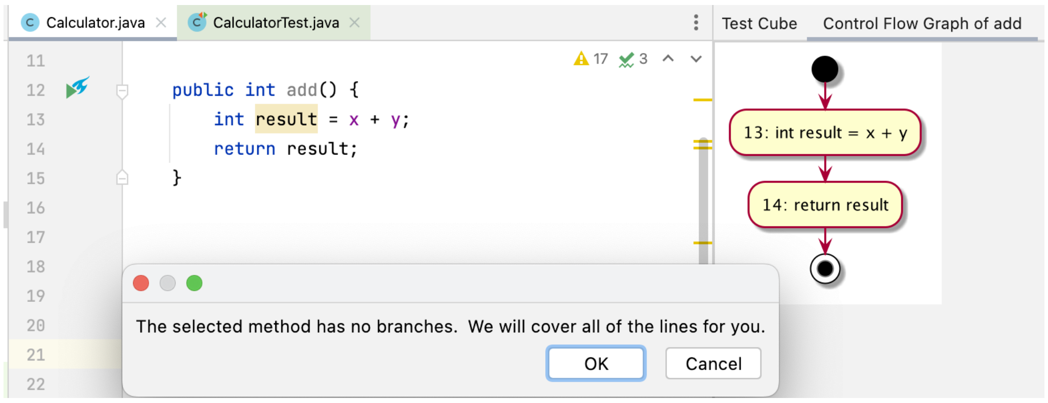

Test Cube
Control Flow Graph of range
34: True
x < 10
34: False
35: return "X is less than 10"
36: True
x > 15
36: False
37: return "X is more than 15"
39: return "X is more than 10 and less than 15"
Generate test to cover the selected branch
Close

Figure 3.5: Example initial coverage in the CFG allowing branch selection.

If the selected method has no branches, TestCube informs the user that it will generate test cases that can cover the method. TestCube will not generate new test cases if the selected method is fully covered. The examples of the two cases are in Figure 3.6 and Figure 3.7.



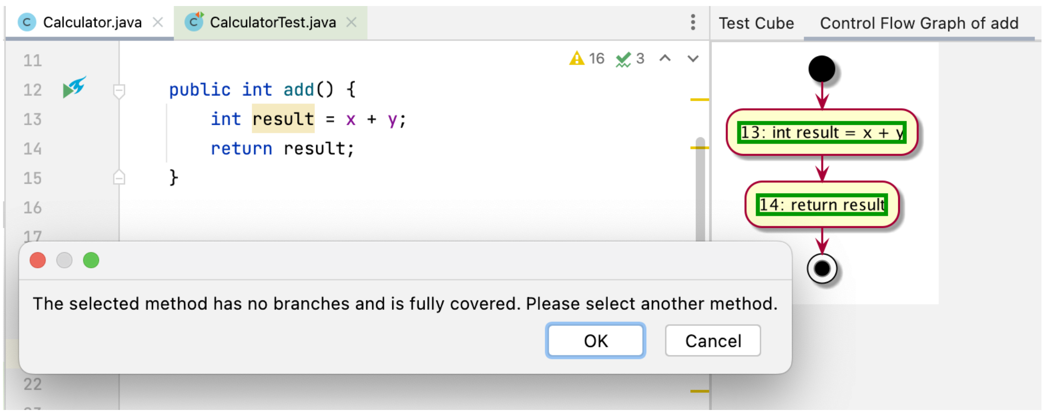
```
public int add() {
    int result = x + y;
    return result;
}
```

The selected method has no branches. We will cover all of the lines for you.

Figure 3.6: Example when the selected method has no branches.

3.2.4 Results Display

After test amplification completes, TestCube will pop up a notification to report the result, indicating the number of amplified test cases covering the selected branch. Developers can choose to inspect the result and explore the result window. Figure 3.8 shows an example of the test amplification result where the left part is IntelliJ's regular editor and the right part is

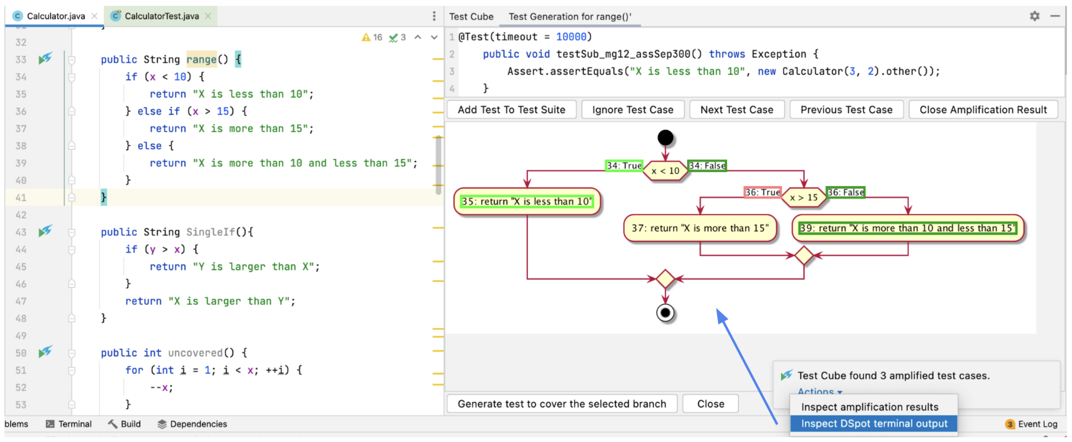


```java
public int add() {
    int result = x + y;
    return result;
}
```

The selected method has no branches and is fully covered. Please select another method.

Figure 3.7: Example when the selected method is all covered.

the TestCube result window. The amplified test case is at the top of the result window, and the CFG of the target method is at the bottom. Besides the existing coverage identified by dark green borders, light green borders identify the new branch coverage and line coverage brought by the amplified test case in the upper part. When developers switch to inspect another amplified test case, the new coverage on the CFG updates correspondingly. Also, developers can continue to select the pink-bordered branches to continue to generate test cases for them. There are a series of buttons below the amplified test cases where developers can switch between amplified test cases and add amplified test cases to their test suite, which follows the existing design of the open test amplification [8].


Calculator.java
CalculatorTest.java
Test Cube
Test Generation for range()'
Add Test To Test Suite
Ignore Test Case
Next Test Case
Previous Test Case
Close Amplification Result
34: True
x < 10
34: False
35: return "X is less than 10"
36: True
x > 15
36: False
37: return "X is more than 15"
39: return "X is more than 10 and less than 15"
Generate test to cover the selected branch
Close
Actions
Inspect amplification results
Inspect DSpot terminal output
Event Log
Terminal
Build
Dependencies

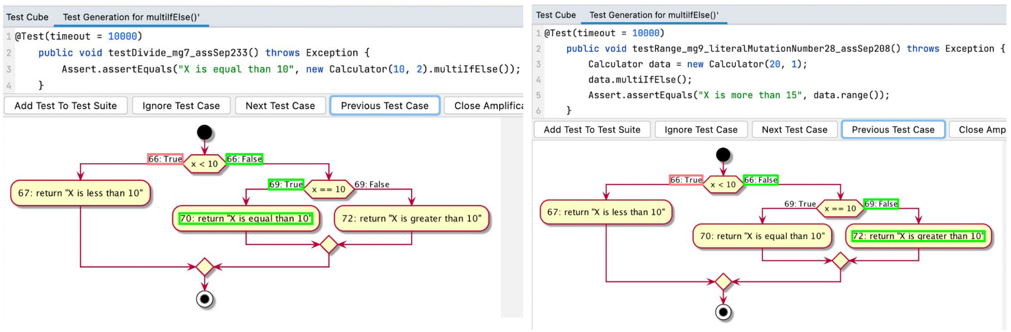
```java
public String range() {
    if (x < 10) {
        return "X is less than 10";
    } else if (x > 15) {
        return "X is more than 15";
    } else {
        return "X is more than 10 and less than 15";
    }
}

public String SingleIf(){
    if (y > x) {
        return "Y is larger than X";
    }
    return "X is larger than Y";
}

public int uncovered() {
    for (int i = 1; i < x; ++i) {
        --x;
    }
```

```java
@Test(timeout = 10000)
    public void testSub_mg12_assSep300() throws Exception {
        Assert.assertEquals("X is less than 10", new Calculator(3, 2).other());
    }
```

Test Cube found 3 amplified test cases.

Figure 3.8: Example of user-guided test amplification result.

We limit the number of amplified test cases to three in the final implementation. We found that most of the amplified test cases covering a specific branch are similar because they amplify the same initial test cases, call the same method, and use similar parameters to satisfy the same decision statement. Providing developers with too many options to analyze and choose from is unnecessary. Nevertheless, there is still potential diversity among the amplified test cases when there are multiple nested branches in the target method. For instance, Figure 3.9 shows two amplified test cases with different coverage. The two amplified test cases are from the amplification result targeting covering the branch "66:False" in the first level of the nested branches. However, they cover different branches of the inner decision statements. Keeping such diversities can provide developers with potential benefits.



```
@Test(timeout = 10000)
    public void testDivide_mg7_assSep233() throws Exception {
        Assert.assertEquals("X is equal than 10", new Calculator(10, 2).multiIfElse());
    }
```

(a) Amplified test case 1

```
@Test(timeout = 10000)
    public void testRange_mg9_literalMutationNumber28_assSep208() throws Exception {
        Calculator data = new Calculator(20, 1);
        data.multiIfElse();
        Assert.assertEquals("X is more than 15", data.range());
    }
```

(b) Amplified test case 2

Figure 3.9: Example amplified test cases with coverage diversity.

3.3 DSpot Extension

DSpot is TestCube's backend. TestCube passes input and configuration settings to DSpot and calls DSpot to perform test amplification. After DSpot finishes execution, TestCube reads and analyzes the result files produced by DSpot and then displays the critical information in the result window.

3.3.1 Extend Configuration Option

To support directed test amplification, DSpot is first extended to be able to accept the direction given by developers in the form of configuration parameters. The direction includes the method developers want to test and the branch they want the amplified test cases to cover.

3.3.2 Coverage Computation

DSpot utilizes OpenClover [31] to support computing branch and line coverage. The branch and line coverage is crucial when assisting developers in inspecting the current coverage status and selecting amplified test cases according to developers' coverage expectations. When assisting developers in choosing the branch they want to cover, DSpot computes the branch coverage and line coverage of the existing tests to help TestCube show the initial coverage in the CFG. After finishing test amplification, DSpot generates a report containing each amplified test case's branch coverage and line coverage. Then TestCube selects the amplified test case according to the coverage report. The report, including all the amplified test cases, can also be reused by the following test amplification round and helps save time in case of repetitive runs of test amplification. When another round of directed test amplification starts, TestCube will look in the previous round's output for amplified test cases that can satisfy the new requirement. The directed test amplification will perform again only when no existing amplified test case meets the new requirement. Otherwise, TestCube will read the existing result and return it.

3.3.3 Directed Amplifier

Brandt and Zaidman's developer-centric test amplification method, which we call open test amplification, uses a series of amplifiers to mutate the input of a test case in diverse ways without a specific direction. To better support directed test amplification and improve the possibility of generating amplified test cases that can cover a specific branch, DSpot is extended with a new directed amplifier, the D-amplifier.

The D-amplifier becomes a directed test amplifier by calling the target method according to the configuration. Figure 3.10 shows the directed test amplification process. D-amplifier is used to call the target method after removing all assertions to set a direction before performing any other input mutation. This way, all the other input mutations will be performed on a test case called the target method. Modifying other elements of the test case is like changing the scenarios the target method will face. As a result, the directed test amplification amplifies the test case in the direction of testing the target method and trying to cover different branches of the method.

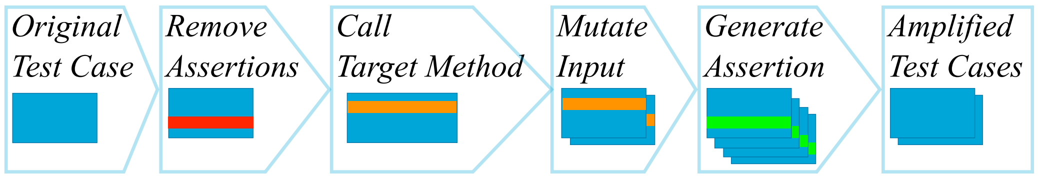


Figure 3.10: Overview of the directed test amplification process within DSpot.

Chapter 4

Evaluation

This chapter introduces the evaluation we conduct to explore the value of user-guided test amplification and compare it with Brandt and Zaidman's open test amplification, including the technical case study and user study. The technical case study evaluates if the directed test amplification method in DSpot can generate more amplified test cases that can cover a given branch than the open test amplification method. The user study investigates developers' opinions on user-guided test amplification.

4.1 Technical Case Study

The technical case study mainly consists of preparing a dataset, using the dataset to conduct test amplification with the directed test amplification method and the open test amplification method in DSpot, and collecting and analyzing the test amplification results of the two methods. Section 4.1.1 describes how we prepare the dataset, and Section 4.1.2 introduces how we conduct the case study and process the results.

4.1.1 Technical Case Study Dataset

As introduced in Section 1.3.2, we need several Java projects and sample their branches to construct the dataset. Therefore, the source code of the projects we use should have many code branches. Besides, when generating test cases for a specific branch, the initial test case that we amplify has to contain one object of the class that the branch belongs to so that we can call the target method containing the branch. Therefore, for each branch, we first find the class that contains it and then use the first test method in the corresponding test class as the directed test amplification's input. It requires that the projects we use have a clearly defined test class for each class.

We selected two open-source Java projects, Javapoet[1] and Stream-lib[2] to conduct the case study. Javapoet is a Java API for generating java source files, and Stream-lib is a Java library for summarizing data in streams. They are both built with Maven and tested with

[1] `https://github.com/square/javapoet`
[2] `https://github.com/addthis/stream-lib`

JUnit. They were also part of the dataset used for the evaluation of DSpot [10]. We selected the two projects because they have many branches in the source code. Specifically, Javapoet has 7026 branches and Stream-lib has 1250 branches. Besides, the two projects' unit tests are clearly structured and defined as most classes have a clearly corresponding test class.

We first filtered all the classes with corresponding test classes to guarantee that we could find the initial test case to amplify and generate new test cases for each target branch. Then we filtered all the methods in the filtered classes that DSpot supports to call on an existing object. For every branch in the filtered methods, we use the first test method in the corresponding test class as the initial test case used to amplify. All the branches and their corresponding initial test cases build up the complete dataset of the case study.

4.1.2 Technical Case Study Execution

We sampled 100 branches and their initial test cases from the dataset described in the previous section for each project to conduct the directed test amplification and the open test amplification.

To ensure that the two methods generate the same number of amplified test cases and limit the execution time, we limit the number of amplified test cases after input mutation to 200 and the number of amplified test cases after generating assertions to 200.

Finally, we collect the amplification result of the two test amplification approaches, including all the amplified test cases generated and the corresponding code coverage they bring to the target branch. We calculate the same two metrics, the ratio of covered branches (Equation 4.1) and the ratio of satisfying test cases (Equation 4.2) for the directed test amplification method and the open test amplification. We use the two metrics to evaluate and compare the two test amplification methods' effectiveness.

For each test amplification method, we filter all the branches covered by the amplified test cases and count their number to calculate the ratio of covered branches. Equation 4.1 shows how we calculate this metric, and the number of sampled branches equals 100. The ratio of covered branches indicates the probability that a test amplification method will generate test cases that can cover a given branch.

