University of Sheffield

Test Suite Health: Automatically Improving the Reliability and Effectiveness of Test Suites

Muhammad Firhard Roslan

Supervisor: Professor Phil McMinn

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

in the

Faculty of Engineering

School of Computer Science

March 2025

Dedicated to my wife, Ayuri, my parents, Fadhilah and Roslan,
my entire family, and Yuki and Aiko.

Declaration

Muhammad Firhard Roslan
March 2025

Abstract

Software systems are inherently prone to faults, stemming from human errors and incorrect assumptions made during the development process. Additionally, due to the evolving nature of codebases, even a correct code behaviour can degrade, becoming faulty. To mitigate these issues, unit testing is widely adopted, providing developers with a systematic way to exercise the system and verify individual units. Typically, developers write tests that exercise the production code, with the goal of achieving a certain level of code coverage and fault detection capabilities. However, writing tests that only achieve high code coverage and mutation scores alone is not enough; tests must also be reliable.

Advancements in automated test generation techniques have decreased the burden of developers. However, developers remain primarily responsible for writing reliable and effective test suites, as these tools often fall short in areas requiring human expertise. Inadequate developer-written tests—whether due to low fault detection capability, flakiness, brittleness, or lack of realism—not only increase the maintenance overhead of the software systems, but also undermine the reliability of the test suite.

This thesis takes a holistic approach to assessing and improving the quality of unit test suites, focusing on its reliability and effectiveness. To address these challenges, I set two primary objectives for this thesis: (1) understanding the factors that limit the reliability and effectiveness of test suites, and (2) developing and empirically evaluating automated techniques to improve and repair existing test suites. Firstly, to ensure that the state-of-the-art automated test generation tool, EvoSuite, could be utilised to enhance existing developer-written tests, I conducted an empirical study to investigate the issue of flakiness in tests generated by such tools. Following this, I evaluated the effectiveness of search-based test generation of EvoSuite capabilities to *amplify* existing developer-written test suites' mutation score. To gain insights into software developer's perspectives on test brittleness, I then conducted a developer survey of 73 professional software developers, evaluated 60 StackOverflow threads, and empirically evaluated 4,801 open-source projects. This is then followed by utilising EvoSuite's search-based test generation technique to replace tests that make calls directly to implementation details with more reliable alternatives. These efforts contribute to understanding and improving four main indicators of test suite quality: fault detection capabilities, test flakiness, test brittleness, and test "realism".

Publications

Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. An Empirical Comparison of EvoSuite and DSpot for Improving developer-written test suites with respect to mutation score. In *International Symposium on Search-Based Software Engineering (SSBSE)*, 2022

Martin Gruber, Muhammad Firhard Roslan, Owain Parry, Fabian Scharnböck, Phil McMinn, and Gordon Fraser. Do Automatic Test Generation Tools Generate Flaky Tests? In *International Conference on Software Engineering (ICSE)*, 2024

Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. Private — Keep Out? Understanding How Developers Account for Code Visibility in Unit Testing. In *International Conference on Software Maintenance and Evolution (ICSME)*, 2024

Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. Viscount: A Direct Method Call Coverage Tool for Java. In *International Conference on Software Maintenance and Evolution (ICSME): Tool Demo Track*, 2024

Phil McMinn, Muhammad Firhard Roslan, and Gregory M. Kapfhammer. Beyond Test Flakiness: A Manifesto for a Holistic Approach to Test Suite Health. In *Proceedings of the 2nd International Workshop on Flaky Tests (FTW)*, 2025

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Phil McMinn, for his invaluable guidance, support, and advice throughout my Ph.D. journey. I am grateful to have him as my supervisor and mentor, and I have learned a lot from him, both academically and in my personal life. Furthermore, I would like to thank Dr. José Miguel Rojas for providing valuable feedback on my research work, being an excellent collaborator, and a very good friend.

I would also like to thank the past and present researchers and academics in the Testing lab at the University of Sheffield: Islam, Owain, Ruizhen, Megan, Michael, Richard, Olek, Zalán, Neil, Nina, Giulia, Yasmeen, Guannan, Donghwan, Arwa, Tony, Kirill, and Sanjeetha. Thank you for sharing your knowledge and experiences with me. I am also deeply grateful to my collaborators: Martin, Gordon, Greg, and Fabian—who helped me in my Ph.D. research. Moreover, I would like to thank Faidzal, Nik, Reezuan, Fahmi, Faiz, Amal, Hikmah, Faris, Alynn, Zaid, Syahrul, and all my other friends from outside the lab, who provided me a lot of moral support.

I would like to express heartfelt thanks to my wife, Ayuri, for her unwavering support throughout this journey, as well as my parents, Fadhilah and Roslan; my parents-in-law, Rie and Rujhan; and my entire family, who stood by my side every step of the way. Finally, I am grateful for the financial support that I received from Majlis Amanah Rakyat, Malaysia, and I give thanks to God for all the blessings that I have received.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Living plants are tender and pliant;
dead, they are brittle and dry.
...
The rigid and stiff will be **broken**.
The soft and yielding will **prevail**."
-Lao Tzu

In 2004, Jeff Bezos remarked, *"It's the software that sends the brown boxes on their way."* [172]. This statement highlights how deeply software plays in everyday modern operations, from logistics to essential services. However, when software fails, the consequences can be severe. On July 19, 2024, a routine software update from CrowdStrike [38] triggered a catastrophic failure that crashed 8.5 million Microsoft Windows machines [9] worldwide and costing Fortune 500 companies over $5 billion in direct losses [43], with total global losses expected to be significantly higher. This incident, which brought critical services such as flights [13], hospitals, and banks to a stop [12], revealed shortcomings in their software update validation. Despite routine automated and manual tests on their core components, gaps in validating certain updates [11] led to the infamous "blue screen of death" [6] caused by an out-of-bounds memory read [8]. Although the update was rolled back quickly, millions of devices [9] were already affected. In response, CrowdStrike is tightening its testing protocols through enhanced software testing [38] and staggered deployments [10] to prevent future failures.

This incident serves as a reminder that software systems are inherently prone to faults [67]. This inevitability stems from the fact that code is mostly written by humans, who are inherently fallible [59]. Faults can emerge from a variety of sources: incorrect assumptions by developers, improper software usage by end-users, and malfunction in the software. Furthermore, code that was once correct can degrade and become faulty over time due to changes and updates [112]. Therefore, software developers will adopt certain software testing practices to protect themselves from such situations [67].

Unit testing provides a practical approach to mitigate this problem, where software developers write tests to verify the behaviour of individual units of the code. The process then involves executing tests and ensuring the units behave as expected. While the concept of writing unit tests is straightforward, inadequate tests can result in anything from small user inconvenience to catastrophic failures [114, 21, 38], and maintaining them effectively can be challenging [166, 189].

Despite advancements in testing techniques that could automatically generate tests from scratch, such as, fuzzing [193, 259] and automated test generation [202, 122, 180], developers remain primarily responsible for writing their own tests [70]. While these tools are effective in achieving high code coverage [125], they still fall short in areas where human expertise is indispensable, such as ensuring that tests are understandable [103] and addressing the hard problems in validating the correctness (e.g., test oracle problem [79]).

However, just as humans are prone to errors when writing code, they are equally prone to make mistakes when writing tests [152]. These flaws can be very obvious, such as failing to meet certain *quantifiable* test adequacy goals, or more diffuse on revealing themselves, like the presence of flaky [210] or brittle [165] tests. Such issues can undermine a test suite's ability to detect faults and provide reliable feedback [166, 189]. Enhancing the reliability and effectiveness of developer-written tests is therefore vital. Achieving this goal requires a broader perspective on how tests are written, moving beyond the immediate concerns of individual test cases, it is essential to consider the overall *health* of the test suite.

1.1 Test Suite Health

A functional test suite is one that provides fast, reliable feedback to developers, and remains effective throughout the lifecycle of a software project. Test suites created solely to meet specific coverage criteria [252] may fail to support software development effectively, as they can become difficult to manage without emphasis on their purpose (Section 2.6.2), execution conditions [155, 148] (Section 2.6.1), and relevance (Section 2.6.2.1).

While test adequacy criteria [67] (Section 2.4) are essential to ensure that a test suite is effective, developers must also consider broader aspects to sustain its health. A healthy test suite should not only fulfill its primary purpose of validating the software's functionality, but does so without introducing unnecessary burdens on the development process (e.g., test flakiness [210]), either at the present time or in the future. This raises critical questions about what a healthy test suite entails and how its continued utility can be ensured.

A Holistic Perspective

A healthy test suite [187] needs to consider a holistic perspective, moving beyond a single indicator, such as the widely used code coverage [153]. Instead, it should encompass a broader range of indicators that collectively determine the test suite's fitness for purpose. Table 1.1 introduces a set of indicators that should be considered, starting from those that are well-understood and measurable, to those that are less so and more diffuse. Together, they provide a basis for evaluating test suite health, focusing on how effectively the test suite provides fast and reliable feedback.

These indicators could be structured into a checklist [131], aimed at maximising their positive impact while minimising the negative presence of each one. Various tools and techniques provide a way to evaluate individual indicators (e.g., [18, 156, 97]), as well as to improve (e.g., [106, 254]) and repair (e.g., [240, 175]) them individually. However, such approaches often fail to consider the interactions between indicators, neglecting a holistic view of test suite health.

While some indicators are complementary, trade-offs also exist between them. For example a test suite with a higher mutation score [113] may potentially contain more brittle test cases [165] if it focuses on the program's implementation details rather than its intended specification. This suggests that optimising only one indicator might not be appropriate. This also means that test suite health could be framed as a Pareto-based optimisation problem [204], where the objective is to achieve a balanced trace-off between conflicting indicators. It is also important to point out that the indicators in Table 1.1 are by no means exhaustive, leaving more opportunities for future works to identify additional indicators for evaluating test suite health.

Distinguishing Test Suite Health from Related Concepts

While closely related, test smells [248], test suite maintenance [196], and test suite health differ in their focus and scope. Test smells typically refer to static properties associated with how individual tests are implemented, often highlighting poor programming practices [206] within a test that indicate potential issues with maintainability [190] and readability [139].

In contrast, test suite health takes a broader view, evaluating how well the entire test suite functions. While test smells focus on issues in individual tests [28], test suite health emphasises the suite's ability to provide developers with reliable feedback about the correctness of their software. This distinction underscores the complementary nature of these concepts: addressing test smells could potentially contribute to improving test suite health, but achieving a healthy test suite requires considering factors beyond individual test case implementation.

Similarly, test suite maintenance involves tasks such as updating and managing existing tests [196] to preserve their relevance [152], as software evolves [173] and to ensure that test suite does not become outdated [258]. While they are necessary for keeping a test suite up-to-date, they do not inherently ensure they are healthy. For instance, a well-maintained test suite could still fail to deliver meaningful feedback if it lacks reliability or adequate coverage of critical functionalities [166].

The concept of test suite health mainly concerns about its overarching ability to reliably validate the software and remain effective throughout the software lifecycle, distinguishing it from test smells, which focus on static properties of individual tests [28], and test suite maintenance, which addresses the upkeep of the test suite [258]. Trying

Table 1.1: *This table provides a list of indicators that could be used to evaluate the health of a test suite, starting with those that are well-understood and could be quantified, to those that are less so and more diffuse. These indicators are not exhaustive and there are others that should also be considered. Taken from [187].*

#	Indicator	Description	Chapter
1	**Code Coverage (Section 2.4.1)**	A test suite that does not meet certain code coverage [265] criteria—skipping certain executable regions of the program under test—is the first and most obvious indicator of unhealthy test suite. Test suite with low code coverage is unable to reveal faults or give useful feedback.	
2	**Pseudo-Testedness**	Pseudo-tested elements (e.g., methods [197, 249] or statements [185]) are executed by tests but can be removed from the program without affecting the tests' passing or failing outcomes. High levels of pseudo-testedness indicate an unhealthy test suite due to a lack of emphasis on writing high-quality assertions.	
3	**Mutation Score (Section 2.4.2)**	A test suite with low mutation score [67] indicates that it is not sensitive to artificially seeded faults [110], and may fail to catch changes that could potentially introduce real faults.	Chapter 4
4	**Execution Time**	Tests should provide fast feedback to developers [189]. If not, this may discourage them from running them as frequently as they should.	
5	**Diversity**	Tests that exercise only certain execution traces are likely to indicate low test diversity. Developers often use a "copy and paste" approach from previous test cases to derive new tests [70]. Syntactically, this means the test cases may execute similar program paths.	
6	**Flakiness (Section 2.6.1)**	Tests that produce inconsistent outcomes—such as passing and failing intermittently, even when executed on the same environment without code changes—are known as flaky tests [211, 182]. Flaky tests undermine the trust in the test suite, making it harder to distinguish real failures from false positives.	Chapter 3
7	**Brittleness (Section 2.6.2)**	Tests that are highly coupled to implementation details may break due to production code changes [165]. Tests should only focus on the behaviour of the program under test, and not its implementation.	Chapter 5 & Appx. A
8	**"Realism" (Section 2.6.2.1)**	Tests that do not imitate how public APIs [165] are used in production—exercising the program differently from real-world users—may lack realism and potentially lead to false outcomes. Similar to brittleness, low realism in tests means they do not verify the behaviour of the production code, resulting in unreliable tests.	Chapter 6
9	**Environmental-Dependent**	Tests that the outcome is dependent on certain operating systems [155] may lack portability as it is tightly coupled with certain environments.	
10	**Variability of Indicators**	A test suite that varies in its coverage [148] or mutation score [239] levels from run to run, is an indication of an unstable test suite. Variability in outcomes is another sure sign of an unhealthy test suite.	

to achieve a healthy test suite requires considering a broader perspective that goes beyond individual test implementation and maintenance updates.

1.2 Aims

The primary aim of this thesis is to understand and develop techniques that could improve the reliability and effectivness of test suites. To address these open challenges, I set two high-level objectives for this thesis:

1. **Understanding:** To identify the challenges that limit the lifespan and effectiveness of test suites.
2. **Improving:** To develop and empirically evaluate automated techniques to improve the effectiveness of test suites and repair unreliable test cases.

1.3 Thesis Contributions

To achieve these objectives, this thesis makes five primary contributions. Figure 1.1 shows the relations between the chapters in this thesis. The following subsections provide a summary of each chapter[1].

1.3.1 Flakiness in EvoSuite-generated tests (Chapter 3)

Chapter 3 is part of a paper that I have published in the proceedings of the 46th International Conference on Software Engineering (ICSE), 2024 [142].

One of the key indicators of an "unhealthy" test suite is flaky tests [187], as described in Table 1.1-#6. Flaky tests have been extensively studied in developer-written tests [168, 141], but their prevalence still remains unexplored in automatically generated tests.

For this, I have made an empirical evaluation of flaky tests generated by EvoSuite [122] on 1,902 Java projects, and compared them to the root-causes of the flakiness with developer-written tests. I also evaluated the effectiveness of flakiness suppression mechanism to remove or repair any flaky tests that are generated by EvoSuite. Among the findings are: (1) flaky tests generated by EvoSuite without flakiness suppression mechanism is as common as developer-written test; (2) the root causes of flakiness in both are distributed differently; and (3) EvoSuite's flakiness suppression mechanism is able to reduce the number of flaky tests by 71.7 % but different root causes of flakiness. The main contributions of this chapter are as follows:

[1]In this thesis, I will use active voice to describe the work I have done, even in a collaborative setup, to highlight my contributions.

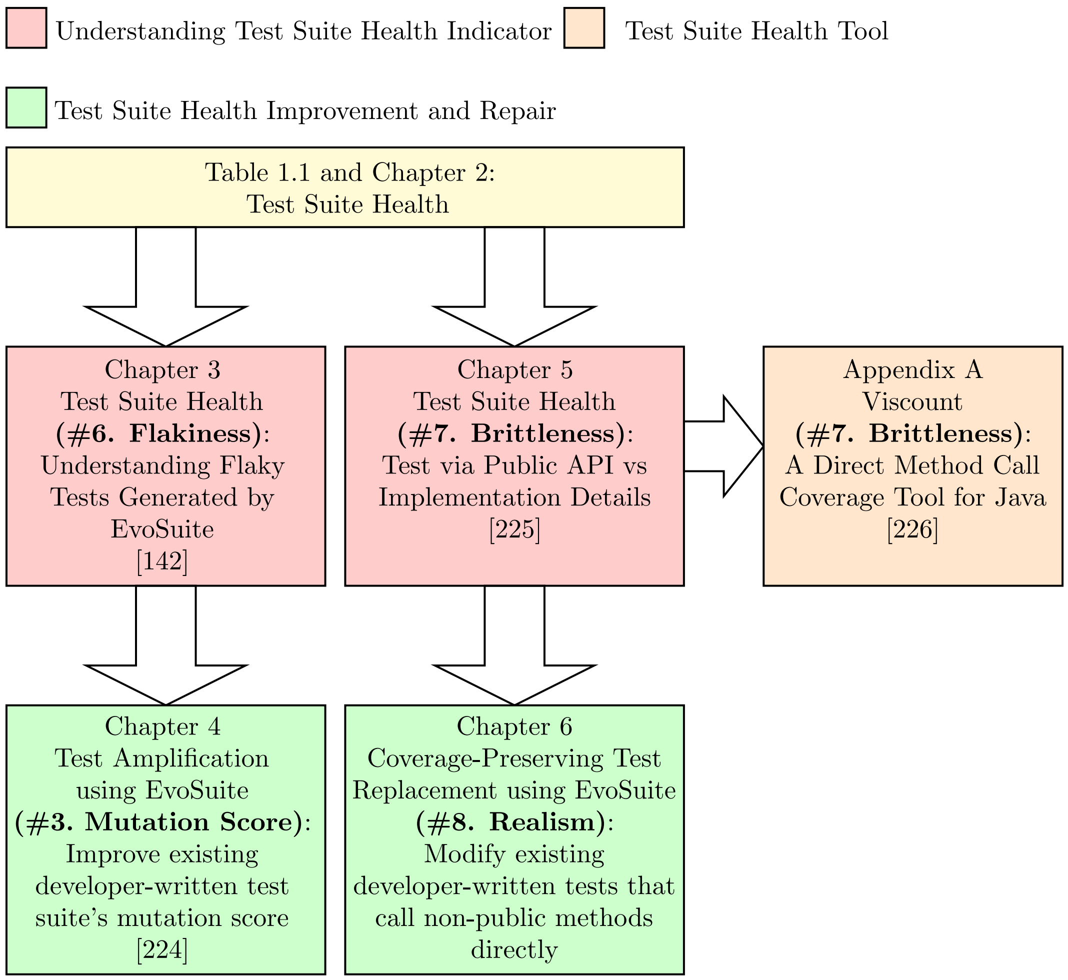

Understanding Test Suite Health Indicator
Test Suite Health Tool
Test Suite Health Improvement and Repair
Table 1.1 and Chapter 2: Test Suite Health
Chapter 3 Test Suite Health (#6. Flakiness): Understanding Flaky Tests Generated by EvoSuite [142]
Chapter 5 Test Suite Health (#7. Brittleness): Test via Public API vs Implementation Details [225]
Appendix A Viscount (#7. Brittleness): A Direct Method Call Coverage Tool for Java [226]
Chapter 4 Test Amplification using EvoSuite (#3. Mutation Score): Improve existing developer-written test suite's mutation score [224]
Chapter 6 Coverage-Preserving Test Replacement using EvoSuite (#8. Realism): Modify existing developer-written tests that call non-public methods directly

Figure 1.1: *The connections between the chapters of this thesis are depicted here. Red blocks focus on "understanding" issues related to Test Suite Health, green blocks are dedicated to techniques to automatically "improve" the quality and health of the test suite, and the orange block represents the tool I developed to evaluate test health-related issues. The arrows between the blocks indicate that each builds upon the previous one, and citations indicate chapters based on my published work.*

1. **Empirical study (Section 3.2):** My empirical evaluation, involving 1,902 open-source Java projects, is the largest conducted to date on both flaky tests and search-based test generation. It is also the first to investigate the root causes of automatically EvoSuite-generated flaky tests.

2. **Recommendations (Section 3.3 and Section 3.5):** The results of this study have significant implications for software developers, maintainers of EvoSuite, and researchers in the field of flaky tests. From these findings, I provide actionable insights and recommendations for them.

3. **Dataset [52]:** The dataset I collected for this study is the first publicly available dataset of flaky tests that includes EvoSuite-generated flaky tests and features a large manually annotated sample. This dataset is included in the replication package to enable further research [52].

1.3.2 Improving Mutation Score of Developer-Written Tests using EvoSuite (Chapter 4)

Chapter 4 is based on a paper that I have published in the proceedings of the 14th International Symposium on Search-Based Software Engineering (SSBSE), 2022 [224].

As stated in Section 2.8.1, the state-of-the-art test amplification tool, DSpot, offers only limited approaches to modify existing developer-written test cases and is not as flexible as EvoSuite's evolution process. After evaluating the reliability of EvoSuite-generated tests as discussed in Chapter 3, in this chapter, I evaluated the "best of both worlds" approach that utilises EvoSuite's Genetic Algorithm while using the developer-written tests as seeds (initial population) to *improve* the fault detection capabilities (Table 1.1-#3)—measured by mutation score—of existing developer-written tests. The main findings from this work are: (1) tests generated by EvoSuite$_{Amp}$ were more effective at killing mutants when compared to DSpot; (2) over 30 repeated runs, EvoSuite$_{Amp}$ tests were able to kill more "unique" mutants when compared to DSpot; and (3) anecdotally, EvoSuite$_{Amp}$ tests were found to be less readable when compared to DSpot.

The contributions of this chapter that improve the quality of existing developer-written test suites are as follows:

1. **EvoSuite$_{\mathbf{Amp}}$ (Section 4.2) [51]:** A new test improvement strategy that utilises the flexibility of EvoSuite, EvoSuite$_{Amp}$, that evolves test cases and leverages fitness information for killing specific mutants.

2. **Dataset (Section 4.3):** An empirical study with seven open-source projects comparing EvoSuite$_{Amp}$ with an existing state-of-the-art test amplification tool, DSpot.

3. **Evaluation (Section 4.4):** Results and analysis of the effectiveness of both tools in terms of mutation score and mutants uniquely killed by each tool.

1.3.3 Private — Keep Out? Test via Public API vs Implementation Details (Chapter 5)

Chapter 5 is based on a paper that I have published in the proceedings of the 40th International Conference on Software Maintenance and Evolution (ICSME), 2024 [225].

This chapter focuses on unit testing practices when it comes to invocation of production methods based on its visibility (Table 1.1-#7). It combines an analysis of 4,801 open-source projects, a developer survey that received 73 responses, a systematic search on StackOverflow involving 60 threads, and provide actionable recommendations. Replication package is available at [56].

Using both numerical and thematic analyses, I made several findings: (1) developers are divided on whether to test exclusively through public APIs; (2) developers who are in favour of only testing through the public APIs tend to have more experience in writing unit tests and believe that the need to test non-public methods directly is due to poor software design; and (3) developers who test non-public methods directly are more concerned about untested areas and overly intricate tests.

This chapter contributes the following to the understanding of another issue in test health:

1. **Open-Source Study (Section 5.4.1):** An empirical analysis of the visibility of methods called directly from the tests across 4,801 Java projects from the Maven Central Repository.
2. **Developer Survey and StackOverflow Threads (Section 5.4.2–5.4.4):** A numerical and thematic analysis of 73 developer survey responses and 60 StackOverflow threads to identify developer attitudes and approaches to testing public APIs versus testing non-public methods directly.
3. **Findings and Recommendations (Section 5.7 and (Section 5.6)):** I discussed a range of findings and summarised the current practices. From there, I offer directions for future work to improve another issue regarding test health.

1.3.4 Replacing developer-written tests that call non-public methods directly (Chapter 6)

This chapter presents a technique for replacing directly called non-public methods in developer-written tests with public APIs that execute them indirectly, to enhance its realism (Table 1.1-#8). As discussed in Chapter 5, some developers find it challenging to test exclusively through public APIs, leading them to test non-public methods directly. This test repair technique utilises EvoSuite's search-based approach to identify candidates that can replace the directly invoked non-public methods with Public API while attempting to preserve the branch coverage of the method. The evaluation which involves 181 Java project revealed the following findings: (1) EvoSuite$_{UTOPIA}$

is able to successfully replace 55.7 % of the directly called non-public methods; (2) successful replacement is able to maintain an average of 84.79 % branch coverage; and (3) unsuccessful replacements were due to not being able to find Public API candidates that are invoking the non-public methods, limitations in EvoSuite's instrumentation, and external dependencies.

The chapter focuses on improving the reliability of developer-written test suites makes the following contributions:

1. **EvoSuite$_{\textbf{UTOPIA}}$ (Section 6.2) [57]:** A novel technique and a new fitness function to replace non-public methods directly invoked in a test with public methods that exercise them indirectly.

2. **Dataset (Section 6.3):** An empirical study conducted on 136 projects to assess the feasibility of EvoSuite$_{UTOPIA}$.

3. **Evaluation (Section 6.4):** Findings and analysis of the effectiveness and usefulness of EvoSuite$_{UTOPIA}$ in replacing non-public method invocations in tests.

1.3.5 Viscount: A Direct Method Call Coverage Tool (Appendix A)

GitHub Repository: `https://github.com/unittesting-nonpublic/viscount`
Appendix A is based on a tool paper that I have published in the proceedings of the 40th International Conference on Software Maintenance and Evolution: Tool Demo Track (ICSME - Tool Demo Track), 2024 [226].

I have developed an open-source tool called Viscount to identify methods and their visibility (access modifiers) that are directly invoked in JUnit test suites. This tool can help developers identify test cases that need to be refactored to ensure that tests are more maintainable and less brittle. The evaluation of JUnit test cases in open-source projects, as discussed in Section 5.4.1, was conducted using Viscount. Appendix A provides details about Viscount's inner working and discuss its current limitations.

1.3.6 Summary

In summary, as shown in Figure 1.1, this thesis contributes to understanding and improving the reliability and effectiveness of test suites. The chapters in this thesis address the challenges that limit the lifespan and effectiveness of test suites, and develop and empirically evaluate automated techniques to improve the effectiveness of test suites and repair unreliable test cases.

Chapter 2

Literature Review

2.1 Introduction

Developing high-quality software should be supported with software testing [196, 67], a critical part of the software development life-cycle. In this chapter, I surveyed relevant concepts and works that are related to the contribution of this thesis. It begins by briefly explain about software testing. It then discusses the topic of software testing, the introduction of Test Suite Health. The chapter then examines the various metrics available for assessing test suite reliability and effectiveness. Finally, the chapter introduces on how automated unit test generation [68], test amplification [105], and test repair [152] techniques could be employed to improve the health of a test suite. It also highlights prior studies that have explored the use of automated test generation tools to improve the reliability and effectiveness of test suites.

2.2 Software Testing

Software testing is a critical part of the software development process [67] that intends to assess the quality of the system-under-test. It is the process of executing a program with the intent of finding faults by evaluating the behaviour of the program on a finite number of inputs. The collection of a finite set of inputs (test case [45]) is known as a test suite that executes the system under test (SUT) [266]. However, it can never show the absence of faults as it is impossible to exhaustively test a software in general [102]. Thus, the main challenge and goal is to find a subset of inputs that will discover as many faults as possible in the software. The consequences of not exercising good test inputs can range from minor issues (e.g., small user annoyance) to catastrophic (e.g., [114, 21]), that lead to data theft or loss of service.

```
public String sort() {
    Collections.sort(items);
    return String.join(",", items);
}
```

Figure 2.1: *An example of* `sort()` *method that is being executed by Figure 2.2*

2.3 Unit Testing

Software testing can be done on multiple different levels (e.g., unit, integration, system) and the first level of testing, also known as unit testing, is performed on the "software units or groups of related units" [45]. It usually serves as the first line of defence against any introduction of software faults. A unit test case [45] is a small piece of code (unit) designed to execute a specific behaviour of the program, which includes certain numbers of observations about the behaviour of the program being executed. If the expected observed output is different from the actual output, the test will fail, and it usually indicates a software *fault*—a static defect in the program [67].

An example of a method that should be unit tested is shown in Figure 2.1. Typically, to test the method, a unit test case similar to the example in Figure 2.2 consists of the four main components: test inputs, invocation of a method-under-test, test output, and test oracle (assertion). The test starts by setting up some *inputs* necessary for the test to run (line 3 to line 5). The test then will execute a *method* (`sort()`) of the program, as shown on line 6, storing the *test output* in a variable. Finally, the *assertion* oracle will check if the output of the variable matches with the expected output using the `assertEquals`, as shown on line 7. A test case will fail if the assertion fails, indicating that either the program has a fault or the assertion is incorrect.

In order to validate the behaviour of a program, test oracles, mainly in the form of assertions are created, and it catches certain expected behaviour. One of the main challenges in writing unit test cases is distinguishing between correct program behaviour and incorrect behaviour. This is known as the *test oracle problem* [79], and it requires human with domain knowledge of the program under test. It would also not be feasible to write test for all possible test inputs and outputs. This means it is important to identify places where the program is most likely to exercise the correct and write test cases to ensure that the program behaves as expected.

2.4 Test Adequacy Criteria

As discussed in the previous section, one of the main goals of testing is to discover faults in a program. However, being able to decide when it is "enough" [265] to stop testing could be challenging, especially if there are no specific engineering goals to be achieved. To address this issue, few test adequacy criteria [252] metrics are being proposed to measure the capability of a test suite to expose faults [136] and determine whether a sufficient range of behaviours of the program have been tested.

2.4.1 Code Coverage

The most widely used test adequacy criteria to assess the quality of a test suite is code coverage. Code coverage refers to the code units that have been executed by the test cases within a test suite [266]. It also gives an approximation of the test suite's ability to capture as much of the program behaviour [126]. Before a developer can detect any faults in the system, a test case must execute the line(s) in which the fault resides.

The simplest and most common form of code coverage is line coverage, which requires that each line of code in the system under test (SUT) is executed at least once in the test suite. However, by simply executing all statements in the program does not guarantee that faults could be found. For example, covering only one side of a conditional statement will not cover all statements. Another improved coverage metric, known as branch (decision) coverage, offers more accurate results than line coverage. Instead of relying on the number of lines of code, branch coverage focuses on control structures such as conditional statements, measuring how many of these structures are exercised by at least one test case in the test suite. The most commonly used coverage metrics in practice are statement coverage and branch coverage [153].

Coverage can be measured at different levels within the program and includes the following:

- Statement / Line coverage: The number of lines of code covered by the test suite.
- Branch coverage: The number of branches (conditional statements) covered by the test suite (e.g., "true" and "false" branches).
- Method coverage: The number of methods covered by the test suite. Measuring method coverage requires less effort than line coverage.

```
@Test
public void testAdd() {
    Example example = new Example();
    example.add("item1");
    example.add("item2");
    String testOutput = example.sort();
    assertEquals("item1,item2",testOutput);
}
```

Figure 2.2: *A simple JUnit test case consisting of four main components: test inputs, method-under-test invocation, test output, and an assertion*

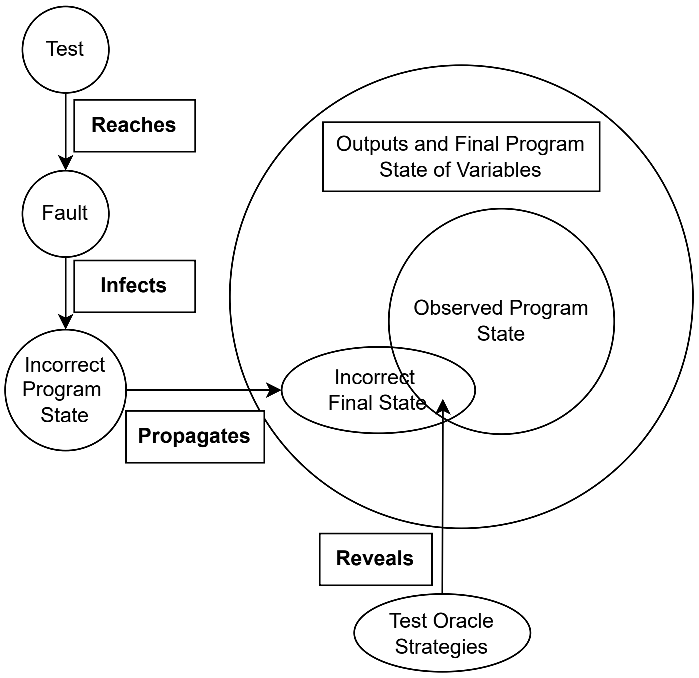


Figure 2.3: *RIPR model. Taken from [67]*

- Class coverage: The number of classes covered by the test suite. This is the simplest form of coverage.

It is often assumed that higher code coverage indicates that it already meets the criteria of having a high-quality and *healthy* test suite. However, a fault will only be detected if a test case executes the line(s) in which the fault resides, and if an assertion is present to check the validity of the expected output, meaning that there is a possibility of having a test case that executes the lines of faulty code without revealing the fault.

Fault Propagation Problem

Based on the Reachability, Infection, Propagation, Revealability (RIPR) model [67], detecting a fault involves four steps:

1. Reachability: The location that contains the software fault must be reached. As shown in Figure 2.4.1.
2. Infection: The state of the program must be incorrect after executing the location of the software fault. As shown in Figure 2.4.2.

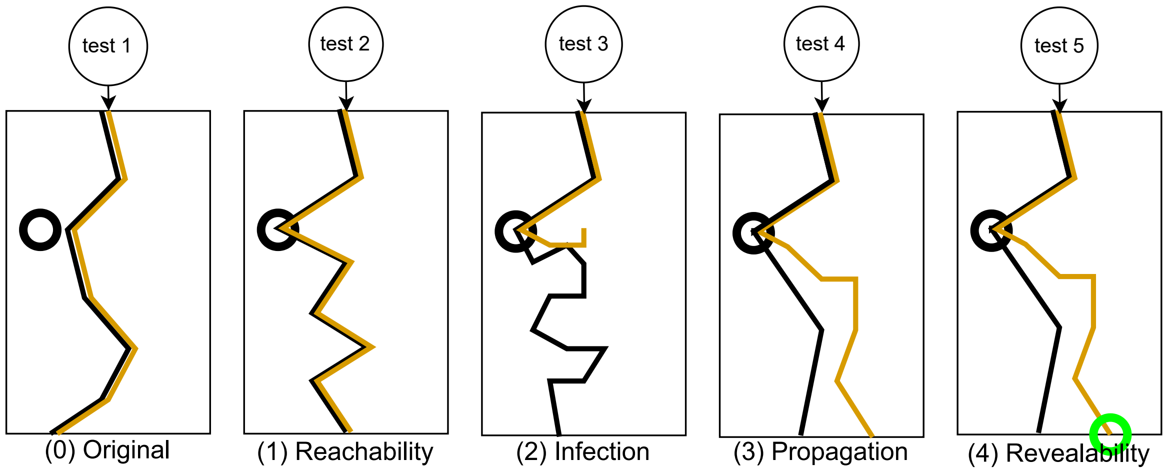


Figure 2.4: *Input that can satisfy the RIPR model. The black line shows the execution trace for the original program, while the yellow line shows if it is being executed on the modified program. The deviation of the two lines indicates that, for the same input, it starts to produce different states. The green circle is an observation on the output that can reveal the failure.*

3. Propagation: The infected state must propagate to cause the output to be incorrect. As shown in Figure 2.4.3.

4. Revealability: A test oracle that can expose the software *failure*—incorrect external behaviour relative to the specified requirements or expected behaviour [67]. As shown in Figure 2.4.4.

Figure 2.3 depicts the RIPR model. The model described is necessary to induce failures in the system, thus achieving only high code coverage is not enough.

2.4.2 Mutation Testing

Based on the RIPR model previously described, since code coverage may not be sufficient to evaluate the quality of a test suite [91], another way to assess the quality of a test suite is through mutation testing [154, 208]. Mutation testing, introduced by DeMillo et al. [110], is a fault-based testing technique that is used to evaluate the quality of a test suite by introducing small changes to the system under test (SUT) and validate if the test suite can detect the changes [69]. By syntactically changing part of the original program, it will create a faulty version of the program. This is inspired by the idea from Competent Programmer Hypothesis [59] where a competent programmer will develop programs that are close to the expected version.

Definitions in Mutation Testing

- **Killed Mutant**: A mutant M, is killed if test suite T, consists of at least one test that fails when run against the mutated program.
- **Survived Mutant**: A mutant M, survives if test suite T, does not have any test case that fails when run against the mutated program.
- **Equivalent Mutant**: A mutant M, of a program P, is equivalent if and only if M semantically equivalent to P.
- **Stubborn Mutant**: A mutant M, of a program P, is hard to kill as it has a limited range of inputs that could kill it.
- **Trivial Mutant**: A mutant M, of a program P, could be easily killed with certain inputs.

The small artificial changes are known as mutants and they are created by applying mutation operators to represent typical programming mistakes. Such operators are used to modify the original expression by inserting, replacing, and deleting certain primitive operators, and this includes:

- Arithmetic Operator Replacement: changing arithmetic operators (e.g., $+$, $-$, $*$, /) to other arithmetic operators.
- Relational Operator Replacement: changing relational operators (e.g., <, >, <=, >=) with another relational operators.
- Conditional Operator Replacement: changing the conditional operator (e.g., AND, XOR, OR) to other conditional operators.
- Statement deletion: removing a statement from the program.
- Replacing variables: replacing variables with other variables.

A mutant can be killed only if the mutated program produces altered output (Figure 2.4.3), and if there is an assertion (green circle in Figure 2.4.4) in the test that can ultimately detect the deviation in the program's output from the original expected result, causing the test to fail [113]. If the mutant could not be killed by any test case in the test suite, the mutant is considered to have survived.

Mutation testing concludes with a *mutation score*, which indicates the effectiveness of the test suites in catching all the artificial faults. Mutation score is the ratio of the total number of mutants that were killed by the test suite to the total number of mutants generated. A high mutation score indicates that the test suite is effective in detecting the mutants in the system. Mutation testing has been shown to be an effective technique for evaluating the quality of a test suite and has been used in many studies to evaluate the quality of test suites [154].

```
// original program
public boolean addLoop() {
    for (int i = 0; i < 10; i++) {
        add(i);
    }
}

// mutated version of the program
public boolean addLoop() {
    for (int i = 0; i != 10; i++) {
        add(i);
    }
}
```

Figure 2.5: *An example equivalent mutant. Taken from [154]*

However, there are some limitations to the usage of mutation testing. Firstly, mutation testing is computationally expensive, as it requires running the test suite against a large number of mutants [208]. As an example, given that a program with 100 mutants and a test suite that consists of 10 tests, it requires *at most* 1,000 test executions to evaluate all mutants, limiting the scalability of this technique. As a result, this has hinder the widespread adoption of mutation testing as a standard practice in the industry (e.g., Google [215, 214], Meta (Facebook) [81]). However, there has been some work in reducing the execution costs of mutation testing, leveraging multi-threading execution of mutants [133] or batching non-conflicting mutants [174].

Secondly, some mutants can be equivalent [183], meaning that the mutated version of the program is *semantically* equivalent to the original program [86]. A simple example of equivalent mutants is shown in Figure 2.5. In this example, although the program has been mutated (changing the operator from '$<$' to '$!=$'), the two programs are semantically equivalent, meaning that for every output, it will produce identical output. There has been some work in trying to detect equivalent mutants using symbolic techniques and mathematical models, however this problem is still undecidable [154] and could hinder the adoption of this technique in practice.

2.5 Regression Testing

As software programs *will* evolve over time, developers may be wondering what are the semantic impact of the syntactical changes to the program, and whether it will introduce new faults that affect the original program. One way of making sure that the original intended behaviour of the program remains the same, developers will execute existing test suite to the newly changed program. Any test failures can potentially indicate a regression [45] in the behaviour. This practice is also known as regression testing, in which it provides a way to ensure that the modifications made to the program do not introduce new faults and that the existing functionalities of the program are still working as expected. It provides confidence that new features or changes to the program do not interfere with the existing behaviours [48]. Developers may add new features or refactor[1] existing behaviour [120] and implementation details, potentially introducing unintended complexities that could impact the original test suite.

Software regression can be detected through re-running existing test suites that consist of test inputs and test oracle that can reveal semantic differences between the original program and the newer modified version of the program. In order to make sure that the regression test suite is in high quality and reliable [152], it relies on *at least* these three assumptions about the test suite:

1. A high-quality regression test suite that covers a wide range of the program behaviours, ensuring if the behaviour of the program changes, existing tests will catch the behavioural differences (discussed in Section 2.4).
2. Tests should behave deterministically when being executed on the same program, ensuring that it will produce reliable and consistent test results [182, 210].
3. Tests should only focus on exercising the behaviour of the system, ensuring that the tests are not tightly coupled with the implementation details, which could lead to false positive failures when the implementation details are updated [165, 83].

2.6 Test Suite Health

A high-quality and functional test suite that only achieves high code coverage and high mutation score does not guarantee a *healthy* test suite that can protect developers against regressions. In cases where a test suite is not healthy, it could lead to a situation where it is not able to detect faults in the system effectively. Instead, the test suite could become unreliable and might hinder developer's day-to-day activities [152]. Labuschagne et al. [166] also found that 64.62 % of 935 build failures resulted from faults in the production code. This means that more than a third (35.38 %) of failures were due to unreliable tests.

Removing unreliable test cases could tremendously affect the quality of the regression test suite, and faults could be missed. Therefore, making sure that unhealthy test cases are being repaired or improved is a crucial task [257]. Table 1.1 shows a list of indicators that should be considered when evaluating the "fitness" of a test suite. Minimally, a healthy test suite should (i) meet the test adequacy criteria (e.g., code coverage) that are required (Section 2.4), (ii) report any regression without giving false positive failures, (iii) can easily adapt or change, and (iv) gives quick feedback.

[1]Refactoring means improving the structure, readability, and efficiency of existing code without changing its behaviour. It helps make the code easier to maintain and more efficient.

```
@Test
public void testRsReportsWrongServerName() throws Exception {
    MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
    MiniHBaseClusterRegionServer firstServer =
    (MiniHBaseClusterRegionServer)cluster.getRegionServer(0);
    HServerInfo hsi = firstServer.getServerInfo();
    firstServer.setHServerInfo(...);

    // Sleep while the region server pings back
    Thread.sleep(2000);
    assertTrue(firstServer.isOnline());
    assertEquals(2,cluster.getLiveRegionServerThreads().size());
    ... // similarly for secondServer
}
```

Figure 2.6: *An example of flaky test caused by asynchronous wait. Taken from [182].*

2.6.1 Test Flakiness

A test is only useful if when being executed on the same program, it will produce the same outcome, meaning that the test is deterministic. Only at that point that the outcome truly representative of the test, making it reliable to reveal failures and to compare across various program versions. A common issue that can impact the health of a test suite, as reported by large software companies such as Meta [146] and Google [189], is the occurrence of non-deterministic test outcomes, also known as flaky tests [182]. A flaky test is a test that can be observed to have both passing and failing outcomes when executed multiple times even on the same code under test without any changes [210]. This is mainly caused by the non-deterministic behaviour of the program, the program execution environment [134], or the test itself [263].

Figure 2.6 is an example of a flaky JUnit test from HBase project. The test uses the `Thread.sleep(2000)` to wait for a server (`firstServer`) to ping back. The test is flaky because it does not explicitly wait for the asynchronous call to complete, relying on a fixed timing (2,000 ms), that could cause the test to sometimes fail. This type of test can hinder the productivity of developers, as they might spend time debugging the test, only to find out that the test is flaky. Rahman et al. also found that ignoring flaky test failures can have a bad effect on the software stability, which leads to a higher incidence of crash reports [217].

There are several studies conducted tries to categorise the root causes of flaky developer-written tests [115, 182, 141, 167]. The categories of root causes slightly differ between studies, mainly due to different programming languages. However, the common root causes of flaky tests still remain the same. Table 2.1 shows the common causes of flaky tests.

Table 2.1: *Different categories of root causes used to classify flaky tests in various literature.*

Category	**Description**
Asynchronous Waits	Test that makes an asynchronous call but does not explicitly wait for it to complete, relying on a fixed timing, leading to non-deterministic test outcomes. [182, 247, 115, 167, 141]
Concurrency	Test that involves multiple threads interacting in unpredictable ways, such as race conditions. [182, 247, 115, 167, 141]
Floating Point	Test that relies on the result of a floating-point operation can encounter discrepancies and inaccuracies, such as precision overflows, underflows, or non-associative addition. [182, 115, 167]
Input/Output (I/O)	Test that uses filesystem (handling input and output operations) can be flaky because of storage space limitations. [182, 115, 167, 141]
Network	Test that relies on a network connection, such as querying a web server, can become flaky if the network is unavailable or the resource is busy. [182, 167, 115, 141]
Non-Idempotent Outcome (NIO)	Test that is flaky due to shared state, e.g., a test that is executed twice fail on the second time because the state was polluted during the first execution and not properly reset. [251]
Ordering Dependency (OD)	Test that relies on a shared resource altered by another test may become flaky if the test execution order changes, leading to inconsistent outcomes.[182, 247, 115, 167]
Platform Dependency	Test that depends on specific OS features, library versions, or hardware can be flaky, as it may behave consistently on one platform but fail in others due to varying hardware or software setups. [247, 115, 141]
Randomness	Test that relies on the output of a random value may fail intermittently if it doesn't handle all potential outcomes. [182, 115, 167, 141]
Resource Leak	Test that mismanages external resources, like failing to release memory, can cause flakiness in later tests that try to reuse those resources. [182, 247, 115, 167, 141]
Test Case Timeout	Test case with an upper time limit can become flaky, as unpredictable execution times may cause it to fail if the test runs slower than expected. [115, 141]
Test Suite Timeout	Test in an upper time limit test suite may fail intermittently if it runs when the suite exceeds the time limit. [115]
Timing	Test that depends on the local system time can become flaky due to variations in precision or timezone settings. [182, 115, 167, 141]
Too Restricted Range	Test where some of the valid output range falls outside of what is accepted in its assertions. This test is flaky, since it does not account for corner cases and thus it may intermittently fail when they arise. [115, 141]
Unordered Collection	Test that assumes a specific order for an unordered collection can become flaky, as such collections have no guaranteed order. Tests relying on a fixed iteration sequence may fail unpredictably due to factors like the implementation of the collection class. [182, 167, 141]

There has also been extensive work in detecting flaky tests. The easiest way to detect flaky tests is by re-running the test multiple times. The inconsistent outcomes of the test after repeatedly executing them indicate that they are flaky. This approach has been universally adopted within large companies such as Microsoft [167] and Google [192], and in most of the popular testing frameworks such as Maven's Surefire Plugin [27] and PyTest [34].

2.6.2 Test Brittleness

Another problem that could affect the health of a test suite is brittle tests. As software code undergoes several changes during its lifetime, such changes might affect the tests. Brittle test is a test that fails when changes are made to the code under test that does not introduce any real faults [165, 152]. Similar to flaky tests, it could hinder the productivity of developers, as they might spend time debugging tests that are failing and not related to the expected behaviour of the system.

Direct Testing of Implementation Details

Brittle tests can be caused by the test being too tightly coupled with the implementation details of the code under test, making it sensitive to changes that are not related to the fault to the system. Bowes et al. [83] suggested that tests should only focus on behaviour of the code under test, rather than the implementation details. The main reason for this is if implementation details are changed or updated, the test becomes obsolete and outdated, and it could lead to tests that breaks easily or produce false positive failures [199]. However, if the test focus on expected behaviour and utilising the public interface, it will avoid such problems [165]. Therefore, it is recommended to focus on the behaviour rather than the implementation details [83].

Most programming languages clearly define where a public interface should reside—typically using the "`public`" access modifier keyword. However, they differ in how they handle the placement of different types of implementation details, also known as non-public members. Most of object-oriented programming languages, such as C++ and Java, support the idea of strict encapsulation [243], which allows developers to clearly separate the parts of the code that can be used externally from those that contain internal implementation details [85]. While all programming languages provide public visibility, each language defines its own rules for the "non-public" levels of access.

For example, as shown in Table 2.2, Java has three levels of access modifiers for implementation details: "`protected`", "`package-private`" (default), and "`private`" [7, 137]. These three levels offer different degrees of accessibility, determining how and from where the method can interact with the internal members of the class [231]. These levels are not considered part of the Public API, as they are inaccessible to outside non-subclass packages (World). The highest level of visibility, '`public`', allows access from anywhere, including those in other applications.

This approach of having more than one level of visibility for implementation details is similar across many other programming languages, such as C++ [2], C# [1], and PHP [42], though its varying degrees are different between languages. For instance, C# has five levels of non-public access modifiers [1]. Nevertheless, in general, most of these languages follow a common pattern with three main levels: public, protected, and private. Public members are globally and externally visible, protected members are usually accessible only within the system, and private members are restricted to the class or file itself.

In more modern dynamically-typed programming like Python, it does not have any built-in notion to define a non-public member, or even a mechanism to differentiate between public interface and non-public member [40]. Instead, developers may adopt conventions such as using underscore ("_") to indicate that a member is non-public, though it is not enforced. However, since this is purely convention and not enforced by the language itself, it is harder to differentiate between public interface and implementation details.

Despite the strict encapsulation strategy found in most object-oriented programming languages, there are ways to bypass the access control mechanism to access non-public members. For instance, reflection APIs in languages such as Java [178], Ruby [118], C# [62] allows such access, while in C++, it is possible to grant special access to certain methods and variables using the "`friend`" keyword [99].

The usage of these mechanisms is generally discouraged, as they can lead to brittle tests that are tightly coupled with the implementation details of the code under test. A study by Yang et al. [255] identified it as a test smell (poor testing practices that are observable in the source code of test suites that can hinder its maintenance [80, 248]) called "Private Method Test", they identified JUnit tests that directly invoke private methods. Since private methods are not visible outside their class, they are also not visible to JUnit tests either. Another test smell known as the "Anal Probe" smell, discussed in a study by Martins et al. [184], broadly refers to a test that "has to use insane, illegal or otherwise unhealthy ways to perform its task" [4]. This could involve accessing private members of the class, such as through the Java Reflection mechanism, which would typically be inaccessible, or manipulating access by extending a class to reach protected members, or placing the test within a specific package to access package-private members, which would otherwise be inaccessible. The "X-Ray Specs" smell [44], as described by Garousi et al.[130], refers to tests that access or alter the internal state of components that should remain hidden. Besides that, Van Deursen et al. [248] also identified "For Testers Only" test smell, where developers write production code that is not intended to be used anywhere in the system and only to support testing. This enables developers to focus on writing test for the implementation instead of exercising its behaviour through its public API. These smells are considered unhealthy because they violate the encapsulation principle, making the test suite more brittle and less reliable.

Table 2.2: *Visibility of different level of access modifier in Java.*

	Public API	Implementation Details		
Visbility	`public`	`protected`	`package-private`	`private`
Same Class	✓	✓	✓	✓
Same Package Subclass	✓	✓	✓	
Same Package Non-Subclass	✓	✓	✓	
Different Package Subclass	✓	✓		
World (Public API)	✓			

Brittleness in Test Assertions

Huo and Clause [150] found that tests which consist of assertion statements that check for values that its test input does not control could lead to brittle assertions and can lead to dependency on other tests to be executed first [262]. They found that if a previously executed test case that modifies the default values (e.g., field values) of the system, the following test case that made an assertion on the assumed default values could potentially fail, due to it has been updated in the previous test.

Verifying Interaction Test Cases Rather Than State

Another reason that a test could also be brittle is that, instead of trying to verify the state of the system, a test tries to verify the expected sequence of actions (interactions) when it is being invoked. Similar to the problem of directly testing implementation details (Section 2.6.2), rather than validating *what* is the result, it checks *how* the system reaches to the result. Carino et al. [90] found that Graphical User Interface (GUI) recorded test cases can easily "break" if changes are made to the GUI. They developed Blackhorse, a tool that removes coupling between the GUI components and test cases, making sure that the modified test cases are not too tightly coupled to the state to exercise the system.

2.6.3 Test Suite Health Summary

Table 1.1 summarises some other key indicators that could affect the health of a test suite that will not be covered in detail in this thesis. Some other examples are the variability of indicator metrics (e.g., Hilton et al. [148] looked into the variability of code coverage results, Alshammari et al. [66] looked into flaky failures when applying mutation testing), and environmental-dependent tests (e.g., tests that focused on certain OS environment [155] or failing on CI/CD environment [166, 241]). These indicators could potentially affect the overall health of a test suite, and it is important to consider them to ensure that the test suite is more reliable.

2.7 Automatic Test Generation

Automated test generation tools have seen a lot of improvement over the last few years. It has lowered the cost of testing software process, while giving equal or better quality in terms of test adequacy criteria when compared to the manually written test. For the next three sections, I will discuss some of the techniques that have been taken to generate test suites automatically:

1. Random Test Generation: Generate test data randomly to test the program (Section 2.7.1).

2. Symbolic Execution: Symbolic execution test generation is performed by solving the collected constraints using the constraint solver (Section 2.7.2).

3. Search-Based Test Generation: Dynamically generate test data using meta-heuristic search techniques (Section 2.7.3).

2.7.1 Random Test Generation

The easiest and simplest form of test generation is random testing. By generating a random sequence of inputs, executing it to the program, and observing the output, it could be used to find faults in the program. This technique can be useful if the program specification is not available or for identifying unknown security faults [193] (e.g., fuzzing [259]). A commonly used random unit test generation tool, Randoop [202, 200, 201], uses random testing with a feedback-directed approach to generate test cases. Randoop generates statements one by one and executes them to ensure that they do not throw any errors. It uses the feedback from previously generated test inputs to generate new sequence of inputs that avoid illegal statements or detect contract violations.

Chen [93] suggested an extension to random testing, known as *adaptive random testing* (ART) to make sure that test inputs are evenly spread across the input domain. ART has been studied widely [92, 149] and has shown to be as effective as traditional random testing. However, Arcuri and Briand [72] conducted a large empirical study and found that it is not as cost-effective when compared to generating tests randomly, as it could be done cheaply and quickly.

2.7.2 Symbolic Execution

A popular white-box testing strategy, known as symbolic execution [160], generates test inputs by exploring program paths symbolically rather than using concrete inputs. It replaces actual values with symbolic variables and traces all possible execution paths. For example, an integer value like 5 is replaced with a symbol χ, which then generates a symbolic representation of the output value. As it explores the program paths, symbolic execution collects constraints (path conditions) that must be satisfied to reach a particular path.

The key advantage of symbolic execution in test generation is its ability to automatically derive inputs that systematically explore different paths in the program. It generates test inputs that cover a wide range of program behaviours, making it effective for improving code coverage. By solving the path conditions with constraint solvers, symbolic execution produces concrete test cases that are guaranteed to exercise specific execution paths.

A major challenge with symbolic execution is its difficulty in covering complex program paths, especially with a limited time. Path explosion [88], where the number of paths grows exponentially, can occur when applied to a bigger project. Besides that, symbolic execution tools often struggle to handle complex objects—data structures with intricate internal states, such as nested components and dynamic interdependencies [77]. This complexity makes it harder for the tool to accurately model all possible states [88], amplifying the path explosion problem and limiting the effectiveness of symbolic execution in test generation.

To mitigate these limitations, symbolic execution is often combined with other dynamic testing techniques, such as fuzzing, which is also known as dynamic symbolic execution [134, 87, 234], it can help overcome some of these limitations. Dynamic symbolic execution allows the symbolic execution to handle parts of the program that it handles well, while using fuzzer to involve in areas that are not covered by symbolic execution.

2.7.3 Search-Based Test Generation

Automation has taken a big role in reducing the cost of doing software testing [64]. For it to achieve an optimal solution while being inexpensive, it needs to reduce the selection of test inputs. While random testing tries to exhaustively apply all possible inputs, it is infeasible due to many possibilities of inputs. Therefore, choosing the test inputs can be also seen as a search problem, and search algorithms are used to derive test data [186]. Testers usually have a target to achieve, whether on finding crashes, maximising code coverage, looking for security vulnerabilities, or even detecting known faults in a program. In the infinite space of inputs, it is a good idea for testers to find inputs that could potentially meet the target that they would want to achieve.

Search-based software testing [144, 145] is known for utilising optimisation techniques such as hill-climbing, simulated annealing, and genetic algorithms for generating inputs to optimise the fitness function. A fitness function is a way to evaluate how close a potential solution is the optimal solution of a problem. It provides a numerical score that reflects the effectiveness of a solution to meet certain criteria.

The usage of meta-heuristic techniques has seen a lot of improvements in automated test generation. It could be guided by one or more fitness functions for scoring the optimality of the chosen inputs [229]. With a goal on the purpose of doing the testing, a

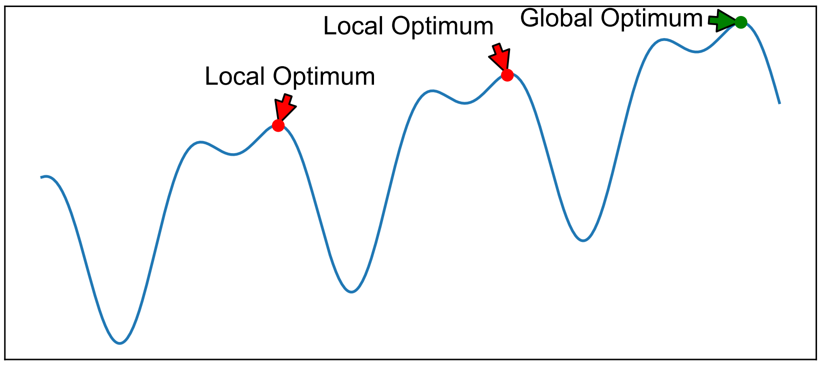


Figure 2.7: *Local optimum problem in hill climbing.*

meta-heuristic optimisation algorithm can systematically search for the possible inputs guided by the fitness function(s). Inside the software engineering area, search-based test generation has been used commonly for automatically generating test inputs [186]. It is a crucial automatic test generation technique that has been widely used in research.

Meta-Heuristic Search Techniques

Hill Climbing. Hill Climbing is a well-known simple search algorithm. The technique will start by finding an initial solution randomly as the starting point and investigating the closest neighbour of the solution. The initial solution will be replaced by a neighbour with a better solution, which is repeated until no better solution can be found around the neighbouring area.

As from the name of the technique, after finding the peak of the hill, the program will assume that is the best solution. We would assume that the solution would be improved over time and the chosen candidate will have better fitness when compared to the initial candidate. The process is simple, and it would give the best result in a short period of time.

However, there is a problem associated with this technique. As shown in Figure 2.7, the search might get stuck only on the peak of the initial candidate's neighbour as it only considers the neighbour from the peak. Search spaces are not usually linear. The chosen candidate could potentially be a local optimum and for a non-linear search space, there is a high possibility that it is not the global optimum. To solve this issue, there are some recommendations for running the hill-climbing technique multiple times, known as multiple restart hill-climbing, with the assumption that the initial solution checks for other areas too.

Simulated Annealing. Instead of only considering looking at a neighbour where the fitness is increasing, simulated annealing [191] is another technique that permits the movement of the fewer fit individuals. Simulated annealing is inspired by the simulation of metallurgical annealing. The idea is that when a highly heated metal is slowly cooled off, the strength could still be increased. It has more freedom to move around the search space. This technique solves the problem of only depending on the starting solution, where it could be chosen badly. The approach will give freedom in the initial stage and slowly follow the hill-climbing approach for the search space. However, if the approach starts to cool off quickly, the search space starts to narrow down quickly, thus making it stuck in a local optimum. This is similar to the hill-climbing problem [186].

Algorithm 1 A high-level description of Genetic Algorithm

```
    Set generation number, m = 0
    Initial population of candidate solutions, P(0)
    Evaluate the fitness for each individual of P(0), F(P(0))
1: GenerateRandomPopulation
2: repeat
3: $Recombination : P(m) = Recombine(P(m))$
4: $Mutation : P(m) = Mutate(P(m))$
5: $Evaluation : F(P(m))$: Compute the fitness for each solution
6: $Selection : P(m+1) = Select(P(m))$
7: $m = m + 1$
8: exit when goal or stopping criteria is met
9: stop
```

Genetic Algorithm. Genetic algorithm [143, 135] is the most used from the family of meta-heuristic search (evolutionary) algorithms. It follows closely the nature of Darwin's "survival of the fittest" theory [242], where a population of initial randomly generated candidate solutions is evolved using genetically inspired search operators. These initial solutions are usually not the best for solving the target problems. To increase the quality of the solutions, it then reproduces more individuals using genetic operators such as crossover, mutation, and selection to improve the initial candidates to better candidate solutions which are guided by the measurement of the fitness function. During crossover in genetic algorithms, two candidates are randomly split into halves, and their segments are combined to produce a new candidate. For mutation technique in genetic algorithm, random modifications are applied to existing candidates to introduce new variations that explore new solutions. To reach beyond the initial "parent" population, new random candidate solutions will also be added to the pool of the population and the process mentioned previously will be repeated. The improvement will be generated on multiple stages and when the search is stopped, hopefully, the optimal solution will be found. The stopping condition will depend on the user, such as time limit, or found the best solution. Other stopping conditions include the maximum number of generations or meeting fitness goals that it desires. Algorithm 1 represents the generic genetic algorithm.

Search-Based Unit Test Generation

Based on Section 2.7.3, it has shown that it is possible to create a very close representation of an ultimate solution. Those techniques are suited for generating unit tests. In this section, I will discuss some of the different approaches that have been done with applying this technique.

Random Search Test Generation. The simplest strategy to create a new test case is by simply generate sequences of statements to the program randomly. If the randomly generated sequence of inputs (test case) found a new branch, it will be added to the test suite, else remove if it did not. This simple approach is similar to the well-known American Fuzzy Lop (AFL) fuzzer [259], where inputs that discover new paths will be kept in the seed. However, for a branch that has a low probability of being executed, they are unlikely to be covered. In order to solve this problem is to enhance the random search by using pre-seeding strategy, where initial inputs are given before starting random search [223].

Seeding Strategy. The process of seeding involves providing an initial set of inputs to navigate the search process towards certain input values that have a higher chance to execute the engineering goals [147]. The initial set of inputs are usually chosen based on previous related knowledge to help solve the problem. In the context of test generation, Fraser and Arcuri [123] proposed gathering seeds statically and dynamically. Static seeding is done by collecting all primitive and string values that appears in the Java bytecode of the class under test. The static seeding approach is similar to what have been proposed by McMinn et al. [188], to use seed values from source code or documentation with the goal to reduce human oracle costs. Primitive and string values that are encountered during the execution (runtime) of the program may also be added to the seed (dynamic seeding). Another strategy that has been proposed by Danglot et al. [106] is to use the existing developer-written test suites to generate new test cases to improve existing developer-written test, which is known as test amplification. Seeding strategy has also been widely used in fuzz testing [219] and found it to be very useful approach.

Evolutionary Search for Test Generation. While random search even with good seeding strategy still relies on encountering new solutions by luck, guided searches aim to find best solutions more directly by using a problem specific “fitness function”. Usage of the search-based algorithm, to be more precise genetic algorithm, for generating unit tests can produce a high quality test suite when compared to the random search.

EvoSuite [124] and Pynguin [180] is the state-of-the-art unit test generation tool that uses a search-based approach to automatically generate test suites to achieve certain test goals (e.g., code coverage). The generated test suite also includes test cases with assertions to verify the expected and actual results. It uses the genetic algorithm to generate test cases that evolves using the evolutionary search and ends with the best optimal test suite. It has also been used to generate test cases for many projects in different domains [157].

Mutation-Test Based Generation. In order to kill a mutant, a test must meet three key conditions: it must be able to reach the mutant, infect the propagation state, the infection needs to be manifested to the program output, and an oracle that could reveal the failure. This is explained in the RIPR [176] formula in Section 2.4.1. These conditions can be used as a fitness function to guide the generation of test cases.

Botacci [82] adopted the fitness function concept of the search-based approach to generate tests capable of killing mutants. Fraser and Zeller [128] implemented similar idea in their tool, μTest, which creates unit tests with assertions aimed at killing mutants. The fitness is measured by combining the distance to execute the mutated statement, the distance of the mutated statement to be reached, and the impact significance of the mutant on the rest of the execution.

In EvoSuite, Fraser and Arcuri [125] used the similar approach as μTest, applying genetic algorithms for weak and strong mutations. It is done by measuring the mutants' impact [128] to satisfy one of the conditions, which is propagation. The mutants' impact is the number of statements with changed paths, between the mutated program and the original program. This is an extension from the Weak Mutation Testing where it only checks the node coverage (Figure 2.4.2). Since a unit test needs to be observed using test assertions, whether it is failing or not, μTest needs to consider the propagation (Figure 2.4.3). The propagation is calculated by observing (Figure 2.4.4) the difference of control flow and data between the original and mutated program when the test is run. The more differences in the infection, the higher chance of it being propagated to different outputs.

There are three parts of fitness function that need to be combined to satisfy the strong mutation testing technique. Using the function proposed by Arcuri [71], where $v(x)$ is a normalising function from $[0, 1]$:

- How close for each branch to be executed:

$$d_{branch\_coverage}(branch, test) = \begin{cases} 0, & \text{if branch has been covered} \\ v(d_{min}(branch, test)), & \text{if predicate executed} \geq 2 \\ 1, & \text{otherwise} \end{cases}$$

  The requirement to satisfy only branch coverage is as follows:

$$f_B(T) = |F| + |F_T| + \sum_{b_k \in B} d_{branch\_coverage}(b_k, T)$$

- How close towards the state of infection:

$$d_{infection}(mutant, test) = \begin{cases} 1, & \text{if mutant not reached} \\ v(d_{min}(mutant, test)), & \text{if mutant reached} \end{cases}$$

The requirement to satisfy the weak mutation is as follows:

$$f_{WM}(T) = f_B(T) + \sum_{M_k \in M} d_{infection}(M_k, T)$$

- How significant the impact of the mutant on remaining execution:

$$d_{propagation}(mutant, test) = \begin{cases} 0, & \text{if mutant can be asserted} \\ 1, & \text{if } d_{infection}(mutant, test) > 1 \\ \dfrac{1}{1 + impact_{min}(mutant, test)}, & \text{if } d_{infection}(mutant, test) = 0 \end{cases}$$

The requirement to satisfy the strong mutation is as follows:

$$f_{SM}(T) = f_B(T) + \sum_{M_k \in M} (d_{infection}(M_k, T) + d_{propagation}(M_k, T))$$

2.8 Test Amplification

As discussed in the previous section (Section 2.7), the software testing research community has developed new techniques [186, 202, 230] to automatically generate new test cases from scratch. However, due to hard problems such as the test oracle problem [79], it is still not yet able to replace humans fully. This means that developers are still mainly *responsible* to write their own test manually [70]. *Test amplification* is a technique which utilises the presence of developer-written tests, and automatically enhancing them to improve quality that can reveal additional failures or modify existing tests to ensure their effectiveness and reliability.

Based on a survey by Danglot et al. [105], test amplification is defined as:

> *"Test amplification consists of exploiting the knowledge of a large number of test cases, in which developers embed meaningful input data and expected properties in the form of oracles, in order to enhance these manually written tests with respect to an engineering goal (e.g., improve coverage of changes or increase the accuracy of fault localisation)."* [105]

Two of the main goals of test amplification that will be covered in this thesis are:

(i) adding new test cases to improve the required test adequacy criteria (e.g., code coverage, mutation score) on existing or newer version of program, and

(ii) alter existing test cases to improve test adequacy criteria, or replace unstable (flaky) outcomes and unreliable results that could cause false-positive failures (brittle).

By improving the quality of tests, test amplification can directly contribute to enhancing the overall health of a test suite. It can address important indicators such as low code coverage and low mutation scores which are critical to improve the effectiveness of a test suite. Besides that, amplifying original developer-written tests can reduce the maintenance burden by modifying brittle test to ensure the long-term reliability of the test suite. In this way, test amplification serves as a powerful tool for improving test suite health, complementing existing developer-written tests.

2.8.1 Adding New Tests

Most of the test amplification tools that are available focused on adding new test cases by modifying existing test cases to improve the quality of test suites [58, 232, 105, 106, 84]. As an example, a developer that has a poor quality written test suite that only covers certain parts of the program can use test amplification tool to increase code coverage of the existing test suite by adding new test.

The state-of-the-art *developer-centric* [84] test amplification tool, known as DSpot [106], amplify existing JUnit test suites by adding new test cases that could improve the quality (e.g., code coverage, mutation score) of the test suite. The foundation of DSpot is mainly inspired by two notable test generation techniques: (i) Tonella's evolutionary test input space exploration [246], and (ii) the regression assertion (oracle) improvement work of Xie [254].

Inspired by Tonella's search exploration, DSpot's input amplification (*I-Amplification*) is the process of generating new test inputs by mutating the original tests that includes:

1. **Literals amplification:** changing literal values used in the tests. As an example, changing the value of a string by removing or replacing existing characters, and adding new characters. For integer values, it could be increasing or decreasing the value. For boolean values, it could be changed by flipping the original value.

2. **Method calls amplification:** changing, removing, or adding accessible method call(s) in the existing test cases.

3. **Test objects amplification:** changing, removing, or adding test object(s) in existing test cases.

Xie's regression assertion improvement, also refer to as assertion amplification (*A-Amplification*), is the process of generating new assertions by creating "observation points" [79] during the execution of the tests. The observation points are used to capture the current state of the system at the specific point. New regression assertions will then be generated by comparing the value at the observation points with the expected value.

Algorithm 2 outlines the steps for DSpot to amplify existing developer-written test suites. The main inputs for DSpot are the developer-written test suite `TS` and the program under test P. In the first stage, DSpot will first add assertions to the existing test case (Line 3), which significantly improve the quality of the test case to catch faults. After that, it applies *I-Amplification* (Line 9) and *A-Amplification* (Line 11) technique

Algorithm 2 Main overview of DSpot's amplification loop. Taken from [106]

```
    Input: Program $P$, $n$ number of iterations of DSpot's main loop
    Input: Developer-Written Test Suite $TS$
    Input: Amplifiers $amps$ to generate new test data input
    Output: An Amplified Test Suite $ATS$

 1: ATS ← 0
 2: for t in TS do
 3:     U ← generateAssertions(t)                                   ▷ Amplify Assertions
 4:     ATS ← {$x$ ∈ U|x improves mutation score / code coverage}
 5:     TMP ← $ATS$
 6:     for i = 0 to n do
 7:         V ← []
 8:         for amp in amps do
 9:             V ← $V \bigcup amp.apply$(TMP)                        ▷ Amplify Inputs
10:         end for
11:         V ← generateAssertions(V)                               ▷ Amplify Assertions
12:         ATS ← ATS $\bigcup${x ∈ U|x improves mutation score / code coverage}
13:         TMP ← V
14:     end for
15: end forreturn ATS
```

n times to explore areas that has not been covered by the existing developer-written test suite. It then produces a new test suite `ATS` that achieves a better code coverage or mutation score.

Similar to DSpot, Small-amp [58] is a test amplification tool for Smalltalk programming language, and Ampyfier [232] is for Python. Both uses the same approach as DSpot to amplify existing test suites. However, as both Smalltalk and Python are dynamically-typed programming languages, it has limited information about the type declaration of the variables in the production code, which is a similar issue found in automatic test generation tool from scratch, Pynguin [181] and Syntext-JavaScript [198]. Both Small-amp and Ampyfier used dynamic type profiling on the existing test suite to capture the type arguments on the production method, and the variable types in the test cases.

Rojas et al. [223] have also proposed different seeding strategies to guide EvoSuite to generate new test cases. The seeding strategies include: using constant values, dynamic values, and concrete types from the SUT, and re-use of objects from existing test cases as seeds. They also found that the proposed re-use of objects from existing test cases seeding strategy performed 2 % better on average in terms of code coverage, when compared to generating test from scratch. This approach is similar to the test amplification technique, rather than start generating tests from scratch randomly, it uses the knowledge of SUT to guide the search for new test cases. However, in the context of test amplification, the seeding strategy is using existing developer-written tests.

2.8.2 Altering Existing Test Cases

Test amplification can also be used to repair or alter existing test cases that are either not in high quality (e.g., code coverage, mutation score), broken due to changes to production code, or not healthy (e.g., brittleness, flakiness).

Improving Quality of Existing Tests

In trying to improve the code coverage of existing test suite, Dallmeier et al. [104] amplify existing tests by adding and removing method being called in JUnit test cases. Their main objective is to modify existing test case to widen the set of execution (coverage) states of the original test suite. Orstra, a tool developed by Xie [254], alter existing test cases by adding new assertions on the observed return value of the method being called and the state of parameters. This is done by instrumenting the CUT, execute the test suite, and collect the state of objects, where assertions are created based on the recorded object state. Xie uses 11 Java classes to evaluate Orstra and shows that it effectively improve the fault-detection capability of the amplified test suite. AutoAssert [260] is a human-in-the-loop assertion generation tool that is able to generate new and modify existing assertions based on the runtime values and feedback from the developers. Zamprogno et al. [260] evaluated AutoAssert on developer-written test cases of 105 JavaScript and TypeScript open-source projects and surveyed developers who used the tool, where they found that AutoAssert is useful tool to improve the assertions for validating the program behaviour.

Evolving Software (Regression) Test Repair

The first work that repair broken JUnit test cases was proposed by Daniel et al. [108]. They developed a tool known as *ReAssert*, that replaces the expected values in assertion statements with observed values during run time. This is done by applying a set of heuristic strategies. They also extended their work [107] to use symbolic execution strategies to overcome some limitations of the previous work when repairing test cases. Symbolic constraints are built based on the literal values which can maintain the assertion's expected value. The symbolic constraints are then being updated to literal values. The test is considered unrepairable for any cases where there are no candidates of solution exists. Another tool known as *TestCaseAssistant*, developed by Mirzaaghaei et al. [195], also repair broken JUnit test cases that are caused by changes (addition, deletion, or modification) to method parameters (signatures) and extension of the existing interfaces. It repairs broken method call by replacing literal values and variables within the broken test to identify candidate of repairs. Li et al. [177] proposed a tool, known as *TRIP*, to repair non-compilable broken Java tests after changes made to production code by using a dynamic symbolic execution technique to find solutions that preserve the original intent—the path conditions—of the broken tests. Yaraghi et al. [256] also proposed a tool, called *TARGET*, that uses pre-trained large language models (LLM)—specifically fine-tuned on Java and JUnit tests—to repair broken test cases. This technique is able to maintain the original quality of the test suite and not discarding broken test cases due to changes in the associated test case production code.

Order-Dependent Test Repair

Shi et al. [240] proposed a tool known as *iFixFlakies* that will recommend fix patches for problematic tests due to test ordering problems. *iFixFlakies* will identify tests that can pass when being run in isolation but fail when other tests are being executed before (victim). *iFixFlakies* then reset or set shared states (*cleaner*) between tests to "clean" the state pollution. However, *iFixFlakies* can only repair patches if there exists polluted shared states in the test suite itself. Li et al. [175] proposed a tool called *ODRepair* to address this problem of *iFixFlakies*, by generating a *cleaner* that will repair order-dependent test cases by using Randoop [202] to generate test input sequences to reset the state. In order to verify that the *cleaner* is able to repair the order-dependent tests, *ODRepair* will pass the generated tests to *iFixFlakies* and execute them again.

Flaky Test Repair

In trying to repair flaky tests attributed to the most frequent category discovered in Microsoft's distributed system, *asynchronous waits* category [182], Lam et al. [167] proposed a tool known as *FaTB* (Flakiness and Time Balancer). It will repair asynchronous waits flaky tests by executing the test 100 times and implement a binary search to find the shortest waiting time so that it does not produce any flakiness asynchronous waits outcome. However, such strategy relies heavily on the specific machine that the test is being executed on, and it will definitely influence the suggested waiting time. FlakeSync, a tool developed by Rahman and Shi [218], automatically repair *asynchronous waits* category by introducing synchronisation in the test execution, making sure that rather than relying on a specific waiting time, it will wait until the asynchronous call is completed. DexFix [261] is another tool that is able to repair failing assertions due to flakiness by applying templated-based repair strategies that is related to non-determinism. The repairs mostly include changing assertions for ordered collections. Lastly, Chen et al. [94] proposed a tool known as *FLAKYDOCTOR* that repair order-dependent tests and flaky tests that are assuming some ordering in an unordered collection by applying large language models (LLM) to create a patch for repairing the flakiness and validate the patch by compiling the generated patch. If the generated patch unable to be compiled, the approach of regenerating again using LLM is applied. The approach of finding the patch is repeated five times.