For each test amplification method and each branch covered by its amplified test cases, we count the number of all generated amplified test cases and the number of amplified test cases that can cover the branch. Equation 4.2 shows how we calculate the ratio of satisfying test cases with the two numbers. The ratio of satisfying test cases indicates the frequency that a test amplification method generates satisfying test cases for a branch it can cover.

$$ratio\ of\ covered\ branches = \frac{number\ of\ branches\ covered}{number\ of\ sampled\ branches} \tag{4.1}$$

$$ratio\ of\ satisfying\ test\ cases = \frac{number\ of\ amplified\ test\ cases\ covering\ the\ given\ branch}{number\ of\ all\ amplified\ test\ cases} \tag{4.2}$$

4.2 User Study

As described in Section 1.3.3, we conduct a user study to investigate developers' opinions on the user-guided test amplification, explore the different value of user-guided test amplification and open test amplification, and analyze the key facets for creating an effective user-guided test amplification interface. Section 4.2.1 introduces the preparation we did before conducting the study. The formal interview mainly consists of two user tasks and a semi-structured interview. Section 4.2.2 and Section 4.2.3 explain the design of the two parts. Finally, Section 4.2.4 describes how we conduct the study, including the interview's procedure and data collection and analysis.

4.2.1 Pre-Study Preparation

As our study involves Human Research, we designed the study following the Human Research Ethics Committee (HERC) guidelines of the Delft University of Technology. We obtained the HERC's approval before conducting the study.

We prepare an informed consent to seek participants' consent for the study and data use. The consent survey is in Appendix C.1. The first part of the consent survey introduces the interview procedure and requests their consent to participate in the study. The second part explains how we collect, store, process, and use their data and requests consent for each step. We also prepare a pre-test [19] questionnaire to investigate the participants' technical background and experience if they agree to take part in our study. Appendix C.2 shows the pre-test questionnaire containing a series of demographic questions.

We use convenience sampling [23] to recruit participants for our user study. Specifically, we invite software developers to participate in our study by posting recruitment ads on Twitter[3] and LinkedIn[4].

4.2.2 User Task Design

To evaluate the user-guided test amplification and compare it with the open test amplification, we ask every participant to use the two methods to generate test cases before the semi-structured interview. We ask participants to try the two methods one by one to allow them to better distinguish between the two methods. In addition, to fairly compare the two methods, we fairly present the two methods and inform participants that both methods are our primary assessment targets.

Specifically, we prepared two Java classes and one initial test case for each class as the starting point where developers start test amplification. Each participant is asked to generate new test cases for each class by one of the two methods. Using two different classes rather than the same class for the two methods avoids the order in which the two methods are used affecting the results. Suppose two methods are used one after the other to generate tests for

[3]`https://twitter.com/danyao_wang/status/1513838978528681987?t=7fH31M00xYrewwfZT6Dkxw&s=19`

[4]`https://www.linkedin.com/posts/danyao-wang-56955a200_testcube-experiment-interview-danyao-wang-activity-6919586914115145730-zwl0?utm_source=linkedin_share&utm_medium=member_desktop_web`

the same class. In that case, participants may become familiar with the class after trying the first method, which may affect their impression of the second method.

The two classes we select are from the project Stream-lib[5] and shown in Appendix B.1 and B.2, `StreamSummary.class` and `ConcurrentStreamSummary.class`. The two classes are used to store and count elements in a stream and have methods with similar functions but are implemented differently in detail. To prepare input for the test amplification, we prepare initial test cases for the two classes that differ only in the class of the object but are otherwise the same, as shown in Appendix B.3 and B.4. The two classes are similar in terms of functionality and complexity, and the two initial test cases are almost the same, so their differences would have little influence on test amplification results.

Besides, we split participants into four groups and exchange the order of them trying the two methods and using the two classes to avoid bias owing to order. The set-up of the four groups of participants is in Table 4.1.

Table 4.1: Set-up of the user tasks.

	First Task	Second Task
Group1	User-Guided Test Amplification *StreamSummary*	Open Test Amplification *ConcurrentStreamSummary*
Group2	User-Guided Test Amplification *ConcurrentStreamSummary*	Open Test Amplification *StreamSummary*
Group3	Open Test Amplification *StreamSummary*	User-Guided Test Amplification *ConcurrentStreamSummary*
Group4	Open Test Amplification *ConcurrentStreamSummary*	User-Guided Test Amplification *StreamSummary*

4.2.3 Semi-Structured Interview Design

We conduct a semi-structured interview to investigate participants' experiences after trying the two test amplification methods. Specifically, we let participants fill out a post-test questionnaire and ask them to explain why they choose a specific option. The questionnaire contains prepared closed-ended questions whose answers are easy to analyze and provide quantitative results. By talking to the participants, we adapted the conversation depending on the respondents' answers, allowing us to probe for a deeper understanding. The discussion allows the participants to converse in-depth, choose their own words, and contribute qualitative data.

The complete questionnaire is in Appendix C.3 and consists of four parts. The first two parts are about user-guided and open test amplification, respectively. The third part explicitly compares the two test amplification methods, and the last part is about participants' overall impression of TestCube. The order of the first two parts is consistent with the order in which a participant tries the two test amplification methods.

[5]https://github.com/addthis/stream-lib

4.2.3.1 User Guided Test Amplification

The five questions used to investigate developers' opinions on the user-guided test amplification are in Appendix C.3.1. All the questions are statements to be agreed or disagreed in various degrees, and participants are supposed to rate on a 1-5 Likert scale [22], where one means "strongly disagree" and five means"strongly agree".

There are two core panels for user-guided test amplification. The first panel is introduced in Section 3.2.3, which contains the CFG with the initial coverage of the method developers want to test, where developers can select one branch they hope to cover and start the test amplification. The second panel is introduced in Section 3.2.4, which shows the test amplification result, including one amplified test case and the CFG of the target method presenting the new coverage brought by the amplified test case.

The CFG is the primary content that we provide to support the user-guided test amplification. The first two questions focus on it. We ask participants if they can understand the CFG easily and find valuable information in it. As the user-guided test amplification interaction aims to help developers generate new test cases according to their requirements, the third question asks if the interaction with the CFG can effectively assist developers in conveying their expectations for the test cases. The last two questions evaluate the usefulness of the CFG in the result panel. We investigate if it helps developers understand the amplification result and select the amplified test cases.

4.2.3.2 Open Test Amplification

We also prepare five questions for the open test amplification interaction to fairly evaluate and compare the two test amplification methods, which are in Appendix C.3.2. The five questions' format is the same as for the user-guided test amplification.

The open test amplification has one window shown in Fig 2.2, which shows the result of the test amplification, containing one amplified test case, a test case information box, and a coverage inspection editor. The information box offers the modification information and an additional instruction coverage list brought by the amplified test case. The editor shows the additional instruction coverage with source code highlighting. To compare the instruction coverage's and the branch coverage's understandability, the first question asks if developers can easily understand the instruction coverage. The second question investigates the valuable information participants find in the open test amplification to compare it with that in the user-guided test amplification. The third question asks if the instruction coverage helps participants understand the amplification result so that we can compare it with the branch coverage. Question 4 and Question 5 ask developers if the modification information and the instruction coverage are helpful when they select to add the amplified test cases, which is used to compare with the CFG's usefulness in result selection.

4.2.3.3 Comparison of Two Test Amplification Methods

Besides comparing the answers to the first two parts of the questionnaire, we ask developers to compare the differences between the two test amplification methods to obtain more direct comparison results.

User-guided and open test amplification have three main differences:

1. The user-guided test amplification uses branch and line coverage, while the open test amplification uses instruction coverage.
2. The user-guided test amplification uses the CFG to display the code coverage, while the open test amplification displays the coverage in text.
3. They are two different types of test amplification. The user-guided test amplification aims at covering a specific method and branch directly, while the open test amplification aims to cover any uncovered instructions.

The four questions to compare the three differences are in Appendix C.3.3. The first two questions ask participants which coverage and display form is easier to understand. The last two questions inquire which test amplification method helps them more in result selection and test generation.

We use multiple choice questions instead of statements and the Likert scale to organize the four questions. We do not use statements to organize the four questions because they involve comparisons, and using statements to describe either of the parties being compared as better could impact the participants' opinions. Every multiple choice question has three options: the two objects being compared and the neutral. The order of the three options is shuffled for different participants to avoid bias due to a fixed order.

4.2.3.4 Overall Impression

We use two questions to investigate participants' overall impression of the amplified test cases proposed by TestCube, which are in Appendix C.3.4. This part assesses the amplified test cases generated by the directed test amplification method from the developer's perspective, which can be a complement to the technical case study.

4.2.4 User Study Execution

We conducted the user study in April 2022 with the 12 recruited participants. They are divided into four groups evenly based on Table 4.1. All the interviews were fully remote and conducted via Zoom. Most of the interviews were finished in a session of about 60 minutes. Three interviews lasted longer, up to 90 minutes, when participants volunteered to spend more time sharing their opinions.

4.2.4.1 Interview Procedure

Figure 4.1 shows an example procedure of the interview for Group1 in Table 4.1.

We first send each developer who registers for the user study a consent survey and a pre-test questionnaire and ask them to read and fill them out before the official interview begins.

When an official interview starts, we first spend about 10 minutes briefly introducing our research, including the concept of test amplification and developer-centric test amplification. Then we explain that the goal of the interview is to evaluate and compare the two

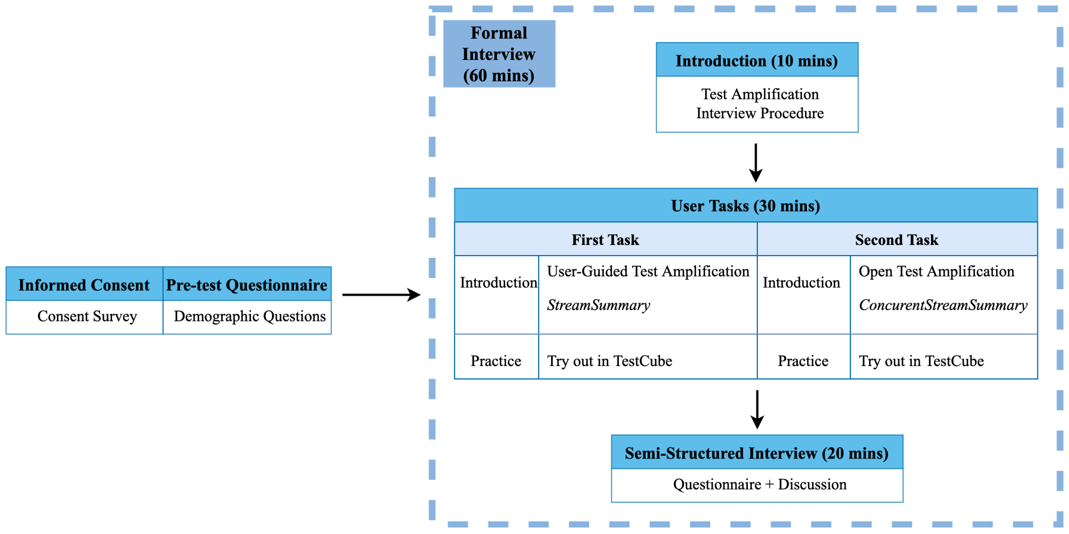


Figure 4.1: Example procedure of the user study interview.

developer-centric test amplification methods in TestCube, the user-guided test amplification and the open test amplification.

In order to make participants distinguish more clearly between the two methods and avoid confusion of concepts, we divide the user task into two separate tasks based on the two methods. We use Group1's first task in Table 4.1 to explain. We first explain the concept of the user-guided test amplification method and use a simple example project different from what the participants will use for the task to show how the method works in TestCube. Then we introduce the project Stream-lib, the class `StreamSummary`, and the initial test case they will use to try out the method to help them understand the environment. After the participants understand everything, they are allowed to remote control the interviewer's computer and interact with the TestCube running on the interviewer's computer. The participants then start test amplification, explore the information shown in the tool window and inspect the test amplification result. The procedure of the second task is the same as the first task. The participants are allowed to spend about 10 minutes on each task, and they usually can finish one or two rounds of test amplification for each task. The two tasks usually take 30 minutes in total, including the task introduction and task practice.

After participants finish the two TestCube tasks, we spend about 20 minutes on the semi-structured interview. We send the participants a link to the post-test questionnaire. Then we ask them to answer every question and explain why they select a specific option. For the opinions that participants do not explain clearly, we encourage them to discuss them in more detail. We also encourage the participants to put forward suggestions to help us improve TestCube. The interview ends when participants finish the questionnaire and submit it.

4.2.5 Data Collection and Analysis

We collected the answers to the pre-test and post-test questionnaires the participants submitted. Besides, we took notes during the interview to collect the participants' opinions. As all the participants agreed to record the interview, we also recorded the semi-structured interview in audio format so that we could check the recording when anything was missing in the notes. Finally, we summarized and categorized the data obtained according to the questionnaire section and question topic.

Chapter 5

Results and Analysis

This chapter presents the technical case study and user study results and analyzes them to answer the research questions.

5.1 Technical Case Study Results

This section presents the technical case study result. Section 5.1.1 displays the dataset we created for the technical case study according to the method described in Section 4.1.1. Section 5.1.2 displays the two metrics of the test amplification result introduced in Section 4.1.2.

5.1.1 Technical Case Study Dataset

Table 5.1 shows the statistical information about the dataset we built to conduct the technical case study, including the number of classes and branches that are filtered according to the rule described in Section 4.1.1. Table 5.2 lists the specific class names of the dataset.

Table 5.1: Technical case study dataset.

Project	Number of Classes	Number of Branches	Number of Sampled Branches
Javapoet	13	160	100
Stream-lib	18	264	100

5.1.2 Test Amplification Results

We collected the amplification result of the directed and open test amplification methods and calculated the two metrics introduced in Section 4.1.2.

Table 5.3 shows the ratio of covered branches for the two example projects obtained by the two test amplification methods. We can see that more branches are covered for Stream-lib by both test amplification methods. Besides, the directed test amplification method can

Table 5.2: Classes of the technical case study dataset.

Project	Class
Javapoet	*AnnotationSpec, TypeName, ClassName, CodeBlock, CodeWriter, FieldSpec, JavaFile, LineWrapper, MethodSpec, NameAllocator, ParameterSpec, TypeSpec, Util*
Stream-lib	*Lookup3Hash, MurmurHash, ConcurrentStreamSummary, SampleSet, StochasticTopper, StreamSummary, AdaptiveCounting, CountThenEstimate, HyperLogLog, HyperLogLogPlus, LinearCounting, LogLog, RegisterSet, CountMinSketch, BloomFilter, Filter, QDigest, DoublyLinkedList*

cover more branches than the open test amplification method for both Javapoet and Stream-lib.

Table 5.3: Ratio of covered branches.

	Javapoet	Stream-lib
Open Test Amplification	23%	35%
Directed Test Amplification	32%	41%

We calculate the ratio of satisfying test cases for each branch covered by the two test amplification methods. Figure 5.1 shows the box plots that describe the distribution of the ratio of satisfying test cases for each project and each test amplification method. Comparing the two projects, we can see that the distribution for Javapoet is relatively concentrated while the distribution for Stream-lib is highly dispersed. Moreover, the mean ratio of satisfying test cases for Javapoet is lower than that for Stream-lib for the same test amplification method. Comparing two test amplification methods, the median ratio of satisfying test cases of the directed test amplification is higher than that of the open test amplification.

We also calculate the average ratio of satisfying test cases for each project and test amplification method, shown in Table 5.4. We can see that the directed test amplification's ratio of satisfying test cases reaches 70% on average for both projects, which far exceeds the result of the open test amplification.

Table 5.4: Average ratio of satisfying test cases.

	Javapoet	Stream-lib
Open Test Amplificatino	24%	45%
Directed Test Amplification	70%	70%

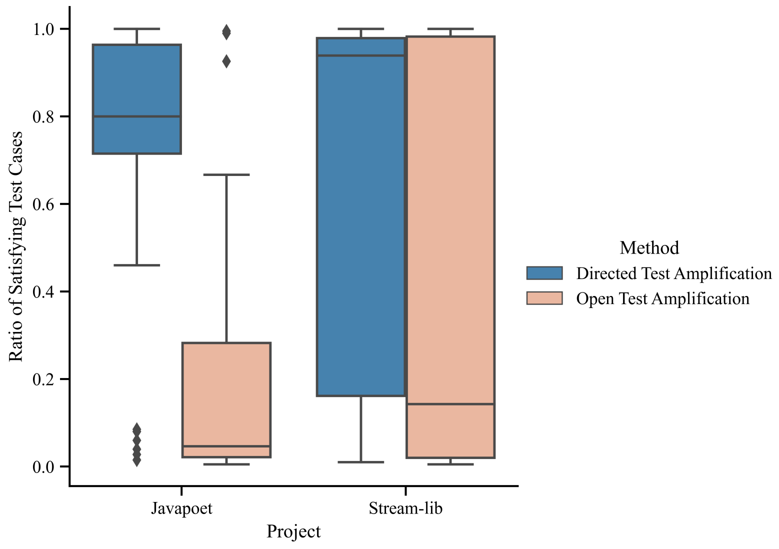


Figure 5.1: Distribution of the ratio of satisfying test cases.

5.2 User Study Results

We present the questionnaires and interview results in five categories. We show the participants' demography and experience in Section 5.2.1 based on the data of the pre-test questionnaire. The other four parts are classified according to the questions of the post-test questionnaire. Section 5.2.2 and Section 5.2.3 show the participants' opinions on the user-guided and open test amplification. We asked participants to compare the differences between the two test amplification methods and present the result in Section 5.2.4. Section 5.2.5 summarizes participants' impression on TestCube and further suggestions.

5.2.1 Participants

We recruited 12 participants for the user study and investigated their technical background and experience through the pre-test questionnaire. The demographic questions in the pre-test questionnaire are in Appendix C.2.

Figure 5.2 demonstrates the participants' development experience. The participants' experience with software development and testing varies from 1 to 9 years. Only 2 participants have used an automatic test generation tool before. One has used a test data generation tool to generate data for batch tests. The other's experience is with JUnit, which she used to build a test framework. Both rated the tool they used before three points on a scale between 1 and 5.

Figure 5.3 shows the participants' technical background. The participants' primary programming languages are Python and Java. Although Java is not the primary language for many participants, they all have experience with Java. Moreover, they come from seven different industry domains.

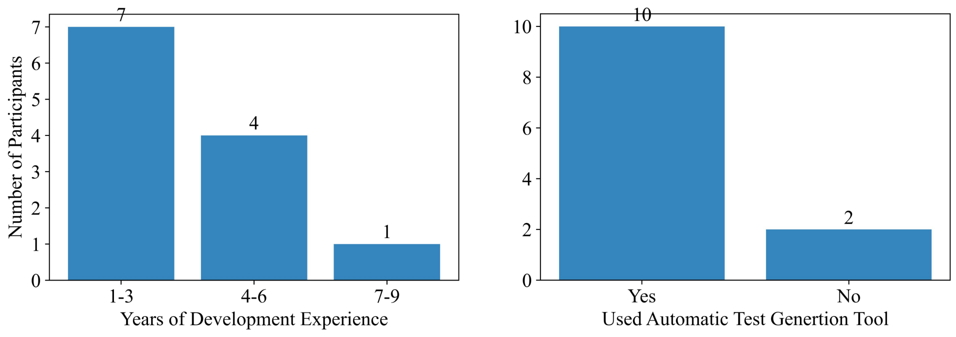


Figure 5.2: Participants' experience of software development and testing.

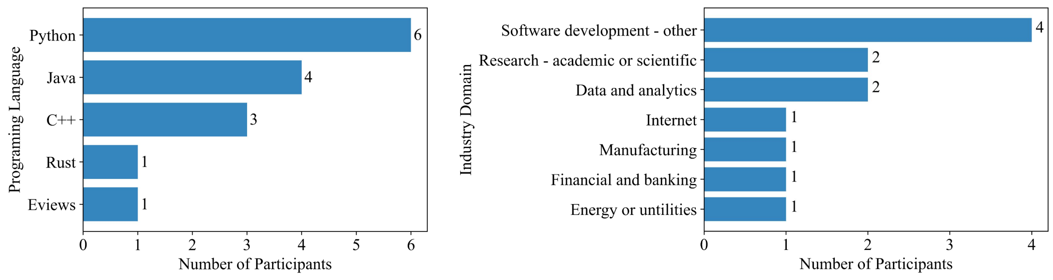


Figure 5.3: Participants' technical background.

5.2.2 User-Guided Test Amplification

The questions to survey participants' perceptions of the user-guided test amplification are in Appendix C.3.1. They focus on four aspects of the user-guided test amplification, its understandability, valuable information it provides, its effectiveness in conveying developers' expectations, and its usefulness in result selection.

5.2.2.1 Understandability

The first and the fourth questions in Appendix C.3.1 ask participants if the CFG and the amplification result shown with the CFG are easy to understand. The distribution of the participants' answers over the Likert scale is shown in Figure 5.4. We also calculate the mean and median scores and mark them in the distribution figure.

Almost all participants found the content in the two windows of user-guided test amplification easy to understand.

For the CFG itself, the participants think it is simple, and the shapes of the components are standardized for use in general flowcharts, making it easy to understand.

For the coverage identified in the CFG, the participants think the colors are pretty distinct. The significant contrast between different colors is clear. One participant who is red-green weak mentioned that even though the old coverage and the new coverage are both green, the significant contrast between the light green and dark green is still large enough

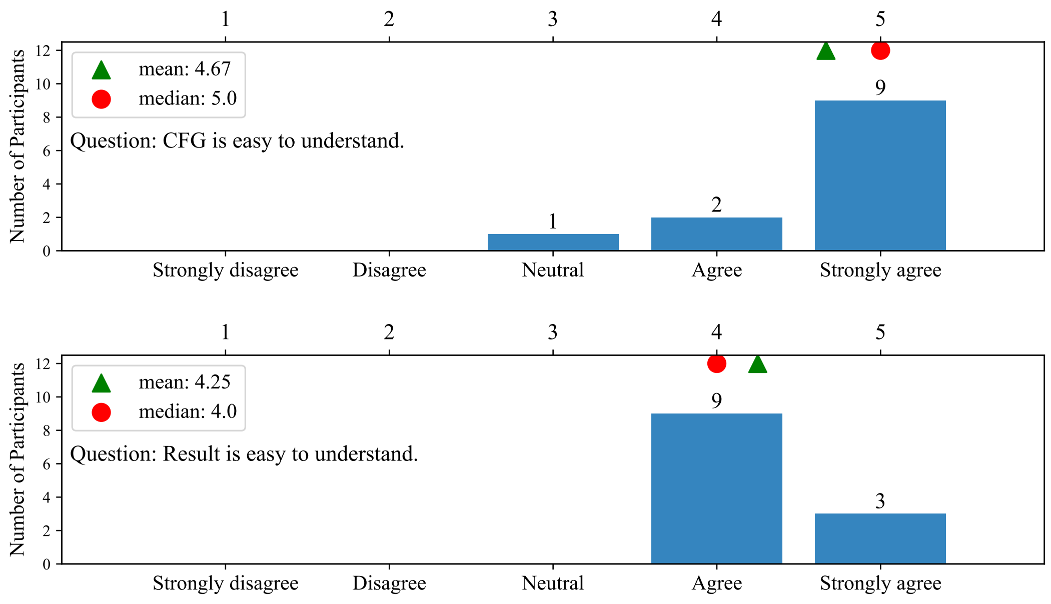


Figure 5.4: Answer distribution of the questions about the user-guided test amplification's understandability.

for him to tell them apart. The coverage is especially easy to understand for the participants familiar with coverage tools as we use the typical colors used in other coverage tools. Only one participant indicated that these colors were not immediately apparent to him as he was unfamiliar with code coverage tools; therefore expressed neutrality for the statement "CFG is easy to understand".

For the result window, participants think it is appropriate to present one amplified test case each time, making it clear and easy to connect the amplified test case with the corresponding new coverage. Participants also think the coverage in the CFG helps them understand the amplified test cases. They believe it is not easy to understand the amplified test cases individually. Nevertheless, the new coverage on the CFG helps them know the modifications applied and what is happening.

5.2.2.2 Valuable Information

The second question in Appendix C.3.1 asks if developers find valuable information in the CFG. The participants' answer distribution is in Figure 5.5.

All participants agree that the CFG provides them with valuable information. The primary information they value is the **code structure and coverage**.

The CFG provides users with much information about the code structure, including the lines, branches, statements, flows, and all possibilities for program execution. It tells developers what kind of situation they are supposed to expect when they test. The line number in the CFG is valuable information when they want to check the source code. The code structure information helps developers understand the code. Participants think they can

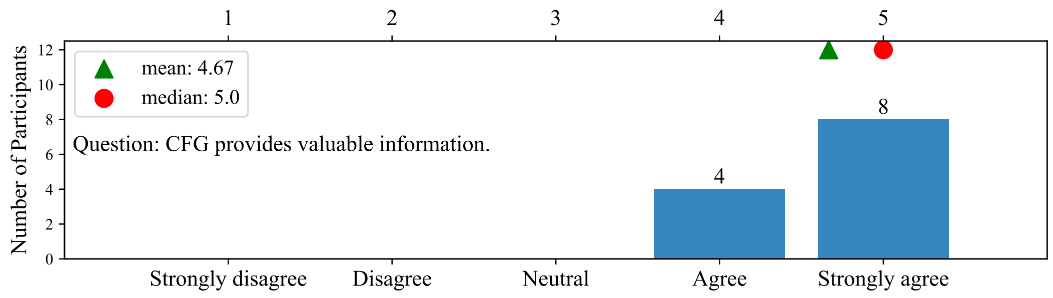


Figure 5.5: Answer distribution of question about the user-guided test amplification's valuable information.

understand the code with the CFG and usually do not need to read the source code. They think the code structure information is especially useful when the method's complexity is high and some of the information tends to be neglected without the CFG, such as some branches.

Participants consider code coverage shown in the CFG essential. The branch coverage tells them which branches are covered and not and helps them learn about the tests. Some participants also notice the importance of line coverage. Sometimes a branch is covered, but the lines that follow it are not all covered, and people usually overlook this kind of situation while line coverage reminds them of this case. Figure 5.6 shows an example of this kind of situation. The stream's size equals its capacity, so it covers the branch "106: False", but its capacity is zero, so there are no items in it, and only part of the following lines are executed.

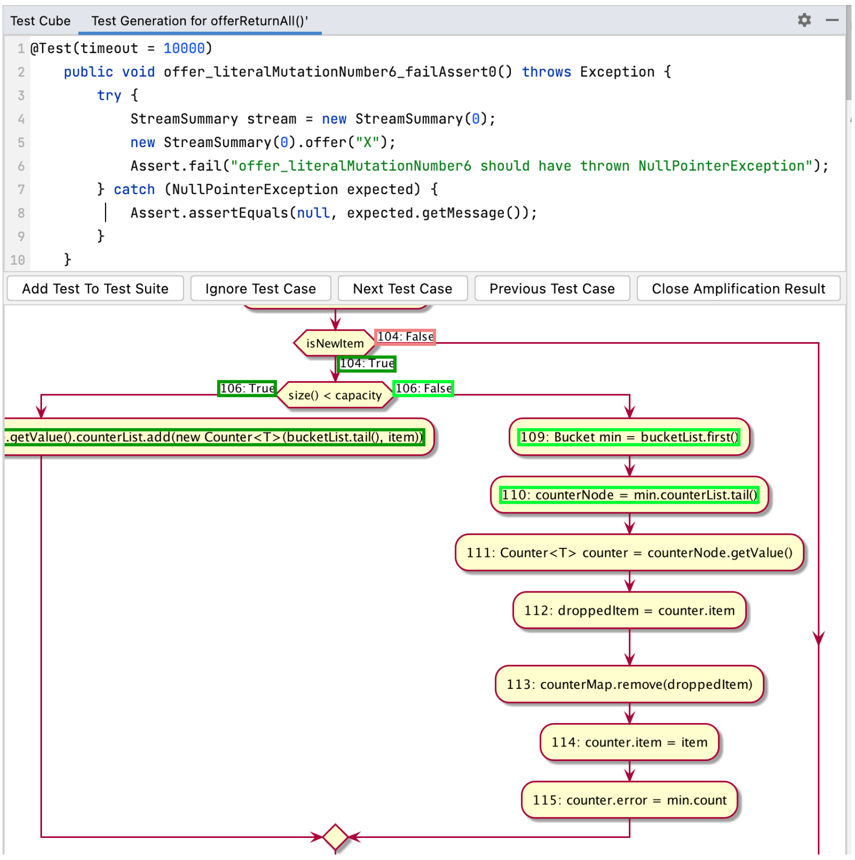
The code structure information in the CFG helps developers understand the code coverage. The CFG gives developers an understanding of the method in an abstract way. Just with the CFG, they can see the flow of the method and understand better with different directions. And then it makes it easier to see what is covered and what is not and understand what kind of situations are covered.

Some participants think the CFG is also helpful in some other situations, such as debugging and when they do not understand code written by others.

5.2.2.3 Effectiveness in Conveying Expectation

The third question in Appendix C.3.1 asks if the user-guided test amplification interaction effectively assists developers in conveying their expectations for the new test cases. The distribution of the participants' answers is shown in Figure 5.7.

All participants think that the user-guided test amplification interaction can effectively assist them in conveying their expectations for amplified test cases. First, the CFG helps them to identify all the possible scenarios. Developers can follow the flow in the CFG and see what the response is, and consider if they want to test the scenario. Sometimes they do not need to cover all branches because some branches are not important to them. Then the current coverage tells developers which branches or lines are already covered. So they can see which ones are not covered and focus on those and make a selection.



```
@Test(timeout = 10000)
    public void offer_literalMutationNumber6_failAssert0() throws Exception {
        try {
            StreamSummary stream = new StreamSummary(0);
            new StreamSummary(0).offer("X");
            Assert.fail("offer_literalMutationNumber6 should have thrown NullPointerException");
        } catch (NullPointerException expected) {
            Assert.assertEquals(null, expected.getMessage());
        }
    }
```

Test Cube
Test Generation for offerReturnAll()'
Add Test To Test Suite
Ignore Test Case
Next Test Case
Previous Test Case
Close Amplification Result
isNewItem
104: False
104: True
106: True
size() < capacity
106: False
.getValue().counterList.add(new Counter<T>(bucketList.tail(), item))
109: Bucket min = bucketList.first()
110: counterNode = min.counterList.tail()
111: Counter<T> counter = counterNode.getValue()
112: droppedItem = counter.item
113: counterMap.remove(droppedItem)
114: counter.item = item
115: counter.error = min.count

Figure 5.6: Example amplified test case that covers partial lines following a covered branch.

5.2.2.4 Usefulness in Result Selection

The fifth question in Appendix C.3.1 asks if the CFG shown in the result window helps participants select the amplified test cases. The distribution of the participants' answers is shown in Figure 5.8.

Ten participants agreed that the coverage is helpful when selecting test cases to add. They know how many branches of the method are already covered, and more branches would be covered if they added the test. The coverage is especially useful when the new coverage brought by the amplified test cases is diverse. The result may offer a surprising coverage besides the target branch selected before when there are multi-level nested branches. Then developers can make a choice based on the different new coverage. One example is shown is Figure 3.9.

Two of the participants express neutrality. They think they do not need to observe the new coverage if they only want to cover the target branch they selected and do not consider other information like the modifications. The result must satisfy their requirement, and they just need to add it to the test suite directly. They even do not need to make a choice. Also, when all the amplified test cases have the same new coverage, the coverage would not be

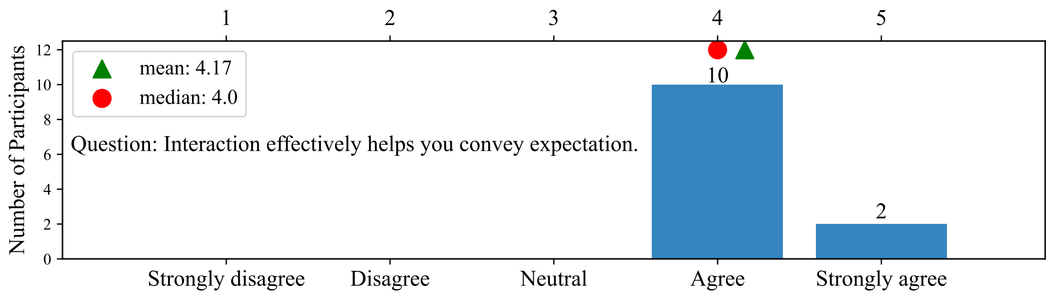


Figure 5.7: Answer distribution of question about the user-guided test amplification's effectiveness in conveying expectation.