2.9 Using EvoSuite to Improve Test Suite Health

EvoSuite [122] is the state-of-the-art search-based test generation tool widely used to generate test suites for Java programs. It has been shown to be able to generate test suites that can achieve high code coverage [221] and mutation score [127]. Additionally, EvoSuite's flexibility has enabled it to be extended in many ways, incorporating various features and techniques [121].

EvoSuite also supports modification to the existing search technique to improve the quality the test suite. For instance, Panichella et al. [204] employs many-objective optimisation technique and Galeotti et al. [129] combines dynamic symbolic execution (Section 2.7.2) approach with search-based technique. Furthermore, EvoSuite also supports multiple seeding strategies [228, 123, 223], adapting new fitness criteria [221, 132, 264], usage of mocking framework to create mock objects that are not easily covered [76], and improving the understandability and readability of the tests [103, 109]. EvoSuite also offers built-in mechanism—from 12 different parameters (Table 3.1)—that suppress non-deterministic outcome in the test that it generates.

These flakiness suppression mechanisms are part of four groups: Test Creation, Sandbox, Test Output, and Test Execution.

- Test Creation group is mainly responsible for ensuring that the static fields are being reset, which this includes the get fields and final fields. This is to ensure

Table 2.3: *Properties for ensuring deterministic behaviour and mitigating flaky outcome in EvoSuite-generated tests.*

Family	**Parameter**	**Description**
Test Creation	Reset Static Fields	Call static constructors only after each static field was modified
	Reset Static Field Gets	Call static constructors also after each static field was read
	Reset Static Final Fields	Remove the static modified in target fields
Sandbox	Sandbox	Executing test cases in an independent testing environment
	Virtual FS (File System)	Using a virtual file system for all File I/O operations
	Virtual Net (Network)	Using a virtual network for all TCP/UDP communications
Test Output	Test Scaffolding	Generate the scaffolding needed to run EvoSuite JUnit tests in a separate file
	No Runtime Dependency	Avoid runtime dependencies in JUnit test
	JUnit Check	Compile and run the resulting JUnit test suite
Test Execution	Replace Calls	Replace any non-deterministic calls/invocation and `System.exit`
	Replace System In	Replacing the InputStream (`System.in`) mechanism with a smart stub/mock
	Replace GUI	Replacing the GUI (`java.swing`) calls with a smart stub/mock

that the static fields are reset before each test case is executed [262].

- Sandbox group consists of three parameters: Sandbox, Virtual FS (File System), and Virtual Net (Network). These parameters are related to executing test cases in an independent testing environment to avoid unsafe operations and restricted security manager, replacing any File I/O operations (only for write, delete, or execute filesystem) with a virtual file system to prevent the real file system from being corrupted [125], and replacing all TCP/UDP communications with a virtual (mock) network [75].
- Test Output group consists of three parameters: Test Scaffolding, No Runtime Dependency, and JUnit Check. These parameters are related to generating all the scaffolding needed to run EvoSuite JUnit tests in a separate file, avoiding runtime dependencies in the JUnit test suite, and compiling and running the JUnit test suite again to ensure that the assertions still hold.
- Finally, the Test Execution group consists of three parameters: Replace Calls, Replace System In, and Replace GUI. These parameters are related to replacing any non-deterministic calls/invocation and `System.exit` [124], replacing the InputStream (`System.in`) mechanism with a smart stub/mock [74], and replacing the GUI (`java.swing`) calls with a smart stub/mock [74].

Due to EvoSuite's flexibility, I will primarily use it in this thesis to improve (amplify) the quality of existing Java developer-written test suite and repair unhealthy Java test cases. In addition, EvoSuite incorporates several default parameters as part of its post-processing step (e.g., minimisation) to ensure that the generated test suite is of good quality [206]. It also includes a timeout mechanism (default = 4,000 milliseconds) that penalises long-running test cases [124].

2.10 Summary

In this chapter, I have surveyed the literature that is closely related to the research topics that will be explored in this thesis.

Firstly, I discussed test adequacy criteria, which are essential to evaluate the quality of a test suite. Next, I examined aspects that are important to ensure that a test suite is healthy (making the test suite more reliable), which includes topics related to test flakiness, brittleness, and the execution time. I emphasised that achieving high code coverage or high mutation score alone is not enough without considering these aspects. I also reviewed literature on automated testing approach, including random testing, symbolic execution, and search-based software test generation. Finally, I explored the topic related to test amplification, focusing on techniques to improve or repair existing developer-written tests. While much of the work in this area has aimed to improve the quality of existing developer-written tests, efforts on improving the reliability aspect of developer-written test cases has been mostly focused on repairing flaky tests [210] or addressing regression test repairs [177].

Overall, I have provided a snapshot on the current state of the research to each following chapters and identified areas for which further work may be beneficial. Given my plan to utilise EvoSuite's search-based technique, the seeding option, and flexibility for modification, it is imporant to evaluate one critical aspect of test suite health: flakiness. Therefore, the next chapter will investigate the flakiness of tests generated by EvoSuite, which is a key step to ensure that it could be used as a test amplification tool.

Chapter 3

An Empirical Study of Flaky Tests in EvoSuite

The contents of this chapter is a part of "Martin Gruber, Muhammad Firhard Roslan, Owain Parry, Fabian Scharnböck, Phil McMinn, and Gordon Fraser. Do Automatic Test Generation Tools Generate Flaky Tests? In *International Conference on Software Engineering (ICSE)*, 2024". The first two authors contributed equally to this research work.

3.1 Introduction

In the previous chapter, I have explained the maintenance issue of developer-written tests and automatically generated tests, and one of the reliability problem is related to the non-deterministic outcome of the tests, even when there are no changes to the code under test. This non-determinism is commonly known as flaky test [210], where a test could pass or fail without any changes. In this chapter, I will look into specifically the flakiness issue of tests that are generated by EvoSuite.

Research around the use of EvoSuite to aid developers when writing unit tests has increased [222]. It has demonstrated good capabilities in achieving high code coverage, good mutation score, and good fault detection rates [237]. Furthermore, EvoSuite offers built-in mechanism—from 12 different parameters (Table 3.1)—that suppress flakiness in the generated test. However, the flakiness issue in automatically generated tests has not been widely studied as it has been in developer-written tests. For this, I will start by understanding flakiness issue in automatically generated tests and whether this problem is as common as has been found previously in developer-written tests.

3.2 Methodology

Figure 3.1 shows an overview of the experimental setup. Sections 3.2.1 to 3.2.9 correspond to the stages of the methodology depicted in Figure 3.1 and will be

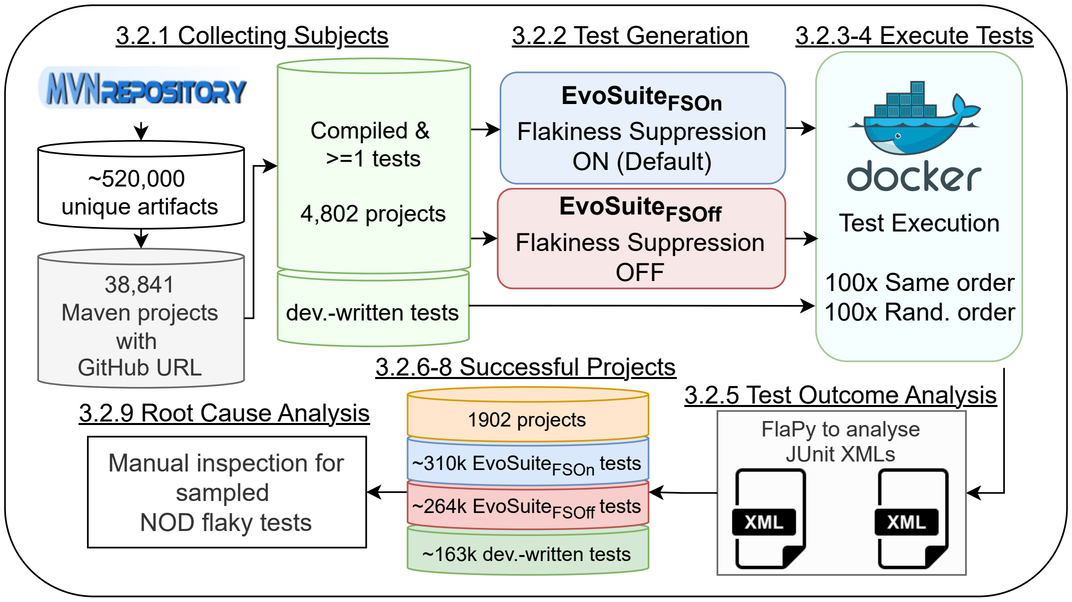

3.2.1 Collecting Subjects
MVNREPOSITORY
~520,000 unique artifacts
38,841 Maven projects with GitHub URL
Compiled & >=1 tests
4,802 projects
dev.-written tests
3.2.2 Test Generation
EvoSuite$_{FSOn}$ Flakiness Suppression ON (Default)
EvoSuite$_{FSOff}$ Flakiness Suppression OFF
3.2.3-4 Execute Tests
docker
Test Execution
100x Same order
100x Rand. order
3.2.5 Test Outcome Analysis
FlaPy to analyse JUnit XMLs
XML
XML
3.2.6-8 Successful Projects
1902 projects
~310k EvoSuite$_{FSOn}$ tests
~264k EvoSuite$_{FSOff}$ tests
~163k dev.-written tests
3.2.9 Root Cause Analysis
Manual inspection for sampled NOD flaky tests

Figure 3.1: *Overview of the experimental setup.*

discussed in detail in the subsequent sections. I set out to answer the following three research questions:

RQ1 (Prevalence): How prevalent is flakiness in tests that were generated without flakiness suppression mechanisms?

RQ2 (Flakiness Suppression): How many flaky tests can EvoSuite's flakiness suppression mechanism prevent?

RQ3 (Root Causes): How do the root causes of generated flaky tests differ from those of developer-written tests?

3.2.1 Subjects

To conduct this empirical study, I used the Maven Central Repository index [24] to collect Java projects. The index is updated every week for newly added projects and updated patches of existing projects. As of 26th of October 2022, the index contains roughly 520,000 unique artefacts. After fetching the unique artefacts, I iterated over each artefact's Project Object Model (POM) file to fetch the URL to the project's repository. To ensure that it could be easily replicated, I only selected projects that are hosted on GitHub (keeping the commit hash) and that use Maven as the build tool. Since the information of the build automation tool that a project uses is not being included in the POM in the Maven Central Repository, I statically crawled through

Table 3.1: *Updated parameters to deactivate flakiness suppression mechanisms (EvoSuite$_{FSOff}$). The description of each parameter is provided in Table 2.3.*

Family	**Parameter**	**Value**
Test Creation	Reset Static Fields	false
	Reset Static Field Gets	false
	Reset Static Final Fields	false
Sandbox	Sandbox	false
	Virtual FS (File System)	false
	Virtual Net (Network)	false
Test Output	Test Scaffolding	false
	No Runtime Dependency	**true**
	JUnit Check	false
Test Execution	Replace Calls	false
	Replace System In	false
	Replace GUI	false

the GitHub URL of each artefact to check if it contains a `pom.xml` file in the root of the repository to make sure that the project uses Maven as the build tool. From this, I was able to collect 38,841 Maven projects and use this as the starting point of the study. As I am interested in understanding the flakiness issue in automatically generated tests and tests written by developers, I also ensured that each project could firstly be compiled and contain at least one passing test case. The filtration of projects that could be compiled and contain at least one passing test case could be easily done by running the Maven compile (`mvn compile`) and test (`mvn test`) commands on each project. After running the commands, I found around 4,801 projects could be compiled and contained at least one passing test case.

3.2.2 EvoSuite

I used the state-of-the-art test generation tool, EvoSuite (v1.2.0), that utilises search-based techniques to generate JUnit test suites. As EvoSuite applies multiple techniques to avoid flakiness, I used EvoSuite with two different configurations: one with the flakiness suppression (EvoSuite$_{FSOn}$) and one without the flakiness suppression mechanism (EvoSuite$_{FSOff}$). This is to understand the impact of flakiness suppression mechanisms in EvoSuite. For the EvoSuite$_{FSOn}$, I did not update any parameters of EvoSuite, as the flakiness suppression is enabled by default. On the other hand, to generate tests without the flakiness suppression, I updated several EvoSuite parameters that were extracted from previous studies [74, 124, 121] and few discussions with one of the EvoSuite maintainers. Table 3.1 shows the updated parameters to deactivate flakiness suppression mechanisms in EvoSuite.

3.2.3 Search Budget

I set the search budget of EvoSuite to be 120 seconds per class for both EvoSuite$_{FSOn}$ and EvoSuite$_{FSOff}$. This ensures that the test generation process can be done within a reasonable time frame, which is similar time budget in previous tool competition have used [233, 250].

3.2.4 Execution of the EvoSuite-generated tests

To detect flakiness in the generated tests, I executed all developer-written and EvoSuite generated tests in each project 100 times in the same order and 100 times in random order. This is similar to what was done in previous studies [141, 168], and it also allows me to identify order-dependent (OD) and non-order-dependent (NOD) flakiness in the tests. The test execution was mainly inspired from iDFlakies [168] and FlaPy [140]. Both tools support the environment to execute tests, which utilises Docker container, to avoid any side effects from the environment, such as infrastructure flakiness [141].

Since Maven-build projects can easily install third-party dependencies of the project via the `'mvn dependency:copy-dependencies'` command, I used this command to make a copy of all the dependencies from the repository into my local machine. I then update the environment variables of the Docker container to include all the dependencies of the projects when executing the tests. This is to ensure that the tests can be executed without any issues related to missing dependencies.

I built a custom JUnit Runner [55] that could execute the tests in the same order and random order. The custom JUnit Runner is built using the JUnit 4 library and the JUnit 5 library. This is to ensure that projects using JUnit 4 or JUnit 5 can be executed without any issues. I had to build a custom test runner since Maven's Surefire Plugin [26] currently does not support executing tests in shuffled order.

3.2.5 Test Outcome Analysis

After the tests have been executed, the custom test runner will output the result of the tests in a JUnit XML format. I then used a customised FlaPy [1] that can analyse the test results of JUnit tests [140]. Firstly, I only include tests that were executed successfully without any errors, failures, or skipped outcome. I discarded any projects that did not contain at least one executable EvoSuite test. I only considered a test as flaky if it has at least one test that has different outcomes.

3.2.6 Successful Projects

The final sample of projects that I could compile, contains at least one developer-written tests, is able to generate tests using EvoSuite (with and without the flakiness suppression mechanism), execute all the tests using the customised test runner, and

[1]FlaPy output JUnit XML files per each test execution for Python projects.

Table 3.2: *Number of (#) developer-written, EvoSuite$_{FSOn}$, and EvoSuite$_{FSOff}$ tests from the 1,902 projects. (FS = Flakiness Suppression)*

Category	Number of tests
Developer-written	163,305
EvoSuite$_{FSOn}$ (Generated With *FS*)	310,193
EvoSuite$_{FSOff}$ (Generated Without *FS*)	264,000

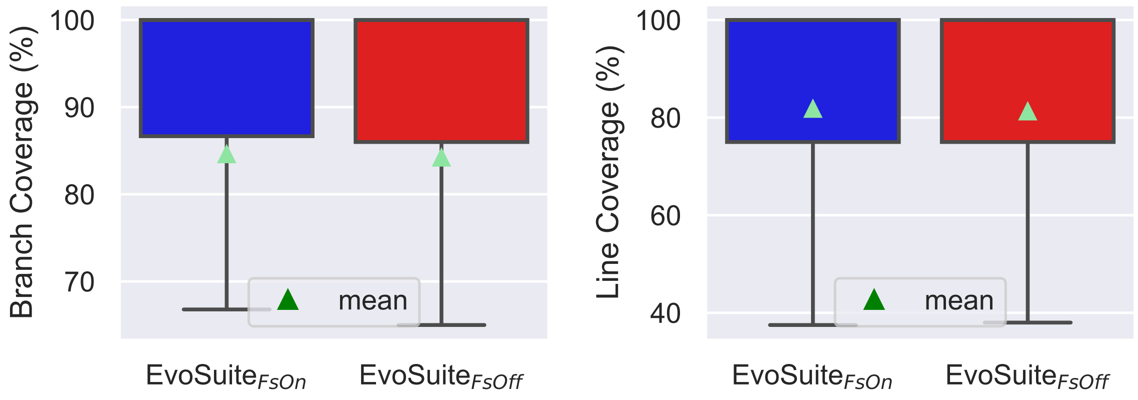


Figure 3.2: *Branch and Line coverage of EvoSuite$_{FSOn}$ and EvoSuite$_{FSOff}$ tests.*

the test results can be analysed using customised FlaPy, consisted of 1,902 projects. Table 3.2 shows the number of developer-written, EvoSuite$_{FSOn}$, and EvoSuite$_{FSOff}$ tests which I was able to execute and generate using EvoSuite.

Figure 3.3 shows the distribution of the lines of code of the projects and the number of developer-written, EvoSuite$_{FSOn}$, and EvoSuite$_{FSOff}$ tests. The mean lines of code for the sample project was 4,948, with a median of 1,395. Only 1.5 % of all projects contain less than 100 lines of code, whereas the largest project (`cosmos-sdk-java`[2]) has more than 500,000 lines of Java code. The total number of lines of code combining all sampled projects is around 9.4 million. The number of developer-written tests per project has a mean of 85.9 and a median of 18. The number of EvoSuite$_{FSOn}$ tests has a mean of 163.1 and a median of 48. The number of EvoSuite$_{FSOff}$ tests has a mean of 138.7 and a median of 43. The EvoSuite$_{FSOn}$ generated test suites with a high code coverage, with mean line coverage of 81.8 % and the mean branch coverage of 84.6 % (Figure 3.2). This is the same for EvoSuite$_{FSOff}$, where the mean line coverage

[2]`https://github.com/cloverzrg/cosmos-sdk-java`

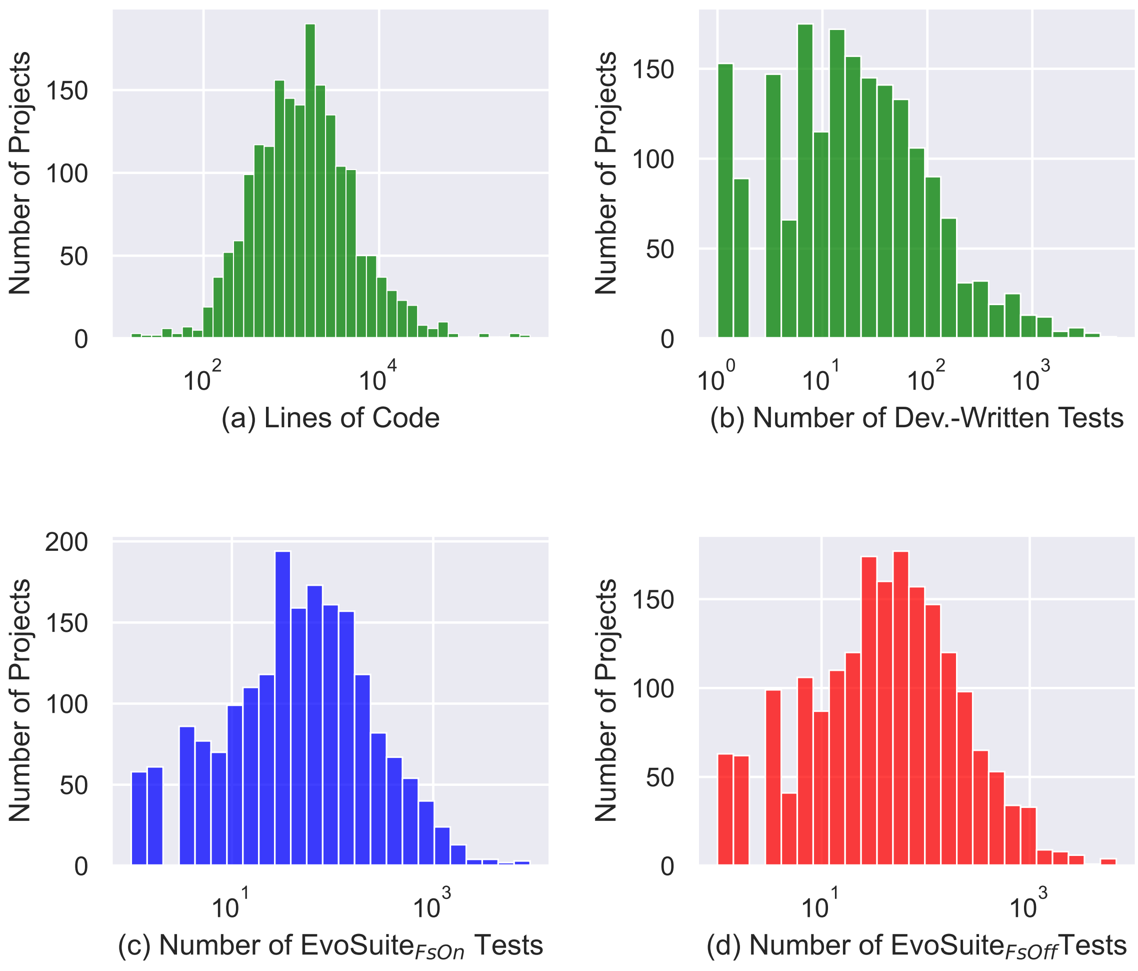


Figure 3.3: *Statistics of the open-source projects studied in this chapter.*

is 81.2 % and the mean branch coverage of 84.2 %. This is similar to the reported code coverage of previous studies on EvoSuite [125, 126].

3.2.7 RQ1: Prevalence

To answer the first research question, I compared the number of flaky tests generated in EvoSuite$_{FSOff}$—without using any flakiness suppression mechanisms—to the number of flaky tests found in the developer-written tests. I considered a test as flaky if it yielded at least one passing and one failing or error outcome [111, 141]. However, tests outcome that switched only between failing and error outcomes are not considered as flaky, since the test will still lead to a build failure and does not assimilate the developer experience caused by flakiness (occasional build failures). Besides that, I look at the proportion of flaky tests between the order-dependent flaky tests and non-order-dependent flaky tests. Finally, I compared the projects that contain at least one developer-written flaky test or at least one EvoSuite$_{FSOff}$ flaky test to identify whether if the flakiness tends to appear in the same projects.

3.2.8 RQ2: Flakiness Suppression

To evaluate the effectiveness of the flakiness suppression mechanisms in EvoSuite, I compared the number of flaky tests generated by EvoSuite$_{FSOn}$ to the number of flaky tests generated by EvoSuite$_{FSOff}$ and to the number of flaky tests in the developer-written tests. I also looked at the proportion of flaky tests to non-flaky tests and used the Wilcoxon signed-rank test [253], which is commonly used as a non-parametric paired difference test to determine the statistical significance difference, rather than using a parametric test as the data are not normally distributed based on the Shapiro-Wilk test [238]:

- Developer-written tests: ($W = 0.1637$, $p = 0.000$)
- EvoSuite$_{FSOn}$: ($W = 0.0822$, $p = 0.000$)
- EvoSuite$_{FSOff}$: ($W = 0.1634$, $p = 0.000$)

3.2.9 RQ3: Root Cause Analysis

To understand the root causes of flaky tests, I manually analysed the flaky tests in the developer-written, EvoSuite$_{FSOn}$, and EvoSuite$_{FSOff}$ tests. Similar to other previous studies [141, 182], I categorised the root causes of the flaky test by labelling the flaky tests manually following the established categories. I used the root causes collected by Parry et al. [210] as the starting point of the root causes of flaky tests. I manually classify only the root causes of the non-order dependent (NOD) flaky tests as order-dependent (OD) flaky tests can be easily identifies via randomly shuffle the execution order of the tests.

Sampling Strategy

I randomly sampled 340 flaky tests from the 1,470 flaky tests that I found across all projects. This is to ensure that I could get a representative sample of flaky tests while keep the labelling of the flakiness feasible. I combined two different sampling strategies to make sure that I avoid creating any bias towards projects with only a few flaky tests (selecting random projects) or tests only from a few projects (selecting random tests). Firstly, I randomly select one NOD flaky test from each project, regardless whether it's developer-written, EvoSuite$_{FSOn}$, or EvoSuite$_{FSOff}$ test, and this sample is called *breadth sample*. Secondly, I randomly select 21 projects and sample all the NOD flaky tests from the projects (*depth sampled*). These projects are evenly distributed in terms of the type of flaky tests that it contains. Using this approach, I sampled a total of 340 flaky tests: 227 from the breadth sample and 134 from the depth sample, with 21 flaky tests appearing in both samples.

Alignment labelling

In order to ensure that I did not create any bias during the labelling of the root causes of the flakiness, three other co-authors of this chapter's publication were involved in the alignment labelling process. We randomly choose 50 flaky tests from the 340 sampled flaky tests and use the project code, test code, the failure/error stack trace, and failure/error messages to identify the root causes of the flakiness. Based on the Fleiss' Kappa Inter-Rater Reliability [119], the agreement between the four raters is 0.41, which is considered as moderate agreement [170]. Although there is some variability in the initial outcomes, the conclusions drawn from these labelling indicate a general consensus between the raters. We discuss any case of disagreement regarding the root cause, which has led to some changes in the established root causes reported by Parry et al. [210]:

- *Unordered collection* category is broadened to include *Unspecified behaviour* in general.
- *Resource leak* category is broadened to also include *Resource unavailability*, and
- *Performance* category is added to include tests that failed due to varying duration of (sequential) processes.

After finishing the alignment labelling, I continue to label the remaining 290 NOD flaky tests in the sample. There are three comparisons that I made to answer the last research question:

1. I compare the root causes that are found in the developer-written tests to those found in previous studies [115, 169].

Table 3.3: *Number of (#) flaky tests found in developer-written, EvoSuite$_{FSOn}$, and EvoSuite$_{FSOff}$.*

Test Type	Flaky (NOD + OD)	
	# Tests	# Projects
Developer-Written	1,528 (0.94 %)	161 (8.46 %)
EvoSuite$_{FSOn}$	1,285 (0.41 %)	133 (6.99 %)
EvoSuite$_{FSOff}$	3,832 (1.45 %)	228 (11.9 %)

Table 3.4: *Number of non-order-dependent (NOD) and order-dependent (OD) flaky tests found in developer-written, EvoSuite$_{FSOn}$, and EvoSuite$_{FSOff}$.*

Test Type	NOD		OD	
	# Tests	# Projects	# Tests	# Projects
Developer-Written	698 (0.43 %)	105 (5.52 %)	830 (0.51 %)	104 (5.46 %)
EvoSuite$_{FSOn}$	175 (0.06 %)	43 (2.26 %)	1,110 (0.35 %)	109 (5.73 %)
EvoSuite$_{FSOff}$	597 (0.22 %)	111 (5.84 %)	3,235 (1.23 %)	163 (8.57 %)

2. I compare the root causes of the flaky tests generated by EvoSuite$_{FSOff}$ against the developer-written tests.

3. I compare the root causes of EvoSuite$_{FSOn}$ flaky tests against EvoSuite$_{FSOff}$ flaky tests.

3.3 Results

As shown in Table 3.2, I am able to execute 163,305 developer-written tests, 310,193 EvoSuite$_{FSOn}$ tests, and 264,000 EvoSuite$_{FSOff}$ tests, which is a total of 737,498 tests across 1,902 projects. Roughly three-fourths of the tests are generated by either EvoSuite$_{FSOn}$ or EvoSuite$_{FSOff}$.

3.3.1 RQ1: Prevalence

I found 1528 (0.94 %) of the developer-written tests to be flaky and the ratio between the number of NOD and OD flaky tests is roughly similar, which is similar to a previous study [168].

For the EvoSuite$_{FSOff}$ tests, I found 3,832 (1.45 %) of the tests to be flaky, which means that the flakiness issue is more prevalent in the EvoSuite$_{FSOff}$ tests compared to the developer-written tests, with 54 % (0.94 % to 1.45 %) increase. The number of NOD flaky tests when compared to OD flaky tests generated by EvoSuite$_{FSOff}$ is 597 (0.22 %) and 3,235 (1.23 %), meaning 84 % of flaky EvoSuite$_{FSOff}$ tests are order-dependent.

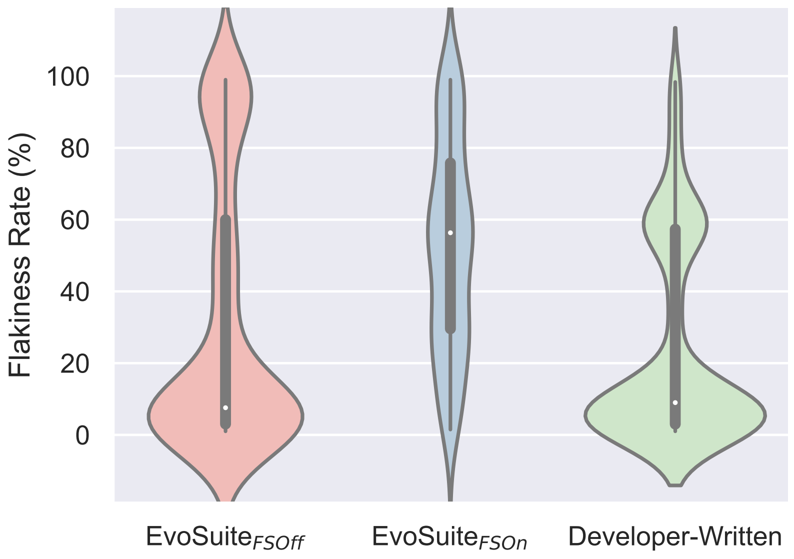


Figure 3.4: *Violin plot showing the non-order dependent (NOD) flaky-failure rate of EvoSuite$_{FSOff}$, EvoSuite$_{FSOn}$, and developer-written tests. Each plot is proportional to the overall number of non-order dependent flaky tests found.*

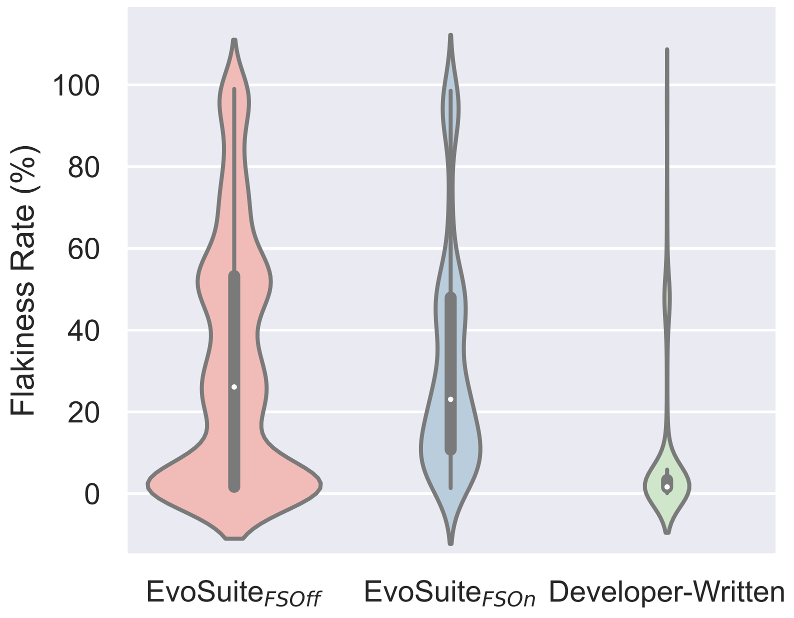


Figure 3.5: *Violin plot showing the order dependent (OD) flaky-failure rate of EvoSuite$_{FSOff}$, EvoSuite$_{FSOn}$, and developer-written tests. Each plot is proportional to the overall number of order dependent flaky tests found.*

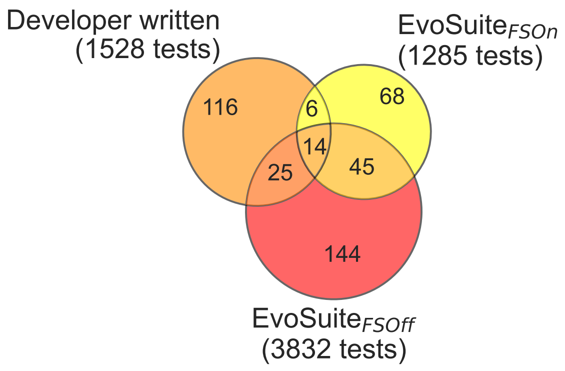


Figure 3.6: *Projects containing flaky tests.*

There is a strong tendency towards order-dependent flakiness when being compared to order-dependent.

Besides that, based on the violin plot shown in Figure 3.4, the rate of flakiness (failing) on non-order dependent EvoSuite$_{FSOff}$ has a roughly similar distribution when compared to the developer-written tests, with most of the tests tend to have a lower failure rate. However, for the order-dependent flakiness rate as shown in Figure 3.5, the rate of flakiness of the 830 developer-written tests are mostly low, but EvoSuite$_{FSOff}$ (3235 tests) has better normalised distributed of flakiness rate, but still majority of them are lower failure rate.

To determine whether generated flaky tests are more likely to appear in projects that already contain developer-written flaky tests, I examined the sets of projects that include at least one flaky test. As shown in Figure 3.6, 161 projects contain at least one developer-written flaky tests, and 228 projects contains at least one EvoSuite$_{FSOff}$ flaky tests. However, the number of overlap between these two sets is small, with only 39 projects (17.1 %) that contains both flaky tests.

Conclusion (RQ1: Prevalance).
As shown in Table 3.3, flakiness is just as prevalent in EvoSuite$_{FSOff}$ tests as it is in developer-written tests, but it does not appear in same projects, due to different in root causes of flakiness (Table 3.5). Besides that, EvoSuite$_{FSOff}$ flakiness issue is also more prevalent when the tests were ran in random orders (order-dependent) compared to same order (non-order-dependent), which is not the case for developer-written tests.

3.3.2 RQ2: Flakiness Suppression Mechanisms

As shown in Table 3.3, EvoSuite$_{FSOn}$ tests are significantly (p-value of Wilcoxon test < 0.001) less flaky when being compared to EvoSuite$_{FSOff}$ tests, with 71.7 % reduction in flakiness. The number of flaky tests in EvoSuite$_{FSOn}$ is also significantly less than the number of flaky tests in developer-written tests, with 56.4 % less.

Table 3.5: *Root causes for non-order-dependent flaky developer-written, EvoSuite$_{FSOn}$, and EvoSuite$_{FSOff}$ tests.*

Root Cause	Developer-written	EvoSuite$_{FSOn}$	EvoSuite$_{FSOff}$
	Number of tests (Number of projects)		
Asynchronous Waiting (Async Wait)	36 (23)	3 (2)	5 (4)
Concurrency	15 (14)	0 (0)	5 (3)
Input/Output (I/O)	9 (9)	0 (0)	0 (0)
Network	36 (9)	1 (1)	13 (13)
Others	0 (0)	34 (22)	0 (0)
Performance	33 (11)	0 (0)	7 (7)
Randomness	13 (12)	2 (2)	23 (22)
Resource Unavailability (Resource Leak)	7 (7)	2 (2)	7 (6)
Test Case Timeout	2 (2)	4 (4)	23 (18)
Time	6 (6)	2 (1)	12 (12)
Too Restrictive Range	2 (2)	0 (0)	0 (0)
Unknown	8 (8)	2 (2)	4 (4)
Unspecified Behaviour (Unordered Collection)	3 (3)	7 (6)	14 (13)
Total	170 (106)	57 (42)	113 (102)

Similar to the results of EvoSuite$_{FSOff}$ in RQ1, the number of order-dependent flaky tests in EvoSuite$_{FSOn}$ is more common than non-order-dependent flaky tests, where 86.4 % of the flaky tests are order-dependent. However, as shown in Figure 3.4, the flakiness (failure) rate distribution of EvoSuite$_{FSOn}$ tests is the opposite of the violin plots of EvoSuite$_{FSOff}$ tests, with EvoSuite$_{FSOn}$ median at 56.34 %. But, the flakiness rate distribution of non-order dependent tests for EvoSuite$_{FSOff}$ is roughly similar with EvoSuite$_{FSOn}$ but lower distribution overall, as shown in Figure 3.5.

Based on Figure 3.6, there is only a small overlap between EvoSuite$_{FSOn}$ (133 projects) and the developer-written tests (161 projects), with only 20 projects that contain both flaky tests. This is similar to the overlap between EvoSuite$_{FSOff}$ and EvoSuite$_{FSOn}$ tests, where there are only 59 projects overlap.

Conclusion (RQ2: Flakiness Suppression).
EvoSuite's flakiness suppression mechanism is effective. It reduced the number of flaky tests by 71.7 %, which is considerably lower than the relative number of developer-written flaky tests (56.4 % fewer flaky tests). The ratio of NOD and OD flaky tests remains strongly leaning towards OD.

3.3.3 RQ3: Root Causes

Table 3.5 shows the root causes of the sampled flaky tests that I have found via manual labelling. The most common root cause of flakiness in developer-written tests is *asynchronous waiting*, 21.2 % of the sampled developer-written flaky tests, which is similar to previous studies [115, 182]. I also found flaky tests that are caused by brittle

```
@Test
public void testTimeoutFailExactly() {
    final List mock = Mockachino.mock(ArrayList.class);
    mock.size();
    mock.size();
    runTimeoutTest(Mockachino.verifyExactly(2)
  ,200,220,200,500,mock,() -> mock.size());
}
// ... omitted ...

private void runTimeoutTest(VerifyRangeStart type,int min,
   int max,int waitTme,int timeout,List mock,Runnable
   runnable) {

    long t0 = System.currentTimeMillis();
    Executors.newSingleThreadScheduledExecutor().schedule(
   runnable,waitTime,TimeUnit.MILLISECONDS);
    long t1 = System.currentTimeMillis();
    long margin = t1 - t0;
    // ... omitted ...
    type.withTimeout(timeout).on(mock).size();
    // ... omitted ...
    long t2 = System.currentTimeMillis();
    long time = t2 - t1;
    assertTrue(time + " expected at most " + max, time <=
   max + margin);
}
```

```
<error message="273 expected at most 220"
 type="junit.framework.AssertionFailedError">
```

Figure 3.7: *Developer-written flaky test with root cause Performance (project* **krka-mockachino**[3]*)*

```
@Test(timeout = 4000)
public void test00()  throws Throwable  {
 RandomJava randomJava0 = new RandomJava();
 randomJava0.gaussian((double) 1057);
 double double0 = randomJava0.gaussian();
 assertEquals(0.8241080392646101, double0, 0.01);
}

Expected:<0.8241080392646101> but was:  <-0.06836772391958745>
```

Figure 3.8: *Randomness-related flaky test generated by EvoSuite$_{FSOff}$ (project `mitchelltech5-jmatharray`)*

```
@Test(timeout = 4000)
public void test05()  throws Throwable  {
 ClassLoader classLoader0 = ClassLoader.
   getSystemClassLoader();
 ClassPathResource classPathResource0 = new
   ClassPathResource("",classLoader0);
 String string0 = classPathResource0.loadAsString("UTF-8")
   ;
 assertEquals("com\nversion.txt\n", string0);
}

org.junit.runners.model.TestTimedOutException:  test timed out after
 4000 milliseconds
```

Figure 3.9: *Test Case Timeout related flaky test generated by EvoSuite$_{FSOff}$ (project `Contrast-Security-OSS-cassandra-migration`)*

assumptions about *Performance* (i.e., duration) of sequential processes, which is a root cause not described previously. Figure 3.7 shows an example of a *Performance* flaky test. The assertion on line 113 is flaky as it assumes that the execution time (line 106) is within a certain range, which is not guaranteed.

For the EvoSuite$_{FSOff}$ generated tests, the root causes tend to be more commonly caused by *Randomness* (20.4 %) and *Test Case Timeout* (20.4 %). Figure 3.8 shows an example of a test that is flaky due to *Randomness* generated by EvoSuite$_{FSOff}$. The test makes an assertion against the value of a random variable that it sampled from a Gaussian distribution using a Box-Muller transform. The *Test Case Timeouts* happens frequently (20.4 %) due to the 4000ms (4 seconds) default timeout EvoSuite$_{FSOff}$ sets

[3] `https://github.com/krka/mockachino/tree/9bcdda05`

```
@Test(timeout = 4000)
public void test17()  throws Throwable  {
    Name name0 = new Name();
    int[] intArray0 = new int[1];
    intArray0[0] = (-3152);
    Blob blob0 = new Blob(intArray0);
    SafeBag safeBag0 = null;
    try {
        safeBag0 = new SafeBag(name0, blob0, blob0);
        fail("Expecting exception:
   IndexOutOfBoundsException");
    } catch(IndexOutOfBoundsException e) {
        verifyException("java.nio.Buffer", e);
    }
}

Exception was not thrown in java.nio.Buffer but in
 java.base/java.nio.HeapByteBuffer.get(HeapByteBuffer.java:169):
 java.lang.IndexOutOfBoundsException:  1
```

Figure 3.10: *Flaky test generated by EvoSuite$_{FSOn}$ due to JIT method inline optimisation*

for each test it generates. An example of this is shown in Figure 3.9.

The root causes of flaky tests generated by EvoSuite$_{FSOn}$ (with flakiness suppression), are very different from the root causes of flaky tests generated by EvoSuite$_{FSOff}$. Flakiness suppression mechanisms in EvoSuite$_{FSOn}$ have successfully reduced the amount of flakiness caused by traditional known root cause. I found that the majority (59.6 %) of the flaky tests do not fit any known root cause category (*Other*). I inspected these cases and found them to be due to two causes, *Verify Expected Exceptions* (18/34) and *StackOverflowErrors* (16/34).

Verify Expected Exceptions is where the test is expecting certain exceptions to be thrown and create an assertion on where (i.e., by a specific class) the exception was thrown. In other words, the generated test case creates an assertion to verify the class name of the top class in the stack trace. Such tests can be flaky since the stack trace can change intermittently, even for the same exception. This issue is caused by the just-in-time (JIT) compilation's optimisation. The JIT compiler will inline methods that are being frequently executed during the test execution, which can lead to the stack trace being different [203, 164].

Figure 3.10 shows an example of such a case: The test is expecting an `IndexOutOfBoundsException` thrown by `java.nio.Buffer`. Sometimes, due to method inlining, this exception is instead thrown by `java.nio.HeapByteBuffer`. This test is flaky due to the default way that the JVM decides to optimise the compilation, where the JIT

```
Exception in thread "main" java.
lang.IndexOutOfBoundsException
 at java.nio.Buffer.checkIndex
  (Buffer.java:743)
 at java.nio.HeapByteBuffer.get
  (HeapByteBuffer.java:169)
 ...
 at SafeBag_ESTest.test17
  (SafeBag_ESTest.java:330)
 ...
```

Before JIT Inline Optimisation

```
Exception in thread "main" java.
lang.IndexOutOfBoundsException


 at java.nio.HeapByteBuffer.get
  (HeapByteBuffer.java:169)
 ...
 at SafeBag_ESTest.test17
  (SafeBag_ESTest.java:330)
 ...
```

After JIT Inline Optimisation

Figure 3.11: *Stack traces of the flaky test in Figure 3.10 before and after JIT method inline optimisation.*

compilation will compile certain parts of the `java.nio.Buffer` class to native code, causing it to no longer appear on top of the stack trace (as shown in Figure 3.11). This flakiness issue does not happen in EvoSuite$_{FSOff}$ generated tests, as one of the flakiness suppression mechanism parameter '`No Runtime Dependency`' is updated to *true*, which prevents EvoSuite from generating tests that verify thrown exceptions. Even though the '`No Runtime Dependency`' parameter decreases the number of flaky tests, it also generates new flaky tests.

Secondly, EvoSuite$_{FSOn}$ generates flaky tests that produce intermittent *StackOverflowErrors*. This was also discovered by a previous study [116]. This error occurs consistently when the flakiness suppression mechanism is turned off. EvoSuite$_{FSOn}$ includes an internal resource threshold—limiting the stack size—to prevent a test case from a *StackOverflowError*, however, the resource checking is non-deterministic and some errors manage to slip through. Such issues do not occur in EvoSuite$_{FSOff}$ because I have disabled the generation of test scaffolding files, which include a check to prevent infinite loops in recursive methods. However, not generating scaffolding files for test classes makes the generated tests more susceptible to traditional causes of flaky tests [121].

Conclusion (RQ3: Root Causes).
EvoSuite$_{FSOff}$ tests are flaky for the same reasons as developer-written ones, however, the distribution among those reasons differs. Developer-written flaky tests are often caused by asynchronous waiting and networking operations, whereas EvoSuite$_{FSOff}$ flaky tests are typically caused by randomness and test case timeouts. When flakiness suppression is enabled (EvoSuite$_{FSOn}$), the picture changes significantly, with the majority of remaining flaky tests not fitting into any previously described categories of flakiness. Instead, they are caused by runtime optimisations and EvoSuite's internal resource threshold, both of which only take effect when certain flakiness suppression mechanisms are activated.

3.4 Threats to Validity

3.4.1 External Validity

Maven Central Repository [25] is the largest official software repositories for Java projects. However, since the custom test runner only support JUnit 4 and JUnit 5, I excluded any projects using JUnit 3 due to not being able to execute the tests in shuffled order. While this could potentially limit the generalisability of the results, JUnit 3 is considered as an outdated version of JUnit and no longer widely used. I also mitigate this threat by using Maven, which is one of the biggest build automation tool, and JUnit, widely used testing framework (JUnit) in Java [54]. In this chapter, I only considered Java projects, and the results might not be generalisable to other programming languages and other test generation tools. However, in this chapter's publication [142], we also looked into the flakiness issue for Python's developer-written tests and Pynguin [180, 179, 181], an automated unit test generation tool for Python, where we found similar flakiness issue.

3.4.2 Construct Validity

Some flaky tests have very low failure rates, which might have been missed within the 100 same order or 100 random order executions. This could potentially lead to underestimation of the number of flaky tests in this dataset. Besides that, the search budget used for EvoSuite could also be a threat to the construct validity, as I only used 120 seconds per class. Allowing more time for the EvoSuite could potentially lead to different rate of flakiness but choosing the right search budget is a non-trivial issue, especially depending on the size of the project and the complexity of the classes [89]. To mitigate this problem, I used the same search budget as in previous tool competitions [233, 250]. I have also measured the coverage of EvoSuite$_{FSOn}$ and EvoSuite$_{FSOff}$ tests and found that the average line and branch coverage per class is high (as shown in Figure 3.2), which indicates that the search budget is sufficient.

3.4.3 Internal Validity

Since I found roughly 1,480 NOD flaky tests, I had to take a sample before manually labelling their root cause, which might pose a potential threat to the validity of our findings. To avoid favouring overly large or small projects, I applied two sampling strategies, using the *breadth* and *depth* sample as explained in Section 3.2.9. Each flaky test was then manually labelled and this might pose a potential threat. To mitigate this issue, I created an alignment sample of 50 flaky tests that were labelled by myself and three other authors of this chapter's publications, and we held discussions about cases in which we disagreed. In our alignment sample, we reached a 'good' inter-rater reliability, meaning that we were aligned in most of the verdicts given even before starting the alignment. Furthermore, the root causes of the flakiness found in the developer-written flaky tests match previous studies [182, 116], which increases my confidence in the validity to the findings.

3.5 Recommendations

The results of this study show that flakiness is a prevalent issue in both developer-written and EvoSuite tests. The flakiness suppression mechanisms in EvoSuite are effective in reducing the number of flaky tests, but they also introduce new flakiness issues. In this section, I will provide recommendations for the (1) maintainers of EvoSuite, (2) developers using EvoSuite, and (3) researchers studying flaky tests.

3.5.1 Maintainers of EvoSuite

I found EvoSuite's flakiness suppression mechanisms to be highly effective and can recommend that they be implemented in other automated test generation tools. However, EvoSuite still generates flaky tests, primarily due to the (1) *Verifying Expected Exceptions* related to the `'No Runtime Dependency'` option, and (2) *StackOverflowErrors* caused by scaffolding. Most notably, both mechanisms are designed—and also accomplish—to prevent traditional causes of flakiness. I recommend that EvoSuite's maintainers revisit the implementation of `'No Runtime Dependency'` parameter to handle for cases where JIT compilation inlined frequently executed methods (as shown in Figure 3.11) and the scaffolding mechanisms to eliminate the flakiness they introduce. Additionally, I recommend studying and addressing order-dependency in EvoSuite, as high numbers of order-dependent flaky tests were found.

3.5.2 Developers Using EvoSuite

For developers using EvoSuite, I strongly recommend using the flakiness suppression mechanisms, as I found them to be highly effective. The remaining flaky tests are primarily caused by issues with *Verifying Expected Exceptions* and *StackOverflowErrors*. The former can be mitigated by disabling the tiered compilation flag (`-XX:-TieredCompilation`) when running the tests. While this prevents the JVM from compiling frequently executed parts of the bytecode into native code (**JIT** compilation), it will negatively affect the performance and execution time of other tests. The flaky *StackOverflowErrors* can be addressed by removing the `NonFunctionalRequirementRule` in the test scaffolding file, although this will cause the test to consistently fail. Developers should also be aware of the potential order-dependencies flakiness in EvoSuite, as I found nearly 6 % of projects were affected by order-dependent flaky tests.

3.5.3 Researchers Studying Flaky Tests

Since generating tests automatically can be done quickly and efficiently, there is a potential for using EvoSuite to significantly aid research on flaky tests. For instance, EvoSuite could also be used to create training data for machine learning models or to systematically expose non-determinisms in specific projects. While I found EvoSuite-generated flaky tests to have similar root causes compared to developer-written flaky tests when flakiness-suppression mechanisms are disabled, there are several differences. First, the distribution of root causes varies; flakiness in EvoSuite$_{FSOn}$ is less likely due to asynchronous waiting or networking issues, and more often results from randomness and test case timeouts. Second, I discovered that projects containing developer-written flaky tests are not particularly prone to also produce EvoSuite-generated flaky tests, and vice versa.

3.6 Chapter Conclusions

Flaky tests are a common and troublesome phenomenon in software testing. In this study, I demonstrated that flakiness does not only affect developer-written tests but also one of the most popular automatic unit test generation tool. I sampled 1,902 open-source Java projects and generated tests for them using EvoSuite. After executing the developer-written and EvoSuite-generated (EvoSuite$_{FSOff}$ and EvoSuite$_{FSOn}$) test suites repeatedly (100 times in same order and 100 times in shuffled order), I found that the flakiness to be even more prevalent among EvoSuite$_{FSOff}$-generated (without flakiness suppression mechanism) tests than developer-written tests. While flakiness suppression mechanisms effectively reduce the flakiness rate among generated tests, they also introduce other, previously unseen forms of flakiness. The root causes of this flakiness are similar, however, the distribution differs: developer-written flaky tests tend to be caused by asynchronous waiting and networking operations, while EvoSuite$_{FSOff}$-generated flaky tests are more frequently due to randomness and test case timeouts. For both developer-written and EvoSuite$_{FSOff}$-generated flaky tests, order-dependency is a frequent cause, suggesting a potential direction for future research. Since the flakiness issue on EvoSuite can mostly be suppressed with the flakiness suppression mechanisms (EvoSuite$_{FSOn}$), in my next chapter, I will look into how to utilise EvoSuite's search-based technique to improve the quality of developer-written tests. I will also explore how EvoSuite can be used to amplify existing developer-written tests with respect to mutation score.

Chapter 4

Automatically Improving the Mutation Score of Developer-Written Test Suites

The contents of this chapter is based on "Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. An Empirical Comparison of EvoSuite and DSpot for Improving developer-written test suites with respect to mutation score. In *International Symposium on Search-Based Software Engineering (SSBSE)*, 2022".

4.1 Introduction

Writing effective unit tests to detect faults is a well-known challenge [70], and manually writing good test is time-consuming and often tedious [235]. Automated test generation techniques, like EvoSuite, have been developed to help ease this burden by generating tests from scratch. As shown in Section 3.3.1, EvoSuite with the flakiness suppression mechanism, generates more stable tests compared to developer-written ones, and it can produce test suites with high mutation score.

Test amplification, as discuss in the Section 2.8, explicitly aims to strengthen existing developer-written test suites [105]. The aim is to generate a new version of the developer's test that covers more *corner cases*—i.e., rare input scenarios and boundary conditions that lie at the edges of the normal range—and is more effective at finding faults. Since the "amplified" test suite is based on the developers set of tests, it is likely more understandable [84]. The current state-of-the-art test amplification tool, DSpot [106], utilises developer-written tests to increase the number of mutants that they kill. It "amplifies" developer-written test cases by changing the values of literals in the tests, method calls, or by adding assertions. Test cases that kill more mutants and have fewer modifications are retained. However, DSpot are subject to some limitations. Since it relies on developer-written tests, it may also potentially inherit its flakiness. This means that if the original tests are flaky, the amplified tests could also be subject to the same problem. As shown in the previous chapter, flakiness is a common problem in developer-written tests, and it is a problem that DSpot does not address when using developer-written tests as the starting point. Since EvoSuite provides flakiness suppression mechanisms (Table 3.1), it is more reliable in generating tests that are not flaky. Additionally, DSpot's changes to tests are less flexible as compared to EvoSuite's evolution process, and it lacks fine-grained fitness guidance of search-based tools. Test cases generated by DSpot that do not kill new mutants, for example, will be discarded even if the test is actually "close" to killing a new mutant and could be usefully improved in future.

The aim of this chapter is to evaluate a potential "best of both worlds" approach. This involves assessing a version of EvoSuite that is capable of reading developer-written tests as a starting point for test case generation. By leveraging its ability to make more fundamental changes to the structure of a test case, it then evolves those tests with the benefit of fine-grained fitness information for killing more mutants. I then evaluate whether EvoSuite's evolution and mutation analysis technique could have a better performance in terms of killing mutants when compared to DSpot's amplification technique. The motivation behind this study is to understand how many more mutants *could* be killed by test amplification tools if the principles of test amplification were applied differently, in the flavour of a more flexible and more guided search-based style of approach.

I compare this modified EvoSuite version, which I refer to as EvoSuite$_{Amp}$, with DSpot using the developer-written tests in open source projects as the starting point for test suite generation—specifically, 42 different versions of 29 different Java classes in 7 different projects of Defects4J (v2.0.0) [157]. This experiment reveals that EvoSuite$_{Amp}$ outperforms DSpot for 35 of the 42 Java class versions studied in terms of mutation score achieved. Over 30 repeated runs, EvoSuite$_{Amp}$ was further capable of killing more "unique" mutants that DSpot was not able to kill in any run for 36 of these 42 subjects. EvoSuite$_{Amp}$ and all the data collected is available in this replication package [51].

In summary, the contributions of this chapter are as follows:

1. A new test improvement strategy that utilises the flexibility of EvoSuite, EvoSuite$_{Amp}$, that evolves test cases and leverages fitness information for killing specific mutants (Section 4.2).

2. An empirical study with seven open-source projects comparing EvoSuite$_{Amp}$ with an existing state-of-the-art test amplification tool, DSpot (Section 4.3).

3. Results and analysis of the effectiveness of both tools in terms of mutation score and mutants uniquely killed by each tool (Section 4.4).

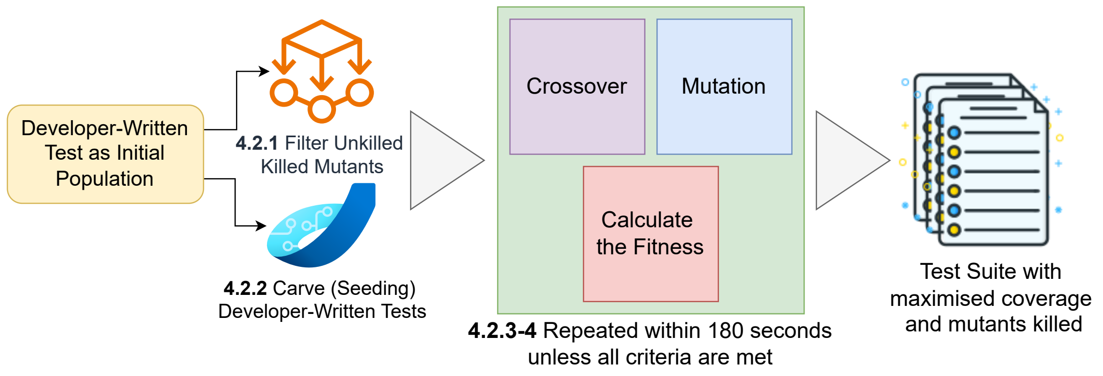

Developer-Written Test as Initial Population
4.2.1 Filter Unkilled Killed Mutants
4.2.2 Carve (Seeding) Developer-Written Tests
Crossover
Mutation
Calculate the Fitness
4.2.3-4 Repeated within 180 seconds unless all criteria are met
Test Suite with maximised coverage and mutants killed

Figure 4.1: *Overview of EvoSuite$_{Amp}$.*

4.2 Modifications made to EvoSuite— EvoSuite$_{Amp}$

EvoSuite was originally designed to generate a test suite from scratch. In this study, I use EvoSuite to read developer-written tests, remove mutants that are killed by developer-written tests, and not to add new random test cases during the search. As shown in Figure 4.1, I made four different modifications to EvoSuite, which I refer to as EvoSuite$_{Amp}$, and are as follows:

4.2.1 Removing Mutants Killed in the Developer-Written Tests

I set the fitness criteria of EvoSuite$_{Amp}$ to both branch coverage and strong mutation testing. Before starting the evolution, I remove the goals that were met by the developer-written tests. This is to ensure that the search focuses on the goals that are not covered yet. As an example, the class under test will have mutant A, X, Y, and Z. If the developer-written test can kill mutant X, Y, and Z, the only criterion that it needs to meet is to kill mutant A only.

4.2.2 Seeding Developer-Written Tests into the Initial Population of the GA

The second modification I made is on the initial population of the test cases. The default behaviour of EvoSuite is to randomly generate new test cases. Instead of randomly generating new test cases, I used the developer-written tests as the initial population of the search. This utilises the developer's domain knowledge of the program. The initial population size on the EvoSuite is set to 50 individuals, but in this study, I changed the population size depending on the developer-written test suite size. This is all done by using the carving technique that has been implemented in EvoSuite [223].

4.2.3 Tuning the Add New Random Test Case Rate to Zero

I tune the settings of the parameter values of the evolutionary algorithm responsible for generating the test suite. The default configuration of EvoSuite is to use crossover, mutation, and randomly add new test cases into the population. However, I change the rate of adding new random test cases to the population of test cases to zero. This change means that the developer-written tests are kept during evolution, without the addition of completely new, randomly generated tests. This is crucial for maintaining similarity of the generated tests to the original test suite, and keeping the test suite free of tests or part of tests that are completely new or alien to the original developer. I still allow modifications to inputs featuring in the tests, however, so that there is scope for improving the original tests to kill more mutants, and for tests to be recombined by the crossover operator. After a few generations, the fitness of all individual chromosomes improves, and the process will stop when all criteria are met or when the search budget is exhausted. A study by Aniche et al. found that developers tend to copy and paste from previous test methods and modify their name, inputs, and assertions [70]. This effect is simulated, in part, by crossover, with mutation focussed on modifying the developer-written test inputs only.

4.2.4 Turning Off Test Suite Minimisation

I turned off the EvoSuite test suite post-process minimisation feature in order to maintain the developer-written tests, or else they may be discarded following test suite evolution. This property does not affect any of the flakiness suppression mechanisms of EvoSuite (as shown in Table 3.1).

4.3 Empirical Study

This section details the experiment design of the empirical study I conducted to assess EvoSuite$_{Amp}$, DSpot$_{Mut}$, and DSpot$_{Cov}$ with respect to killing mutants. I also include DSpot$_{Cov}$ into the experiment because the EvoSuite$_{Amp}$ fitness criteria include branch coverage. In the following, I refer to EvoSuite$_{Amp}$ and DSpot as distinct "tools", while I break down the analysis of DSpot in terms of the two configurations DSpot$_{Mut}$ and DSpot$_{Cov}$. I designed this study to answer the following four research questions:

RQ1: Which tool (EvoSuite$_{Amp}$ or DSpot) kills the most mutants?

RQ2: Which tool kills the most "unique" mutants (mutants not killed by the alternative tool)?

RQ3: Which tool kills the most mutants with the smallest test suites?

RQ4: Which tool provides the most consistent results when re-run multiple times?

Table 4.1: *Subject programs used in this study, by Lines of Code, average number of mutants (# of Mutants), number of unique classes evaluated (# of Classes), and number of versions evaluated (# of Versions).*

Subject (Acronym)	**Lines of Code**			**# of**	**# of**	**# of**
	Min	**Max**	**Avg.**	**Mutants**	**Classes**	**Versions**
Commons-Cli (**Cli**)	56	200	104	261	2	3
Commons-Codec (**Cdc**)	162	355	242	253	3	4
Commons-Compress (**Crs**)	92	370	205	182	5	5
Commons-Csv (**Csv**)	105	1152	675	117	3	9
Jsoup (**Jsp**)	85	280	193	48	5	7
Commons-Lang (**Lng**)	52	1366	907	568	3	5
Commons-Math (**Mth**)	148	1091	469	662	8	9
Total					29	42

4.3.1 Subjects

I performed this experiment on the widely used benchmark Defects4J (v2.0.0) [157], which contains 835 reproducible real faults on 17 open-source projects. Although I am not specifically interested in the individual bugs provided by this benchmark, it provides an ideal set of subject classes and utilities with which I can evaluate the performance of both the EvoSuite$_{Amp}$ and DSpot tools. This includes an interface for test generation, which among other things help with removing flaky tests—tests that pass and fail non-deterministically without any changes to code [211]. It also incorporates the Major [156] mutation analysis tool, which I use as an independent arbiter of the mutants killed by the test suites generated by both EvoSuite$_{Amp}$ and DSpot tools (since DSpot relies on PITest [97], while EvoSuite uses its own in-built mutation analysis).

I selected subject classes from Defects4J with which to perform this experiment based on the following rules:

1. The project includes developer-written tests;
2. DSpot$_{Mut}$, DSpot$_{Cov}$, and EvoSuite$_{Amp}$ were capable of using the provided original developer-written tests,
3. Major [156], PITest [97], and EvoSuite could generate mutants for the project.

After running every faulty version of each project in the Defects4J dataset, 42 faulty versions of 29 unique classes in 7 libraries met the requirements above. Table 4.1 shows the details of these subjects. I found a large number of Defects4J's classes/versions to be unusable for this study due to an issue with DSpot's interface with its mutation analysis tool PITest needed for the study, and problems compiling the class under test.

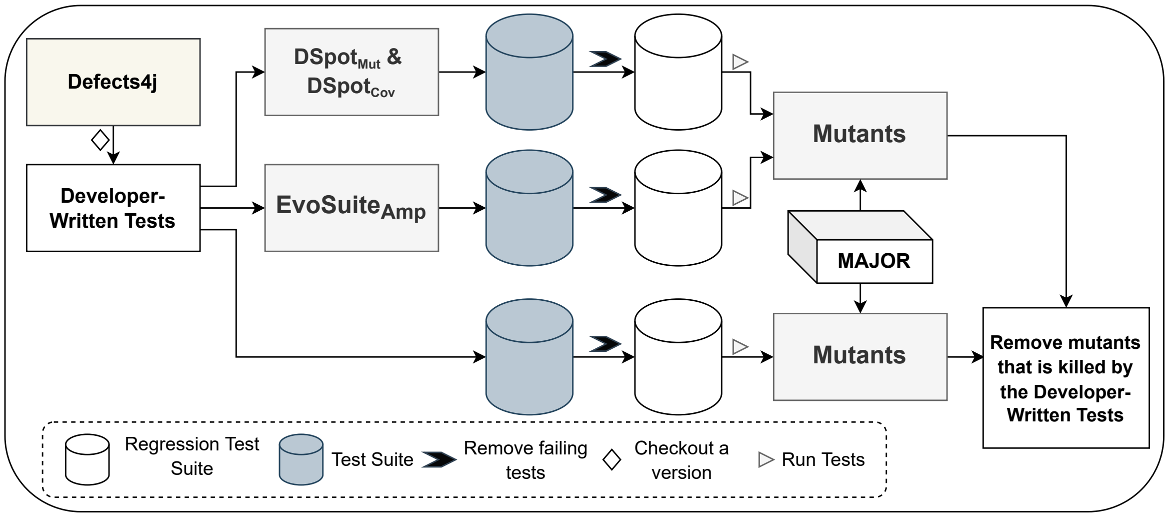


Figure 4.2: *Overview of the experimental setup.*

I have contacted the owner of the DSpot project, but the issue has not been resolved to date. Despite this, this final subject set comprises a wide and diverse set of classes over a number of projects that are suitable for this study.

4.3.2 Experimental Procedure

Figure 4.2 shows the overview of this experiment. The tools are fed with developer-written tests that were gathered from the test files in every project version (with a particular fault) from Defects4J. For each version, as shown in Table 4.2, I improved each class of the study's original developer-written test suite (as provided by Defects4J) using EvoSuite$_{Amp}$ (using EvoSuite v1.2.0), DSpot$_{Mut}$, and DSpot$_{Cov}$ (both using v3.2.0 of DSpot). I ran all experiments on the same workstation, with 32GB RAM and Intel i5 CPU at 3.10GHz, running Ubuntu 20.04.4 LTS. For both tools, I set the search time budget (Algorithm 1, line 8) to 120 seconds, a commonly used value for test suite generation, and one that is applied in the search-based testing tool competition [207].

To take into account the non-deterministic nature of the tools, I repeat test suite generation 30 times for each tool/configuration studied. While I did not perform any internal modifications to DSpot—the build was downloaded from their repository[14] and configured to form DSpot$_{Mut}$ and DSpot$_{Cov}$. I made the modifications to EvoSuite to form EvoSuite$_{Amp}$ detailed in Section 4.2.

Since I have turned off the test suite minimisation post-process on EvoSuite (Section 4.2), which is part of the checks performed in EvoSuite's JUnit Check parameter (Table 3.1), I need to ensure that no failing (*flaky*) tests are present. This checking is also necessary for both DSpot's configuration, as DSpot does not have any dedicated flakiness suppression mechanism in the tool. To ensure there are no flaky tests generated by either tool, I used the *fix_test_suite* feature of Defects4J that removes failing tests from the test suite until all tests pass. Without removing these failing tests, flaky tests could interfere with the mutation score result. For all the developer-written and generated regression test suites, I used the Major mutation testing tool [156] to compute the mutation score. Major includes a summary of which mutants are killed by each test suite. The summary helps in finding the additional number of mutants that the generated test suites kill. I calculate the relative increase of mutation score for each automatically improved test suite, over the original developer-written version as:

$$\%IncreaseKilled = \frac{AverageMutantsKilled_{Amplified}}{TotalNumberOfMutants} \times 100$$

Generated Test Suites

EvoSuite$_{Amp}$ and DSpot generate the test cases in a single test file. There are some cases where it has dependencies from other test files that developers wrote, such as a utility class. Without importing dependencies in EvoSuite$_{Amp}$ and DSpot, the improved test suite files will have compilation errors. For this reason, I made sure the improved test suite files always imported these test suite dependencies.

Handling of Mutation Analysis

In this experiment, I used strong mutation testing to evaluate the amplified tests. There were, in effect, three different mutation testing tools involved in the study. EvoSuite uses its own mutation analysis tool, while DSpot uses the PITest mutation analysis tool as part of its test amplification process. Since both EvoSuite and DSpot use different mutation analysis tools, it is not fair to compare the number of mutants it kills with different tools, which could produce different results for the same test suite. To avoid any bias in this study, I used a third mutation analysis tool, in the form of Major [156] to perform mutation analysis after both EvoSuite and DSpot generate the improved test suite. Since Major is Defects4J's default mutation analysis tool, it was straightforward to apply this analysis in my study.

Statistical Analysis.

Since I am assessing algorithms that are making random choices, I analysed the data that I collected using well-established statistical analysis recommendations [73]. I repeated each experiment 30 times. I then used the Mann-Whitney U-test to check for the significant differences regarding the number of mutants killed, comparing EvoSuite$_{Amp}$ with DSpot$_{Mut}$ improved test suites, and then EvoSuite$_{Amp}$ with DSpot$_{Cov}$ test suites, for each version of each subject class. I used the 99% confidence interval, which means that if the p-value is less than 0.01, this result is statistically significant. I further calculate effect sizes, using Vargha-Delaney's ($\hat{A}$) test. Again, I compared EvoSuite$_{Amp}$ with DSpot$_{Mut}$, and then EvoSuite$_{Amp}$ with DSpot$_{Cov}$. An $\hat{A}$ value that is over 0.5 indicates that EvoSuite$_{Amp}$ outperforms DSpot. Another statistical analysis that I performed was finding the correlation between the size of the test suite, and the mutation score. I used Spearman's rank correlation coefficient to find the relationship between the two variables, with 99% confidence interval to indicate if the result is statistically significant.

4.4 Results

RQ1: Mutation Score

Table 4.2, part B, shows the mean of the mutants killed by EvoSuite$_{Amp}$, DSpot$_{Mut}$, and DSpot$_{Cov}$. The table further shows that EvoSuite$_{Amp}$ is more effective at killing mutants for 35 out of the 42 versions (83.3%) than DSpot$_{Mut}$. It is also better at killing mutants for 27 out of the 42 (64.3%) versions compared to DSpot$_{Cov}$. EvoSuite$_{Amp}$ is most effective at killing mutants with classes from the *Math* project. All the class versions that EvoSuite$_{Amp}$ achieves a better mutation score have a p-value less than 0.01. Where DSpot$_{Cov}$ and DSpot$_{Mut}$ achieve a better mutation score than EvoSuite$_{Amp}$, the p-value is less than 0.01. When using the $\hat{A}$ statistic to measure effect size, I found that EvoSuite$_{Amp}$ has a score that favours it over DSpot$_{Mut}$ in 35 out of the 42 projects and DSpot$_{Cov}$ in 25 out of 42 versions (59.5%), each time with an $\hat{A}$ value greater than 0.8 (i.e., a large effect size).

Conclusion (RQ1: Mutation Score.).
EvoSuite$_{Amp}$ performs better than DSpot$_{Mut}$ and DSpot$_{Cov}$ in terms of killing mutants.

RQ2: Unique Mutants

I also evaluated the cumulative number of uniquely killed mutants after executing each of the 30 test suites on every tool. This means that mutants that are still alive after 30 runs are considered as either stubborn mutants or equivalent mutants. I found that EvoSuite$_{Amp}$ killed more unique mutants in 36 out of the 42 versions (85.7%) when compared to DSpot$_{Mut}$, and 27 out of the 42 versions (64.3%) when compared to DSpot$_{Cov}$. This and more detailed information regarding the performance of each tool on each class version can be seen in part D of Table 4.2.

Conclusion (RQ2: Unique Mutants).
EvoSuite$_{Amp}$ kills more unique mutants after 30 runs when compared to DSpot$_{Mut}$ and DSpot$_{Cov}$.

RQ3: Size of Test Suite

Even though EvoSuite$_{Amp}$ can kill more mutants, it generates bigger test suites in general. When comparing DSpot$_{Mut}$ to EvoSuite$_{Amp}$, 39 out of the 42 class versions

Table 4.2: *The result of test amplification on 42 versions after 30 runs for* $EvoSuite_{Amp}$ (***Evo***), $DSpot_{Mut}$ (***DS***), *and* $DSpot_{Cov}$ (***DJ***).