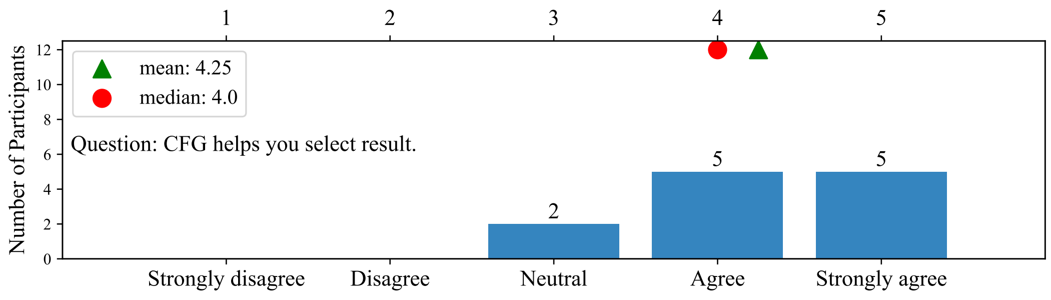


Figure 5.8: Answer distribution of question about the user-guided test amplification's usefulness in result selection.

helpful for them to choose among them.

5.2.3 Open Test Amplification

The questions to survey participants' perceptions of the open test amplification are in Appendix C.3.2, which are mainly about three aspects of the open test amplification, its understandability, valuable information it provides, and its usefulness in result selection.

5.2.3.1 Understandability

We use the first and the third questions in Appendix C.3.2 to investigate the understandability of all the content in the open test amplification result window. Specifically, the first question asks developers if the coverage display is easy to understand, and the third question asks if the overall result window is easy to understand. The distribution of the participants' answers is shown in Figure 5.9.

Eight participants agree that the information in the result window is easy to understand. They like the coverage list, especially the highlighting in the editor, which helps them understand the effect of the amplified test cases. Other participants expressed neutrality or did not think some of the information was easy to understand.

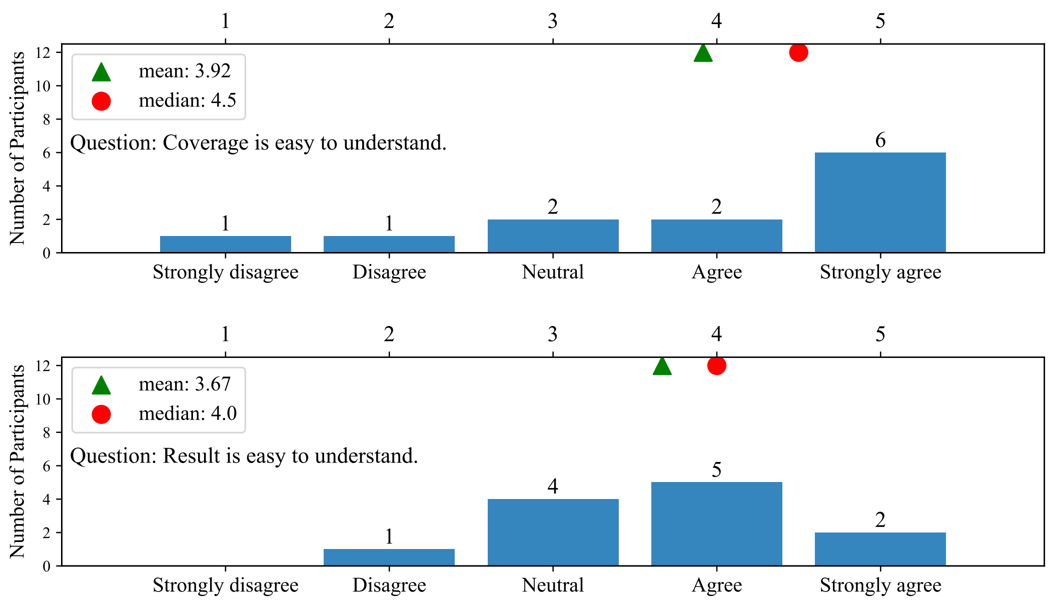


Figure 5.9: Answer distribution of questions about the open test amplification's understandability.

There are three major topics mentioned by participants.

- **Instruction Coverage**

  A common complaint mentioned by the participants is **the number of additional instructions.** They think it is too detailed and not easy to understand.

  The eight participants that still selected agree or strongly agree did think that the absolute numbers of instructions were not informative. Nevertheless, they think the class, method, and corresponding lines of the coverage are enough for them to understand the coverage. They tend to understand the coverage by reading the highlighted code and think the hyperlinks help them locate the code conveniently, and they do not need to look for it by themselves.

  The other four participants were more critical of the instruction coverage. They think the **connection between the instructions and the code** is not straightforward even with the hyperlinks. They still need to click each line to read the code. Then they still do not know what instructions they are, even though they know the corresponding code. The instruction coverage is not clear enough to help them determine whether the branch or path is covered if they care about a branch or path, especially when there are nested branches.

- **Scope of Code Coverage**

  The open test amplification displays the additional coverage for the whole project, causing two problems.

First, the instruction coverage comes **from different classes** making it complicated to understand. It is easy to understand only if they understand the code very well. However, if they do not, it can be difficult because then they have to go through the code to find the different methods called in the code to understand what is going on. Besides, the participants do not know what the test case is really testing as the coverage comes from different classes, which is an obstacle to maintaining and developing the test suite in the future.

Second, it is possible that the instruction coverage list becomes long when there is **much additional instruction coverage**. Then it is not easy for the participants to get the information they need from a long list when they care about specific coverage, such as a branch or line. Participants are worried that the additional instruction coverage would become too much to understand for a large project.

- **Lack of Existing Coverage**

  Some participants think **displaying only the additional instruction coverage** makes it difficult for participants to understand the amplified test case. Participants believe the extended coverage is so scattered without the existing coverage that it is difficult to understand the relation between the highlighted code and why the amplified test case covers the highlighted code.

5.2.3.2 Valuable Information

The second question in Appendix C.3.2 asks if developers find valuable information in the result window. The distribution of the participants' answers is shown in Figure 5.10.

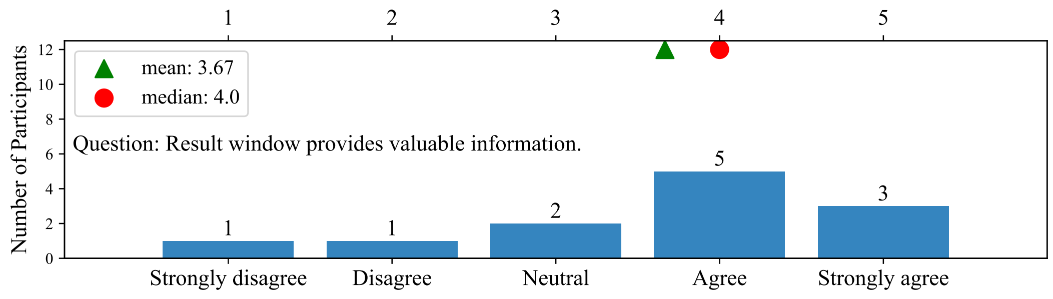


Figure 5.10: Answer distribution of question about the open test amplification's valuable information.

Eight participants agree that the result window provides them with valuable information, including the **additional instruction coverage and the modification information**. The additional instruction coverage tells them what is being newly covered by the amplified test case. The modification information helps them understand the amplified test cases by connecting them with the original test case. They have to find the difference between the amplified test case and the original one by themselves without the modification information.

The other four participants questioned the value of the information provided. For the modification information, some participants think **the number of modifications** is not very valuable as it does not specify the modification in detail. Others think they do not care about the modifications and read the amplified test cases directly to understand them. For the instruction coverage, some participants think it may not be able to offer meaningful information to them if they care about some specific code to have covered. One participant thinks the instruction coverage **is informative but possibly not important to her**, “it is like the shop attendant tells you that they have lots of pretty clothes, but does not say which one would suit you.”

5.2.3.3 Usefulness in Result Selection

The fourth and fifth questions in Appendix C.3.2 ask if the modification information and the additional coverage in the result window help participants select the amplified test cases. The distribution of the participants’ answers is shown in Figure 5.11.

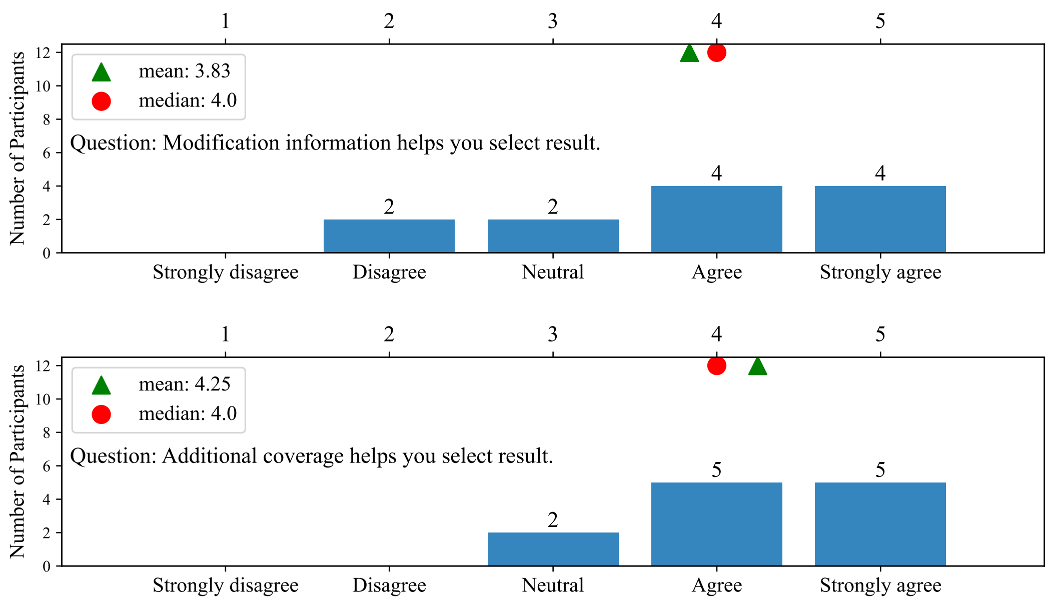


Figure 5.11: Answer distribution of questions about the open test amplification’s usefulness in result selection.

- **Modifications Information**

  Eight participants agree that the modification information helped them select the amplified test cases. Some participants think the modifications help them understand the effect of the amplified test cases by **assisting them in checking what has been changed**. Then they know how the impact is supposed to vary based on that. Others think it is **helpful when they are picky with the test cases** and not only care about the effect of the amplified test cases. They may have some requirements for the new test case in terms of code statement and would observe the modifications to determine if the amplified one is satisfying.

Other participants think they do not consider the modifications when selecting results. Some believe modifications are unimportant and **only care about the effect** of the test cases. Some think the **modifications are too minor**, so there is no need to introduce them or consider them. Some participants shared their opinions by **connecting with the usage scenarios**: they would use open test amplification when they want to use it to generate test cases that can cover as much code as possible. Then they would add all the results directly to reach high coverage and would not consider the modifications.

- **Additional Coverage**

  Ten participants think the additional instruction coverage is helpful for them to make the selection, and two participants expressed neutrality. The reason why it is helpful is that it explains the effect of the amplified test case. If the developers hope to cover as much as possible, they can select the amplified test cases with a lot of additional coverage.

  A common concern mentioned by participants that chose agree and neutrality was that the coverage only contains additional coverage without existing coverage. They think it is also important to know the initial coverage. They are **not sure if it is good enough to add the amplified test case with only knowing the additional coverage**. For example, if they hope to cover a specific line, the additional coverage information helps only when the line is newly covered. They do not know the previous coverage, so it is difficult to determine if the line is already covered when it is not in the additional coverage.

  One participant also reflected on the idea that test cases bringing more coverage are good. She thinks the test cases with more coverage and highlighting look good, but she is unsure if the coverage is really important. The extensive range of highlighting may mislead the developers. **Bringing much coverage does not definitely mean it is a good test case.**

5.2.4 Comparison of Two Test Amplification Methods

We ask the participants to compare the differences between the user-guided and open test amplification, and the related questions are in Appendix C.3.3.

5.2.4.1 Branch Coverage vs. Instruction Coverage

The first question in Appendix C.3.3 asks the participants which type of code coverage is easier to understand. The distribution of the participants' answers is shown in Figure 5.12.

All participants think branch and line coverage is easier to understand than instruction coverage.

First, branch and line coverage are known by all the participants, while some participants indicated that it was their first exposure to instructional coverage. Secondly, the

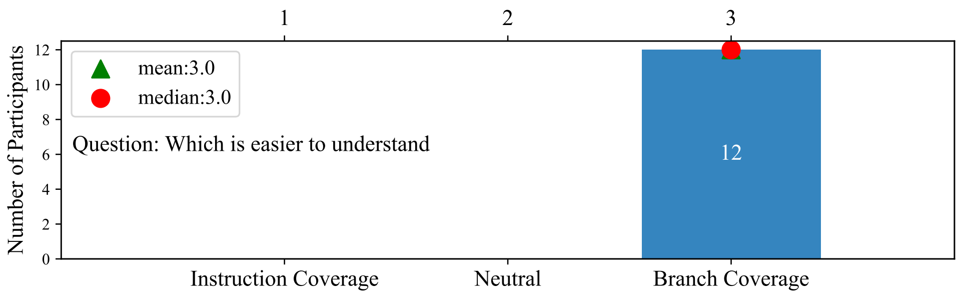


Figure 5.12: Answer distribution of question about understandability of different code coverage.

branch and line coverage can be mapped to the source code easily. Participants think they only consider the line where the instructions are by checking the highlighting in the source code when inspecting instruction coverage and do not understand what the instructions are. They said they needed to know the code sufficiently to understand the instructions. In addition to the understandability, some participants also mentioned the importance. They think branch coverage is more important than instruction coverage when considering a test suite's code coverage.

5.2.4.2 Control Flow Graph vs. Text

The second question in Appendix C.3.3 asks the participants which representation of code coverage is easier to understand. The distribution of the participants' answers is shown in Figure 5.13.

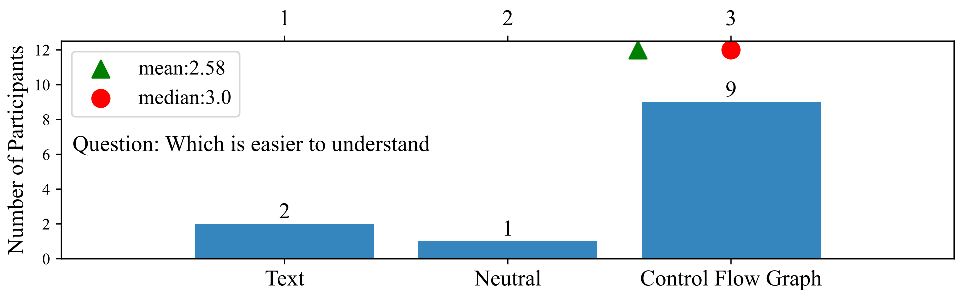


Figure 5.13: Answer distribution of question about understandability of different display form of code coverage.

Nine participants think the coverage shown in the CFG is easier to understand. The most important reason is that they all like visualization, making the coverage clear and straightforward. They do not need to read the source code in the editor, and the flow of the code in the CFG also helps them understand the coverage. For the coverage listed and highlighted in the text, they need to read and understand the code first, and then they can have a good understanding of the coverage.

Only one participant thinks he likes the coverage in the text because he likes looking at the code instead of the graph.