(A)	(B)						(C)			(D)			(E)	(F)			(G)		
Fault Version	# Killed Mutants						Std. Dev.			# Unique Mutants Killed			Orig. Mutation Score (%)	Inc. Killed (%)			# KLoC		
	Mean			Median										Mean					
	Evo	DS	DJ	Evo	DS	DJ	Evo	DS	DJ	Evo	DS	DJ		Evo	DS	DJ	Evo	DS	DJ
Cdc-11	†18.2	16.5	20.0	†18.0	16.0	20.0	$^{\circ}$1.2	1.2	0.0	20	20	20	63.9	†21.9	19.9	24.1	*0.2	0.1	0.3
Cdc-16	$^{\dagger\circ}$90.5	2.0	7.0	$^{\dagger\circ}$90.5	2.0	7.0	$^{\dagger\circ}$23.3	1.0	0.0	$^{\dagger\circ}$139.0	3	7	50.6	$^{\dagger\circ}$9.9	0.2	0.8	$^{*\bullet}$0.7	<0.1	0.1
Cdc-17	$^{\dagger\circ}$7.3	1.0	2.0	$^{\dagger\circ}$7.5	1.0	2.0	$^{\dagger\circ}$1.4	0.0	0.0	$^{\dagger\circ}$9.0	1	2	43.5	$^{\dagger\circ}$31.6	4.3	8.7	$^{*\bullet}$0.3	0.2	0.2
Cdc-18	$^{\dagger\circ}$6.9	1.5	2.0	$^{\dagger\circ}$7.0	2.0	2.0	$^{\dagger\circ}$1.5	0.5	0.0	$^{\dagger\circ}$9.0	2	2	43.5	$^{\dagger\circ}$30.0	6.7	8.7	$^{*\bullet}$0.3	0.2	0.3
Cli-37	$^{\dagger\circ}$13.7	1.0	1.0	$^{\dagger\circ}$13.0	1.0	1.0	$^{\dagger\circ}$4.3	0.0	0.0	$^{\dagger\circ}$23.0	1	1	76.5	$^{\dagger\circ}$3.7	0.3	0.3	$^{*\bullet}$1.4	<0.1	<0.1
Cli-38	$^{\dagger\circ}$10.8	2.0	2.0	$^{\dagger\circ}$11.0	2.0	2.0	$^{\dagger\circ}$2.4	0.0	0.0	$^{\dagger\circ}$15.0	2	2	76.1	$^{\dagger\circ}$2.9	0.5	0.5	$^{*\bullet}$1.5	<0.1	<0.1
Cli-39	1.0	2.0	2.0	1.0	2.0	2.0	0.0	0.0	0.0	1	2	2	14.3	7.1	14.3	14.3	$^{*\bullet}$0.2	<0.1	0.2
Crs-34	$^{\dagger\circ}$39.0	35.0	35.0	$^{\dagger\circ}$40.0	35.0	35.0	$^{\dagger\circ}$3.0	0.0	0.0	$^{\dagger\circ}$44.0	35	35	48.2	$^{\dagger\circ}$34.8	31.2	31.2	$^{*\bullet}$0.4	0.2	0.2
Crs-39	$^{\dagger\circ}$57.5	36.0	36.0	$^{\dagger\circ}$58.5	36.0	36.0	$^{\dagger\circ}$4.0	0.0	0.0	$^{\dagger\circ}$62.0	36	36	32.1	$^{\dagger\circ}$51.3	32.1	32.1	$^{*\bullet}$0.8	<0.1	<0.1
Crs-40	†9.0	4.4	17.0	†9.0	4.0	17.0	$^{\circ}$2.1	3.1	0.0	†15.0	12	17	37.2	†0.9	0.5	1.8	$^{*\bullet}$0.2	<0.1	<0.1
Crs-44	12.1	15.3	17.0	12.5	14.0	17.0	$^{\dagger\circ}$3.1	1.5	0.0	$^{\dagger\circ}$18.0	17	17	0.0	52.6	66.5	73.9	$^{*\bullet}$0.2	<0.1	<0.1
Crs-45	$^{\dagger\circ}$108.4	56.3	63.0	$^{\dagger\circ}$110.0	57.0	63.0	$^{\dagger\circ}$14.6	1.3	0.0	$^{\dagger\circ}$132.0	57	63	54.2	$^{\dagger\circ}$18.7	9.7	10.9	$^{*\bullet}$0.8	0.2	0.2
Csv-01	$^{\dagger\circ}$6.7	2.1	3.0	$^{\dagger\circ}$6.5	3.0	3.0	$^{\dagger\circ}$2.9	1.1	0.0	$^{\dagger\circ}$16.0	3	3	41.7	$^{\dagger\circ}$8.0	2.5	3.6	*0.3	<0.1	0.5
Csv-02	$^{\dagger\circ}$9.6	2.0	7.0	$^{\dagger\circ}$12.0	2.0	7.0	$^{\dagger\circ}$3.3	0.0	0.0	$^{\dagger\circ}$12.0	2	7	36.8	$^{\dagger\circ}$50.5	10.5	36.8	$^{*\bullet}$0.3	0.1	0.1
Csv-04	†16.8	14.8	27.0	†17.0	15.0	27.0	$^{\dagger\circ}$1.4	1.0	0.0	†20.0	18	27	40.0	†24.0	21.2	38.6	*0.3	0.3	0.9
Csv-06	$^{\dagger\circ}$12.2	8.4	9.0	$^{\dagger\circ}$12.0	8.0	9.0	$^{\dagger\circ}$0.6	0.5	0.0	$^{\dagger\circ}$13.0	9	9	35.0	$^{\dagger\circ}$61.2	41.8	45.0	$^{*\bullet}$0.4	0.2	0.2
Csv-07	†14.7	12.9	23.0	†14.0	12.0	23.0	$^{\dagger\circ}$2.3	1.5	0.0	†19.0	16	23	47.1	†21.0	18.4	32.9	*0.3	0.2	1.0
Csv-10	16.2	56.0	108.0	14.5	54.5	108.0	$^{\circ}$6.3	10.3	0.2	33	75	109	28.2	5.7	19.7	38.0	$^{*\bullet}$0.6	0.2	0.5
Csv-11	†18.1	13.4	31.0	†18.0	13.0	31.0	$^{\circ}$2.0	2.1	0.0	†23.0	18	31	42.0	†22.3	16.5	38.3	*0.3	0.2	1.0
Csv-12	†31.7	0.0	108.0	†33.0	0.0	108.0	$^{\dagger\circ}$3.4	0.0	0.0	†36.0	0	108	50.3	†10.1	0.0	34.6	$^{*\bullet}$2.1	0.1	1.0
Csv-16	†22.4	12.2	46.0	†22.5	8.0	46.0	$^{\circ}$4.3	7.4	0.0	†31.0	26	46	36.8	†19.6	10.7	40.4	*0.8	0.3	1.3
Jsp-58	9.7	20.1	29.0	8.0	19.0	29.0	$^{\dagger\circ}$4.4	2.2	0.0	23	25	29	22.5	13.7	28.4	40.8	*0.1	<0.1	0.2
Jsp-69	†14.3	2.0	24.0	†15.0	2.0	24.0	$^{\dagger\circ}$2.8	0.0	0.0	†18.0	2	24	2.6	†37.6	5.3	63.2	*0.2	0.1	0.4
Jsp-79	$^{\dagger\circ}$2.3	0.0	0.0	$^{\dagger\circ}$2.0	0.0	0.0	$^{\dagger\circ}$0.5	0.0	0.0	$^{\dagger\circ}$4.0	0	0	53.8	$^{\dagger\circ}$9.0	0.0	0.0	*0.2	0.2	0.3
Jsp-80	36.6	46.0	49.0	35.0	46.0	49.0	$^{\dagger\circ}$4.4	2.2	0.0	45	49	49	11.1	45.1	56.8	60.5	$^{*\bullet}$0.3	<0.1	0.2
Jsp-84	16.0	26.0	27.0	16.0	26.0	27.0	$^{\dagger\circ}$2.2	0.0	0.0	20	26	27	0.0	38.2	61.9	64.3	$^{*\bullet}$0.2	0.1	0.1
Jsp-86	$^{\dagger\circ}$6.9	1.5	0.0	$^{\dagger\circ}$7.0	1.5	0.0	$^{\dagger\circ}$2.1	1.5	0.0	$^{\dagger\circ}$12.0	3	0	51.4	$^{\dagger\circ}$19.6	4.3	0.0	*0.2	<0.1	0.2
Jsp-93	†15.7	4.0	18.0	†16.0	4.0	18.0	$^{\dagger\circ}$2.6	0.0	0.0	$^{\dagger\circ}$19.0	4	18	2.5	†39.2	10.0	45.0	*0.3	0.2	0.4
Lng-03	†517.8	484.2	545.0	†517.5	485.0	545.0	$^{\dagger\circ}$12.2	8.9	0.0	†543.0	499	545	0.4	†58.2	54.5	61.3	$^{*\bullet}$2.2	1.2	1.6
Lng-04	$^{\circ}$0.8	1.0	0.0	$^{\circ}$1.0	1.0	0.0	$^{\dagger\circ}$0.5	0.0	0.0	$^{\dagger\circ}$2.0	1	0	82.9	$^{\circ}$2.0	2.4	0.0	$^{*\bullet}$0.3	<0.1	<0.1
Lng-05	$^{\dagger\circ}$104.0	5.0	8.0	$^{\dagger\circ}$104.0	5.0	8.0	$^{\dagger\circ}$3.1	0.0	0.0	$^{\dagger\circ}$112.0	5	8	0.0	$^{\dagger\circ}$73.8	3.5	5.7	$^{*\bullet}$0.4	0.2	0.2
Lng-07	$^{\dagger\circ}$530.8	406.0	492.0	$^{\dagger\circ}$530.5	407.5	492.0	$^{\circ}$11.0	13.3	0.0	$^{\dagger\circ}$554.0	449	492	0.4	$^{\dagger\circ}$59.3	45.4	55.0	$^{*\bullet}$2.4	1.0	1.5
Lng-16	$^{\dagger\circ}$520.7	416.2	490.0	$^{\dagger\circ}$520.0	415.0	490.0	$^{\circ}$10.9	12.8	0.0	$^{\dagger\circ}$545.0	445	490	0.5	$^{\dagger\circ}$59.6	47.7	56.1	$^{*\bullet}$2.3	1.1	1.5
Mth-09	$^{\dagger\circ}$26.2	20.0	20.0	$^{\dagger\circ}$26.0	20.0	20.0	$^{\dagger\circ}$3.4	0.0	0.0	$^{\dagger\circ}$32.0	20	20	51.6	$^{\dagger\circ}$28.8	22.0	22.0	0.4	0.9	1.7
Mth-25	$^{\dagger\circ}$112.9	41.9	82.0	†74.5	32.0	82.0	$^{\dagger\circ}$72.3	13.2	0.0	$^{\dagger\circ}$233.0	59	82	0.0	$^{\dagger\circ}$32.2	12.0	23.4	$^{*\bullet}$0.3	<0.1	0.1
Mth-26	$^{\dagger\circ}$157.0	51.4	65.0	$^{\dagger\circ}$157.0	51.0	65.0	$^{\dagger\circ}$4.2	0.7	0.0	$^{\dagger\circ}$168.0	53	65	45.6	$^{\dagger\circ}$33.3	10.9	13.8	*1.0	0.8	1.2
Mth-27	$^{\dagger\circ}$152.1	51.2	65.0	$^{\dagger\circ}$152.0	51.0	65.0	$^{\dagger\circ}$3.4	0.6	0.2	$^{\dagger\circ}$157.0	53	65	46.0	$^{\dagger\circ}$32.6	11.0	13.9	*1.1	0.8	1.1
Mth-36	$^{\dagger\circ}$85.6	54.0	71.0	$^{\dagger\circ}$86.0	54.0	71.0	$^{\dagger\circ}$10.7	0.0	0.0	$^{\dagger\circ}$101.0	54	71	48.4	$^{\dagger\circ}$23.3	14.7	19.3	1.5	2.2	2.8
Mth-52	$^{\dagger\circ}$1509.6	181.0	187.0	$^{\dagger\circ}$1497.0	181.0	187.0	$^{\dagger\circ}$244.0	0.0	0.0	$^{\dagger\circ}$1869.0	181	187	6.0	$^{\dagger\circ}$55.1	6.6	6.8	$^{*\bullet}$1.1	0.2	0.5
Mth-53	†244.1	88.3	307.0	†255.5	82.5	307.0	$^{\dagger\circ}$60.0	24.2	0.0	$^{\dagger\circ}$311.0	142	307	26.5	†46.5	16.8	58.5	1.4	3.1	13.1
Mth-55	$^{\dagger\circ}$536.2	266.0	266.0	$^{\dagger\circ}$542.0	266.0	266.0	$^{\dagger\circ}$24.3	0.0	0.0	$^{\dagger\circ}$567.0	266	266	19.0	$^{\dagger\circ}$68.5	34.0	34.0	$^{*\bullet}$1.5	0.8	1.0
Mth-56	$^{\dagger\circ}$89.8	45.0	47.0	$^{\dagger\circ}$90.0	45.0	47.0	$^{\dagger\circ}$9.5	0.0	0.0	$^{\dagger\circ}$111.0	45	47	0.0	$^{\dagger\circ}$57.2	28.7	29.9	$^{*\bullet}$0.5	<0.1	<0.1

† $EvoSuite_{Amp}$ performs significantly better than $DSpot_{Mut}$ (p-value < 0.01)
$^{\circ}$ $EvoSuite_{Amp}$ performs significantly better than $DSpot_{Cov}$ (p-value < 0.01)
* $EvoSuite_{Amp}$ generates more lines of code than $DSpot_{Mut}$
$^{\bullet}$ $EvoSuite_{Amp}$ generates more lines of code than $DSpot_{Cov}$

Table 4.3: *Spearman correlation value (ρ) between test suite size (LOC) and mutation score.*

Tool	p-value	Correlation (ρ)
EvoSuite$_{Amp}$	<0.01	0.698
DSpot$_{Mut}$	<0.01	0.588
DSpot$_{Cov}$	<0.01	0.608

studied resulted in improved test suite with a smaller number of lines of code[1] (KLOC) when DSpot$_{Mut}$ was used. Similarly when comparing DSpot$_{Cov}$ to EvoSuite$_{Amp}$, 26 out of the 42 class versions had smaller test suites with DSpot$_{Cov}$. Furthermore, when comparing DSpot$_{Mut}$ to EvoSuite$_{Amp}$, the improved test suites for 27 out of 42 class versions (64.3%) had a better ratio of killing mutants per line of code with DSpot$_{Mut}$, and similarly 26 out of 42 versions (61.9%) were better with DSpot$_{Cov}$ than EvoSuite$_{Amp}$. In all seven versions in which DSpot$_{Mut}$ has a better mutation score, it improves test suites with a smaller KLOC compared to EvoSuite$_{Amp}$. As an example, Cli-39 as shown in Table 4.2 part G, EvoSuite$_{Amp}$ generates 0.2 KLOC to kill one mutant, while DSpot$_{Mut}$ generates 0.1 KLOC to kill two mutants. When comparing EvoSuite$_{Amp}$ to DSpot$_{Cov}$, where DSpot$_{Cov}$ has a better mutation score, 8 out of 15 class versions (53.3%) have a lower number of LOC. As an example, for Cdc-11, EvoSuite$_{Amp}$ generates 0.2 KLOC while killing around 18 mutants and DSpot$_{Cov}$ generates 0.3 KLOC, while killing 20 mutants. On the contrary, Jsp-84, EvoSuite$_{Amp}$ generates 0.2 KLOC while killing around six mutants, and DSpot$_{Cov}$ generates 0.1 KLOC, while killing 27 mutants.

In order to verify whether there is a correlation between the generated test suite KLOC size and the increase of mutation score, I used the Spearman rank correlation measure. Table 4.3 presents the correlation coefficients of each tool. There is a strong correlation (ρ) between the EvoSuite$_{Amp}$ size of the test, and the increase in mutation score. In both DSpot$_{Mut}$ and DSpot$_{Cov}$, there is a moderate ($\rho >$0.4) correlation between the size and increase of the mutation score, and high correlation ($\rho >$0.7) for EvoSuite$_{Amp}$. All the tools' p-values are less than 0.01, meaning that there is statistical significance.

Conclusion (RQ3: Test Suite Size).
Overall, EvoSuite$_{Amp}$ tends to generate a larger final test suite when compared to DSpot$_{Mut}$ and DSpot$_{Cov}$.

[1]The lines of code metric was derived using the `cloc` (Count Lines of Code) library, which counts only lines that contain executable code while omitting blank lines and comments.

RQ4: Consistency

In order to investigate the non-determinism rate on each tool, I calculated the mean, median, and standard deviation (σ) of the mutation score for all 42 subject class versions over each of the respective 30 re-runs. Table 4.2 (parts B and C) shows the results of the calculations. The mutation score EvoSuite$_{Amp}$ produces has a greater standard deviation when compared DSpot$_{Mut}$ and DSpot$_{Cov}$. There were only 7 out of the 42 versions (16.6%) for which the DSpot$_{Mut}$ produced a higher standard deviation, while there was zero for DSpot$_{Cov}$.

Conclusion (RQ4: Consistency).
EvoSuite$_{Amp}$ tends to show more varied behaviour when compared to DSpot$_{Mut}$ and DSpot$_{Cov}$.

4.5 Threats to Validity

Naturally, there are threats to validity associated with this study [162]. The first is associated with subject selection. I chose to use versions of classes that are part of the Defects4J benchmark, yet not all of the classes it provides could be used in this study, due to problems in getting DSpot to work. However, I was able to use 42 versions of 29 unique classes in 7 projects, which still provides a suitable number and diversity of subjects to carry out these experiments and draw conclusions from the results. Another threat is related to how mutation score is calculated, since EvoSuite and DSpot use different mechanisms. EvoSuite provides its own implementation of a mutation analysis pipeline, while DSpot uses PITest. To control this threat, I used a third tool, Major, to provide an unbiased assessment across the results of the two tools. To control the threats related to the non-deterministic behaviour of both tools, I repeated this experiments 30 times. To mitigate the threats associated with this statistical analysis, and assumptions about the normality of the statistical distributions of this results, I used non-parametric statistical tests. Finally, after generating the test cases using both EvoSuite$_{Amp}$ and DSpot, there are some cases where it needs other test files to run, due to dependencies. This could have an impact when calculating the mutation score. To mitigate this problem, I ensure that all improved test suites retained access to any dependent libraries and code.

4.6 Discussion

Mutation Score

The EvoSuite$_{Amp}$ tool, in general, kills more mutants than DSpot, which implies that using the distance to mutation fitness function that is provided in EvoSuite can kill mutants that DSpot finds hard to kill. In the case of amplifying developer-written tests using DSpot$_{Cov}$, it is not surprising that an increase in code coverage also helped to increase the mutation score, as mutants that are not reached by developer-written tests could not be detected. Overall, the results show that EvoSuite's evolution and mutation analysis technique is much more suited to improving test suites to kill mutants than DSpot.

Unique Mutants

Furthermore, EvoSuite$_{Amp}$ finds and kills more unique mutants after 30 runs when compared to DSpot$_{Mut}$ and DSpot$_{Cov}$. This shows that EvoSuite explores more parts of the program than DSpot within the 30 runs and that it could find more unique mutants, further adding to this finding that it is better at improving test suites to kill mutants than DSpot.

Test Suite Size

In answering RQ3, I found that EvoSuite$_{Amp}$ usually creates a bigger test suite when compared to the two configurations of DSpot, and that there is a high correlation between killing mutants and a big test suite. However, by looking at the mutants killed per number of lines of code, the value is not significantly bigger. I set EvoSuite$_{Amp}$ to not run the minimisation technique that the default EvoSuite does (see Section 4.2), to avoid original developer-written tests being discarded—however, enabling this technique could reduce the lines of code while maintaining the mutation score. I leave this experiment as an item for future work.

Consistency of Mutation Score Results

Finally, RQ4 shows that both DSpot$_{Mut}$ and DSpot$_{Cov}$ give more consistent mutation score results over the 30 runs with each subject class version. This potentially means, however, that EvoSuite$_{Amp}$ has a higher chance of exploring more edge cases due the higher degree of stochasticity that it evolves the developer-written tests, and thereby could find more unique mutants to be killed, as shown by the answer to RQ2.

Readability of Final Tests

Anecdotally, I noted that the tests produced by EvoSuite$_{Amp}$ were less readable than DSpot's. Some of this was due to the inevitable disruption caused by the evolutionary operators (although I deliberately turned some of these off for this reason—see Section 4.3.2). In particular, the carving procedure adapts developer-written tests to EvoSuite's internal test case representation, which causes them to lose some of their original qualities. This is something that needs to be investigated in future work.

4.7 Chapter Conclusions and Future Work

The current state-of-the-art test amplification tools aim to improve developer-written tests, but are limited in the changes they can make and are not guided by fine-grained fitness information. Search-based test case generation tools like EvoSuite, on the other hand, can benefit from the guidance provided by fitness functions, and have many more control over the structure of tests, but are limited in terms of reusing of developer-written tests and the final readability of the tests they generate.

In this chapter, I introduce a modified version of EvoSuite, EvoSuite$_{Amp}$, that uses EvoSuite's carving functionality to start the search from developer-written test, and evolves these tests with the goal of killing more mutants. When evaluating it against the state-of-the-art Java test amplification tool DSpot, EvoSuite$_{Amp}$ was better at killing more mutants and killing more unique mutants that DSpot was found to never kill in any of the 30 re-runs of these experiments.

This chapter highlights that automated tools can kill more mutants by using developer-written tests as seeds, providing greater flexibility in modifying those tests, as well as providing proper fitness guidance. However, the downside is less readability of the final tests, since they do not have similar structure as the original tests written by developers.

Besides that, in Section 2.6, I have discussed test brittleness as a critial aspect of test suite health. As discussed in Section 2.6.2, developers are generally advised to avoid directly invoke implementation-detail methods, which is another indicator that a test suite is not healthy. To better understand about this issue, an empirical study on open-source projects and a survey of developers' opinion regarding this advice are necessary. Such study could provide a more detailed understanding of how to balance fitness goals with best practices in writing maintainable and healthy test suite.

Chapter 5

Testing via Public APIs vs Implementation Details

The contents of this chapter is based on “Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. Private — Keep Out? Understanding How Developers Account for Code Visibility in Unit Testing. In *International Conference on Software Maintenance and Evolution (ICSME)*, 2024”.

5.1 Introduction

As software evolves, maintaining the health of regression test suites is critical but time-consuming [216, 165]. New tests must be added, outdated tests must be removed, and existing tests often become brittle, breaking due to minor changes in the codebase [152]. This maintenance effort can be challenging and developers might opt to discard these false-positive broken tests, potentially weakening the test suite overtime [152, 166]. As discussed in Section 2.6.2, good practices recommend writing tests that validate the behaviour of the software—focusing on the Public APIs—rather than implementation details, to ensure that tests does not spuriously break unnecessarily [165, 83, 117].

In this study, I examine 226,915 JUnit tests of 4,801 open-source Java projects collected from the Maven Central Repository to understand whether developer follows this principle in practice or not. Besides that, to investigate this issue in more detail, I conducted the first qualitative study on why developers choose to test through a unit’s public API or directly test its non-public methods, risking the reliability of test suite in the future. This study involved a survey of 73 developers and a systematic analysis of 60 StackOverflow threads from 2008 [46] to 2023 [53]. With the help of two other collaborators of this paper, I was able to identify few insights, through numerical and thematic analysis of questionnaire responses and StackOverflow posts.

In the open-source projects, I found that more than a quarter (28.04 %) of the projects consists of at least one non-public method directly invoked in their tests and proportionally, the percentage of package-private method being the highest when compared to other access modifiers. Besides that, approximately two-thirds of developers surveyed, plus those posting on StackOverflow, strictly adhere to testing public APIs only, believing that testing non-public methods reflects poor code design and suggest refactoring of production code is needed instead. In contrast, developers that test non-public methods directly argue that it's necessary to ensure complex implementation details are well-tested, and the undesirable complexity involved in having to write tests solely from the public API. These developers often use various strategies to make non-public accessible for testing, such as raising the visibility of methods, so they statically become accessible to test frameworks. The finding from the open-source Java projects particularly supports this, with a large number of non-public methods being called directly from tests, including a disproportionate number designated as "package-private" — i.e., with enough visibility that they can be invoked by JUnit tests without using any reflection mechanism.

I provide several implications for future work, including how to provide automated support to developers to help them avoid the temptation of bypassing public APIs. This chapter contributes the following to the understanding another issue in test health:

1. **Open-Source Study:** An empirical analysis of the visibility of methods called directly from the tests across 4,801 Java projects from the Maven Central Repository (Section 5.4.1)

2. **Developer Survey and StackOverflow Threads:** A numerical and thematic analysis of 73 developer survey responses and 60 StackOverflow threads to identify developer attitudes and approaches to testing public APIs versus testing non-public methods directly (Section 5.4.2, 5.4.3, and 5.4.4);

3. **Findings and Recommendations:** A discussion of this findings, summarising the current practices (Section 5.6), and suggesting directions for future work to improve another issue regarding to test health in (Section 5.7).

5.2 Research Questions

A unit typically comprises several accessible methods, that form its "public API", which can called from other units. Developers also include non-public methods in units to implement functionality that should not be invoked externally outside of the unit. These methods tend to contain useful routines that are utilised within the bodies of other methods, whether it's public or otherwise. Assuming that the unit does not contain any unreachable code, all non-public methods should be invoked by a public method at some point, implying that the entire unit should be testable through its public API.

My focus is on understanding how developers consider different visibility levels when writing unit testing. Given the long-term benefits of a maintainable test suite, do developers strictly test units through public API? Or, do they bypass public APIs and directly test non-public methods, and if so, why? I aim to address these four research questions:

RQ1 (Open-Source Testing). How frequently are public and non-public methods directly invoked from tests in open-source code? Do open-source developers test against public APIs only, or do they also make direct calls to non-public methods in their tests?

RQ2 (Stance). What proportion of developers believe that tests should be written against a unit's public API only, compared to those willing to test non-public methods directly?

RQ3 (Rationale). What are the reasons why developers take a particular stance on the issue of testing public APIs only versus testing non-public methods directly?

RQ4 (Practice). How do developers go about unit testing code that contains both public and non-public methods? Do they test non-public methods directly, for example, or do they test indirectly via public methods as part of a behaviour-driven testing approach? How do developers test when their language does not have access modifiers? What, if any, changes to testing frameworks developers would like to see in the future in respect of these topics?

5.3 Methodology

5.3.1 Open-Source Study (RQ1)

For this research question, I developed a tool called Viscount [226], to measure the number of times methods with different levels of visibility are being directly called from tests in open-source software.

I selected open-source Java projects as subjects of this evaluation because Java offers various levels of access modifiers (i.e., public, protected, package-private, and private) and is a mature language with a vast number of publicly available open-source projects.

The initial dataset of open-source projects is similar to the one used in Chapter 3, where I collated the URLs of 38,841 Github-hosted Java projects listed in the index of the Maven Central Repository (as of 2023-04-13). Through scripted automation, I found 7,998 of these projects could be built without errors using Java 8 (the most commonly used version of Java in 2023 [54]) and Maven 3.9.6.

Viscount begins by statically extracting the access modifiers of each project's production methods using the Spoon framework [213]. It then employs a dynamic approach to trace methods called directly from tests, accurately accounting for calls made via Java Reflection, via virtual dynamic calls of the intended method, or through a mocking library that use similar techniques as Java Reflection. To achieve this, Viscount instruments each project's bytecode using Javassist [96], inserting log statements at the method and constructor entry and exit points (i.e., return statements and throw

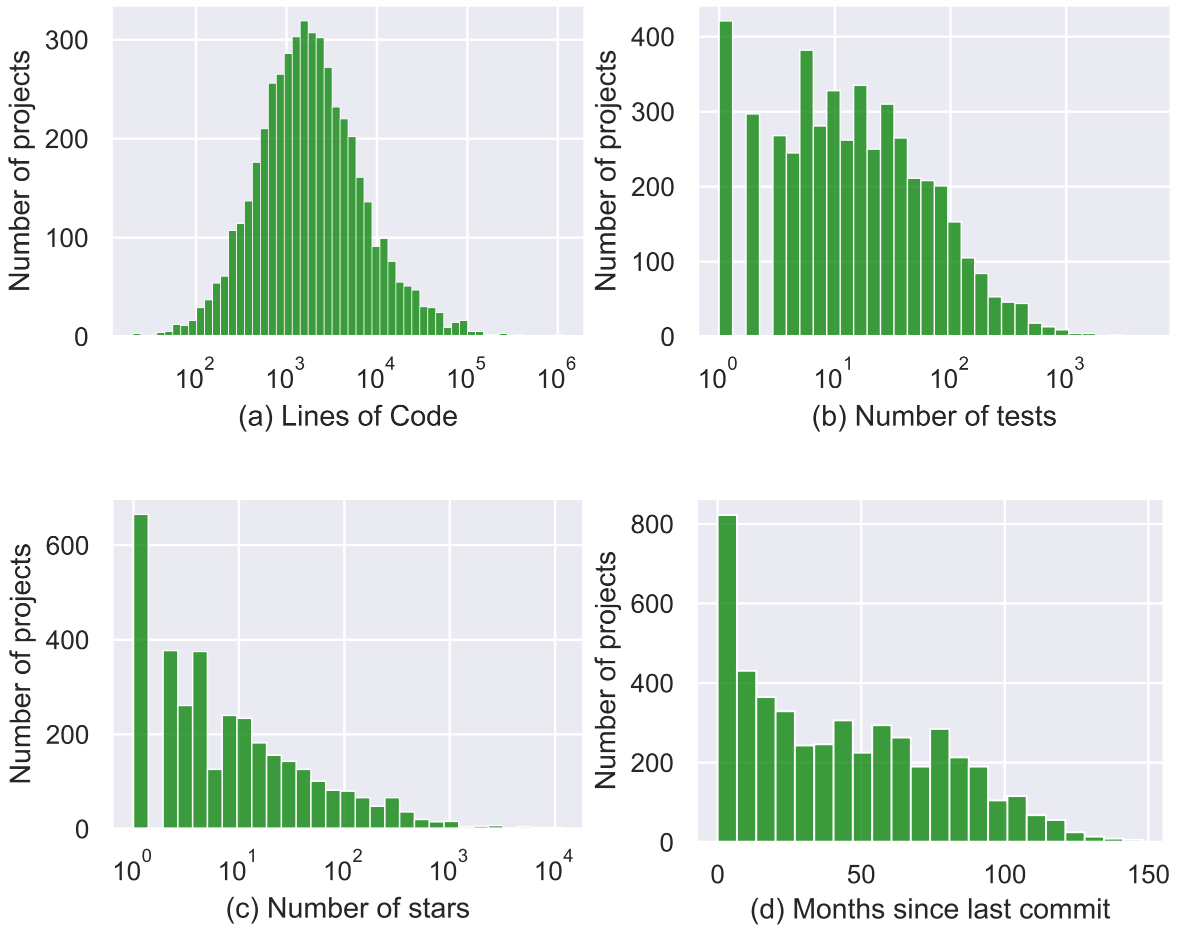


Figure 5.1: *Statistics of the Maven open-source projects studied in RQ1.*

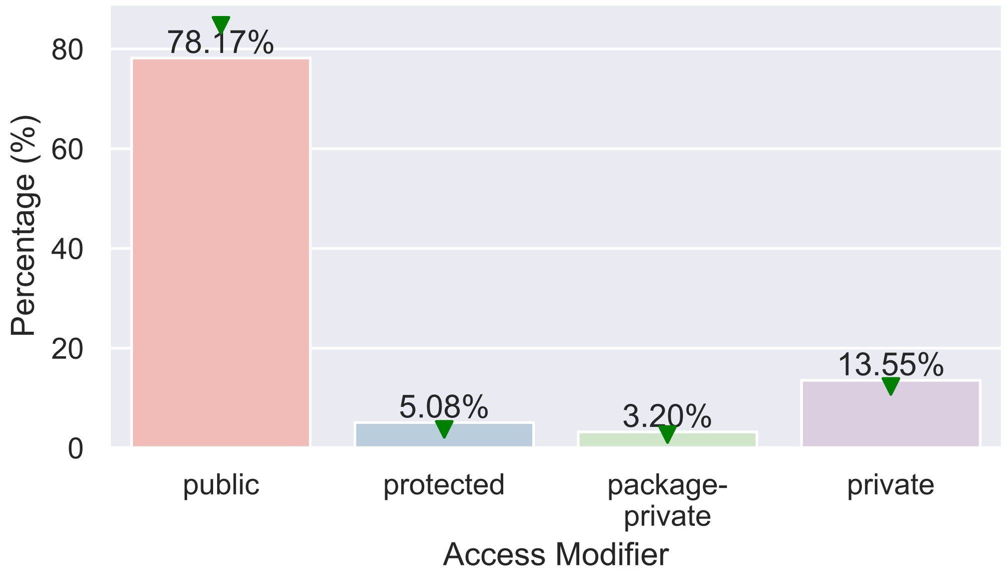


Figure 5.2: *Distribution of Access Modifier being used per project in the production method, by percentage, across all Maven open-source projects studied in RQ1.*

exceptions). It then runs the project's test suites to collect logging information and identify all direct invocation made from any test. I allocated a maximum of three hours for the execution Viscount per each project, discarding projects that did not complete within this time frame. Projects that failed to produce any logs, typically due to a lack of test suites, were also discarded. Additionally, Viscount excludes any tests that involve multi-threaded code, as accurately capturing traces in such cases is challenging due to interleaved method entry and exit points in the logs (a common issue affecting similar tools, e.g., [138]). Viscount then performs static analysis on each project's source code to confirm that the logged production methods were directly from tests, rather than through third-party libraries or the Java API itself (possible, for example, through Java's Serialisation libraries). I then manually inspected any ambiguous cases that Viscount could not automatically resolve. More detailed information about Viscount will be discussed in Appendix A.

The final sample for RQ1 consisted of 226,915 tests from 4,801 Java projects. Project statistics are as shown in Figure 5.1, highlighting a range of projects that are large to small in size. The mean lines of code per project in the sample was 5841.02, with an average of 47 tests and 54.9 GitHub stars. The average time since the last commit at the time of collection was 43 months. The average distribution of the production method visibility in these 4,801 is shown in Figure 5.2, with majority of the production methods being public.

5.3.2 Developer Survey (RQs 2–4).

I designed a questionnaire that included a mixture of multiple-choice and open-ended questions. I conducted a pilot study with professionals and academic colleagues, and carried out several iterations with a small group of PhD students, refining the design after each trial (e.g., improving the clarity of what constitutes a non-public method and reordering the questions). The final survey included questions from four different perspectives:

- *Demographic* questions gathered information about the developer's years of experience in development and testing, and their main programming language.
- *Stance*-related questions explored the developer's opinions on testing only public APIs vs testing non-public methods directly, which I used to address RQ2.
- *Rationale*-related questions focussed on the reasons behind the developers' opinions, providing data to answer RQ3.
- *Practical*-related questions aimed at understanding the approaches developers use in their everyday development practice. The responses to these questions were used to answer RQ4.

The full developer survey questionnaire and background information sheet is as shown in Appendix C. The background information sheet provides a language-agnostic definition of "non-public" methods. This included explanation of each access modifiers and common conventions to denote visibility across various programming languages, such as C++, C#, Java, Go, Kotlin, and Rust, as well as dynamically typed languages such as JavaScript, Python, and Ruby, and those enhanced with enforced type annotations, e.g., TypeScript.

The questionnaire was made available for three weeks using Google Forms and was distributed via LinkedIn, X (formerly Twitter), regional technology forums, and through personal industrial contacts, who were also asked to share it with their colleagues. To mitigate the risk of bot infiltration, I required each participant to sign in to their Google account, which also successfully prevented multiple submissions from the same person. I did not collect their account information or track their identity.

In total, I received 73 responses. The median range of development experience among participants was 10-15 years, with the mode being 15+ years; for unit testing experience, both the median and mode fell within the 5–10 years range. Figure 5.3 shows the distribution of years of experience in software development and Figure 5.4

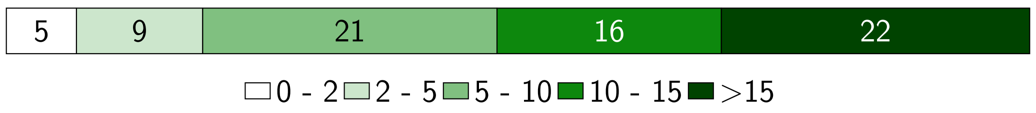


Figure 5.3: *Years of experience in software development*

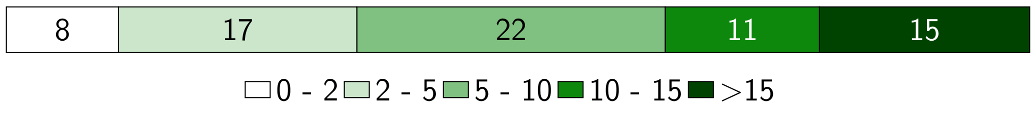


Figure 5.4: *Years of experience in writing unit tests*

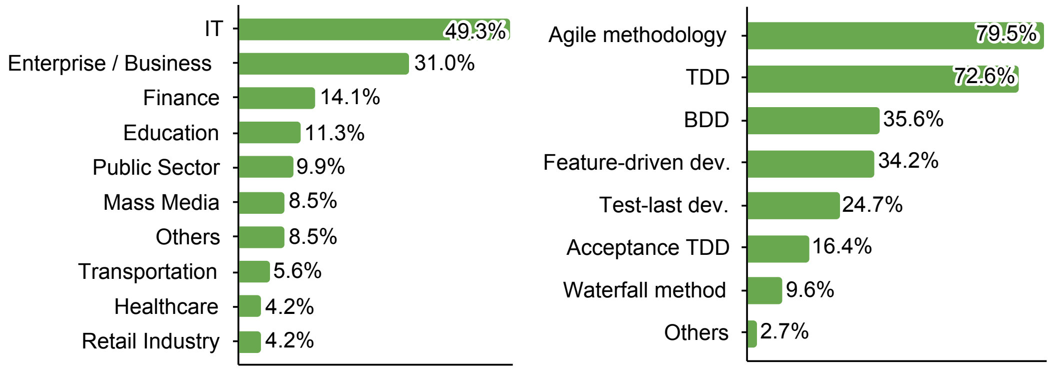


Figure 5.5: *Distributions of participants current working industry and software development methodologies that they practices*

shows the distribution of unit testing experience among the participants. When asked about their primary programming language, participants responded with Java (25), Python (15), TypeScript (9), C# (7), PHP (6), Kotlin (4), C++ (2), JavaScript (2), Ruby (2), and Scala (1). I analysed the frequency of responses for the fixed-choice questions using Kendall's tau-b correlation coefficient to evaluate the strength of relationship between different ordinal answers [20], as ordinal data typically do not follow a normal distribution and non-parametric tests like Kendall's tau-b are more appropriate for this type of analysis [98]. For open-ended questions, I applied inductive thematic analysis [100] to the answers, coding each response that capture its key concepts. Since free-text responses to open-ended questions often contain multiple components, I divided the responses into seperate elements when conjunctions such as *"and"*, *"or"*, and *"but"* were being used. I conducted this analysis collaboratively with the other two collaborators of this chapter's publication to keep this coding as consistent as possible and to minimise any individual biases. I then grouped similar codes into overarching themes. In the results section (Section 5.4), I discuss the major themes to which I attributed more than two comments.

As shown in Figure 5.5, the majority of the participants indicated that they work in the IT industry, use agile development methodologies (79.5%) and practice test-driven development (72.6%). (Both questions allowed for multiple responses.)

5.3.3 StackOverflow Analysis (RQs 2–4)

To better capture a variety of perspectives and gain a comprehensive understanding of developers' views on testing non-public methods, I used triangulation [212] to complement the results of the developer questionnaire by analysing online discussions on StackOverflow, one of the biggest question-and-answer website for software developers [37].

I formulated the following four search queries on StackOverflow: "*test non-public method*", "*test private method*", "*test protected method*", and "*test package-private method*".

The rationale behind these queries was to retrieve threads discussing the testing of non-public methods in the general context, as well as threads focusing on the testing of specific types of non-public methods. This led to the inclusion of the keywords "private", "protected", and "package-private" as search terms. To refine the scope of the search results, I added the "*[unit-testing]*" tag to each query term. I deliberately avoid further narrowing the search queries to avoid missing relevant threads in the search results. Similarly, I did not specifically mention any programming language in these queries. The keywords "private" and "protected" were chosen because they represent different levels of visibility across various programming languages, e.g., C#, C++, Java, Kotlin, and PHP. Although "package-private" is specific to Java, including it allows me to triangulate these results with those from the empirical study of Java projects for RQ1 (Section 5.3.1).

I executed this search queries on 16th of November 2023, and gathered the top 25 threads for each query, sorted by relevance. I eliminated duplicates across queries and also discarded any insubstantial threads; i.e., those where a genuine discussion did not materialise, resulting in fewer than two responses. This process left me with 60 unique threads in total, including original posts from between 2008 to 2023.

I then manually analysed these 60 threads. Firstly, I categorised each of the thread as either "Debate" or "Practical", based on the style of the question. "Practical" threads, which were used to answer RQ4, primarily involved posters asking specific questions related to testing non-public methods within the context of particular programming language or testing framework. "Debate" threads, used to address RQs 2 and 3, were more general and/or conceptual, where the poster's main intent was to seek the community's opinions on testing non-public methods, sometimes in the context of the specific scenario presented in the question. The categorisation I made was usually clear from the text of the thread, with some posters explicitly stating their technical question not to devolve into a discussion about the benefits and drawbacks of directly testing non-public methods (e.g., [49], *"I would not like to discuss whether I should test privates or not but [...] focus on how to test it."*). In the case of "Practical" threads, I focused only on the top answer—i.e., the one with the highest number of up-votes. For "Debate" threads, I extended this to the top three answers to account for a variety of opinions and provide a more complete understanding of the discussion. I did not extend this analysis to further posts to maintain the quality and manage the number of posts I had to manually examine. In cases where "Debate" thread has fewer than three answers, I just analysed the one or two responses available. I then applied inductive thematic analysis to the questions and answers in these threads, just as what I did for free-text responses in the developer questionnaire.

5.4 Results

5.4.1 RQ1 (Open-Source Testing)

Figure 5.6 illustrates the distribution of open-source Java projects based on how frequently unit tests directly invoke production code methods with different access modifiers.

Out of 4,801 projects, 3,455 (72%) have tests that exclusively call public methods, while the remaining 1,346 projects (28%) have tests that make at least one call to a non-public method. Among these, projects with tests directly calling package-private methods were most common (828 projects), followed by those with direct calls to protected (698 projects) and private (42 projects) methods. In 64 projects (1.3%), the tests exclusively call non-public methods, with 37, 23, and 1 projects' tests *exclusively* targeting package-private, protected, and private methods respectively. Additionally, 3 projects have tests that only call package-private and protected methods directly. The one project that exclusively calls only private methods contains only a single test [32], which directly tests a private method using Java Reflection [31] to bypass the visibility restriction. Regarding tests that directly call private methods, as indicated in Table 5.2, 68 used Java Reflection to override visibility restrictions to make the method accessible for invocation. The remaining 48 used third-party libraries to achieve the same effect, including mechanisms in mocking frameworks such as PowerMock Whitebox [33] and EasyMock- ReflectionUtils [15].