Two participants expressed neutrality. One said she likes both and it is better to combine the two formats; the other said his favorite coverage display form is a percentage, so the CFG and text are the same.

5.2.4.3 User-Guided Test Amplification vs. Open Test Amplification

The third and fourth questions in Appendix C.3.3 ask the participants which type of test amplification is more helpful for them in result selection and test generation. The distribution of the participants' answers is shown in Figure 5.14.

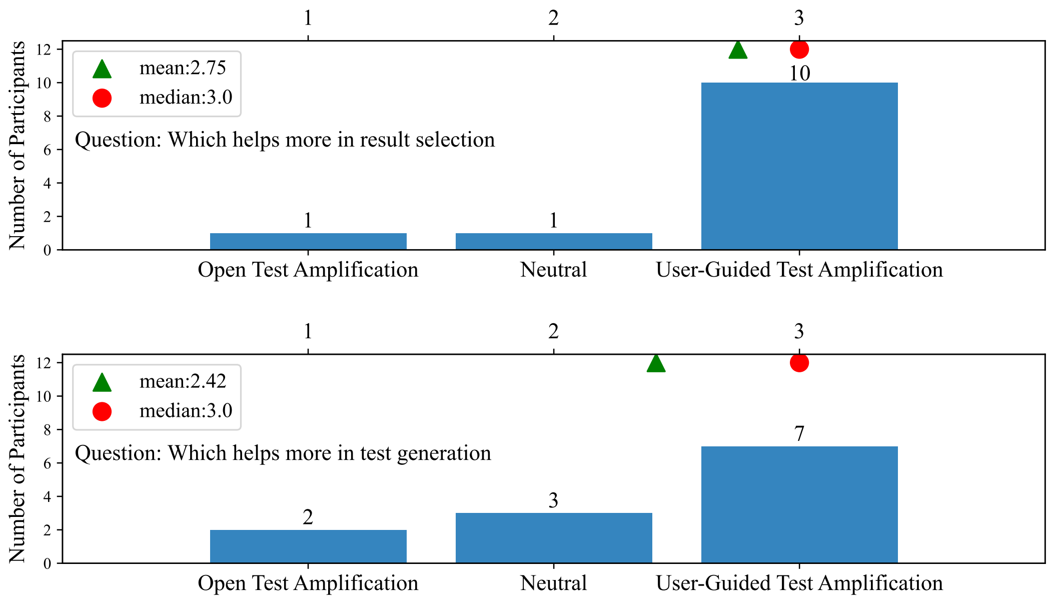


Figure 5.14: Answer distribution of questions about helpfulness of different test amplification.

- **Result Selection**

  Ten participants think directed test amplification enables them to select the result more easily. When writing tests, they usually have a specific coverage goal and consider what kind of coverage the test cases can bring, not only the amount of coverage. Then it is challenging to select the result according to an open coverage as they need to check every class, method, and line to decide if they want to keep the amplified test case. It is easier with a specific goal and checking if the amplified test case realizes the goal.

One participant thinks she prefers open test amplification. She typically hopes to cover the whole project as much as possible. So it is better to have more new coverage and add them based on the coverage amount.

One participant thinks selecting test cases depends on the usage scenario instead of the result. It is easy to select the result with both methods. He would add all of the results provided by TestCube no matter which method he used.

- **Test Generation**

  Seven participants prefer directed test amplification. They expressed their appreciation of the idea of user-guided test amplification because it is closer to the perspective of writing tests in real-life scenarios. They usually focus on specific features and write tests for them, while open test amplification is like looking around without any goal and generating test cases for the sake of generating test cases.

  Two participants prefer open test amplification. One participant thinks she would use open test amplification first to cover as much as possible and then use directed test amplification to refine the test suite. So he thinks the open test amplification has more application scenarios and is more helpful. The other participant thinks she is actually doing test amplification instead of generating test cases. She likes to connect the new test cases with the existing test cases. This connection is more explicit in open test amplification as it starts from selecting an existing test method and then amplifies it.

  Three participants expressed neutrality and would like to combine the two types of methods. They would use open test amplification when they do not have a coverage goal and want much coverage. If they have a specific coverage goal, they will use directed test amplification. It also depends on how much control they want over the test cases. If they want to have less control, the open test application is good enough to provide more additional coverage. If they want more control over each test case's coverage, then directed amplification is better.

5.2.5 Overall Impression and Suggestions

We investigate the participants' opinions on the test amplification result and TestCube. The related questions are in Appendix C.3.4. We also encourage them to put forward further suggestions for improving TestCube.

5.2.5.1 Satisfaction with TestCube

We ask developers if they are satisfied with the amplified test cases TestCube generates and if they want to use TestCube in the future. The distribution of the participants' answers is shown in Figure 5.15.

- **Amplified Test Case**

  All participants think they are satisfied with the amplified test cases provided by TestCube. The amplified test cases are simple and easy to understand. They can know

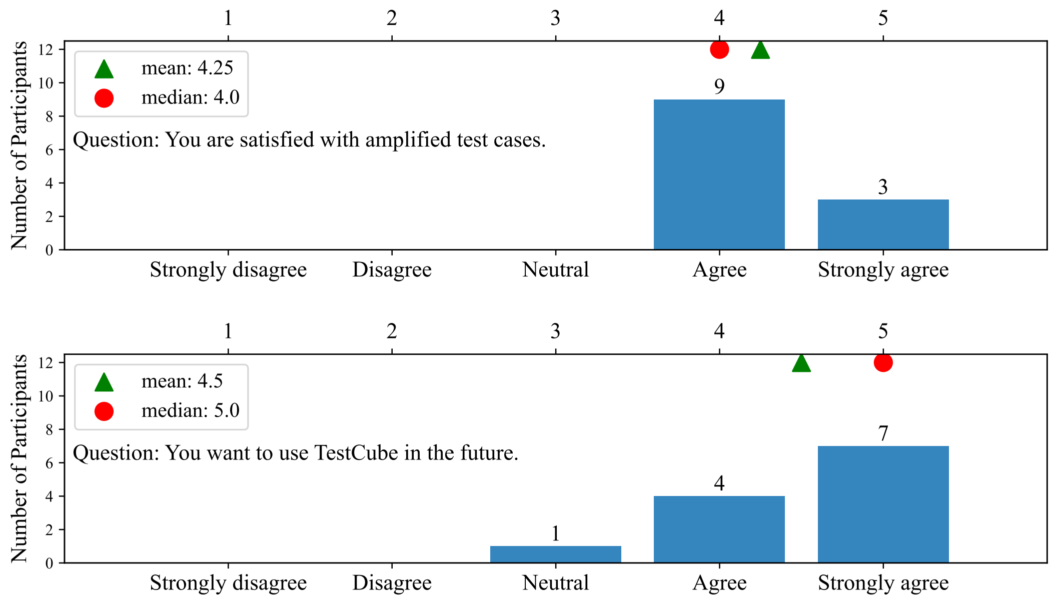


Figure 5.15: Answer distribution of questions about satisfaction with TestCube.

the effect the amplified test cases can bring to the test suite by inspecting the code coverage brought by them and edit them. Clear presentation of results is essential to their satisfaction with the amplified test cases.

- **TestCube**

  Eleven participants think they want to use TestCube in the future. They think TestCube can help them write simple test cases for simple test scenarios. It can reduce much mechanical and repetitive work and save time. Besides, they can also see what is happening, so they are not blinded when generating tests with TestCube.

  One participant expressed neutrality because he enjoys writing tests manually and does not use Java, even though he was surprised by the good quality of amplified test cases.

5.2.5.2 Suggestions

Developers put forward many suggestions for TestCube during the experiment. They are mainly about the display format of code coverage and amplification result and the user-guided test amplification interaction.

- **Code Coverage Display**

  S1 Developers suggest that there should be an explanation for the information provided. The text “instr” in the open test amplification result window should be replaced by "instructions." The meaning of different colors used to identify different coverage in the CFG should be explained.

S2 Although developers think the coverage shown in the CFG is good enough to understand, some still believe it is good to combine CFG with hyperlinks and coverage highlighted in the editor. It provides convenience when they sometimes want to read the code in the editor.

S3 Some developers are interested in other coverage metrics and suggest we add them. They think there should be an overall introduction about the coverage brought by the amplified test cases, such as the total number of additional coverage and the code coverage in percentage form.

S4 Some developers think it would be better to provide some preference settings. For example, they want the autonomy to choose different color themes for the code coverage highlighting. Some of them hope to have the option not to show some details when they think they are not necessary.

- **Amplification Result Display**

S5 Some developers hope we can return the result in a specific order that they can determine, such as the new coverage amount.

S6 Several developers think the distribution of the information panel in the result window should be organized well for the open test amplification result window. The amplified test cases should not separate the information box and the editor with code coverage highlighting.

S7 Some developers think highlighting code coverage in the regular editor is better than a new editor in the tool window.

- **Test Amplification Interaction**

S8 One participant thinks starting the test amplification from the code in the editor would be a pretty straightforward way to generate test cases. It provides another option for users in addition to starting from the CFG.

S9 Several participants think supporting to select multiple branches can help developers convey their branch coverage expectations better.

S10 One participant thinks it would be better if we only make the branches that we can eventually cover by amplified test cases available to select.

5.3 Analysis and Discussion

5.3.1 RQ1: Does the directed test amplification method generate test cases that satisfy the developers' branch coverage expectations?

We discuss this research question based on the technical case study result and the user study result.

5.3.1.1 Technical Perspective

The result in Table 5.3 shows that 32% and 41% of the sampled branches are successfully covered for Javapoet and Stream-lib by the directed test amplification method implemented with the D-amplifier in DSpot. The result means that using one initial test case and one random branch as input, the directed test amplification method has a 30% to 40% chance of generating an amplified test that can cover the given branch. Since covering a given branch is the developers' expectation, the directed test amplification can generate test cases that satisfy developers' expectations in 30% to 40% of cases.

To find out why more than half of the branches cannot be covered by the amplified test cases generated, we checked the test amplification process of 80% of the uncovered branches. We found that the core reason we cannot cover a branch is that the objects under test and target methods' parameters are not initialized with the right parameter values to fulfill the target branch condition. Sometimes DSpot cannot call the target method because DSpot does not support the parameters' type, which is often a client class without public constructors. Sometimes it does generate the parameters and calls the target method. However, the parameters it generates are null or empty and often lead to breaking down when it comes to assertion generation. Even though DSpot generates non-empty parameters and calls the target method, the object under test often does not have the right field value to meet the target branch.

We also analyzed the initial test case's influence on the technical case study result. We only amplify one existing test method to generate new test cases in our experiment. We are curious if amplifying more test methods would cover the branch uncovered in the case study. Therefore, we sampled five branches covered neither by the amplified test cases obtained in the case study nor the existing test suit for each project to explore the influence of the initial test case. For each of the sampled ten branches, we amplify all the existing methods instead of only the first test method in the corresponding test class with the directed test amplification approach. The result shows that the directed test amplification still cannot generate amplified test cases that can cover the ten sampled branches. We found that using more and different initial test cases still cannot solve the problem of not being able to generate proper parameter values for the target branch because these parameters are often objects' fields and complex to initialize and mutate. Not being able to generate proper parameter values is the core obstacle to obtaining satisfying amplified test cases.

Taking the above analysis together, we further analyze the reason for the different technical case study results of the two projects, Javapoet and Stream-lib. We found that Javapoet's source code has more methods whose parameters are classes without public constructors, which DSpot does not support. Conversely, Stream-lib has relatively more methods whose parameters are simple data types, such as the integer. Besides, most Stream-lib classes have public constructors, so DSpot supports generating them. The differences between the two projects make generating proper parameter values for target branches in Stream-lib easier than in Javapoet. Therefore, more branches are covered in Stream-lib. It also indicates that the simpler and more flexible the methods' parameters in a project, the better the performance of the directed test amplification method.

5.3.1.2 Developers' Perspective

According to the result in Section 5.2.5.1, all 12 participants are satisfied with the amplified test cases generated with TestCube. It indicates that the directed test amplification method can generate test cases that satisfy the developers' expectations from the developers' perspective.

During the user study, developers are surprised and happily add them to the test suite when they can get new test cases that precisely cover the branch they hope to be covered. There is also the case when developers select one branch and start test amplification but do not get amplified test cases that can cover the selected branch. However, developers do not think it is a big issue. They think that the result of not always meeting expectations is acceptable as long as the tool reports the result clearly. It indicates that presenting a clear result and providing feedback on developers' expectations is also crucial to making the amplification result satisfying.

RQ1: Does the directed test amplification method generate test cases that satisfy the developers' branch coverage expectations?

The directed test amplification can generate test cases that satisfy developers' branch coverage expectations in 30% to 40% of cases. Although developers cannot always obtain the expected result, they are satisfied with the amplified test cases generated by the directed test amplification method.

5.3.2 RQ2: Does the directed test amplification method generate more test cases that fulfill developers' branch coverage expectations than the open test amplification method?

According to Table 5.3, the ratio of covered branches increases from 23% to 32% and from 35% to 41% after adding the D-amplifier. Moreover, according to Table 5.4, for the covered branches, the average ratio of satisfying test cases has increased significantly from 24% to 70% and from 45% to 70% for Javapoet and Stream-lib. The result indicates that the directed test amplification has a higher probability of generating test cases that fulfill developers' expectations than the open test amplification method.

The advantage of the directed test amplification comes from the D-amplifier. The D-amplifier calls the target method before amplifying the initial test case with all the other amplifiers. However, the open test amplification method amplifies the initial test case without direction; therefore has a more negligible probability of calling the target method. As a result, only a tiny part of amplified test cases call the target method in the open test amplification method's amplification result. In contrast, most amplified test cases call the target method in the directed test amplification method's result, which accounts for the gaps between the two methods' ratio of satisfying test cases. As we limit the number of amplified test cases to 200, the directed amplification method has a higher probability of keeping amplified test cases containing the target method than the open test amplification, which accounts for the differences between their ratio of covered branches.

Comparing the two projects, Javapoet and Stream-lib, we can see that the improvement in generating test cases that fulfill developers' expectations brought by the D-amplifier is more significant for Javapoet than Stream-lib. By analyzing the two projects, we found that the number of methods in Javapoet's classes is higher than that in Stream-lib. Therefore, the probability of calling the target method by the open test amplification method is smaller for Javapoet than Stream-lib, and the D-amplifier's effect is more significant for Javapoet.

As the directed test amplification increases the ratio of satisfying test cases, it can obtain satisfying test cases by generating fewer amplified test cases than the open test amplification. Then the directed test amplification method will generate assertions and calculate code coverage for fewer amplified test cases. It helps reduce test amplification execution time and makes it more efficient. The directed test amplification's efficiency improvement is particularly significant for projects with many methods in a single class.

RQ2: Does the directed test amplification method generate more test cases that fulfill developers' branch coverage expectations than the open test amplification method?

The directed test amplification method improves the probability of generating tests that meet developers' branch coverage expectations by 9% and 6% for the two example projects. Besides, the directed test amplification method increases the proportion of amplified tests that meet requirements among all amplified test cases from 24% and 45% to 70%, making it possible to reduce the total number of amplified test cases needed to get the amplified test cases covering a given branch.

5.3.3 RQ3: How do developers perceive the user-guided test amplification?

We summarize three main impressions that participants have of the user-guided test amplification by analyzing the result in Section 5.2.2 and Section 5.2.5.1.

5.3.3.1 Easy to Understand

According to the result in Section 5.2.2.1, almost all participants found the content in the user-guided test amplification interaction is easy to understand, including the CFG, the coverage shown in the CFG, and the amplified test case. The CFG and branch coverage helps the developers to understand the amplified test cases.

5.3.3.2 Provide Valuable Information

User-guided test amplification interaction provides developers with valuable information by CFG according to the result in Section 5.2.2.2. First, the CFG contains much information about the code structure, which helps them understand both the source code and the coverage of test cases. Second, the coverage shown in the CFG is also valuable. The new coverage indicates the impact of the amplified test cases. The initial coverage is also essential information for developers to understand the status of the test suite and determine what needs further testing.