While the Venn diagram provides a useful overview of method being invoked directly in test on a per-project basis, differences in project sizes could give a misleading impression of the overall frequencies. To address this, I therefore tracked the number of production methods being called by access modifier type across all projects, as detailed in Table 5.1. I counted a production method as executed if directly invoked at least once by at least one test. The vast majority of direct calls made by tests are to public methods. This might also be due to the average number of public methods in production code per project is an order of magnitude higher, as shown in Figure 5.2. The number of calls to package-private exceeds that of protected and private, and overall, the proportion of package-private methods that are directly called is greater than that for public.

Since package-private methods can be accessed by JUnit tests without breaking the encapsulation of the method outside of its package[1], this might indicate that developers

[1]Although Maven organises production code and tests into separate directories, the same logical package names can be used, meaning that tests can be in the same package as the classes that they

Table 5.1: *Numbers of production code methods, by access modifier, directly invoked from test suites ("Invoked") out of the total number of methods ("Total") in all* 4,801 *Java projects studied in RQ1.*

Access Modifier	Invoked	Total	Percentage
public	179,640	1,468,185	12.2%
protected	2,601	70,579	3.7%
package-private	4,243	27,459	15.5%
private	118	132,701	0.1%

intentionally make methods package-private to test them, rather than leaving them private. However, because I cannot know the original intentions of the developers of these projects, I revisit this in the subsequent developer survey and StackOverflow analysis as part of the next research questions.

Conclusion (RQ1 (Open-Source Testing)). Open-source developers directly invoke both public and non-public methods in their tests. In 72% of the projects studied, tests only call public methods directly, while the remaining 28% include at least one call to a non-public method. Although the raw number of non-public methods invoked from tests is small compared to the number of public methods, this is largely because to the number of public methods being an order of magnitude higher (Figure 5.2). Proportionally, package-private methods are the most frequently called type of method by access modifier.

5.4.2 RQ2 (Stance)

I asked developers whether they agreed with the statement "*In general, developers should write unit tests that only invoke public methods, avoiding direct calls to non-public methods*". Figure 5.7 summarises their responses, showing that nearly two-thirds (64%) of participants either *agreed* or *strongly agreed*; while 30% *disagreed* or *strongly disagreed*, and the remaining 6% were unsure.

I also asked developers *"How often do you write tests that directly invoke non-public methods?"*. The most common answer selected by partipiants were "Never" and "Rarely", as illustrated in Figure 5.8. This figure also shows that the frequencies of answers ("Always", "Often", "Sometimes", "Rarely", "Never") aligns with the levels of agreement observed in Figure 5.7. Kendall's tau-b indicating a significant strong association between these sets of answers ($\tau_b = 0.463$, $p < 0.01$).

Additionally, I also analysed the 18 StackOverflow threads I classified as "Debate"-based (Section 5.3.3). I categorised the top three responses in each thread as either

test despite being in different directories on the file system [39, 23].

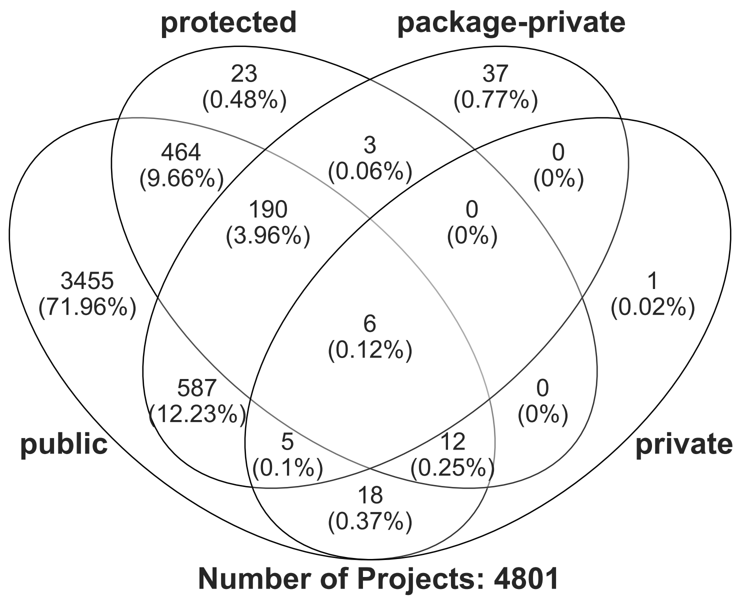


Figure 5.6: *Venn diagram of open-source Java projects, grouped by visibility of production code methods called directly from their tests.*

Table 5.2: *Mechanisms used to directly invoke private methods in tests, by numbers of tests ("# Tests") and projects ("# Projects"), in the dataset of Java projects.*

Framework	# Tests	# Projects
Java Reflection [31]	68	31
JMockit Deencapsulation [19]	14	1
PowerMock Whitebox [33]	13	5
Apache Commons Lang3 MethodUtils [5]	12	4
Manifold Jailbreak [22]	6	1
EasyMock ReflectionUtils [15]	1	1
Spring Framework ReflectionTestUtils [36]	1	1
tvd12 test-util MethodInvoker [41]	1	1

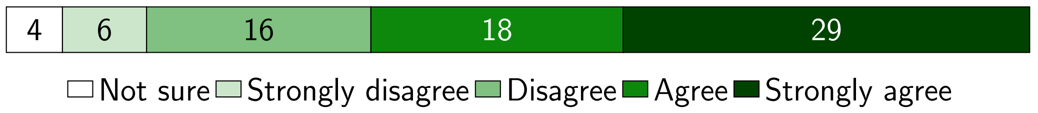


Figure 5.7: *Developer responses to the question "To what extent do you agree with the following statement? 'In general, developers should write unit tests that only invoke public methods, avoiding direct calls to non-public methods.'"*

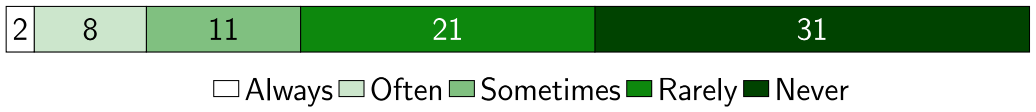


Figure 5.8: *Developer responses to the survey question: "How often do you write tests that directly invoke non-public methods?"*

in favour of testing only public APIs (19; ∼40%), supporting of testing non-public methods directly (17; 35%), or being neutral in stance (12; ∼25%). Similar to the developer survey, I observe that there was more support for testing public APIs only, though by a smaller margin. These posts were also more likely to receive more upvotes from users of the site, and consequently be ranked first in the list of answers to the original question.

Conclusion (RQ2 (Stance)). Among the developer survey participants, nearly two-thirds prefer to test only public APIs, while the remaining third are open to testing non-public methods directly. A similar trend was observed in the answers in StackOverflow "Debate" threads, where responses advocating to test only via public APIs were more frequently the top answers than those supporting directly testing of non-public methods.

5.4.3 RQ3 (Rationale)

To gain deeper understanding on developers' stance, I divided participants into two groups: "agree" and "disagree", based on their responses to the statement *"In general, developers should write unit tests that only invoke public methods, avoiding direct calls to non-public methods"* (Figure 5.7). The "agree" group consists of participants who *agreed* or *strongly agreed* with the statement, indicating a preference for testing only through public methods. In contrast, the "disagree" group includes those who *disagreed* or *strongly disagreed*.

I also asked developers to assess the importance of eight different qualities of test suites, as shown in Table 5.4. The table highlights some interesting differences between the two groups. Participants in the "agree" group most frequently rated all the qualities as "Very Important", with the exception of code coverage, which they rated as "Important". On the other hand, the "disagree" group marked code

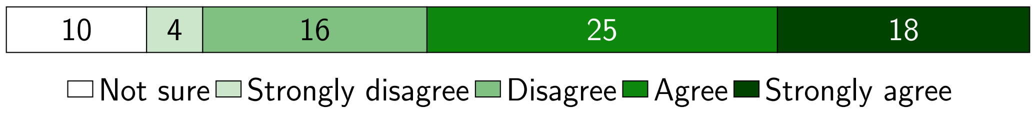


Figure 5.9: *Developer responses to the question: "Testing non-public methods leads to more tests failing spuriously when modifications are made to those methods"*

Table 5.3: *Classification of posters' answers to StackOverflow "Debate" threads as either arguing to test public APIs only or test non-public methods directly, or were neutral in stance. "Rank" refers to whether the answer was first, second, or third in the list of responses to the original post, depending its "upvotes". (NB: Some posts received fewer than three responses.)*

Classification	# Answers	Mean Rank	Mode Rank
Test public APIs only	19	1.89	1
Test non-public methods directly	17	1.94	2
Neutral	12	1.83	2

coverage as "Very Important". This suggests that testers in the "disagree" group place greater emphasis on exercising implementation, as achieving higher coverage may require directly invoking non-public methods from a test to execute a statement that are otherwise hard to cover. In contrast, members of the "agree" group, who favour public-only testing, appears more concerned with testing behaviour. Some participants explicitly express this view in their free-text responses in this survey. For instance, Participant P_{49} remarked *"It's important that unit tests test the behaviour and not the implementation"*, while P_{72} stated *"I think that testing public rather than non-public methods leans more towards testing behaviour rather than implementation which is usually preferable"*.

I also asked developers to what extent they agree with the statement "*Testing non-public methods leads to more tests failing spuriously when modifications are made to those methods.*" I found that those in the "agree" group also tended to agree or strongly agree with this statement. Kendall's tau-b test revealed a strong and significant association: ($\tau_b = 0.53$, $p < 0.01$). However, there was no such obvious correlation for the "disagree" group as I only found Kendall's tau-b test value of ($\tau_b < 0.2$, $p < 0.01$). Figure 5.9 summarised the distribution of responses. Public-only testers are thus more concerned about tests breaking during an internal refactoring if they are tightly coupled to non-public methods through direct invocations, further supporting the idea that this group prefer to test behaviour over implementation. This perspective was echoed

Table 5.4: *Developer responses to the question "To what extent do you value the following aspects when writing unit tests?"*

The participant could choose from the options "No opinion" □ , "Not Important" 🟩 , "Somewhat Important" 🟩 , "Important" 🟩 , and "Very Important" 🟩 .

Aspect	**"Agree" group**	**"Disagree" group**
Code coverage	10 19 16	7 5 10
Capturing behaviour via assertions	9 36	3 17
Ease of debugging	5 16 23	5 10 6
Robustness after refactoring	11 32	3 8 11
Sensitivity to change	14 28	3 7 10
Realism	10 7 28	4 5 11
Confidence in code	6 38	3 3 15
Conciseness	6 10 13 17	8 9 3

by several participants in free-text responses. For example, P$_{65}$ said *"Writing tests that are coupled to the private implementation of a class hurts the ability to refactor in the future"*. Additionally, the "agree" group pointed out potential process or design flaws within the production code. P$_{21}$ noted, *"If you have to do it, that means you probably have a code smell. . . "*; P$_{43}$ agreed with this, saying *". . . that may be a sign that there is a design/decomposition problem"*, while P$_{58}$ stated it could indicate *"a sign of a leaky abstraction"*. P$_{25}$ also pointed out that not testing through public methods could lead to unrealistic tests: *"even if a private method CAN handle a variety of inputs, it's pointless to test for every possible combination, because you know the code that is calling the private method will never present those possibilities"*.

In contrast, developers who disagreed with the public-only approach cited the complexity of logic in non-public methods and the need to thoroughly tests them. For instance stated by P$_{55}$: *"If the logic within the non-public is complex and critical, then do what you need to do to test them"*. P$_{31}$ also stated that testing non-public methods directly is often less verbose and potentially easier and clearer: *". . . if its [sic] clearer/easier to test an internal part of a flow rather than setting up for testing the public method then that's a better idea"*.

Finally, I conducted a thematic analysis of the original poster questions that initiated the 18 "debate" threads I retrieved from StackOverflow.

These threads provided additional insights into the motivations of developers considering testing non-public methods, revealing the following themes:

– Avoiding breaking encapsulation for testability's sake: This theme included question posts whereby the original posters, unlike some survey respondents who viewed the need to directly test non-public methods as a code design flaw, felt they would have to compromise code design to *avoid* such testing of non-public methods. One poster wrote, *"If I wrote this class this way, I could have unit tests [. . .] but I feel like this pattern is not correct. Is there a better way?"* [50].

– Curiosity: This theme featured posters that are curious about whether testing non-public methods is good practice or not.

– Complex logic: Echoing participants in this developer survey, this theme characterised by posters who wanted to directly test non-public methods due to the complexity.

– Complexity of testing through the public interface: Posters described another reason that did not emerge as an explicit theme in the developer survey: they felt that testing through public methods was too laborious and repetitive, leading to duplicated code across tests — *"the method contains logic shared between other methods in the class and it's tidier to test the logic on its own"* [47]; or simply too complex in its own right: *"I want to test logic used in synchronous threads without having to worry about threading problems"* [47].

Conclusion (RQ3 (Rationale)). Public API testers prioritise of testing *behaviour*, while developers who directly test non-public methods focus more on the correctness of *implementation*. Public API testers view the need to test non-public methods directly as indicative of code design issues, which could lead to reliability and refactoring challenges in the future. On the hand, implementation-driven testers are more concerned about complex parts of non-public code remaining untested and about the increased complexity of test code, as testing non-public methods through the public API often require a lot of shared setup between tests.

5.4.4 RQ4 (Practice)

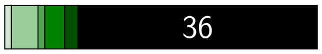
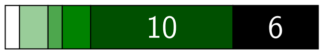
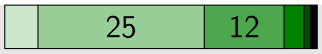
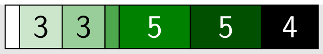
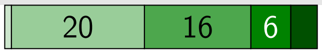
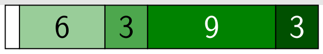
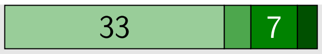
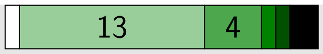
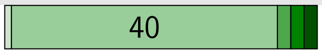
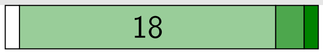
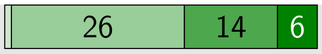
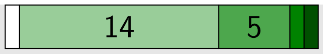
Access Modifiers in Programming Languages

I also asked developers the question: *"If your main language does not have access modifiers, do you follow any conventions to denote visibility of methods and instance variables?"*. Of the repondents, 54 participants indicated that their main programming language includes access modifiers, while the other 19 said theirs did not. Among those 19, 16 mentioned that they followed specific conventions to denote visibility instead. I then asked these participants, in a free-text follow-up question to those who said they followed conventions, which conventions they used.

This thematic analysis of responses revealed the following major themes:

Table 5.5: *Developer responses to the question: "How do you go about testing non-public methods?"*

The participant could choose from the options: "Not sure" □, "Not a feature in my language" □, "Never" 🟩, "Rarely" 🟩, "Sometimes" 🟩, "Often" 🟩, and "Mostly" ■.

Means	**"Agree" group**	**"Disagree" group**
Via public methods only		
By directly invoking		
Using reflection / mocks		
Adding test code in production		
Temporary switch to public		
Permanent change to public		

– Underscore prefixes: Almost all respondents reported using underscores to indicate non-public methods.
– Project-specific: Some participants noted that they use different visibility conventions depending on the project.
– Visibility is irrelevant for understanding: Some participants questioned the necessity of such mechanisms altogether. For example, P_{50}: *"Visibility is usually irrelevant for understanding the code"*.
– Conventions increase maintenance: Participants observed that encoding visibility in the method names (e.g., using underscores) adds to maintenance refactoring effort — P_{50}: *"encoding it in the name increases the work required to change visibility."*

Approaches to Testing Non-Public Methods

I asked developers *"How do you go about testing non-public methods?"*, and analysed the responses within the two groups defined in the previous research question. Unsurprisingly, those in the "agree" group, those who advocated for testing through public APIs only, most frequently answered "mostly" for "via public methods only" and "never" to the other options shown in Table 5.5. In contrast, the "disagree" group most commonly only rated "via public methods only" as "often", and indicated "sometimes" for testing by direct invocation, using reflection, or employing mocks.

I also asked developers about any additional methods that they use to directly test non-public methods. The thematic analysis of their free-text responses revealed the following major themes, listed in order of prevalence:

Table 5.6: *Top 3 themes for the question — "Are there any guidelines, best practices, or specific rules you follow when testing non-public methods?"*

Theme
1: Avoid testing non-public methods directly
2: Apply good software engineering process
3: Test non-public methods directly if necessary

– Refactor: Developers believe that testing non-public methods directly indicates *code smell* related to its poor design, suggesting the need for refactoring. As stated by P_{21}: *"If you HAVE TO test non-public methods, maybe they are in the wrong place (i.e. maybe they should be extracted elsewhere)"*.
– Elevate access modifier: Some developers elevate the visibility of a method to make it accessible and thus it became directly callable from a unit test. As noted by P_{39}, *"Mostly what I do (in Java) is have the method be 'package private' (the default visibility)"*, sometimes using test framework annotations to indicate the reason for the visibility change (P_{39}: *"... and use an annotation (@VisibleForTesting) to mark the reason for the visibility"*).
– Via public methods: Developers emphasised that non-public methods should be tested indirectly by invoking the public methods.

I then asked developers *"Are there any guidelines, best practices, or specific rules you follow when testing non-public methods?"*. From their responses as shown in Table 5.6, I identified two new main themes not mentioned in the previous question:
– Apply good software engineering process: In this theme, developers additionally emphasised that if a proper process or design principle had been followed, there should not be a need to directly test non-public methods. These perspectives align with the previously identified "refactor" theme.
– Test non-public methods directly if necessary: This theme captured the view of developers who believe that non-public method should be tested directly when they are complex.

Following this, I asked participants whether they took a different approach for different levels of visibility for non-public methods (protected and public). The majority (59) answered "No", while the remaining 14 replying "Yes". In a follow-up free-text question I asked participants to explain further. From their responses, I identified the following themes:
– Test what the tests have access to: Respondents focused on testing non-public methods that were visible to tests and does not produce any syntactic error. As an example, within JUnit, tests can access protected and package-private methods but not private

ones (P_{56}: *"In Java at least, protected methods can be called from classes in the same package (production and test sources can have the equivalent packages)"*; and P_{47}: *"I would look at it as non-private and private. Any method with accessibility outside of it's [sic] own class can be accessed by a unit test, and thus *could* be unit tested"*).
– Test via public methods only: Respondents emphasised that developers should focus on testing through public methods only.
– Make it visible: Similar to the previously identified "elevate access modifier" theme, if the method cannot be accessed by the tests, increase its visibility level (e.g., P_{36}: *"I elevate private to package private, and leave the others as is"*), or make it visible or accessible for testing through a wrapper class (e.g., P_{56}: *"Private methods cannot [be directly accessed], so the class must be extended to put a public wrapper around the private method."*)

Finally, I analysed StackOverflow answers in "Practical" threads (Section 5.3.3) where posters specifically ask how to test a non-public method in a given scenario. This thematic analysis revealed similar types of responses, but in a different order of prevalence, largely because the original poster was asking *how* to test non-public methods, and the responses tended to be a mix of direct answers along with opinion-based ones that accounted for the bigger picture:
– Use language-specific mechanism: Posters discussed a variety of mechanisms to gain access to non-public methods normally inaccessible for testing, including using reflection (e.g., in Java) to override access controls, and C++'s "friend" construct.

I also identified themes that I had previously seen in responses to this developer survey, thereby helping to corroborate its results:
– Test via public methods only: Posters advised testing solely via the public API; i.e., not to test non-public methods directly.
– Refactor: Posters advised to refactor production code to avoid the need to test non-public methods directly.
– Elevate access modifier: Posters in favour of side-stepping the public API advising raising the access level of the non-public so that it is visible for testing.

Tooling

Finally, I asked developers if they would like to see any features or improvements in unit testing frameworks to better support of testing non-public methods. The responses revealed the following themes, listed in order of prevalence:
– None: Most respondents did not feel that any new features were necessary, making this the most common response.
– Reduce support in current tools: Some respondents felt that the current testing frameworks made it too easy to test non-public methods directly (using the mechanisms featured in Table 5.2, for example), and suggested that these frameworks should instead

do more to discourage or even prevent the practice. (P_{21}: *"Current tools make it seem bad enough, which makes the developer think about it twice. Making it easier to test private methods will lead to more code smells"*).

– Support for inaccessible methods: Other respondents expressed a desire for more support in testing framework to handle inaccessible methods (e.g., private methods in Java) without needing to use technique like reflection to bypass access restriction.

Conclusion (RQ4 (Practice)). The majority of developers using programming languages that do not adopt any syntactic visibility modifiers conventions use underscores in method names to indicate that they are meant to be non-public. Testers who prefer to exclusively test through a unit's public API view the need to test non-public methods directly as a sign of a code smell, suggesting poor process design or the need to refactor production code. Developers who do test non-public methods often resort to "back-door" technique, with one of the most common being to elevate the method's visibility to make it accessible for the purpose of testing.

5.5 Threats to Validity

I will address the threats to validity [162] and the strategies used to mitigate them in this section.

Questionnaire Participants

As with most questionnaires, the sampling of participants may affect the *external validity* of this study. Given the lack of a reliable way to quantify the entire population of software developers, I employed the *non-probability purposive sampling* method to recruit participants who were most likely to provide useful responses for this study [78]; I distributed this questionnaire widely via social media and professional networks to reach a large number of participants.

Questionnaire Evaluation

The *internal validity* of this study depends on the design of this questionnaire. Following empirical software engineering guidance [161], I conducted a pilot study before releasing the questionnaire, consisting of four iterations with ten respondents from diverse backgrounds (CS PhD students and professional developers). This pilot study helped mitigate potential researcher biases and refine the questionnaire's format, length, and clarity.

Thematic Analyses

For both the free-text responses in the questionnaire (Section 5.3.2) and StackOverflow threads (Section 5.3.3), I opted for a *collaborative* thematic analysis between three authors to ensure this interpretations are valid (*construct validity*) and mitigate any individual researcher bias.

Programming Language

This open-source study focused exclusively on Java projects hosted on GitHub that use Maven as the build automation tool which poses a potential threat to *external validity*. Although Java remains one of the most popular programming languages [245] and Maven is the primarily build automation tool for Java [54], further replications are necessary to determine how these results generalise to other programming languages. Additionally, I only included projects that I could build with Java 8, the most popular version of the language in 2023 [245]. While I believe this is a good representation of Java projects, further research is needed to assess whether these results apply to more recent versions of the language.

Test Instrumentation

Collecting data from open-source projects requires implementing a test instrumentation mechanism to capture the execution trace of existing tests. As with any study of this nature, there is a risk that implementation errors could impact the results which may have an impact on *internal validity*. To mitigate this risk, I carefully review, debug and test the implementation, and thoroughly discuss the results with the co-author of this chatper's publication. I also acknowledge certain limitations in the current instrumentation as it cannot handle invocation of multi-threaded in the tests (similar to java-callgraph [138]) or some obscure method invocation mechanisms used in a small number of projects. However since these features are orthogonal to the use of access modifiers, therefore I do not expect them to affect this analysis. I mitigate this further by using a sufficiently large number of projects (4,801) and tests (226,915).

Triangulation

The nature of the problem under investigation in this study led me to use three complementary methods: a developer questionnaire, an analysis of online question-and-answer discussions (StackOverflow), and a quantitative experiment on open-source Java projects. This *triangulation* helps me to mitigate threats to validity across research methods (e.g., self-reporting biases) and contributes to a comprehensive understanding of the problem, allowing me to provide more confident answers to the research questions [212].

Reproducibility and Replicability

To promote reproducibility and replicability, I made this research materials available online [56], including this questionnaire, its responses, analysis spreadsheets, and experiment scripts for open-source projects.

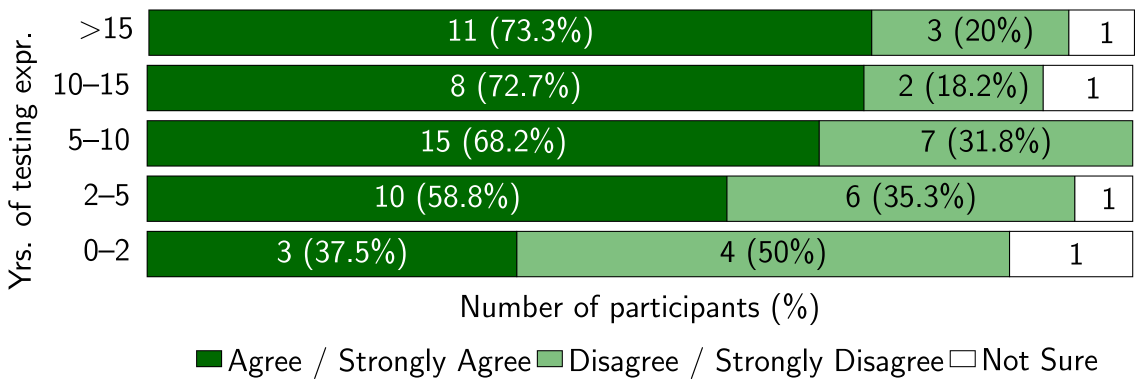


Figure 5.10: *Participants years of testing experience, grouped by stance on testing through public methods only (distribution shown in Figure 5.7).*

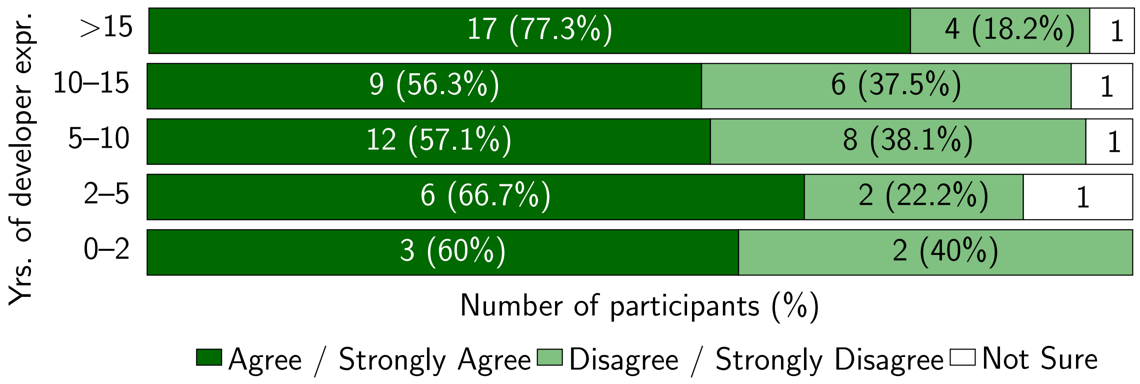


Figure 5.11: *Participants years of developer experience, grouped by stance on testing through public methods only (distribution shown in Figure 5.7).*

5.6 Discussion

Further Analyses

I explore potential correlations between developer experience and the survey results, as well as between the size of a project and the frequency of direct calls to non-public methods in the tests examined in RQ1.

In the demographic survey, I asked developers about their years of experience in development and testing that they had. Figure 5.10 displays a stacked bar chart of *testing experience*, divided into groups based on whether respondents “agree” with testing through public APIs only or the “disagree” group from RQ3, or were unsure, as determined based on their responses to the question shown in Figure 5.7. The chart indicates an increasing proportion of those in the “agree” group as years of testing experience increases. Kendall’s tau-b revealed a moderate positive correlation between these two traits, which is significant at $\alpha = 0.05$ ($\tau_b = 0.240$, $p = 0.014$). This suggests that more experienced tester are more likely to oppose to test non-public methods directly, and highlighting the need for additional organisational guidance is required for more junior team members to ensure that they will avoid side-stepping the public API and test non-public methods directly. Conversely, I found less evidence of a similar relationship with *development experience* (Figure 5.11); Kendall's tau-b test showed only a weak and non-significant association ($\tau_b = 0.135$, $p = 0.17$).

I also investigated whether the project size was somehow correlated with specific practices related to testing of non-public methods. It would be conceivable that directly testing non-public methods is less prevalent in larger projects due to having larger development teams with potentially more development and testing experience, who likely adhere to a better software engineering practices. However, Kendall's tau-b revealed only a weak correlation between the number of tests and the proportion of non-public methods directly called in them ($\tau_b = 0.179$, $p < 0.01$). While both large and small projects in this dataset generally focused on testing public methods (in accordance with the data shown in Table 5.1), they also frequently included direct calls to non-public methods.

Summary of Findings

The results indicates that while the majority of developers favour of testing solely through public APIs, the preference is not overwhelming, with about a third disagreeing and in favour of testing through non-public methods directly depending on the circumstances (RQ2). Developers in the "agree" group that oppose direct testing of non-public methods, arguing that it will complicates future code refactoring and maintenance, and that the desire to test non-public methods directly often coming from poor code design, suggesting refactoring as the solution instead (RQ3, RQ4). However, this requires to be done with caution, as it could lead to repetitive in tests that keep calling the same public methods—a concern that is raised in some literature suggested this as another code smell in its own right [80]. Some studies suggests such repetitive tests themselves should be refactored [248] to adhere the DRY principle ("Don't Repeat Yourself"), while others advocate for keeping tests "DAMP" ("Descriptive And Meaningful Phrases"); e.g., [165]. Another refactoring issue involes the temptation to make parts of the implementation visible solely for testing purposes, a practice that is also considered as a test smell (referred to as "For Testers Only" [248, 80, 63], discussed in Section 2.6.2.

The drawbacks of the alternatives to direct testing non-public methods are exactly those expressed by participants in this survey and those on StackOverflow users who are in favour of testing non-public methods, often depending on the situation. Some also cited the complexity of the code within non-public method—perhaps considering code coverage of the tests in mind — and the difficulty of testing it indirectly, leading to complex tests and/or the challenges in isolating the specific parts of the code they wish to exercise (RQ3). Survey respondents and posters on StackOverflow proposed various solutions, with some of the most common being the *use of language-specific mechanisms* to bypass access modifiers or *elevating access modifiers*—effecitvely making the internals more accessible for testing. This findings related to RQ1 suggest that developers do often make methods package-private in Java for the testing purposes, indicating that they do not strictly adhere to testing exclusively through the public API only.

Finally, the majority of developers in this survey did not believe that additional tooling support is needed to address the challenges (RQ4). However, they perhaps may not be aware of potential future research techniques that could help convert direct calls to non-public methods into tests that solely exercise the public API, or manage other perceived drawbacks of testing non-public methods over the public API, such as repetition in tests. I will expore more implication and ideas for future work in the next section.

5.7 Implications and Future Work

5.7.1 Software Testing Education

The results of the survey revealed that attitudes varied according to levels of experience. Rather than allowing developers to learn through the "hard way", I suggest that educators be informed of some of these different perspectives and make an effort to highlight the best software engineering practices in their teaching.

Behavioural-Driven Testing

The survey results indicate that less experienced software developers and testers are more likely to be in favour of testing non-public methods directly. This suggests a need to improve education and resources focused around behaviour-driven development and testing approaches. Providing this would give developers with a better understanding of the longer-term drawbacks, such as reduced reliability of the test suites, offering a clearer incentives to avoid this practice.

Attitudes towards Coverage

While developers often shows good and healthy skepticism towards code coverage quality in general (e.g., [35]), this study found a correlation between code coverage and the favouring of testing non-public methods directly. Increasing code coverage may thus be a driving factor behind testing implementation rather than behaviour (a pattern also noted by Bowes et al. [83]). Ideally, developers should use code coverage to identify untested behaviours rather than focusing on individual lines that their tests execute in the production code. Therefore, testers should place less concerned with code coverage as a quality metric for their test suites, as tests that are more closely coupled to implementation are of lower quality due to the higher maintenance costs they incur over time.

Identifying and Discouraging Anti-Patterns in Testing

These findings reveal that developers sometimes take alternative shortcuts to test non-public methods' implementation or make them accessible by elevating methods access levels, using language-specific mechanisms like the "friend" keyword in C++ [99], or other employable technique such as Reflection in Java to access code normally inaccessible to tests. These practices should also be highlighted as anti-patterns to avoid in software testing within education materials.

5.7.2 Automated Techniques

The results reveal that there is extensive scope for automated techniques to assist with the problem of testing non-public methods.

Automated Support To Replace Direct Non-Public Calls

In the survey, some developers described the complexity or difficulty of testing exclusively through public interfaces. Future research therefore focus on techniques to refactor existing developer-written tests by removing direct calls to non-public methods and replace them with call sequences that rely solely on available public methods only. This could be achieved using test generation tools like EvoSuite [122]. As an example, when a direct non-public method call is detected, a search-based technique [186] could be used to find any equivalent set of public method calls that will invoke the non-public method the same way. However, the resulting generated tests may face readability challenges [60, 103, 227]. For this reason, applying a large language model to assist in refining the tests could be beneficial, similar to the recent work by Alshahwan et al. [65] and Yaraghi et al. [256].

Refactoring Production Code Based on Problematic Tests Calling Non-Public Methods Directly

In the survey, several developers pointed out that the desire to bypass the public API and directly test a non-public method often stems from poorly design in the production code in the first place. This suggests that tests that involve in calling non-public methods directly might indicate production code in need of refactoring. However, such refactoring would need to be more intricate than simply making non-public methods accessible to tests, as it would undermine the main objectives of keeping an implementation details method to be private. While there has been research on refactoring *tests* to eliminate test smells (e.g., [244]), to the best of my knowledge, poorly designed test cases have not yet been used as the basis for automated refactoring of *production code*. Direct calls to non-public methods in tests could be a potential direction for guiding automated refactoring tools towards improved code design, ensure that tests only need to invoke the public API to better effect.

Improved Automated Test Smell Detection Tools

Current test smell detection tools primarily target private methods [255], however direct calls to any non-public methods, such as private-package ones, can couple tests to the implementation details, making them difficult to maintain over time. I therefore argue for extending these tools to detect calls to all other non-public methods, e.g., package-private. Moreover, identifying when a test directly invoke a non-public method is surprisingly a non-trivial task. Static methods often miss cases when mocking libraries are used to bypass visibility restrictions to call private methods. In this study, I resorted to use dynamic approaches, and yet still, I encountered some cases that were challenging to analyse due to the tests that involved threaded code or starting threads themselves—both of which indicative of further test smells. Additionally, callbacks through external APIs posed further complications. I checked these cases by hand or excluded them entirely from this study. Future work should focus on addressing these challenges.

Automatic Test Generation and Non-Public Methods

The respondents in the survey or those using StackOverflow did not comment anything on automatically generated test suites (such as those being produced by EvoSuite [122], Randoop [202], or AgitarOne [3]). These tools maintain the option and incentivise to call non-public methods directly with the goal of increasing coverage [76, 237], meaning tests generated by these tools will still be hard to maintain. In a study conducted by Gay [132] shows that directly covering a branch from a method is less constrained in the choice of input, which means that it is easier to search for call sequences invoking those methods. Further work needs to establish what the longevity is of automatically generated test suites, since this might not be a problem if the tests are to be thrown away and regenerated anyway [236].

5.7.3 Developer Support

The survey responses suggest different approaches to improve existing tools to help developers to ensure their unit tests do not directly call non-public methods.

Stricter Test Frameworks / Build Tools

As previously mentioned, developers often take alternative shortcuts in their tests to directly exercise non-public methods, but stricter test frameworks could eliminate these bad practices entirely. For instance, JUnit, when used with build tools like Maven and

Gradle, organises unit tests within the same package as the classes they test. This setup makes certain non-public methods, such as those declared package-private or protected, accessible to tests. I found such calls to be particularly to be common in this study of open-source Java projects, especially in the case of package-private. However, by placing the test suites *outside* the package of the classes they are designed to test, these direct calls would be impossible due to the visibility rules of the Java language. This suggestion is directly based on developer feedback in response to RQ4 (Section 5.4.4). Additionally, mocking tools and third-party libraries, such as those shown in Table 5.2, should consider deprecating or remove the functionality for allowing developers to access private methods through Java Reflection.

Integrated Development Environment support

Integrated Development Environments (IDEs) could incorporate support for some of the proposed future work on automated techniques mentioned earlier. For instance, when a developer is inclined to call a non-public method directly, the IDE could automatically suggest an alternative sequence of calls that utilise the public interface of the class instead. Similarly, these tools include bad smell detectors implementation to provide developers more useful indicators of test suite quality, beyond just code-based metrics like code coverage.

5.8 Chapter Summary

Test suites that directly test internal implementations requires additional maintenance whenever those implementation changes—a cost that could be avoided by focusing solely on testing public APIs. In this study, I conducted a developer survey and analysed relevant StackOverflow threads to present the first qualitative study of developer opinions, rationale, and practice when faced with the decision of how to test units with methods of different visibility levels. This qualitative study is further supported by a quantitative analysis of 4,801 open-source Java projects, which I obtained from the Maven Central Repository. Among several findings, this work revealed that while the majority of testers prefer to test only the public API of a unit, a significant number are still willing to bypass it and test a non-public method directly, particularly if those methods are complex even if it means raising the visibility level of the method. While many developers suggested that such code needs to be refactored to enable testing through its public methods, others resort to various techniques to directly access those non-public methods if needed, including making these methods visible for the purpose of testing only, such as using "friend" [99] or `@VisibleForTesting`. This behaviour was supported by the results of this open-source Java code study, where an unexpectedly high number of methods directly tested were declared "package-private"—not fully private but with just "enough" visibility to be accessible to unit tests. I proposed several research directions and strategies to help developers reduce the need to test non-public methods directly, aiming to improve test reliability in the future.

Chapter 6

Coverage-Preserving Test Repair: Automatically Replacing Direct Non-Public Method Call in Test

6.1 Introduction

The aim of this chapter is to evaluate the feasibility of using EvoSuite to replace direct invocations of non-public methods in existing developer-written tests. This is to ensure that non-public method are exercised through public methods while maintaining the branch coverage of the non-public method covered in the tests. I refer to this extension of EvoSuite as EvoSuite$_{UTOPIA}$—**U**nit **T**est **O**nly **P**ubl**I**c **A**PI—leveraging EvoSuite's meta-heuristic technique [186] and its ability to easily adopt new fitness functions.

To evaluate EvoSuite$_{UTOPIA}$, I used the projects with at least one non-public methods directly invoked in the tests identified in the evaluation of open-source projects in Chapter 5 as well as projects where I was able to generate tests using EvoSuite out-of-the-box (v1.2) in Chapter 3. From the 181 projects, EvoSuite$_{UTOPIA}$ successfully replaced 1,146 (55.7 %) non-public method invocations out of 2,057 non-public methods being directly called from 933 tests across 136 projects. On average, the successful replacements generated by EvoSuite$_{UTOPIA}$ were able to maintain 84.79 % of the coverage as achieved by the original non-public method being exercised directly. For the remaining non-public methods, EvoSuite$_{UTOPIA}$ was unable to replace them because the non-public methods were either not reachable from any existing public methods, or at least one of the method arguments are dependent on an external library.