5.3.3.3 Easy to Use

The user-guided test amplification is easy to use as it provides practical assistance in conveying developers' expectations and result selection according to the result in Section 5.2.2.3 and Section 5.2.2.4.

The user-guided test amplification effectively assists developers in conveying their expectations for amplified test cases, where the CFG and coverage display play a crucial role. The CFG helps developers identify the possible scenarios that need to be tested, and the coverage helps them focus on uncovered ones. Developers are satisfied with selecting one branch each time, although they also expressed a desire for TestCube to offer a more flexible branch selection.

The user-guided test amplification effectively assists developers in result selection with the CFG and coverage, especially when the amplified test cases have different coverage. The coverage in the CFG indicates the new coverage the amplified test cases can bring to the test suite, which is a critical factor for developers to consider when making their choices. It is pretty useful when the amplified test cases can cover other parts of the target method besides the target branch, and developers need to inspect them and make a choice.

5.3.3.4 Satisfactory Test Amplification Method

According to the result in Section 5.2.2.4, almost all participants are satisfied with the amplified test cases generated with TestCube and want to use TestCube in the future. TestCube is especially useful for people who do not like to write tests. It can help them save the time of writing simple repetitive tests and provide valuable information about the new tests to help them understand the effect and further development.

RQ3: How do developers perceive the user-guided test amplification?

In summary, the developers think the user-guided test amplification is easy to understand, provides valuable information, and is easy to use. The CFG contains code structure and presents code coverage, which is valuable information and easy to understand. The user-guided test amplification effectively assists developers in conveying their branch coverage expectations for amplified test cases and result selection. The user-guided test amplification can generate satisfying amplified test cases and help developers write tests well.

5.3.4 RQ4: What different value do the user-guided test amplification and the open test amplification bring to developers?

We analyze the value of the open test amplification based on the result in Section 5.2.3 and compare it with the user-guided test amplification. Besides, we use the result in Section 5.2.4 to compare the value of different aspects of the two test amplification methods.

5.3.4.1 Different Way of Test Amplification

Open test amplification follows the logic of traditional test amplification, starting from selecting an existing test method and then amplifying it. The explicit connection between the amplified test cases and the original test case makes the test amplification result easy to understand. The goal of increasing instruction coverage for the whole project displays the effect of amplified test cases and provides open results to choose from.

User-guided test amplification also uses test amplification but changes the starting point of the test amplification from the test method to the method that needs testing. It is a new perspective of test amplification and closer to the perspective of writing tests in real-life scenarios. It also assists developers in conveying their expectations for the new test cases, giving test amplification a defined goal rather than open exploration.

User-guided test amplification facilitates developers having a clear objective for the new test. They can obtain what they want by expressing their requirement and having more control over the generated test cases. The fact that each new test has an apparent effect also provides a sound basis for the maintenance and further development of the test suite.

In addition, the user-guided test amplification solves the issues about the additional instruction coverage in the result selection part of the open test amplification in Section 5.2.3.3. The coverage goal of covering a specific branch focuses on the meaning of the coverage, preventing developers from being misled by the amount of coverage. The coverage of results obtained based on user requirements also solves users having difficulty finding their target of interest in open coverage.

The user-guided test amplification is a good complement to TestCube's existing interaction. More than half of the participants think they prefer the directed interaction to the open interaction. The remaining participants also believe they would use the two methods for different situations and purposes. The open test amplification allows them to cover more code, while the directed one is more convenient when they have a specific coverage goal. Combing the two methods is always better than only using one type of interaction.

5.3.4.2 Different Valuable Information

The valuable information that open test amplification provides is amplified test cases' modification information and additional instruction coverage. The modification information helps developers understand the change of the amplified test cases compared with the original test case. The additional instruction coverage displays the effect of the amplified test cases. Most developers think this information is easy to understand and valuable, helping them understand the amplification result and select it.

According to the result, the number of participants agree that user-guided test amplification provides valuable information and is easy to understand is more than that of open test amplification. Also, most participants think the user-guided test amplification helps them more in result selection. The valuable information that the user-guided test amplification provides to contribute to the result is the CFG and visualized code coverage.

- **Control Flow Graph**

The user-guided test amplification visualizes the method developers want to test by the CFG. Most participants prefer the visualization of code to plain text because it is more intuitive and easier to understand. Therefore, the CFG provides them with a better way to understand the code structure. It is useful not only when generating tests but also whenever they want to understand the code structure.

- **Code Coverage**

  The user-guided test amplification shows different code coverage and visualizes the code coverage in the CFG. The different types of coverage provide new valuable information missing in the open test amplification. The CFG offers a more intuitive way of understanding the code coverage. There are three key points that avoid the problems the participants mentioned in the open test amplification, make the amplification result easier to understand, and assist developers in result selection.

  - Use Branch/Line Coverage

    Developers think the instruction coverage is difficult to understand and usually only check the highlighted code to understand the coverage. Some developers think they still do not know if a branch or path is covered with the instruction coverage. The user-guided test amplification interaction uses line coverage and branch coverage directly to provide the information developers need explicitly. Also, instead of clicking the hyperlinks one by one to check the source code, the CFG displays the source code directly to make it more convenient to read.

  - Limit Scope of Code Coverage

    The open test amplification presents the additional instruction coverage for the whole project. The user-guided test amplification only displays the coverage of the method that the developers want to test and focuses on the new branch and line coverage required by developers.

    The user-guided test amplification limits the scope of the code coverage so that there is not too much covered code needing to be displayed in the result. It avoids overloading with information and makes the content easy to understand. More importantly, the developer's requirements determine the scope of code coverage. It solves the problem that developers think it is not easy to find critical information from the open coverage list. It makes it easier for developers to select the amplified test cases as it focuses on developers' coverage goals and excludes the information that developers are not interested in.

  - Display Existing Coverage

    The open test amplification only shows the additional instruction coverage of the amplified test cases. The user-guided test amplification shows the new coverage brought by the amplified test cases and the existing coverage, which helps developers understand the coverage. It also allows developers to learn about the coverage status and ensure that the amplified test case is good enough to add to the test suite.

RQ4: What different value do the user-guided test amplification and the open test amplification bring to developers?

The user-guided test amplification brings a new perspective of test amplification, starting test amplification from the method developers want to test, and assisting developers in conveying their branch coverage expectations for the new test cases. The user-guided test amplification is more convenient when developers have a specific coverage goal, while the open test amplification allows them to cover more code. Besides, the user-guided test amplification uses the CFG and directed branch coverage to help developers understand the amplification result and select results better.

5.3.5 RQ5: What are the key facets to creating an effective interface for user-guided test amplification?

We summarized some principles for creating an effective user-guided test amplification interface by analyzing the evaluation result of the user-guided test amplification prototype we designed and the suggestions provided by developers in Section 5.2.5.2.

- **Make the content in the interface clear and easy to understand for all kinds of users.**

  We provide the CFG and present the branch coverage to guarantee the understandability of the content during the interaction. The result shows that making the content easy to understand helps developers interact better with the interface and get satisfying results. Several developers further suggest adding some explanation for the content as they are unfamiliar with software testing tools (S1 in Section 5.2.5.2).

  We should consider all types of users, including both new users of software testing tools and experienced developers, and provide valuable information with a clear introduction to help them get started quickly and generate test cases efficiently.

- **Consider preferences of different developers.**

  The user-guided test amplification's core goal is to assist developers in test amplification based on their preferences for the new test cases. We allow developers to convey their preferences by selecting one method and branch they want to test. Developers express their preferences and requirements for the user-guided test amplification interface (S2, S3, S4, S5, S9 in Section 5.2.5.2), such as supporting different color themes for the code coverage display. We should consider the diversity of developers' preferences and expectations to help them generate test cases they want and provide better information presentation.

- **Make every user action count.**

  We design a series of notification windows in our prototype to respond to users' every click and save their time understanding what is happening. We also keep the number of amplified test cases small to save the developer's time. It allows developers to interact with the interface efficiently.

Some developers also put forward suggestions that can make the interaction more efficient (S5, S9, S10 in Section 5.2.5.2), such as supporting selecting multiple branches when conveying coverage expectations.

An important point to improve the efficiency of developers using the tool is to maximize the value of each user action and provide positive feedback to the user for each action. This saves the users' time and motivates them to use the tool through positive feedback.

- **Use the interface features developers are familiar with.**

  The user-guided test amplification starts from the ordinary editor developers work with. It uses standardized flow graph elements and typical colors that other code coverage tools use to display the CFG and code coverage. These features allow developers to understand the content easily and get used to TestCube quickly. Developers also suggested reusing the features they are familiar with (S7, S8 in Section 5.2.5.2), such as conveying their expectations in the editor based on the source code instead of the CFG. It indicates that using elements that developers are familiar with to create the interface can make it easier to use.

RQ5: What are the key facets to creating an effective interface for user-guided test amplification?

First, we should make the content in the interface clear and easy to understand for all kinds of users, as understanding the content is essential for users to think about their expectations for test cases and select the result. Second, we should consider all types of users' preferences and cater to different preferences for test cases and information display. Third, we should maximize each user action's value and provide positive feedback for each action to improve the efficiency of conveying users' expectations and generating test cases. Last but not least, we should use the interface features developers are familiar with to integrate the test amplification process into developers' daily development environment and make the interface easy to use.

5.4 Threats to Validity

This section discusses some threats to the validity of the technical case study result and the user study result.

5.4.1 Technical Case Study

5.4.1.1 Internal Validity

The technical case study evaluates the performance of the directed test amplification methods in generating new test cases to cover a given branch. When conducting test amplification, we limit the number of amplified test cases after input mutation and after assertion generation to 200, which can threaten the internal validity. However, the core factor determining if the directed test amplification method can generate an amplified test case that can cover a given branch is whether it can generate right parameter values to fulfill the target branch. The number of amplified test cases has little influence on the result.

The technical case study also compares two test amplification methods' performance. Except for using two different test amplification methods, the experiment is conducted on the same computer with the same DSpot configuration and input data. Therefore, only one factor, the test amplification method, influences the final result. We avoided threats to the internal validity of the comparison result by keeping all the experimental settings the same except the test amplification method.

5.4.1.2 Construct Validity

The directed test amplification method is implemented by adding one D-amplifier based on Brandt and Zaidman's open test amplification method. One threat to the construct validity is that the result only represents the performance of one type of implementation as the open test amplification could use different amplifiers. The case study result might be different if the open test amplification uses different amplifiers, which can lead to a different conclusion.

5.4.1.3 External Validity

Since it is a case study, the result we obtained only works for the projects we selected to construct the dataset. The two projects we selected provide an evaluation result of the directed test amplification for java projects with many code branches, complex data types, and test classes corresponding to the classes. The data types' complexity in a project significantly influences the effectiveness of the directed test amplification. It might perform better in other projects with more simple data types. The result might differ from our case study result in other projects with fewer branches and without test classes corresponding to the classes.

5.4.2 User Study

5.4.2.1 Confirmability

To ensure that all the user study results are from the participants instead of the interviewer, we keep the results in Section 5.2 closely based on the interviews. The result structure is wholly based on the questionnaire used in the interview. For each part, we first present the participants' answers to all the questions and then describe the reasons that explain their answers to avoid information omission. All the explanations for the answers are from the participants' words, and we keep their words as much as possible.

5.4.2.2 Internal Validity

All the user study interviews were conducted by the same interviewer online. The user tasks were all conducted through remote control, and TestCube was running on the same computer. The interview was finished in one go with no pauses for each participant. To avoid the influence of the order in which the two test amplification methods are tried and the influence of the two exampled classes used, we divided the participants evenly to experience every possible set-up of the user task. One threat to the internal validity is the groups of participants, and we mitigate the threat by dividing the participants randomly.

5.4.2.3 Construct Validity

One threat to the construct validity is that the interview result mainly follows the pre-prepared questionnaires' structure and may not contain all aspects of the test amplification methods we tend to evaluate. We mitigate this threat by encouraging participants to share any opinions in their minds. We also motivate their thinking by asking their opinions about the content in TestCube but do not share any personal opinions of the interviewer. We can see from the results that the participants' views were vibrant and far exceeded the framework of the questionnaire, making the final result comprehensive. Another fact that makes the results comprehensive in evaluating the test amplification methods is that the participants experienced all kinds of situations and results when practicing the test amplification tasks, including when the method they wanted to test was already covered and when they did not get satisfying test cases, etc.

Another threat to the construct validity is that the developers are only trying out a prototype of the user-guided test amplification. Design and implementation flaws affect the developers' experience and evaluation result of the general user-guided test amplification. Furthermore, all the participants in our user study knew nothing about the example project Stream-lib and TestCube before participating in the user study. Developers familiar with the example project and TestCube may spend less time understanding the tool and relevant information, thus influencing their opinions.

5.4.2.4 External Validity

There are two threats to external validity.

The participants influence the generalization of our results to all developers. According to the result in Section 5.2.1, the participants are from diverse industry domains and have experience in the development of different duration. We also have participants with and without experience in using automatic test generation tools, enjoying and not enjoying writing tests, which should contribute to generalizing results. One threat is that no participants have more than ten years of development experience, making the result potentially not generalizable to very senior developers.

The example project Stream-lib and the two example classes used in the user tasks also impact the result. The two projects contain all kinds of methods, including simple and complicated methods, methods with and without branches. The two initial test cases are very simple, composed of three statements. The initial test cases also create all kinds of coverage situations, including methods uncovered, partially covered, and fully covered. We expect the diversity of the example classes to make the result generalizable.

Chapter 6

Conclusions and Future Work

This thesis proposes a novel type of developer-centric test amplification, user-guided test amplification. The user-guided test amplification involves developers in the test amplification process by allowing developers to guide the test amplification to generate new test cases according to their branch coverage expectations. Based on the current design of developer-centric test amplification in TestCube, which we call open test amplification, we implemented the user-guided test amplification interaction in TestCube. Besides, we extended DSpot with a directed test amplification method to support TestCube in the back-end. We explore how user-guided test amplification helps developers generate new test cases by evaluating our prototype. This chapter summarizes the conclusions based on our proposed research questions and contributions. We also put forward some opportunities for future work.

6.1 Conclusions

We proposed five research questions in Chapter 1. We studied the current state-of-the-art, implemented a user-guided test amplification prototype, and conducted a technical case study and a user study. By analyzing all the information obtained during the study, we propose answers to the five research questions.

Research Question 1

Does the directed test amplification method generate test cases that satisfy the developers' branch coverage expectations?

We define the developers' branch coverage expectations for a new test case as covering a specific branch of a method they want to test. We answer this research question from both a technical and a user perspective.

Technically, we selected two Java projects, Javapoet and Stream-lib, and sampled 100 branches for each project to simulate the branch developers want to cover. We conduct the directed test amplification with the sampled branches and evaluate how many branches are covered by the amplified test cases. The result shows that the directed test amplification can generate test cases that satisfy developers' expectations for the two example projects in 30% to 40% of cases.

Besides, we recruited 12 software developers to generate test cases with the directed test amplification method and ask for their opinion on the result. All the developers are satisfied with the amplified test cases they get. We conclude that although developers cannot always obtain the expected result, they are satisfied with the amplified test cases generated by the directed test amplification method.

Research Question 2

Does the directed test amplification method generate more test cases that fulfill developers' branch coverage expectations than the open test amplification method?

We implemented the directed test amplification method in DSpot by adding a directed amplifier, D-amplifier, to amplify the initial test case directly before all the amplifiers used in the open test amplification amplify the test case.

To evaluate the effect of the D-amplifier, we conducted a technical case study with the same dataset used in the case study conducted for answering RQ1. We conducted the directed and open test amplification with the same input and configuration and compared their amplification result.

The result shows that the directed test amplification method improves the probability of generating tests that meet developers' branch coverage expectations by 9% and 6% for the two example projects. Besides, the directed test amplification method increases the proportion of amplified tests that meet requirements among all amplified test cases from 24% and 45% to 70%, making it possible to reduce the total number of amplified test cases needed to get amplified test cases covering a given branch.