In summary, this chapter makes three main contributions:

1. A novel technique, EvoSuite$_{UTOPIA}$, that will replace non-public methods directly invoked in a test with public methods that exercise them indirectly.

2. An empirical study was conducted on 136 projects to assess the feasibility and usefulness of EvoSuite$_{UTOPIA}$.

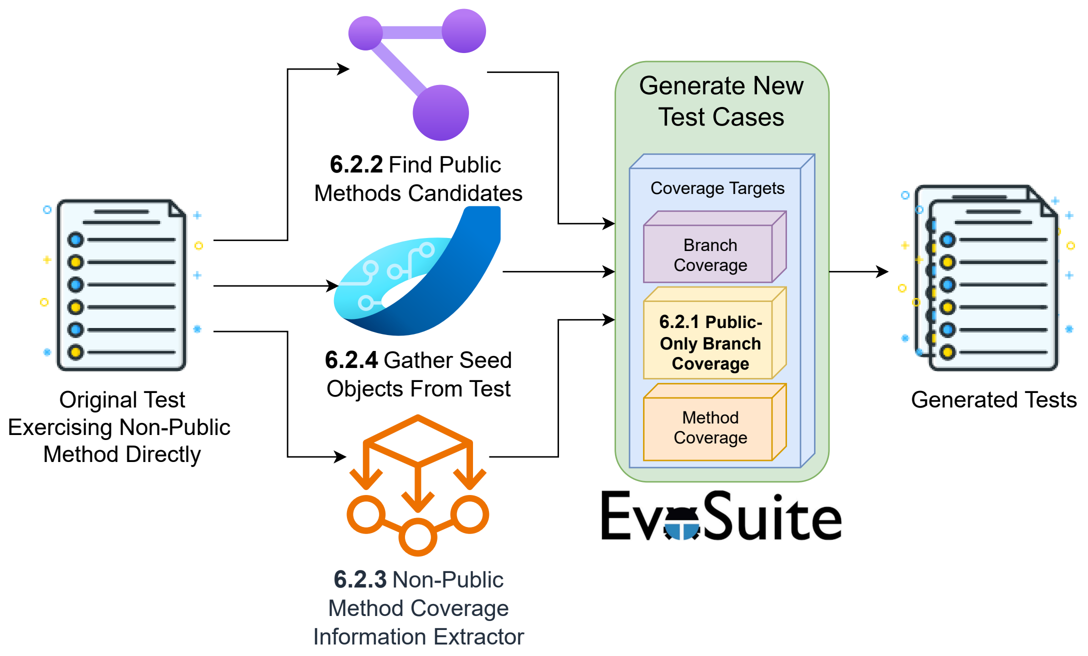

6.2.2 Find Public Methods Candidates
6.2.4 Gather Seed Objects From Test
6.2.3 Non-Public Method Coverage Information Extractor
Original Test Exercising Non-Public Method Directly
Generate New Test Cases
Coverage Targets
Branch Coverage
6.2.1 Public-Only Branch Coverage
Method Coverage
EvoSuite
Generated Tests

Figure 6.1: *Overview of EvoSuite$_{UTOPIA}$. The modification made will be explain in Section 6.2*

3. Findings and analysis of the effectiveness and usefulness of EvoSuite$_{UTOPIA}$ in replacing non-public method invocations in tests.

6.2 Modifications Made to EvoSuite— EvoSuite$_{UTOPIA}$

As mentioned earlier in Section 4.2, the original design of EvoSuite [122] is to generate a test suite for a given class-under-test (CUT) from scratch. In this chapter, I require EvoSuite to be able to read the coverage information from the original test case that directly invokes non-public method and track the coverage of the non-public method being exercised. Secondly, I modified EvoSuite's original direct branch coverage [132] and instrumented other related production classes that call the non-public method. Lastly, to ensure the non-public methods are called in a similar manner, I modified the seeding strategy to reuse the primitives and objects from the original test, when possible. I have made EvoSuite$_{UTOPIA}$ available online [1]. Figure 6.1 depicts an overview of EvoSuite$_{UTOPIA}$.

[1]https://anonymous.4open.science/r/evosuite-utopia

6.2.1 Public-Only Branch Coverage

Firstly, I modified EvoSuite's original direct branch coverage. The original fitness value for the branch coverage in EvoSuite estimates how close the test suite is to cover all branches (as discussed in Section 2.7.3). The direct branch coverage, also known as `Context Branch` (CBranch) will cover a specific branch coverage directly through the method of the branch, meaning that it will penalise any test cases that cover a branch without directly invoking the method [221]. However, this restriction does not apply to branches within private methods due to the need to use Reflection [31] mechanism to directly invoke them in JUnit tests. In short, the fitness function of Direct Branch Coverage is similar to the standard Branch Coverage, but it will only consider a branch as being covered if the method is directly invoked by the test case.

In EvoSuite$_{UTOPIA}$, since the aim is to avoid calling any non-public methods directly in a test case, the newly created Public-Only branch coverage fitness function penalises test cases that cover a branch in a non-public method directly. Unlike CBranch Coverage, which rewards direct invocation of all non-private methods, Public-Only branch coverage will only reward test cases that cover branches indirectly via a public method. This approach ensures that non-public methods are only exercised *exclusively* through public methods. Public-Only branch distance is defined as follows:

$$d_{PO\_branch}(b,t,visibility) = \begin{cases} 0, & \text{if branch has been covered } \textbf{via public method} \\ v(d_{min}(b,t,visibility)), & \text{if predicate executed} \geq 2 \textbf{ via public method} \\ 1, & \text{otherwise} \end{cases}$$

The requirement to satisfy public-only branch coverage fitness function is as follows:

$$f_{PO\_Branch}(T) = |F| + |F_T| + \sum_{b_k \in B} d_{PO\_branch}(b_k, T, v)$$

6.2.2 Identifying Existing Coverage Information of the Directly Invoked Non-Public Method

In order to replace a test case that directly invoke non-public methods, EvoSuite$_{UTOPIA}$ gathers information of the coverage exercised specifically for the non-public method in the test. This coverage information includes branch coverage, the newly created public-only branch coverage, and method coverage. For method coverage, EvoSuite$_{UTOPIA}$ specifically targets the non-public method that is directly called. For both branch coverage and public-only branch coverage, EvoSuite$_{UTOPIA}$ collects the information on the branches of the non-public method that are covered by the test.

6.2.3 Instrument Other Relevant Production Classes

To identify suitable candidate public methods that can replace direct invocations of non-public method, EvoSuite$_{UTOPIA}$ instruments additional relevant production classes. Since methods with package-private or protected visibility can be accessed by other classes in the same package, EvoSuite$_{UTOPIA}$ will try to find public methods beyond the class itself. This is not an issue in the original EvoSuite, as it is designed to generate a test suite for a specific given class-under-test (CUT), and with the `CBranch` fitness function being enabled by default, protected and package-private methods are incentivised to be invoked directly. Private methods are also still accessible within the same class, meaning that to cover them indirectly, EvoSuite only needs to find non-private methods within the same class to invoke them.

To extend beyond the targeted class, EvoSuite$_{UTOPIA}$ first analyses the static method-level call graphs generated by EvoSuite and conservatively finds any public methods that are calling the non-public ones. I set a depth limit of five for the call-graph, meaning that if the targeted non-public methods invoked beyond five level of other non-public methods, it will not be considered as a candidate for replacement. Lower call graph depth values indicate a closer relationship to the non-public method and will usually be preferred as candidates. In contract, higher values suggest a more distant relationship and the inputs to exercise the non-public method may potentially be more constrained.

If there is no candidate found within the depth of five, EvoSuite$_{UTOPIA}$ will stop, and do not continue to find any replacement. After identifying potential public methods that can replace the non-public method, EvoSuite$_{UTOPIA}$ includes the instrumentation of the classes containing the public methods that can potentially replace the non-public method. This enables EvoSuite to calculate the coverage of other relevant classes, ensuring it can evolve the test cases accordingly.

6.2.4 Seeding Strategy via Reusing Non-Public Invocation Test Case Objects

Instead of using the original primitive value and object seeding strategies of EvoSuite [123], EvoSuite$_{UTOPIA}$ will use the objects from the test class where the original test case resides as the seeds to ensure that the generated test cases is similar to the original test case. This ensures that, when possible, the test cases generated by EvoSuite$_{UTOPIA}$ will use similar primitive values and objects as the original test case. This seeding strategy can be useful in terms of improving readability and efficiency during the search process [223].

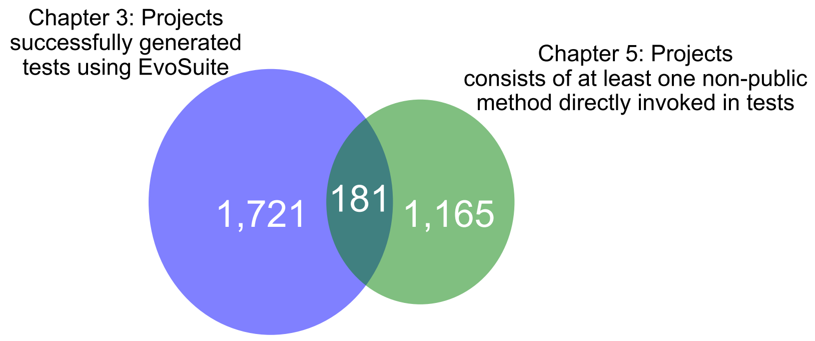


Figure 6.2: *Overlapping projects from Chapter 3 and Chapter 5 that will be used to evaluate EvoSuite$_{UTOPIA}$.*

6.3 Methodology

In this section, I will outline the experimental setup details of this study to evaluate EvoSuite$_{UTOPIA}$. I also set out to answer the following research questions:

RQ1: How effective is EvoSuite$_{UTOPIA}$ in replacing direct non-public method invocations in a test while maintaining the original coverage?

RQ2: How many test cases being generated by EvoSuite$_{UTOPIA}$ to replace the non-public method invocations?

RQ3: Does EvoSuite$_{UTOPIA}$ able to generate meaningful tests?

6.3.1 Subjects

I initially started the experiment with the dataset of 181 projects, overlapping projects that I have used in Chapter 3 and Chapter 5 (Figure 6.2). The projects were first selected based on the availability of the developer-written test cases that directly invoke at least one direct non-public (protected, package-private, or private) method invocation. Secondly, the projects were selected based on the ability to generate test cases using EvoSuite (v1.2). Finally, I only analyse the projects that I could generate test cases using EvoSuite$_{UTOPIA}$. All the projects are written in Java, using Maven as the build tool, and hosted on GitHub. I used the same commit hash as in both previous chapters to clone it.

6.3.2 Experimental Procedure

Using the dataset of 181 projects, I provided EvoSuite$_{UTOPIA}$ with the test cases that directly invoke non-public methods, along with the class name where the non-public method resides, and the full name of the non-public method (including the class name

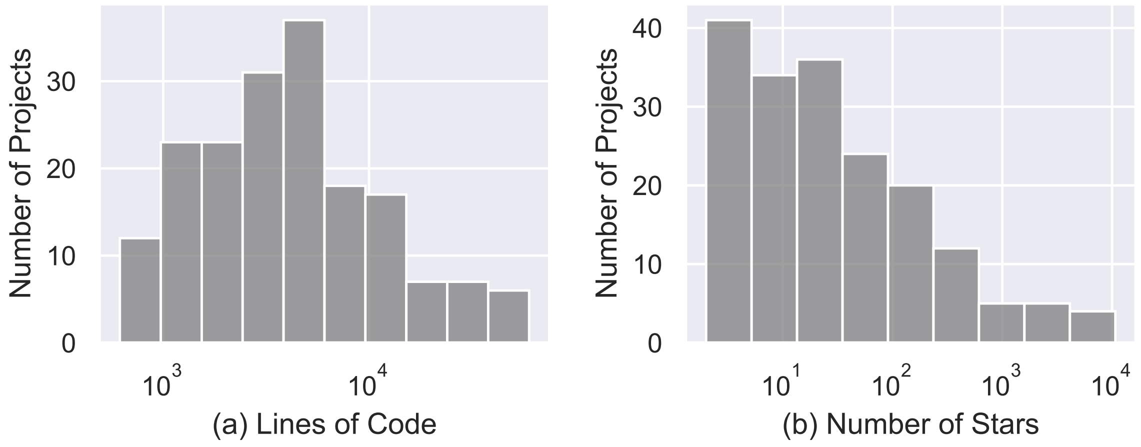


Figure 6.3: *Statistics of the* 136 *successful open-source projects used to evaluate EvoSuite*$_{UTOPIA}$.

and its arguments). With the four modifications that I made to the original EvoSuite, EvoSuite$_{UTOPIA}$ generates test cases that replace the non-public method invocations. I set the search time budget (Algorithm 1, line 8) to 60 seconds. I did not impose any restrictions on the number of test cases that could be generated, allowing flexibility to cover the same branch coverage as the original test case that exercises the non-public method. To ensure the generated test cases will not be flaky, I did not modify any of EvoSuite's flakiness suppression mechanism properties (Table 3.1). I also did not modify the EvoSuite's search algorithm (Section 2.7.3), as I am only interested in generating test cases to replace the non-public method invocation. The final sample consisted of 136 projects for which EvoSuite$_{UTOPIA}$ could generate at least one test to modify direct calls to non-public method.

Statistics for the successful projects are shown in Figure 6.3, indicating a range of project sizes from large to small. The average (mean) number of lines of code across 136 projects is 7,306.46 (σ = 9,888.19), with a median of 3,986 (Figure 6.3a). The average (mean) number of GitHub stars is 307.3 (σ = 1,226.63), with a median of 18 (Figure 6.3b).

Retrieving Generated Tests Coverage Information

Since EvoSuite also provides a built-in method for determining the coverage information of the final test suite without using third-party code coverage libraries, I will use this internal coverage information to evaluate the effectiveness of EvoSuite$_{UTOPIA}$ in replacing non-public methods. I chose this approach because it is simpler and avoids adding the complexity of incorporating additional code coverage libraries. Besides that, I am not comparing the results of EvoSuite$_{UTOPIA}$ tests with other tools.

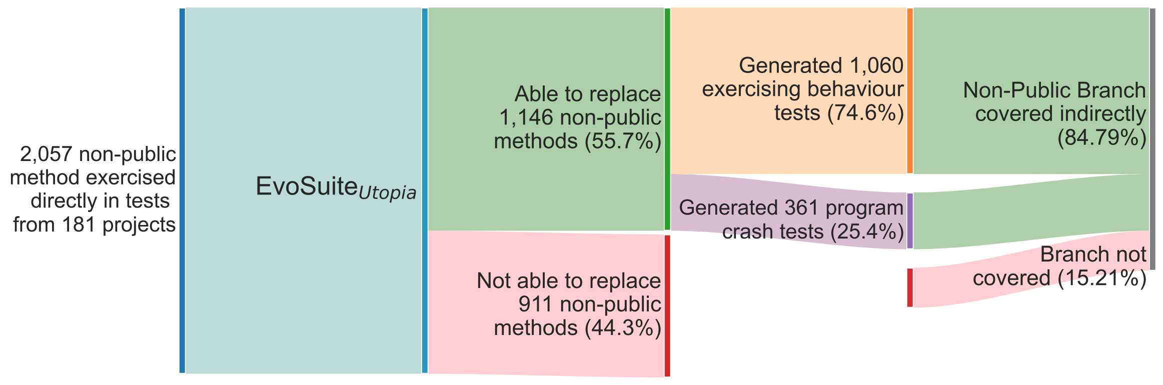


Figure 6.4: *EvoSuite$_{UTOPIA}$ was successful at replacing* 55.37 % *of* 2,057 *non-public methods being directly called in* 933 *tests (RQ1). EvoSuite$_{UTOPIA}$-generated tests predominantly focused on exercising the behaviour rather than program crash tests (unexpected behaviour exceptions), while maintaining* 84.79 % *of the non-public branch coverage indirectly through Public APIs.*

Evaluating the Intent of EvoSuite$_{UTOPIA}$ Generated Tests

To evaluate whether EvoSuite$_{UTOPIA}$ generated meaningful test cases, I assessed the number of successfully generated test cases that included at least one assertion and did not include any automated oracle related to unexpected exceptions scenario, such as undeclared exceptions. These types of test cases usually focus on program crashes and are highly likely unrelated to the original test cases. Such test cases can help increase code coverage; however, without any assertions included, they provide little real semantic value [205]. Therefore, I will address RQ3 by assessing the ratio of generated test cases with more than one test assertion, to those with unexpected exception test cases, in order to evaluate whether it is exercising a Public API behaviour or not.

6.4 Results

RQ1: Effectiveness of EvoSuite$_{UTOPIA}$ in Replacing Direct Non-Public Method Invocations

Among the 136 projects, EvoSuite$_{UTOPIA}$ successfully replaced 1,146 of 2,057 non-public method invocations identified in 933 tests. On average per project, it successfully replaced 65.74 % of the non-public method invocations (σ = 29.67). Besides that, all EvoSuite$_{UTOPIA}$-generated tests that successfully replaced the non-public method could be compiled and passed when being executed 5 times. Of the 1,146 successful replacements, EvoSuite$_{UTOPIA}$ is able to maintain on average 84.79 % of the coverage (branch, public-only branch, and method coverage) that was originally achieved by directly invoking the non-public methods.

Figure 6.4 summarises the proportion of successful replacements of direct non-public method invocations in the tests. The results show that EvoSuite$_{UTOPIA}$ successfully replaced 55.7 % of these non-public method calls, covering 1,146 methods, while 44.3 % (911 methods) could not be replaced. Among the generated tests, the majority—74.6 % (1,060 tests)—focused on exercising the expected behaviour of the methods, whereas 25.4 % (361 tests) led to program crashes, highlighting unexpected behaviours or exceptions. EvoSuite$_{UTOPIA}$ maintains non-public branch coverage indirectly. The findings indicate that 84.79 % of non-public branches were successfully covered through public API calls, reducing the need for direct access to non-public methods. However, 15.21 % of branches remained uncovered.

RQ2: Test Suite Size

I found that the mean number of successfully generated test cases by EvoSuite$_{UTOPIA}$ is 1.24 ($\sigma = 0.731$). The median number of test cases generated by EvoSuite$_{UTOPIA}$ is 1. However, there were 13 instances where EvoSuite$_{UTOPIA}$ generated more than five test cases to achieve similar code coverage for the non-public method. An example of the original test case is shown in Figure 6.6, where it is directly invoking a non-public method called `cleanPrimitives` (Figure 6.5) multiple times, aiming to exercise different parts of the method. Therefore, to maintain the same branch coverage, EvoSuite$_{UTOPIA}$ generated nine test cases. The largest number of test cases generated to exercise the same coverage of a non-public method was 14.

RQ3: Relevancy of EvoSuite$_{UTOPIA}$-Generated Tests

Of the 1,146 non-public methods that have been successfully replaced, I found that 74.62 % of 1421 test cases generated by EvoSuite$_{UTOPIA}$ are exercising the implementation details via a public method with assertion based on the behaviour of the public method. The remaining 25.38 % of the test cases are related to unexpected exceptions. This indicates that roughly three-quarters of the generated test cases are meaningful and are not just creating program crash tests [101] to exercise the non-public method.

6.5 Threats to Validity

As with any empirical study, there are threats to validity that must be considered [162]. I will discuss the threats to the validity of this study in this section. First, the generalisability of the results may be limited to the specific projects that I have selected. These projects were selected based on the availability of the developer-written test cases that directly invoke non-public methods (based on Chapter 5) and the ability to generate test cases using EvoSuite (based on Chapter 3). The results may not be

```
protected static String cleanPrimitives(String className){
  // ... omitted ...
  switch (cleanedClassName) {
   case "boolean":
       cleanedClassName = "Boolean";
       break;
   case "char":
       cleanedClassName = "Character";
       break;
    case "byte":
       cleanedClassName = "Byte";
       break;
   // ... omitted ...
  }
  return cleanedClassName + arrayToken;
}
```

Figure 6.5: *`cleanPrimitives` method from project `Thomas-S-B-visualee`*

```
@Test
public void testCleanPrimitives() {
 String inputString;
 String actual;

 inputString = "boolean";
 actual = ExaminerImpl.cleanPrimitives(inputString);
 assertEquals("Boolean", actual);

 inputString = "char";
 actual = ExaminerImpl.cleanPrimitives(inputString);
 assertEquals("Character", actual);

 inputString = "byte";
 actual = ExaminerImpl.cleanPrimitives(inputString);
 assertEquals("Byte", actual);
 // ... omitted ...
}
```

Figure 6.6: *Original developer-written test that is invoking a protected method, `cleanPrimitives` (Figure 6.5) directly more than once (project `Thomas-S-B-visualee`)*

applied to other projects that do not meet these criteria. However, the sample of 136 projects is large enough to provide a representative basis for assessing EvoSuite$_{UTOPIA}$.

Second, consideration must be given to how the code coverage information is calculated. Since I am only interested in three coverage criteria, namely branch coverage, public-only branch coverage, and method coverage, I will use EvoSuite's internal code coverage information. This specifically targets the coverage criteria met by the non-public method being invoked in the original test cases.

Thirdly, previous studies that developed test repair tools [256, 95] are evaluated using code generation task techniques, such as Exact Match Accuracy [61], BLEU [209], or CodeBLEU [220] to evaluate the quality of generated repair candidates. However, since my focus is on replacing non-public method invocations, the test that is generated by EvoSuite$_{UTOPIA}$ could be different from the original test due to calling of different variables and methods. Therefore, for RQ1, I will only evaluate the effectiveness by comparing the coverage and whether it could be compiled and executed.

Fourthly, since I did not manually examine the intent behind each of the original test cases, I cannot definitively determine their original purpose. Therefore, the evaluation of the intent of EvoSuite$_{UTOPIA}$-generated test cases is based on whether the generated test cases are related to program crash (unexpected behaviour) scenarios. If not, the generated tests are considered as exercising the implementation details—encapsulated in a non-public method—via a Public API. Additionally, each public method could represent different behaviours or intents.

Finally, as EvoSuite generates tests non-deterministically, the results may vary between runs. Most of the previous studies generate more than one test suite to evaluate the effectiveness of the tool. However, since I am doing my evaluation on a large number of projects on 933 tests, the evaluation on EvoSuite$_{UTOPIA}$ should still be generalisable.

6.6 Discussion

6.6.1 Example of EvoSuite$_{UTOPIA}$-Generated Test

An example of a test being replaced using EvoSuite$_{UTOPIA}$ is as shown in Figure 6.7. The original test case (`testCompare`), taken from the Apache Commons Validator project[2], making time-based comparisons by invoking the protected method, `compare` (highlighted in Figure 6.7), directly to ensure the implementation details are correct. In the same figure, EvoSuite$_{UTOPIA}$-generated test case is able to replace the invocation of the protected method by invoking a public method, `compareDates`, that will maintain the same coverage as what is being exercised by the original test case.

[2]https://github.com/apache/commons-validator/blob/f38ea

```
Developer-written test
@Test
public void testCompare() {
 // ... omitted ...
 assertEquals(0, calValidator.compare(value, diffHour, Calendar.
   DAY_OF_YEAR), "date(B)"); // same day, diff hour
 // ... omitted ...
}

EvoSuite_UTOPIA-generated test
@Test(timeout = 4000)
public void testCompare() throws Throwable {
 CalendarValidator calendarValidator0 = new CalendarValidator();
 Locale locale0 = Locale.UK;
 MockGregorianCalendar mockGregorianCalendar0 = new
   MockGregorianCalendar(locale0);
 ZoneOffset zoneOffset0 = ZoneOffset.MAX;
 TimeZone timeZone0 = TimeZone.getTimeZone((ZoneId)zoneOffset0);
 Calendar calendar0=MockCalendar.getInstance(timeZone0,locale0);
 int int0 = calendarValidator0.compareDates(calendar0,
   mockGregorianCalendar0);
 assertTrue(calendarValidator0.isStrict());
 assertEquals(1,int0);
}
```

Figure 6.7: *The first test case in this figure is an original developer-written test, from the project* ***apache-commons-validator****, that is invoking the protected method* ***compare*** *directly. The second test case is generated by EvoSuite$_{UTOPIA}$ targeting to replace the invocation of the method* ***compare*** *directly by calling the public method* ***compareDates*** *that will maintain the same coverage as the original developer-written test.*

6.6.2 Effectiveness of EvoSuite$_{UTOPIA}$

From RQ1, EvoSuite$_{UTOPIA}$ was able to achieve 84.79 % of the coverage of the non-public methods when it is able to generate tests successfully to replace the non-public methods being called directly. Overall, this demonstrates that EvoSuite's evolution technique and the new public-only branch coverage fitness function (See Section 6.2.1) can be an effective approach for replacing non-public methods called in a test. However, EvoSuite$_{UTOPIA}$ could not replace 911 of the remaining non-public method invocations in the original test cases.

6.6.3 Test Suite Size

The median number of test cases generated by EvoSuite$_{UTOPIA}$ is 1. This is expected, as EvoSuite$_{UTOPIA}$ focuses on replacing a specific non-public method directly called in the original test case. However, as shown in Section 6.4, there are instances where EvoSuite$_{UTOPIA}$ generated more than one test case to exercise similar coverage for the non-public method. There are two main reasons I have found where EvoSuite$_{UTOPIA}$ generates more than one test case to achieve similar coverage. Firstly, there are instances where the original test case invokes the same non-public method multiple times with different input parameters to explore different branches. Secondly, the data-flow required to exercise the non-public method from a public method is often more constrained [132], causing EvoSuite$_{UTOPIA}$ to call the public method multiple times with different arguments to exercise similar coverage.

6.6.4 Mutation Score

I also noticed that the generated test cases do not have a high strong mutation score when only considering mutants within the targeted non-public method. This is mainly because the generated test case assertions are based on the observable behaviour of the public method that is being used to replace the non-public method, making it difficult to kill mutants in the non-public method. Besides that, in this study, I did not include either weak (satisfying up to Figure 2.4.2) or strong (satisfying both Figure 2.4.3 and Figure 2.4.4) mutation coverage criteria for EvoSuite$_{UTOPIA}$. Future work of this study should include to address this limitation.

6.6.5 Unrepairable Tests that Invokes Non-Public Method

As discussed in Section 6.6.2, there are 34.26 % of the non-public methods where EvoSuite$_{UTOPIA}$ could not replace the non-public, meaning it failed to generate any test cases to cover the non-public method indirectly via a public method.

No Public Method Candidate Found

Certain replacement failures may not meet the preconditions of the repair strategy. This is because no public method candidate exists that can reach to the public method that is at the depth of five.

Test Framework Restrictions

In cases where the project is using JUnit 5 framework, EvoSuite$_{UTOPIA}$ is unable to retrieve (instrument) the existing coverage information of the tests. This is also part of the preconditions before EvoSuite$_{UTOPIA}$ starts the search/evolution process.

External Libraries

This is a known limitation of EvoSuite [76], as it is difficult or even impossible to generate test cases that involve external dependencies. Certain tests are dependent on external dependencies such as database connections, network connections, or file system operations.

6.7 Chapter Conclusion and Future Work

As shown in Section 5.4.1, approximately a quarter of open-source projects have at least one non-public production method directly invoked in their tests. In this chapter, I have developed EvoSuite$_{UTOPIA}$, a new approach that uses EvoSuite's search-based capabilities to replace direct non-public method invocations in a test. This approach identifies public methods that can indirectly exercise the non-public method, while trying to maintain similar branch coverage.

I evaluated the effectiveness of EvoSuite$_{UTOPIA}$ on 136 projects and demonstrated its ability to automatically replace and repair tests that directly call non-public methods. However, there are still some limitations that need to be addressed. Firstly, EvoSuite$_{UTOPIA}$ could not replace some non-public methods when their arguments requires call to external libraries. Secondly, the generated test cases have a low mutation-killing rate for mutants within the targeted non-public method. Finally, EvoSuite$_{UTOPIA}$ cannot retrieve existing coverage information for tests written using the JUnit 5 framework. Future work should aim to address these three challenges.

Furthermore, to ensure that original EvoSuite-generated tests are not brittle, I also plan to evaluate whether the public-only branch coverage fitness function (Section 6.2.1) and the reverse call graph strategy (Section 6.2.3) can be easily adopted for generating test cases from scratch.

Chapter 7

Conclusions and Future Work

To summarise, the primary aim of this thesis is to understand and develop techniques that could improve the reliability and effectiveness of test suites. To address these challenges, I set out to achieve two high-level objectives:

1. understanding the factors limiting test suite lifespan and effectiveness, and

2. developing automated techniques to improve test suites and repair unreliable test cases.

The following sections summarise how my work advances these objectives.

I have investigated four of the main indicators in test suite health, that is test flakiness, mutation score, test brittleness, and test realism. I have also developed and evaluated automated techniques to improve the quality of existing developer-written tests and repair poorly designed developer-written test cases, conducting empirically studies on large dataset of tests from open-source projects.

Understanding Test Suite Health Indicators

In Chapter 3, I empirically evaluated the flakiness in EvoSuite-generated test outcome. Through an evaluation of 1,902 Java projects, I found that EvoSuite flakiness suppression is largely effective at controlling and removing flaky tests, when compared to developer-written tests. Based on this finding, I provided recommendations for software developers using EvoSuite, maintainers of EvoSuite, and researchers studying flaky tests. Besides that, this empirical study reinforced confidence in the reliability of EvoSuite-generated tests.

In Chapter 5, I conducted another empirical evaluation of brittleness in developer-written tests, focusing on whether developer directly test non-public methods directly or not. This study involved in analysing the visibility of methods invoked directly in 4,801 open-source Java projects, conducting an online developer survey which received 73 responses, and performed thematic analysis on 60 StackOverflow threads. I found that developers are divided on whether to test non-public methods directly or not. I also found that inexperience developers are more likely to test non-public methods directly.

Improving Test Suite Health

Building on EvoSuite's flakiness suppression mechanism (Table 2.3), which effectively mitigate most of the flaky tests, I proposed EvoSuite$_{Amp}$, in Chapter 4. This new test amplification tool utilises EvoSuite's search-based technique to improve the mutation score of existing developer-written tests. I evaluated EvoSuite$_{Amp}$ on the Defects4J dataset and compared it to DSpot, the state-of-the-art test amplification tool. I found that EvoSuite$_{Amp}$ was 83.33 % more effective at killing mutants compared to DSpot.

In Chapter 6, I presented EvoSuite$_{UTOPIA}$, a modification on EvoSuite to replace non-public method being directly called in a test with Public API that execute them, while trying to preserve the branch coverage of the non-public method. This is to improve *realism* of the test and made it less brittle. From an evaluation involving 181 Java projects, I demonstrated that EvoSuite$_{UTOPIA}$ successfully replaced more than half of non-public method with a public method, while retaining 84.79 % of the branch coverage of the original non-public method.

7.1 Limitations

One key limitation of this thesis is its focus on Java projects, particularly in the context of automated test generation with EvoSuite. As an example, while similar flakiness issues were observed with Pynguin for Python in [142], the specifics of how test generation tools behave across different programming languages may vary. The extent to which these findings apply to other languages, such as JavaScript, remains uncertain, as test execution environments, dependencies, and tool implementations differ across programming languages.

7.1.1 Uncertainty in the Empirical Evaluations

The results presented in Chapter 3, where I empirically evaluated flakiness in EvoSuite-generated tests, are based on datasets generated by executing developer-written and EvoSuite-generated tests from 1,902 open-source projects 100 times in fixed and 100 times in shuffled order. However, running the tests a larger number of times may reveal additional flaky tests, meaning some tests in the dataset could have been inaccurately labelled as non-flaky. This could potentially impact the results. Given the time constraints and available computational resources, I conducted as many reruns as possible.

Besides that, the selection of projects from the Defects4J benchmark in Chapter 4, while widely used, may not fully represent all types of software projects, particularly large-scale or industrial codebases. Additionally, due to compatibility issues with

DSpot, not all classes within Defects4J could be analysed, potentially introducing a bias in the subject selection. Another factor to consider is the dependence on a specific mutation analysis tool—while Major was used to ensure consistency, other tools like EvoSuite's internal mutation analysis or PITest might produce different mutation scores. This variability in mutation analysis could impact how improvements in test effectiveness. Moreover, ensuring that the improved test suites retained access to their dependent libraries and code required careful handling, and this process could have inadvertently influenced the mutation score results.

The results of RQ1 (Section 5.4.1) in Chapter 5 (shown in Table 5.1 and Figure 5.6) are based on an empirical analysis of the access modifiers invoked in 4,801 open-source projects using Viscount, a tool that I developed (Chapter A). To assess these projects, Viscount had to insert lightweight instrumentation and modify the test bytecode, which could have introduced delays that may have affected the results, such as timeouts that potentially cause some test failures. I excluded failing tests from the analysis, but this could still have impacted the results.

Finally, the evaluation of Chapter 6 was conducted on projects where EvoSuite could successfully generate tests—from Chapter 3—and where direct non-public method calls were present—from Chapter 5. This selection criterion might not be representative of all Java projects. Additionally, EvoSuite$_{UTOPIA}$ was unable to replace non-public methods when their arguments required calls to external libraries, which could be a limiting factor in its broader applicability. Furthermore, the tool could not retrieve existing coverage information for JUnit 5 tests, which was a necessary precondition for its operation, making its effectiveness dependent on projects using JUnit 4. Lastly, limitations in EvoSuite's instrumentation itself may have contributed to unsuccessful replacements, indicating potential areas for further tool refinement.

7.1.2 Open-Source Projects

In this thesis, all empirical evaluations that have been conducted focus on open-source programs written in Java. Therefore, the findings may not be fully applicable to other programming languages, such as JavaScript or Python. However, for the flakiness evaluation in Chapter 3, in [142], we also evaluated the flakiness of Pynguin-generated tests on Python programs, and the results are consistent with the findings in Chapter 3.

7.1.3 Limitations Summary

By acknowledging these limitations, this thesis provides a more comprehensive understanding of the constraints and potential areas for improvement. While these factors do not undermine the validity of the findings, it highlights important considerations for future work and the generalisability of the results.

7.2 Future Work

7.2.1 Investigating the Synergy and Trade-offs between Test Suite Health Indicators

The work that I have undertaken in this thesis has focused on evaluating and automatically improving individual indicators of test suite health—flakiness (Chapter 3), mutation score (Chapter 4), brittleness (Chapter 5 & Appendix A), and "realism" (Chapter 6)—by themselves. To holistically enhance test suite health, the software testing research community could further explore the synergy, subsumption, and trade-offs between the indicators. For instance, one could investigate whether the brittleness of a test suite is correlated with low test suite's mutation score [113] and high pseudo-testedness [185, 249]. This could provide valuable insights into the relationships among these metrics and their mutual impacts.

7.2.2 Measurability of Test Suite Health Indicators

While several indicators in Table 1.1 already have established metrics, such as code coverage [18] and mutation score [156], other indicators such as "realism" lack obvious means of quantification. Future work could focus on the development of metrics for these indicators, which could help in the evaluation of test suite health. Measurement metrics for all of the indicators should also take into account non-determinism in as much as possible, to remove inconsistencies when being evaluated.

7.2.3 Chapter 3: Root-Causing EvoSuite-generated order-dependent flaky tests

As shown in the violin plot in Figure 3.5, EvoSuite with the flakiness suppression mechanism turned on still produce more order-dependent flaky tests than the developer-written tests. I could extend the empirical evaluations of Chapter 3 to investigate the root causes of EvoSuite-generated order-dependent flaky tests. This could help to identify the reasons behind the flakiness and whether the root causes are similar to what I have found for the non-order-dependent tests. If not, it would raise interesting questions about some lack of validation in the current flakiness suppression mechanism. Applying existing techniques of repairing order-dependent flaky tests [175] could help in reducing order-dependent flaky tests in EvoSuite.

7.2.4 Chapter 6: Evaluating the effectiveness of EvoSuite$_{UTOPIA}$'s tests with developers.

Similar to the work of Rojas et al. [222], where they evaluated the effectiveness of EvoSuite to help software developers task when writing unit tests, for my future work of Chapter 6, I could conduct a similar study with developers to evaluate the effectiveness of EvoSuite$_{UTOPIA}$ in replacing tests that are calling non-public methods directly. The work requires an observational study where developers are asked to perform test refactoring, and give them EvoSuite$_{UTOPIA}$-generated tests to evaluate whether they see the automated approach as helpful or not.

7.2.5 Investigating Project Composition by Application Type used in Chapter 3, Chapter 5, and Chapter 6

Inspired by recent work in analysing project builds—such as AROMA study [159] by Keshani et. al., which demonstrated how small configuration changes can dramatically improve the reproducibility of Maven artefacts, and investigation by Mir et al. [194] of transitivity and granularity in vulnerability propagation on Maven projects, which revealed that dynamic method-level call graphs analysis drastically reduce false positives third-party dependencies compared to naive dependency-level analysis—an independent avenue for future work should similarly consider the role of projects being evaluated in this thesis. By categorising the composition of projects used in Chapter 3, Chapter 5, and Chapter 6 by their application type (e.g., streaming libraries, business libraries) researchers can uncover domain-specific patterns in testing practices and maintainability challenges. Just as reproducibility issues vary across ecosystems due to differences in build configurations and tooling, some of the indicators in test suite health (Table 1.1) may also be influenced by the characteristics of a project's domain. For instance, streaming libraries might face unique concurrency and real-time processing challenges, exhibit test flakiness (Table 1.1-#6) tied to concurrency flaky root cause (Table 2.1), while math libraries (e.g., Apache Commons-Math) could exhibit complexity related to intricate code complexity, exhibiting test brittleness due to test focusing on implementation details (Table 1.1-#7). Analysing these patterns could not only advance our understanding of test suite health but also connect this thesis with broader topics in software engineering, such as vulnerability management and development practices, potentially leading to better automated techniques for enhancing test reliability.

Appendix A

Viscount: A Direct Method Call Coverage Tool for Java

The contents of this chapter is based on "Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. Viscount: A Direct Method Call Coverage Tool for Java. In *International Conference on Software Maintenance and Evolution (ICSME): Tool Demo Track*, 2024".