Research Question 3

How do developers perceive the user-guided test amplification?

We implemented the user-guided test amplification in TestCube. We visualize the method developers want to test by a Control Flow Graph and display the branch and line coverage in the CFG. Developers convey their branch coverage requirement to the amplified test cases by selecting one uncovered branch in the CFG. Finally, developers inspect the amplified test cases with the CFG showing the new coverage of the amplified test cases.

We conducted a user study to investigate developers' opinions on the user-guided test amplification we designed and implemented in TestCube. We recruited 12 developers and invited them to try the user-guided and open test amplification in TestCube. Then we conducted a semi-structured interview by asking them to fill in a questionnaire and discuss the reason for their answers.

We conclude that the developers think the user-guided test amplification is easy to understand, provides valuable information, and is easy to use by analyzing the answers to the questionnaire and their discussion. The CFG contains code structure and presents code coverage, which is valuable information and easy to understand. The user-guided test amplification effectively assists developers in conveying their branch coverage expectations for amplified test cases and result selection. The user-guided test amplification can generate satisfying amplified test cases and help developers write tests well.

Research Question 4

What different value do the user-guided test amplification and the open test amplification bring to developers?

We also use the user study conducted for RQ3 to explore the new value the user-guided test amplification brings to developers compared with the open test amplification. We compare the two test amplification methods by asking developers to evaluate the open test amplification method and compare the two methods directly.

We summarized the new value of the user-guided test amplification by analyzing the user study result. The user-guided test amplification brings a new perspective of test amplification, starting test amplification from the method developers want to test, and assisting developers in conveying their branch coverage expectations for the new test cases. The user-guided test amplification is more convenient when developers have a specific coverage goal, while the open test amplification allows them to cover more code. Besides, the user-guided test amplification uses the CFG and directed branch coverage to help developers understand the amplification result and select results better.

Research Question 5

What are the key facets to creating an effective interface for user-guided test amplification?

During the user study, we encourage developers to put forward suggestions for TestCube. We discover the critical points for designing an effective interface for user-guided test amplification by analyzing the feedback on our prototype and developers' suggestions.

We concluded four critical points for creating an effective interface for user-guided test amplification. First, we should make the content in the interface clear and easy to understand for all kinds of users, as understanding the content is essential for users to think about their expectations for test cases and select the result. Second, we should consider all types of users' preferences and cater to different preferences for test cases and information display. Third, we should maximize each user action's value and provide positive feedback for each action to improve the efficiency of conveying users' expectations and generating test cases. Last but not least, we should use the interface features developers are familiar with to integrate the test amplification process into developers' daily development environment and make the interface easy to use.

6.2 Contributions

The contributions of this thesis are:

1. Extended TestCube: A new user-guided test amplification interaction in the TestCube plugin, which visualizes code coverage and assists users in conveying their branch coverage expectations for amplified test cases and displays the corresponding result for user selection.

2. New amplifier in DSpot: A directed amplifier that supports directed test amplification that aims at generating new test cases meeting specific branch coverage requirements.

3. New selector in DSpot: A selector that can compute branch and line coverage of tests and select amplified test cases based on coverage requirements.

4. An evaluation of the effect of the new directed amplification method in test amplification aiming at specific branch coverage requirements.

5. Evaluate the value of user-guided test amplification and differences with the open test amplification.

6. Four suggestions for creating an effective user-guided test amplification interface.

6.3 Future work

We can do much work to enhance the value of the user-guided test amplification even though our study shows that our current design is already valuable for software developers. We put forward the following suggestions for future work based on the research.

Empower the Parameter Generation in DSpot. Since the core limitation of our current directed test amplification method stems from DSpot not supporting parameters of complicated data types, we can improve the DSpot implementation to support more powerful parameter generation capabilities.

Find the Optimal Configuration of Test Amplification. Many configurations need to be settled when conducting test amplification with DSpot. We should find the best set of amplifiers and combine them with the D-amplifier to realize the best performance. We also need to find a decent number to limit the number of amplified test cases after input mutation and assertion generation. This number should ensure that we can get an amplified test case that satisfies developers' coverage expectation and minimize the time spent on DSpot execution.

Optimize the Information Display. First, we should provide more information visualization options to meet different developers' preferences, such as different color themes. We should also maximize the value of the user's action by only making the branches that we can generate test cases to cover available to select. Besides, we can optimize the amplified test cases that we keep and present to developers by maximizing the diversity of results and removing duplicates.

Support Diverse User Expectations. We can consider all kinds of expectations developers would have for the new test cases and integrate them into the user-guided test amplification design. For example, the code coverage requirement includes not only the branch coverage but also the path coverage. There are also other metrics to measure a test case's value, such as the mutation score. We can support generating new test cases according to developers' expectations on all kinds of metrics in the future.

Support Diverse Expectation Delivery Method. We only support conveying developers' expectations for the amplified test cases by selecting one branch in the Control Flow

Graph. We can also support other methods to meet different developers' preferences, such as selecting branches from the source code in the editor.

Bibliography

[1] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[2] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[3] Taweesup Apiwattanapong, Raul Santelices, Pavan Kumar Chittimalli, Alessandro Orso, and Mary Jean Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06)*, pages 137–146. IEEE, 2006.

[4] Andrea Arcuri. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering*, 23(4):1959–1981, 2018.

[5] Arnaud Roques. Plantuml. `https://plantuml.com`, 2009.

[6] Luciano Baresi and Matteo Miraz. Testful: Automatic unit-test generation for java classes. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 281–284. IEEE, 2010.

[7] Boris Beizer. *Software Testing Techniques (2nd Ed.)*. Van Nostrand Reinhold Co., USA, 1990. ISBN 0442206720.

[8] Carolin Brandt and Andy Zaidman. Developer-centric test amplification. *Empirical Software Engineering*, 27(4):1–35, 2022.

[9] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A snowballing literature study on test amplification. *Journal of Systems and Software*, 157:110398, 2019.

[10] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. Automatic test improvement with dspot: a study with ten mature open-source projects. *Empirical Software Engineering*, 24(4):2603–2635, 2019.

[11] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie Van Deursen. Botsing, a search-based crash reproduction framework for java. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1278–1282. IEEE, 2020.

[12] Pouria Derakhshanfar, Xavier Devroey, and Andy Zaidman. It is not only about control dependent nodes: Basic block coverage for search-based crash reproduction. In *International Symposium on Search Based Software Engineering*, pages 42–57. Springer, 2020.

[13] Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 31–36, 2014.

[14] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[15] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[16] Tony Gorschek, Per Garre, Stig Larsson, and Claes Wohlin. A model for technology transfer in practice. *IEEE software*, 23(6):88–95, 2006.

[17] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12. IEEE, 2015.

[18] Josie Holmes, Iftekhar Ahmed, Caius Brindescu, Rahul Gopinath, He Zhang, and Alex Groce. Using relative lines of code to guide automated test generation for python. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–38, 2020.

[19] Shu Hu. *Pretesting*, pages 5048–5052. Springer Netherlands, Dordrecht, 2014. ISBN 978-94-007-0753-5. doi: 10.1007/978-94-007-0753-5_2256. URL `https://doi.org/10.1007/978-94-007-0753-5_2256`.

[20] Yunho Kim, Zhihong Zu, Moonzoo Kim, Myra B Cohen, and Gregg Rothermel. Hybrid directed test suite augmentation: An interleaving framework. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 263–272. IEEE, 2014.

[21] Kiran Lakhotia, Mark Harman, and Hamilton Gross. Austin: An open source tool for search based software testing of c programs. *Information and Software Technology*, 55(1):112–125, 2013.

[22] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[23] Clifford E Lunneborg. Convenience sample. *The Blackwell encyclopedia of sociology*, 2007.

[24] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, pages 95–111. Springer, 2011.

[25] Bogdan Marculescu, Robert Feldt, and Richard Torkar. A concept for an interactive search-based software testing system. In *International Symposium on Search Based Software Engineering*, pages 273–278. Springer, 2012.

[26] Bogdan Marculescu, Robert Feldt, and Richard Torkar. Objective re-weighting to guide an interactive search based software testing system. In *2013 12th International Conference on Machine Learning and Applications*, volume 2, pages 102–107. IEEE, 2013.

[27] Bogdan Marculescu, Robert Feldt, and Richard Torkar. Practitioner-oriented visualization in an interactive search-based software test creation tool. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 2, pages 87–92. IEEE, 2013.

[28] Bogdan Marculescu, Robert Feldt, Richard Torkar, and Simon Poulding. An initial industrial evaluation of interactive search-based testing for embedded software. *Applied Soft Computing*, 29:26–39, 2015.

[29] Bogdan Marculescu, Simon Poulding, Robert Feldt, Kai Petersen, and Richard Torkar. Tester interactivity makes a difference in search-based software testing: A controlled experiment. *Information and Software Technology*, 78:66–82, 2016.

[30] Bogdan Marculescu, Robert Feldt, Richard Torkar, and Simon Poulding. Transferring interactive search-based software testing to industry. *Journal of Systems and Software*, 142:156–170, 2018.

[31] Marek Parfianowicz, Grzegorz Lewandowski. Openclover. `https://openclover.org`, 2012.

[32] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[33] Aidan Murphy, Thomas Laurent, and Anthony Ventresque. The case for grammatical evolution in test generation. *EVOLUTION*, 10:3520304–3534042, 2022.

[34] Aurora Ramírez, Pedro Delgado-Pérez, Kevin J Valle-Gómez, Inmaculada Medina-Bulo, and José Raúl Romero. Interactivity in the generation of test cases with evolutionary computation. In *2021 IEEE Congress on Evolutionary Computation (CEC)*, pages 2395–2402. IEEE, 2021.

[35] José Miguel Rojas and Gordon Fraser. Is search-based unit test generation research stuck in a local optimum? In *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, pages 51–52. IEEE, 2017.

[36] Conor Ryan, John James Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *European conference on genetic programming*, pages 83–96. Springer, 1998.

[37] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227. IEEE, 2008.

[38] Sina Shamshiri, José Miguel Rojas, Juan Pablo Galeotti, Neil Walkinshaw, and Gordon Fraser. How do automatically generated unit tests influence software maintenance? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 250–261. IEEE, 2018.

[39] Mozhan Soltani, Pouria Derakhshanfar, Annibale Panichella, Xavier Devroey, Andy Zaidman, and Arie van Deursen. Single-objective versus multi-objectivized optimization for evolutionary crash reproduction. In *International Symposium on Search Based Software Engineering*, pages 325–340. Springer, 2018.

[40] Bharti Suri and Prabhneet Nayyar. Coverage based test suite augmentation techniques-a survey. *International Journal of Advances in Engineering & Technology*, 1(2):188, 2011.

[41] Maneela Tuteja, Gaurav Dubey, et al. A research study on importance of testing and quality assurance in software development life cycle (sdlc) models. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(3):251–257, 2012.

[42] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Genetic and Evolutionary Computation Conference*, pages 1400–1412. Springer, 2004.

[43] Claes Wohlin, Martin Höst, and Kennet Henningsson. Empirical research methods in software engineering. In *Empirical methods and studies in software engineering*, pages 7–23. Springer, 2003.

[44] Zhihong Xu and Gregg Rothermel. Directed test suite augmentation. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 406–413. IEEE, 2009.

[45] Zhihong Xu, Myra B Cohen, and Gregg Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1365–1372, 2010.

[46] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 257–266, 2010.

[47] Zhihong Xu, Yunho Kim, Moonzoo Kim, and Gregg Rothermel. A hybrid directed test suite augmentation technique. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, pages 150–159. IEEE, 2011.

[48] Zhihong Xu, Yunho Kim, Moonzoo Kim, Myra B Cohen, and Gregg Rothermel. Directed test suite augmentation: an empirical investigation. *Software Testing, Verification and Reliability*, 25(2):77–114, 2015.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

Test Amplification: A technique that generates new test cases by adapting existing, manually written test cases and can improve the code coverage of the existing test suite [9].

Developer-Centric Test Amplification: Test amplification that provides amplified test cases that developers can take over into their manually maintained test suite, where the developer accepting the test case is central [8].

Open Test Amplification: A prototype of develoepr-centric test amplification designed by Brandt and Zaidman [8], which provides amplified test amplification that can bring any additional instruction coverage to the existing test suite.

Amplifier: DSpot used a series amplifiers to modify the input of test cases.

D-amplifier: A directed amplifier that modify test cases by calling a given target method.

Directed Test Amplification: A test amplification method supported by the D-amplifier, which can amplify test cases directly to cover a given branch.

User-Guided Test Amplification: A test amplification that allows developers to guide the test amplification to generate new test cases according to their branch coverage expectations.

Control Flow Graph (CFG): A directed graph in which the nodes represent basic blocks and the edges represent control flow paths [1].

Appendix B

Materials for User Tasks

This appendix includes the materials we use for the user task in the user study.

B.1 StreamSummary.java

```
package com.clearspring.analytics.stream;

import java.io.ByteArrayInputStream;
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

import com.clearspring.analytics.util.DoublyLinkedList;
import com.clearspring.analytics.util.ExternalizableUtil;
import com.clearspring.analytics.util.ListNode2;
import com.clearspring.analytics.util.Pair;

/**
 * Based on the <i>Space-Saving</i> algorithm and the <i>Stream-Summary</i>
 * data structure as described in:
 * <i>Efficient Computation of Frequent and Top-k Elements in Data Streams</i>
 * by Metwally, Agrawal, and Abbadi
 *
 * @param <T> type of data in the stream to be summarized
 */
```

```java
public class StreamSummary<T> implements ITopK<T>, Externalizable {

    protected class Bucket {

        protected DoublyLinkedList<Counter<T>> counterList;

        private long count;

        public Bucket(long count) {
            this.count = count;
            this.counterList = new DoublyLinkedList<Counter<T>>();
        }
    }

    protected int capacity;
    private HashMap<T, ListNode2<Counter<T>>> counterMap;
    protected DoublyLinkedList<Bucket> bucketList;

    /**
     * @param capacity maximum size (larger capacities improve accuracy)
     */
    public StreamSummary(int capacity) {
        this.capacity = capacity;
        counterMap = new HashMap<T, ListNode2<Counter<T>>>();
        bucketList = new DoublyLinkedList<Bucket>();
    }

    public int getCapacity() {
        return capacity;
    }

    /**
     * Algorithm: <i>Space-Saving</i>
     *
     * @param item stream element (<i>e</i>)
     * @return false if item was already in the stream summary, true otherwise
     */
    @Override
    public boolean offer(T item) {
        return offer(item, 1);
    }

    /**
     * Algorithm: <i>Space-Saving</i>
```

```
 *
 * @param item stream element (<i>e</i>)
 * @return false if item was already in the stream summary, true otherwise
 */
@Override
public boolean offer(T item, int incrementCount) {
    return offerReturnAll(item, incrementCount).left;
}

/**
 * @param item stream element (<i>e</i>)
 * @return Pair<isNewItem, itemDropped> where isNewItem is the return value
 *     of offer() and itemDropped is null if no item was dropped
 */
public Pair<Boolean, T> offerReturnAll(T item, int incrementCount) {
    ListNode2<Counter<T>> counterNode = counterMap.get(item);
    boolean isNewItem = (counterNode == null);
    T droppedItem = null;
    if (isNewItem) {

        if (size() < capacity) {
            counterNode = bucketList.enqueue(new Bucket(0)).getValue().
                counterList.add(new Counter<T>(bucketList.tail(), item));
        } else {
            Bucket min = bucketList.first();
            counterNode = min.counterList.tail();
            Counter<T> counter = counterNode.getValue();
            droppedItem = counter.item;
            counterMap.remove(droppedItem);
            counter.item = item;
            counter.error = min.count;
        }
        counterMap.put(item, counterNode);
    }

    incrementCounter(counterNode, incrementCount);

    return new Pair<Boolean, T>(isNewItem, droppedItem);
}

@Override
public void writeExternal(ObjectOutput out) throws IOException {
    out.writeInt(this.capacity);
    out.writeInt(this.size());
```

```
        for (ListNode2<Bucket> bNode = bucketList.tail(); bNode != null; bNode =
             bNode.getNext()) {
            Bucket b = bNode.getValue();
            for (Counter<T> c : b.counterList) {
                out.writeObject(c);
            }
        }
    }

    protected void incrementCounter(ListNode2<Counter<T>> counterNode, int
        incrementCount) {
        Counter<T> counter = counterNode.getValue();          // count_i
        ListNode2<Bucket> oldNode = counter.bucketNode;
        Bucket bucket = oldNode.getValue();              // Let Bucket_i be the
            bucket of count_i
        bucket.counterList.remove(counterNode);              // Detach count_i
            from Bucket_i's child-list
        counter.count = counter.count + incrementCount;

        // Finding the right bucket for count_i
        // Because we allow a single call to increment count more than once,
             this may not be the adjacent bucket.
        ListNode2<Bucket> bucketNodePrev = oldNode;
        ListNode2<Bucket> bucketNodeNext = bucketNodePrev.getNext();
        while (bucketNodeNext != null) {
            Bucket bucketNext = bucketNodeNext.getValue(); // Let Bucket_i^+ be
                Bucket_i's neighbor of larger value
            if (counter.count == bucketNext.count) {
                bucketNext.counterList.add(counterNode);      // Attach count_i to
                    Bucket_i^+'s child-list
                break;
            } else if (counter.count > bucketNext.count) {
                bucketNodePrev = bucketNodeNext;
                bucketNodeNext = bucketNodePrev.getNext();   // Continue hunting
                    for an appropriate bucket
            } else {
                // A new bucket has to be created
                bucketNodeNext = null;
            }
        }

        if (bucketNodeNext == null) {
            Bucket bucketNext = new Bucket(counter.count);
            bucketNext.counterList.add(counterNode);
            bucketNodeNext = bucketList.addAfter(bucketNodePrev, bucketNext);
        }
        counter.bucketNode = bucketNodeNext;

        //Cleaning up
        if (bucket.counterList.isEmpty())           // If Bucket_i's child-list
            is empty
        {
            bucketList.remove(oldNode);          // Detach Bucket_i from the
                Stream-Summary
        }
    }

    @Override
    public List<T> peek(int k) {
        List<T> topK = new ArrayList<T>(k);

        for (ListNode2<Bucket> bNode = bucketList.head(); bNode != null; bNode =
             bNode.getPrev()) {
            Bucket b = bNode.getValue();
            for (Counter<T> c : b.counterList) {
                if (topK.size() == k) {
                    return topK;
                }
                topK.add(c.item);
            }
        }

        return topK;
    }

    public List<Counter<T>> topK(int k) {
        List<Counter<T>> topK = new ArrayList<Counter<T>>(k);

        for (ListNode2<Bucket> bNode = bucketList.head(); bNode != null; bNode =
             bNode.getPrev()) {
            Bucket b = bNode.getValue();
            for (Counter<T> c : b.counterList) {
                if (topK.size() == k) {
                    return topK;
                }
                topK.add(c);
            }
        }
        return topK;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append('[');
        for (ListNode2<Bucket> bNode = bucketList.head(); bNode != null; bNode =
            bNode.getPrev()) {
            Bucket b = bNode.getValue();
            sb.append('{');
            sb.append(b.count);
            sb.append(":[");
            for (Counter<T> c : b.counterList) {
                sb.append('{');
                sb.append(c.item);
                sb.append(':');
                sb.append(c.error);
                sb.append("},");
            }
            if (b.counterList.size() > 0) {
                sb.deleteCharAt(sb.length() - 1);
            }
            sb.append("]},");
        }
        if (bucketList.size() > 0) {
            sb.deleteCharAt(sb.length() - 1);
        }
        sb.append(']');
        return sb.toString();
    }

    @SuppressWarnings("unchecked")
    @Override
    public void readExternal(ObjectInput in) throws IOException,
        ClassNotFoundException {
        this.bucketList = new DoublyLinkedList<Bucket>();
        this.capacity = in.readInt();

        int size = in.readInt();
        this.counterMap = new HashMap<T, ListNode2<Counter<T>>>(size);

        Bucket currentBucket = null;
        ListNode2<Bucket> currentBucketNode = null;
        for (int i = 0; i < size; i++) {
            Counter<T> c = (Counter<T>) in.readObject();
            if (currentBucket == null || c.count != currentBucket.count) {
                currentBucket = new Bucket(c.count);
                currentBucketNode = bucketList.add(currentBucket);
            }
            c.bucketNode = currentBucketNode;
            counterMap.put(c.item, currentBucket.counterList.add(c));
        }
    }

    /**
     * @return number of items stored
     */
    public int size() {
        return counterMap.size();
    }

    /**
     * For de- serialization
     */
    public StreamSummary() {
    }

    /**
     * For de- serialization
     *
     * @param bytes
     * @throws IOException
     * @throws ClassNotFoundException
     */
    public StreamSummary(byte[] bytes) throws IOException,
        ClassNotFoundException {
        fromBytes(bytes);
    }

    public void fromBytes(byte[] bytes) throws IOException,
        ClassNotFoundException {
        readExternal(new ObjectInputStream(new ByteArrayInputStream(bytes)));
    }

    public byte[] toBytes() throws IOException {
        return ExternalizableUtil.toBytes(this);
    }
}
```

B.2 ConcurrentStreammary.java

```
package com.clearspring.analytics.stream;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.atomic.AtomicReference;

/**
 * Based on the <i>Space-Saving</i> algorithm and the <i>Stream-Summary</i>
 * data structure as described in:
 * <i>Efficient Computation of Frequent and Top-k Elements in Data Streams</i>
 * by Metwally, Agrawal, and Abbadi
 * <p/>
 * Ideally used in multithreaded applications, otherwise see {@link
     StreamSummary}
 *
 * @param <T> type of data in the stream to be summarized
 * @author Eric Vlaanderen
 */
public class ConcurrentStreamSummary<T> implements ITopK<T> {

    private final int capacity;
    private final ConcurrentHashMap<T, ScoredItem<T>> itemMap;
    private final AtomicReference<ScoredItem<T>> minVal;
    private final AtomicLong size;
    private final AtomicBoolean reachCapacity;

    public ConcurrentStreamSummary(final int capacity) {
        this.capacity = capacity;
        this.minVal = new AtomicReference<ScoredItem<T>>();
        this.size = new AtomicLong(0);
        this.itemMap = new ConcurrentHashMap<T, ScoredItem<T>>(capacity);
        this.reachCapacity = new AtomicBoolean(false);
    }
```

```
@Override
public boolean offer(final T element) {
    return offer(element, 1);
}

@Override
public boolean offer(final T element, final int incrementCount) {
    long val = incrementCount;
    ScoredItem<T> value = new ScoredItem<T>(element, incrementCount);
    ScoredItem<T> oldVal = itemMap.putIfAbsent(element, value);
    if (oldVal != null) {
        val = oldVal.addAndGetCount(incrementCount);
    } else if (reachCapacity.get() || size.incrementAndGet() > capacity) {
        reachCapacity.set(true);

        ScoredItem<T> oldMinVal = minVal.getAndSet(value);
        itemMap.remove(oldMinVal.getItem());

        while (oldMinVal.isNewItem()) {
            // Wait for the oldMinVal so its error and value are completely
                up to date.
            // no thread.sleep here due to the overhead of calling it - the
                waiting time will be microseconds.
        }
        long count = oldMinVal.getCount();

        value.addAndGetCount(count);
        value.setError(count);
    }
    value.setNewItem(false);
    minVal.set(getMinValue());

    return val != incrementCount;
}

private ScoredItem<T> getMinValue() {
    ScoredItem<T> minVal = null;
    for (ScoredItem<T> entry : itemMap.values()) {
        if (minVal == null || (!entry.isNewItem() && entry.getCount() <
            minVal.getCount())) {
            minVal = entry;
        }
    }
```

```
        return minVal;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        for (ScoredItem entry : itemMap.values()) {
            sb.append("(" + entry.getCount() + ":␣" + entry.getItem() + ",␣e:␣"
                + entry.getError() + "),");
        }
        sb.deleteCharAt(sb.length() - 1);
        sb.append("]");
        return sb.toString();
    }

    @Override
    public List<T> peek(final int k) {
        List<T> toReturn = new ArrayList<T>(k);
        List<ScoredItem<T>> values = peekWithScores(k);
        for (ScoredItem<T> value : values) {
            toReturn.add(value.getItem());
        }
        return toReturn;
    }

    public List<ScoredItem<T>> peekWithScores(final int k) {
        List<ScoredItem<T>> values = new ArrayList<ScoredItem<T>>();
        for (Map.Entry<T, ScoredItem<T>> entry : itemMap.entrySet()) {
            ScoredItem<T> value = entry.getValue();
            values.add(new ScoredItem<T>(value.getItem(), value.getCount(),
                value.getError()));
        }
        Collections.sort(values);
        values = values.size() > k ? values.subList(0, k) : values;
        return values;
    }

    public long size(){
        return this.size.get();
    }
}
```

B.3 StreamSummaryTest.java

```
package com.clearspring.analytics.stream;

import org.junit.Test;

import static org.junit.Assert.*;

public class StreamSummaryTest {

    @Test
    public void offer() {
        StreamSummary stream = new StreamSummary(3);
        stream.offer("X");
        assertEquals(1, stream.size());
    }
}
```

B.4 ConcurrentStreamSummaryTest.java

```
package com.clearspring.analytics.stream;

import org.junit.Test;

import static org.junit.Assert.*;

public class ConcurrentStreamSummaryTest {

    @Test
    public void offer() {
        ConcurrentStreamSummary stream = new ConcurrentStreamSummary(3);
        stream.offer("X");
        assertEquals(1, stream.size());
    }
}
```

Appendix C

Questionnaires

This appendix presents the questionnaires we used before and during the user study experiment interview. Section C.1 and Section C.2 show the consent survey and pre-test questionnaire we send to the participants before the user study interview. Section C.3 shows the questionnaire we use to investigate developers' opinions on the user-guided test amplification during the interview.

C.1 Consent Survey

Welcome to the informed consent survey for the Test Cube evaluation interviews. This study is done by Danyao Wang from the TD Delft for her master thesis. In this survey we will inform you of the data we are collecting during the interview, how we are handling the data and how you can request for your data to be deleted.

C.1.1 Taking Part in the Study

This part is about the general procedure of the study and consent to take part in the study.

The purpose of this interview is to explore what developers think of the Test Cube plugin's function of generating test cases. We will introduce the Test Cube plugin and the example project to you and then ask you to generate test cases with Test Cube in two different ways for two example java classes. Finally, we will ask you to fill out a survey containing some questions about your opinion on the Test Cube plugin and discuss why you choose a specific answer. The survey and discussion will be analyzed to extract aggregated and anonymized information for the final thesis report.

The interview will be conducted over a video call (e.g., Zoom) and will take about 60 minutes. Below we ask you whether you are okay with us recording the video call so we can re-analyze the interview afterwards. Please tell us beforehand if you do not want to be recorded, and we will come up with an alternative.

You can also choose whether we can anonymously quote you in any research output and include the data from your interview in an aggregated and anonymized data set. If you choose 'no', we will exclude your quotes from our research outputs, and not include your data in the dataset we will publish after finishing the research.

The interviews will be conducted by Danyao Wang (d.wang-9@student.tudelft.nl). Carolin Brandt (c.e.brandt@tudelft.nl) and Andy Zaidman (a.e.zaidman@tudelft.nl), her supervisors, are also part of the research team.

As with any online activity, the risk of a breach is always possible. To the best of our ability, your answers in this study will remain confidential. We will minimize any risks by making the survey completely anonymous, keeping the data only accessible to the research team, ensuring any result in the publications completely anonymous, and destroying all the personal research data(your email address and interview recording) after finishing the research.

If you have any further questions about the interview procedure, the data processing or publishing afterwards, or you wish for your personal data to be deleted, you can ask during the interview or contact Danyao (d.wang-9@tudelft.nl) at any time.

1 To match your answers in this survey to you, please fill in the email we used to contact you for the interview.

[open]

2 I have read and understood the study information above. I have been able to ask questions about the study, and my questions have been answered to my satisfaction.

[multiple choice]: Yes/No

3 I consent voluntarily to be a participant in this study and understand that I can refuse to answer questions and I can withdraw from the study at any time, without having to give a reason.

[multiple choice]: Yes/No

C.1.2 Use of Information in the Study

1 I consent to record the interview video call (audio and video, including the shared screen). The recordings will be destroyed after the study is completed. Any results that will be shared in publications will be anonymized.

[multiple choice]: Yes/No

2 I agree that the information I give during the interview can be anonymously quoted in research outputs resulting from this study.

[multiple choice]: Yes/No

3 I understand that information I provide during the interview will be used for research and scientific publications from the TU Delft and an inspiration for further features and versions of the Test Cube.

[multiple choice]: Yes/No

4 I understand that personal information collected about me that can identify me, such as my e-mail address and the interview recordings, will not be shared beyond the research team.

[multiple choice]: Yes/No

5 I give permission for aggregated data and anonymized quotes (if you agreed above) from this interview to be archived on the Git(lab)/subversion repository at TU Delft so it can be used for future research and learning.

[multiple choice]: Yes/No

6 Do you have any further comments on how we should process your data? If questions arise later, please contact Danyao: d.wang-9@student.tudelft.nl.

[open]

C.2 Pre-Test Questionnaire

We ask you a set of general questions to determine your experience with software development. Test amplification tools propose new test cases based on your existing test cases. They can be used to automatically improve test suites, e.g., their coverage or their mutation score.

1 For how many years have you developed software and written tests for it?

[open]

2 Which programming languages do you primarily use?

[open]

3 Which industry domain are you developing software for?

[open]

4 You enjoy writing tests for software.

[Likert 1-5]

5 Have you ever used automated tools for generating test cases?

[multiple choice] Yes/No

6 How would you rate your experience with automated test case generation tools?

[Likert 1-5]

C.3 Post-Test Questionnaire

This questionnaire consists of a series of questions about your experience using the Test Cube's two different ways of generating test cases. Please select an answer for each question and explain why you choose the specific option.

C.3.1 User-Guided Test Amplification

This section is about the first method you tried to generate test cases.

1 The Control Flow Graph of methods is easy to understand.

[Likert 1-5]

2 The Control Flow Graph of methods provides valuable information.

[Likert 1-5]

3 The interaction with the Control Flow Graph effectively assists you in conveying your expectations for the test cases.

[Likert 1-5]

4 The test generation results displayed with the Control Flow Graph are clear and easy to understand.

[Likert 1-5]

5 The the Control Flow Graph and branch/line coverage is helpful when you select test cases.

[Likert 1-5]

C.3.2 Open Test Amplification

This Section is about the second method you tried to generate test cases.

1 The instruction coverage and corresponding code highlighting is easy to understand.

[Likert 1-5]

2 The test case information provides valuable information.

[Likert 1-5]

3 The test generation result displayed with additional instruction coverage is clear and easy to understand.

[Likert 1-5]

4 The modifications applied to test cases are helpful when you select test cases.

[Likert 1-5]

5 The instruction coverage and highlighting code are helpful when you select test cases.

[Likert 1-5]

C.3.3 Compare the Two Types of Test Amplification

1 Which type of coverage is easier to understand?
[Multiple choice]: Line and branch coverage/Instruction coverage/Neutral

2 Which display form of coverage is easier to understand?
[Multiple choice]: The coverage shown in CFG/The coverage shown in text/Neutral

3 Which type of test amplification goal helps you select the amplified test cases more?
[Multiple choice]: Aiming to cover a specific method and branch/Aiming to cover any additonal instructions/Neutral

4 Which type of test amplification is more helpful for you to generate test cases?
[Multiple choice]: User-Guided Test Amplification/Open Test Amplification/Neutral

C.3.4 Overall Impression

1 The amplified test cases provided by TestCube satisfy your expectations.
[Likert 1-5]

2 You would want to use TestCube to help you write tests in the future.
[Likert 1-5]