A.1 Introduction

When deciding which tests to write, developers must consider a number of factors, including which parts of the production code to test and how to test them. In doing this, developers usually aim to achieve high code coverage and to capture the behaviour by creating meaningful assertions to the production code. Advocacy for and guidance on what good unit tests are and how to write them is plentiful both in formal [163] and grey literature [16].

A common recommendation available is to test the behaviour (i.e., usually in public methods) of the program instead of implementation details (often embodied in non-public methods). The main reason is because testing the implementation details directly may lead to brittle tests that are prone to break when implementation details are updated or modified. Tests that directly call non-public methods are tightly coupled to the implementation details of the production code. As been documented in Chapter 5, this is considered as an anti-pattern in testing which is also often referred to as a test smell [248, 83, 255], and has been discredited in practice [165, 46, 28].

Despite the existing advice against directly call non-public methods in tests, I found 28% of 4,801 open-source Maven projects contain at least one test that directly calls non-public methods (i.e., with `protected`, package-private, or `private` visibility), as discussed in previous chapter.

```
@Test
public void testResize() {
    Wallet wallet = new Wallet(2);
    Method method =wallet.getClass().getDeclaredMethod
     ("resize"); // method-under-test
    method.setAccessible(true);
    method.invoke(wallet);
    assertEquals(3, wallet.capacity());
}
```

Figure A.1: *An example of a test that examines the implementation of the method* `resize`*.*

```
@Test
public void testAddCard() {
    Wallet wallet = new Wallet(1);
    wallet.addCard(new Card("VISA"));
    wallet.addCard(new Card("AMEX"));
    assertEquals(2, wallet.size());
}
```

Figure A.2: *This test examines the behaviour of the code via using* `addCard` *method to change the size (via* `resize()`*) of the Wallet.*

Figure A.1 shows an example of what this anti-pattern looks like in practice. The test case is using Java *Reflection* to gain access and call the private method `resize` directly in an object of type `Wallet`. As shown in Figure A.3, it would be plausible that the implementation of the `Wallet` class could change in the future to use a resizeable collection instead of an array that needs resizing when its capacity is reached. In this scenario, method `resize` would have to be removed and the test would break, needing refactoring or removal. Figure A.2 presents an alternative test that also tests the ability of the wallet to resize beyond its initial capacity when a new card is added. Notice that method resize is still being tested, but this time via the execution of public method `addCard` instead. This test is more realistic because they form explicit contract: if it fails, it implies that the code is broken and what end users will receive as the output is incorrect.

One of the main motivations that lead developers to engage in this practice is trying to achieve higher coverage [225]. As the practice of testing non-public methods directly is more prevalent than expected, it poses serious threats to the reliability of a test suite [152].

One of the reasons why this is not being addressed is the lack of tool support to analyse *how* existing tests achieve coverage. Once a project has a large number of tests, there is no easy way to identify non-public methods being directly called. Code coverage tools such as JaCoCo provide a way to measure method coverage, but as illustrated in this example, methods can be covered directly or indirectly in tests, and tools like JaCoCo cannot distinguish direct or indirect calls to non-public methods because they do not track call hierarchies [17] and treat production and test code similarly. In a smaller-sized project, it is possible to manually analyse each test case to identify non-public methods being called directly. However, it is not feasible to analyse for projects with thousands of test cases.

To address this problem, I developed **Viscount**, a tool that can determine *direct method call coverage*—i.e., the percentage of methods of each level of access modifiers in Java that are directly invoked from JUnit tests. This tool is designed to help developers identify tests that call non-public methods directly to facilitate test suite maintenance. Viscount's core features include:

- Retrieving every method's visibility in the production code;
- Retrieving production methods that are called directly in test code; and
- Summarising direct method call coverage in a clear format.

A.2 Viscount

Viscount is a tool that allows testers, developers, and researchers to identify methods directly invoked in their test suites within a Maven project. It is primarily written in Java and can be used through a command-line interface.

To use Viscount, the user needs to provide access to the Maven project in question. Viscount parses the project's source code to find every production method's visibility. It then installs the Surefire Report plugin to the Maven project (if it is not already included), and sets up an execution environment ready with a Java agent that applies instrumentation to production code and test code. It then runs the project's test suite to collect direct method call coverage information—similar to the work of Huo and Clause [151]—about the tests using this instrumentation. Viscount outputs details about each production code method's visibility, the methods invoked directly by the project's tests, and a direct method call coverage report. Viscount is available on GitHub for evaluation and extension[1].

The main entry point of Viscount is `viscount.sh`. As part of the tool requirement, the user needs to include the project name, the project path, and a directory to output the results. Suppose the project name is `javapoet`, the project is located in the directory `/path/to/javapoet`, and the output of the results in the directory `/path/to/results`. The command to run Viscount would be:

```
./viscount.sh javapoet /path/to/javapoet /path/to/results
```

[1]`https://github.com/unittesting-nonpublic/viscount`

```java
public class Wallet {
    // ... omitted ...
    public Wallet(int initialCapacity) {
        cards = new Card[initialCapacity];
        size = 0;
    }

    public void addCard(Card card) {
        if (isCardPresent(card)) {
            return;
        }
        if (size== cards.length) {
            resize(); cards[size++] = card;
        }
    }

    private void resize() {
        // ... omitted implementation details ...
    }
    protected int capacity() {
        // ... omitted implementation details ...
    }
    protected boolean isCardPresent(Card card) {
        // ... omitted implementation details ...
    }
    // ... omitted ...
}
```

Figure A.3: *Example of a class named* ***`Wallet`*** *that can keep multiple* ***`Card`*** *object. The public method* ***`addCard`*** *is to add new card into the Wallet object. It will not add existing card (****`isCardPresent()`****) and update the* ***`size`*** *field (****`resize()`****) if new card is added into Wallet.*

Table A.1: *Production code methods and their visibility for **square-javapoet**, as outputted by Viscount in a TSV file.*

METHOD	VISIBILITY
com.squareup.javapoet.TypeName.isPrimitive()	public
com.squareup.javapoet.ClassName.simpleName()	public
com.squareup.javapoet.CodeBlockJoiner.join()	package-private
com.squareup.javapoet.Builder.isNoArgPlaceholder(char)	private
⋮	⋮

A.3 Dependencies

Viscount is primarily built on top of two Java libraries.

First, I used Spoon [213], a meta-programming library to analyse and transform Java source code. Spoon parses the source code to build an abstract syntax tree (AST) meta-model for performing AST transformations and analysis. It also provides its own Launcher (application runner), specifically for Maven-built projects, called MavenLauncher. It loads a Maven project by reading the `pom.xml` file and setting up the project dependencies. In this tool, I used Spoon to extract the visibility of production code methods and to distinguish between production code and test code.

Second, I used Javassist [96], a Java-bytecode analysis library to transform bytecode at compile or load time. Javassist parses class files into objects representing the classes, methods, or fields, and can be used to modify the objects. Modifications can be done by inserting, deleting, or replacing bytecode instructions, similar to other Java bytecode frameworks such as ASM [29]. In this tool, I used Javassist to instrument production methods and constructors, and test methods and helpers (inserting probes at entry and exit points) during the test execution.

A.4 Viscount's Architecture

Figure A.4 depicts the overall architecture of Viscount. I will discuss the tool's main components and how it works in the following sections. Viscount's main components are:

- Extracting production code method visibility
- Including the Surefire Report plugin in the Maven project
- Performing runtime instrumentation of the project during test execution
- Analysing the test reports

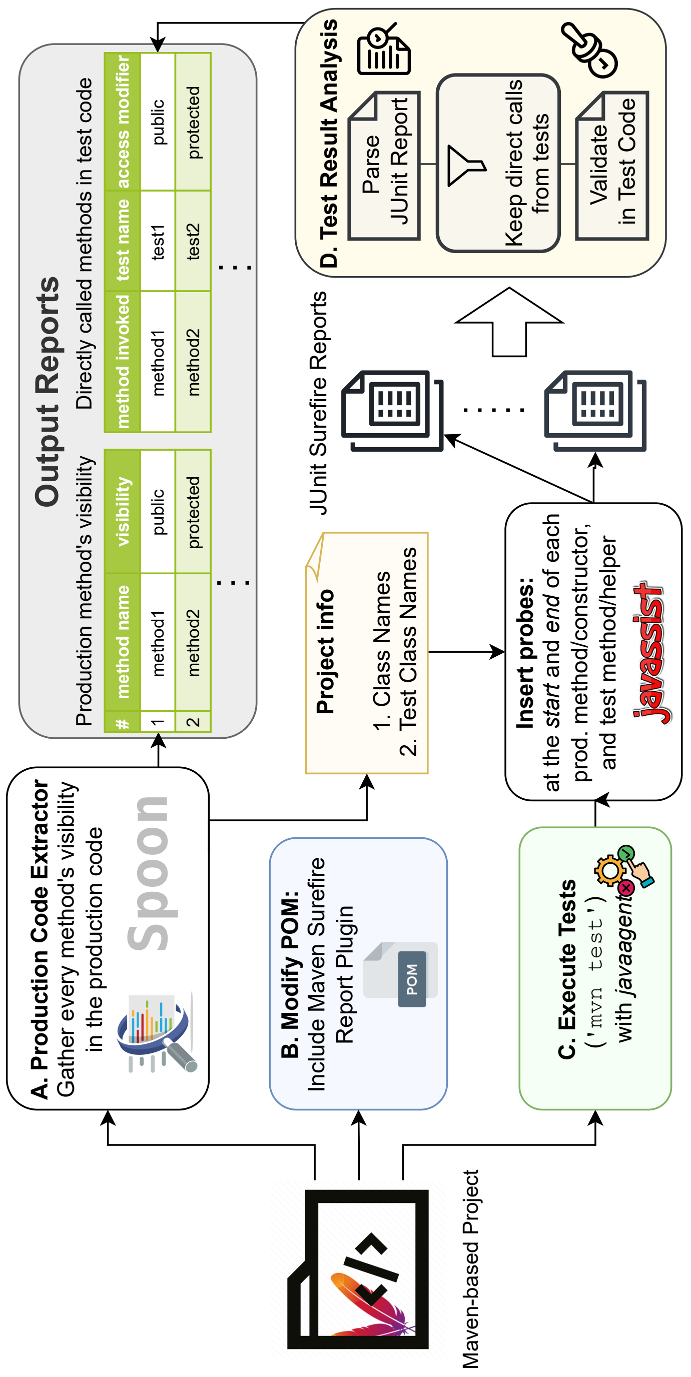

Maven-based Project
A. Production Code Extractor
Gather every method's visibility in the production code
Spoon
B. Modify POM:
Include Maven Surefire Report Plugin
POM
C. Execute Tests
('mvn test')
with javaagent
Project info
1. Class Names
2. Test Class Names
Insert probes:
at the start and end of each prod. method/constructor, and test method/helper
javassist
JUnit Surefire Reports
. . . . .
D. Test Result Analysis
Parse JUnit Report
Keep direct calls from tests
Validate in Test Code
Output Reports
Production method's visibility
# | method name | visibility
1 | method1 | public
2 | method2 | protected
. . .
Directly called methods in test code
method invoked | test name | access modifier
method1 | test1 | public
method2 | test2 | protected
. . .

Figure A.4: *Overall architecture of Viscount*

Table A.2: *Production methods directly called in test for **square-javapoet**. The output will be provided in a TSV file.*

TEST CASE (TC)	METHOD NAME	VISIBILITY
...TypeNameTest.isPrimitive()	...TypeName.isPrimitive()	public
...UtilTest.characterLiteral()	...Util.characterLiteralWithou...	package-private
...ClassNameTest.peerClass()	...ClassName.get(java.lang.Class)	public
⋮	⋮	⋮

To further describe the operation of Viscount, I will use the following two projects as running examples:

1. **square-javapoet**[2], a Java API to generate `.java` source files; and
2. **viscount-example**, an example project that I created to demonstrate the tool, partially shown by Figures A.1–A.3, and which is located in Viscount repository.

A.4.1 Extracting Production Code Methods

The first step in this analysis is to investigate the visibility of each method in the production code. Viscount starts the analysis by using Spoon's MavenLauncher to build the AST of the production code. It collects every method name and its visibility in all classes (including nested classes) in the production code (via Spoon's `CtMethod`). It also includes the parameters of each method to make sure that handling of method overloading is handled correctly. Table A.1 shows an example of the TSV file output of this step. Since Spoon's MavenLauncher can distinguish between production classes and test classes, Viscount stores the names of these two types of classes in a temporary file. This is an important step during the execution of the tests, which will be discussed in Section A.4.3.

A.4.2 Including Surefire Report Plugin

Before executing the tests, Viscount automatically includes the Surefire Report plugin [26] as part of the project, which generates reports for the executed unit tests. Viscount does this by adding the report plugin to the project's parent `POM.xml` file (a build configuration file used in Maven projects). The plugin is used to generate the test reports, which are then used in the next step of Viscount to analyse the test suites. This is an important step to ensure that the test reports are generated with the correct formatting.

[2]`https://github.com/square/javapoet`

A.4.3 Runtime Instrumentation and Test Execution

Next, Viscount launches '`mvn test`' to run the project's tests. It attaches a Java agent [30] that dynamically inserts instrumentation into the production methods and test code using Javassist [96] before the class is loaded by the Java Virtual Machine. Since Viscount is only interested in direct method call coverage of the production code, it will only insert probes into methods and constructors that the classes are part of the project, and not imported libraries or Java's own APIs. The agent uses the information collected in the previous step (Section A.4.1) to distinguish between the production classes and test classes, inserting a probe at the beginning and end of each constructor and method. Viscount does this by using the `CtMethod` (representing a method in Java) and `CtConstructor` (representing a constructor) `insertBefore/After` method in Javassist. Since some tests potentially throw exceptions, Viscount also insert an exit log when an exception is being thrown from `Exception` class, as it can handle any type of exceptions. The probe logs the method name, the parameters, and its visibility on every entry and exit point (Figure A.5). In the test code, the agent inserts a probe at the beginning and end of each test method and helper (Figure A.6). During the execution of the tests, these logs will be included in the test reports generated by Surefire Report, which will be used in the next step (Section A.4.4).

The overall process of instrumentation is similar to `java-callgraph` [138] when generating call graph dynamically. The current limitation of Viscount, similar to the one in `java-callgraph`, is that it does not work reliably for multithreaded and concurrent programs, as the probes could interleave between threads, causing inaccuracies in later processing. Therefore, Viscount skips any tests involving concurrent execution in the analysis, as discussed in the next section.

A.4.4 Analysing Test Reports

The final step for Viscount is to analyse the JUnit XML test reports generated by the Surefire Report plugin. Since Viscount inserts the probes into both production and test code, as described in the previous subsection, it can now extract the methods that are being directly called [221] in the test code from the logs produced. Figure A.7 shows an example of the output from one test case in the JUnit XML report.

To analyse the test reports (e.g. Figure A.7), Viscount parses the XML file and extracts the methods directly invoked from the tests. It discards any methods or constructors called from other production methods, as these are not directly invoked from the tests. The highlighted methods from Figure A.7 is an example of those directly invoked from the test. Finally, to ensure that a method is directly invoked from the test, Viscount statically checks if the method name is called directly from the test. Table A.2 shows an example of the output.

Viscount does not include any test cases that failed or were skipped. It also discards any test for which it could not find both entry and exit points or where the points are interleaved between methods. This is to ensure that the results are accurate and

```java
public String newName(String suggestion, Object tag) {
  logStartMethod("newName(String,Object)");
  checkNotNull(suggestion, "suggestion");
  checkNotNull(tag, "tag");
  // ... omitted ...
  return suggestion;
  // This is conceptually after the return, but
   actually runs before the return in bytecode
  logEndMethod("newName(String,Object)");
}
```

Figure A.5: *Example of added probes at entry/exit points in production method.*

```java
@Test
public void characterMappingSubstitute() throws
   Exception {
  logStartTest("characterMappingSubstitute()");
  NameAllocator nameAllocator = new NameAllocator();
  assertThat(nameAllocator.newName("a-b", 1)).
   isEqualTo("a_b");
  logEndTest("characterMappingSubstitute()");
}
```

Figure A.6: *Example of added probes at entry/exit points in test method.*

```
<testcase name="isPrimitive" classname="TypeNameTest" ...>
<system-out>
   START TEST: com.squareup.javapoet.TypeNameTest.isPrimitive()
   Start method call:  1 com.squareup.javapoet.TypeName.isPrimitive()
   End method call:  1 com.squareup.javapoet.TypeName.isPrimitive()
   Start method call:  137 com.squareup.javapoet.ClassName.get(...)
      Start constructor call: 2 com.squareup.javapoet.ClassName
   (...)
         // ... omitted ...
      End constructor call: 2 com.squareup.javapoet.ClassName
   (...)
   End method call:  137 com.squareup.javapoet.ClassName.get(...)
   // ... omitted ...
   Start method call:  137 com.squareup.javapoet.ClassName.get(...)
      Start constructor call: 2 com.squareup.javapoet.ClassName
   (...)
      // ... omitted ..
      End constructor call: 2 com.squareup.javapoet.ClassName
   (...)
   End method call:  137 com.squareup.javapoet.ClassName.get(...)
   // ... omitted ...
   END TEST: com.squareup.javapoet.TypeNameTest.isPrimitive()
</system-out>
</testcase>
```

Figure A.7: *Output of `TypeNameTest.isPrimitive()` test from `square-javapoet` in Surefire Report*

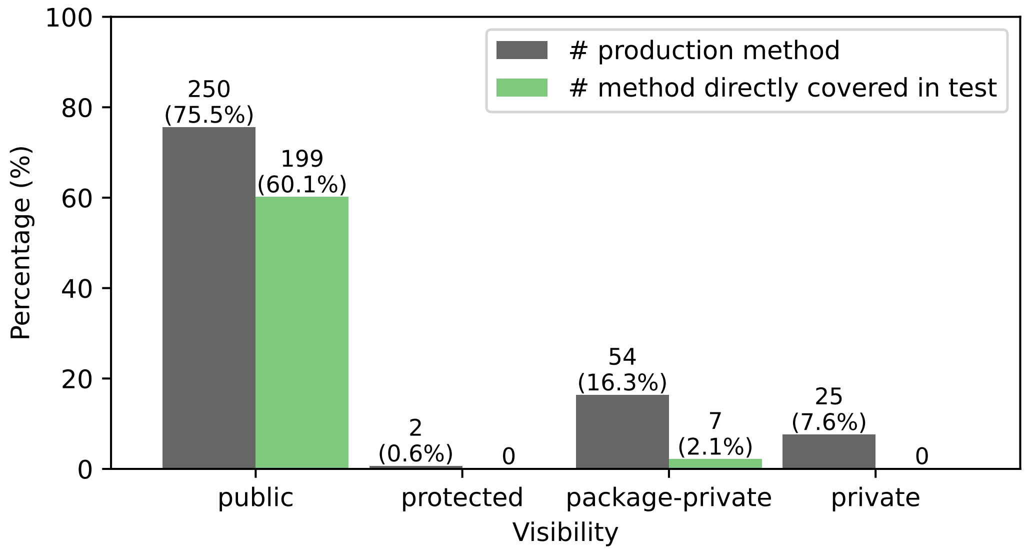


Figure A.8: *Direct method call coverage of* ***square-javapoet***

reliable, as some methods could be called from multiple threads or concurrent calls in the production code.

Using all of this information, Viscount can now calculate the direct method call coverage. The direct method call coverage is calculated by dividing the number of *unique* methods being directly invoked in the test code by the total number of methods in the production code, as shown in Figure A.8 for **square-javapoet** and Figure A.9 for **viscount-example**.

A.5 Applying the Tool

To demonstrate the capability of the tool, I replicated the analysis on **viscount-example**. In this example project, it contains two test cases: `testResize` (Figure A.1) and `testAddCard` (Figure A.2). The production methods and the visibility are shown in Table A.3 and the methods being directly invoked in the test code are shown in Table A.4. As shown in Figure A.9, there are two non-public methods being called directly in the test suite, suggesting to developers that they have tests that are more coupled to the implementation details.

I have also applied Viscount on a larger scale (4,801 projects) to evaluate the tool, as shown in previous chapter. These projects are Maven-built projects from the Maven Central Repository [25] that contain at least one passing test with source code available on GitHub. I extracted the visibility of each method in the production code and identified the methods directly called in the test code. I was able to analyse 226,915 tests from 4,801 projects, where I found 28% of the projects have at least one direct call to a non-public method in the test code. Overall, 3.73% of methods directly called

Table A.3: *Every production methods' access modifier in **viscount-example**.*

METHOD	VISIBILITY
`wallet.Wallet.capacity()`	`protected`
`wallet.Wallet.addCard(wallet.Card)`	`public`
`wallet.Wallet.isCardPresent(wallet.Card)`	`protected`
`wallet.Wallet.size()`	`public`
`wallet.Wallet.resize()`	`private`
`wallet.Example.main(java.lang.String[])`	`public`
`wallet.Card.toString()`	`public`
`wallet.Wallet.toString()`	`public`
`wallet.Wallet.latestCard()`	`package-private`

Table A.4: *Production methods directly called in tests for **viscount-example**.*

...	TEST CASE (TC)	METHOD NAME	VISIBILITY	...
...	`...testAddCard()`	`...addCard(wallet.Card)`	`public`	...
...	`...testAddCard()`	`...size()`	`public`	...
...	`...testResize()`	`...resize()`	`private`	...
...	`...testResize()`	`...capacity()`	`protected`	...

in the tests across all projects are non-public methods.

A.6 Current Limitations

As discussed in Section A.4.3, Viscount cannot compute direct method call coverage for test cases that execute multithreaded code. This is because it cannot guarantee the entry and exit points of each method do not interfere with other methods. Additionally, since Viscount executes a project's tests with instrumentation (inserting probes at entry and exit points) to both production code and test code, the execution time of the tests can be long for projects that include recursive calls. One solution to these limitations is by statically analysing the call graph of each test method/helper to determine the direct method calls [158]. However, most static call graph cannot identify methods invoked through Java Reflection [171]. Finally, as Viscount uses Spoon's MavenLauncher to analyse the source code, it does not support other build systems, such as Gradle or

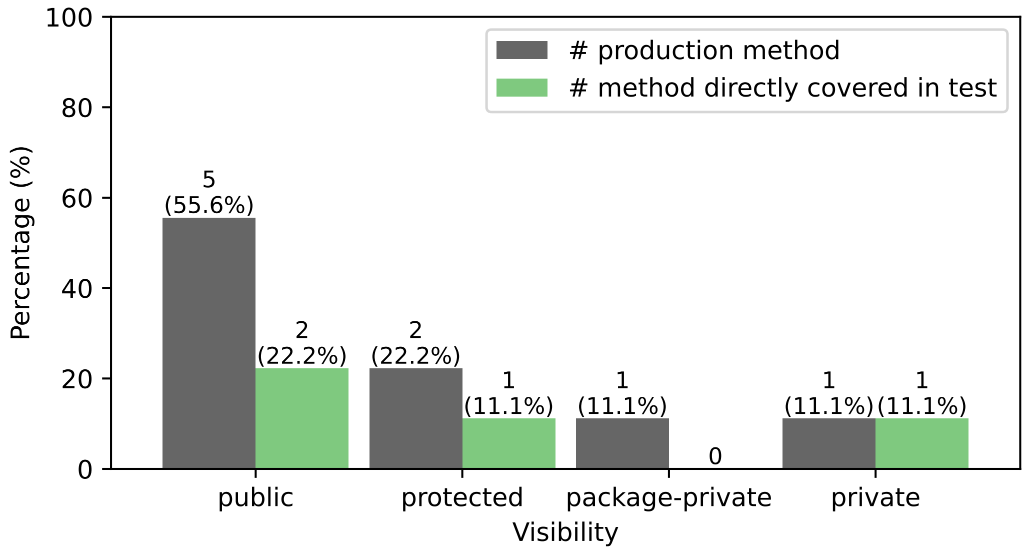


Figure A.9: *Direct method call coverage of* ***viscount-example***

Ant.

A.7 Related Tools

JaCoCo, a popular code coverage tool for Java provides a way to measure method coverage. Since it works independently and does not rely on any build tools (e.g. Maven, Gradle), it cannot distinguish direct or indirect calls to production methods as it does not track call hierarchies [17]. Mainly, since Viscount could do both (as shown in Section A.4.1 and Section A.4.3), it can determine direct method invocations by tests. Yang et al. [255] developed a test smell detector that can statically detect direct invocation of a private method in test code, also known as “Private Method Test” (PMT). Unlike Viscount, their test smell detector could only detect private methods that are being directly called in the test code, where Viscount can detect all levels of access modifiers (i.e., public, protected, package-private, and private).

A.8 Conclusions and Future Work

This chapter presents Viscount, a tool to help developers analyse which methods are being directly called in their test code. It is designed to assist with test maintenance activities, such as ensuring that tests focus on behavioural aspects of code rather than implementation details. It can aid in identifying tests that directly exercise non-public methods directly (i.e., protected, package-private, and private), which is considered a bad practice in unit testing.

Viscount supports three features that will help developers identify test cases that are calling non-public methods directly, facilitating test maintenance tasks:

- Retrieving every production method and its visibility;
- Identifying directly called methods in the test code; and
- Summarising *direct method call coverage* in the test code.

In future work, I plan to improve Viscount's report generation and enhancing it's handling of different type of members (which include fields, constructors, and classes) that are directly called from the test code. I also plan to extend the tool by incorporating static analysis (Spoon [213]) and bytecode analysis (SootUp [158]) to identify methods being invoked in the tests rather than relying on test execution. This will improve the tool's efficiency and reduce the time taken to analyse the test code.

Appendix B

Private — Keep Out? Ethics Application

The University Of Sheffield.

Application 053606

Amendment - Complete (Submitted on 27/09/2023)

Delete

Description of changes

As we aim to increase the number of participants for our questionnaire survey, we intend to promote our survey on the Prolific website.

Additional ethical considerations

Do the proposed changes pose any additional ethical considerations?
No

Additional risks

Do any of the proposed amendments to the research potentially change the risk for any of the researchers?
No

Supporting documentation revisions

Do the proposed amendments require revisions to any of the supporting documentation? Please note that when uploading new versions of documents which you have previously provided, you should give a description of the document which clearly indicates that this is a new version, e.g. by providing an appropriate version number. It is also helpful to the reviewers if you clearly mark the changes you have made in the document itself (e.g. by highlighting new text or using tracked changes in Word).
No

Other relevant information

Decision

should be approved

Amendment - Complete (Submitted on 03/08/2023)

Delete

Description of changes

In an effort to encourage more participants to join this survey, we would like to include a lucky prize draw for them. Thus, we have included an additional statement in the participant consent section: "- I understand that if I do not include my email address, I will not be eligible for participation in the lucky draw." By doing so, only participants who choose to share their email addresses will be eligible for entry into the lucky prize draw, which will be limited to a maximum of 5 winners. Participants who wish to remain completely anonymous can still do so, but they will not be included in the draw. We believe that this addition to the consent form will enhance engagement and participation in our survey. We have followed the UREC Participation Form format to include a prize draw in the consent form.

Additional ethical considerations

Do the proposed changes pose any additional ethical considerations?
No

Additional risks

Do any of the proposed amendments to the research potentially change the risk for any of the researchers?
No

Supporting documentation revisions

Do the proposed amendments require revisions to any of the supporting documentation? Please note that when uploading new versions of documents which you have previously provided, you should give a description of the document which clearly indicates that this is a new version, e.g.

by providing an appropriate version number. It is also helpful to the reviewers if you clearly mark the changes you have made in the document itself (e.g. by highlighting new text or using tracked changes in Word).
Yes

Uploaded documentation
- Exploring_Unit_Testing_Practices_in_the_Industry.pdf

Other relevant information

Decision

should be approved

Amendment - Complete (Submitted on 19/07/2023)

Delete

Description of changes

In the participant consent section, participants were asked to agree to this statement: "- I agree to assign the copyright I hold in any materials generated as part of this project to The University of Sheffield." However, we have decided not to store the data in a data repository, making this copyright assignment unnecessary. As stated in the UREC Participant Consent Form about this point, there are no intentions to utilize the data beyond the scope of this project. Moreover, participants cannot preview the survey questions, and requesting them to grant copyright beforehand may lead to confusion and reluctance to participate. We have therefore decided to drop this statement from the consent form, as no transfer of copyright is required in our survey. No more changes have been made to the application.

Additional ethical considerations

Do the proposed changes pose any additional ethical considerations?
No

Additional risks

Do any of the proposed amendments to the research potentially change the risk for any of the researchers?
No

Supporting documentation revisions

Do the proposed amendments require revisions to any of the supporting documentation? Please note that when uploading new versions of documents which you have previously provided, you should give a description of the document which clearly indicates that this is a new version, e.g. by providing an appropriate version number. It is also helpful to the reviewers if you clearly mark the changes you have made in the document itself (e.g. by highlighting new text or using tracked changes in Word).
Yes

Uploaded documentation
- Participant_Consent_Form.pdf

Other relevant information

UREC Participant Consent Form: https://www.sheffield.ac.uk/media/3507/download

Decision

should be approved

Original application

Section A: Applicant details

Date application started:
Wed 10 May 2023 at 10:50

First name:
Muhammad Firhard

Last name:
Roslan

Email:
mfroslan2@sheffield.ac.uk

Programme name:
Computer Science

Module name:
n/a
Last updated:
29/11/2023

Department:
Computer Science

Applying as:
Postgraduate research

Research project title:
Exploring the Common Themes Among Developers When Writing Unit Tests: Behaviour-Driven vs Implementation-Based Testing

Has your research project undergone academic review, in accordance with the appropriate process?
No

Similar applications:
- not entered -

Section B: Basic information

Supervisor

Name	Email
Phil McMinn	p.mcminn@sheffield.ac.uk

Proposed project duration

Start date (of data collection):
Sat 15 July 2023

Anticipated end date (of project)
Sat 30 December 2023

3: Project code (where applicable)

Project externally funded?
No

Project code
- not entered -

Suitability

Takes place outside UK?
No

Involves NHS?
No

Health and/or social care human-interventional study?
No

ESRC funded?
No

Likely to lead to publication in a peer-reviewed journal?
Yes

Led by another UK institution?
No

Involves human tissue?
No

Clinical trial or a medical device study?
No

Involves social care services provided by a local authority?
No

Is social care research requiring review via the University Research Ethics Procedure
No

Involves adults who lack the capacity to consent?
No

Involves research on groups that are on the Home Office list of 'Proscribed terrorist groups or organisations?
No

Indicators of risk

Involves potentially vulnerable participants?
No
Involves potentially highly sensitive topics?
No

Section C: Summary of research

1. Aims & Objectives

Unit tests are short tests designed to test small components (units) of a larger piece of software. Developers write and maintain unit tests throughout the software development lifecycle. Unit tests are written before or after a unit is written to ensure it has been developed correctly. After this, unit tests serve as "regression" tests, to ensure that the unit continues to work correctly, and hasn't broken – either as the result of direct or indirect changes to it or the software as a whole.

There are two main schools of thought when it comes to writing unit tests: "behaviour-driven testing" and "implementation-based testing". With behaviour-driven testing, unit tests test the externally visible behaviours of the unit only. That is, for example, they can only call and access public methods and instance variables of a class. Implementation-based testing, on the other hand, can invoke protected and/or package-private methods/instance variables as well, to ascertain at a finer-grained level that the unit is implemented correctly. As part of this strategy, developers may even resist making methods/instance variables completely private, so that they can be called or checked directly by their unit tests.

While implementation-based testing allows developers to gain more confidence that units are working correctly by essentially treating the unit as a "white box", other software engineers prefer to test behaviour only – believing that internal implementation details may change over time, causing implementation-based tests to require maintenance or be thrown away. In other words, implementation-based tests are seen as "brittle" tests. Arguably, behaviour-driven testing is a harder, more disciplined approach to testing, but with a longer-term pay-off.

Each approach has its pros and cons, and developers may fall into one camp or the other.

The aim of this study is twofold:
Firstly, we will empirically assess real software developers by means of a Google questionnaire to see if there is a preference for one or the other in real software development practice and what the deeper, underlying reasons are for choosing one approach over the other.

Secondly, we will also search for blog posts and question-and-answer (Q&A) sites such as StackOverflow related to the topic to also study development practices and opinions. To ensure that we minimize potential biases, we will do a comprehensive systematic review to gather all the information from the Q&A sites.

Our objectives are to:
1) Identify preferences in practicing behaviour-driven and/or implementation-based testing among real software developers.
2) To uncover developer opinions surrounding perceived effectiveness between behaviour-driven or implementation-based practices in unit testing among software developers.
3) To discover the factors that might influence software developers to adopt one/both of the practices in unit testing.
4) To develop recommendations to software developers on how to avoid creating hard-to-maintain tests, based on the findings of this

study.

2. Methodology

We plan to conduct two studies:

1) An empirical assessment of software developers in the industry:
We will distribute a questionnaire (via Google Forms) using social media (LinkedIn, Twitter), and through email to our industrial connections. The collected answers will be stored in the University's Google Drive.
Our questions will centre around their behaviours in terms of writing tests. All questions will require an answer, but we will include 'No Opinion' as an option to avoid participants responding with a random selection. The questions we will ask developers will be around the following themes:
1) The main motivation behind their approach of testing public/non-public methods
2) The approach that they take to write tests for public/non-public methods
3) Unintended consequences of different practices

A draft of our questionnaire survey is available at
https://docs.google.com/forms/d/e/1FAIpQLSfvJ521hm3BKaOhQOS5nC1h7wWLTjU7XvCzqYEqW5gb7xzxVQ/viewform

We will also collect demographic information (e.g., years of experience, main programming languages used). We will not ask for personal information, although we will keep the email addresses of participants who wish to be kept informed of the results of the study. When writing up the work, we will not identify participants individually nor use any of this personal information and the data will not be accessible to other researchers.

2) Supporting information from Blogs and Q&A websites.
To complement the developer survey responses, we will also collect information from questions-and-answer (Q&A) sites – such as StackOverflow and StackExchange – as well as relevant blog posts, systematically. This will allow us to gather further opinions on the topic of implementation-based / behaviour-driven testing from developers.
We will search for related keywords in StackOverflow to the topic of our research and gather opinions from the StackOverflow threads and blog posts. After gathering relevant StackOverflow threads and blog posts, we will create a dataset of relevant threads. We will not store nor use information about users who post this information in our analysis or in publication. All information collated will be stored on the university's Google Drive.

3. Personal Safety

Have you completed your departmental risk assessment procedures, if appropriate?

Not Applicable

Raises personal safety issues?

No

This questionnaire survey does not raise any issues of personal safety or physical or mental well-being for the researchers involved in the project. We believe that there are no personal safety issues to be concerned about.

Section D: About the participants

1. Potential Participants

Potential participants should be an individual that is involved in the development of proprietary software or open-source project. They should also be involved in the software testing process of a project.

2. Recruiting Potential Participants

Social Media (Twitter, LinkedIn), Emails, and Direct Contacts

2.1. Advertising methods

Will the study be advertised using the volunteer lists for staff or students maintained by IT Services? No

- not entered -

3. Consent

Will informed consent be obtained from the participants? (i.e. the proposed process) Yes

Since the questionnaire survey will be in a Google Form, we will attach the consent form that the participants need to fill out before starting the questionnaire survey. Link:

https://docs.google.com/forms/d/e/1FAIpQLSfvJ521hm3BKaOhQOS5nC1h7wWLTjU7XvCzqYEqW5gb7xzxVQ/viewform

4. Payment

Will financial/in kind payments be offered to participants? No

5. Potential Harm to Participants

What is the potential for physical and/or psychological harm/distress to the participants?

As this is going to be a questionnaire survey related to software testing practice, we do not anticipate any physical or psychological harm to the participants. Since we will not be storing the participant’s personal information, it will not risk the participant’s reputation. If there is any personal information that is being revealed intentionally or unintentionally in the 'free text' questions, we will remove and filter them from the raw data.

How will this be managed to ensure appropriate protection and well-being of the participants?

Participants will be free to withdraw from the study at any point without having to give any justification. This option will only be available for the participant that includes their email address, which is not mandatory, but not for the ones that do not provide their email address.

6. Potential harm to others who may be affected by the research activities

Which other people, if any, may be affected by the research activities, beyond the participants and the research team?

There are no other people who may be affected by the questionnaire survey beyond the participants. No personal information will be collected from the participants.

What is the potential for harm to these people?

The potential for harm to participants is none, as this questionnaire survey does not involve any physical or psychological risks.

How will this be managed to ensure appropriate safeguarding of these people?

To ensure the appropriate safeguarding of the participants, the survey will be voluntary and anonymous, and their consent will be obtained before they participate in the survey.

7. Reporting of safeguarding concerns or incidents

What arrangements will be in place for participants, and any other people external to the University who are involved in, or affected by, the research, to enable reporting of incidents or concerns?

Participants will be provided with my university email address, and are encouraged to report any concerns. The survey will also include a free-text comment box at the end of the questionnaire where participants can provide any feedback.

Who will be the Designated Safeguarding Contact(s)?

The Designated Safeguarding Contact(s) for this study will be the Head of the Department of Computer Science (g.j.brown@sheffield.ac.uk), and will be responsible for handling any concerns that are reported by participants.

How will reported incidents or concerns be handled and escalated?

Any concerns reported by participants will be taken seriously and handled promptly. If necessary, the concern will be escalated to the relevant authorities.

Section E: About the data

1. Data Processing

Will you be processing (i.e. collecting, recording, storing, or otherwise using) personal data as part of this project? (Personal data is any information relating to an identified or identifiable living person).
Yes

Which organisation(s) will act as Data Controller?

Other

We intend to use Google Forms for the questionnaire and it will be stored in the University's Google Drive. Since we would like the participant to have an option of revoking their participation, we will need to collect participants' email addresses. In the survey, participants are not required to fill in their email addresses as we only make it an optional free text answer for them to fill in. We can only withdraw participants that include their email addresses and the option to withdraw expires one month following their submission.

2. Legal basis for processing of personal data

The University considers that for the vast majority of research, 'a task in the public interest' (6(1)(e)) will be the most appropriate legal basis. If, following discussion with the UREC, you wish to use an alternative legal basis, please provide details of the legal basis, and the reasons for applying it, below:

- not entered -

Will you be processing (i.e. collecting, recording, storing, or otherwise using) 'Special Category' personal data?
No

3. Data Confidentiality

What measures will be put in place to ensure confidentiality of personal data, where appropriate?

As we would like to stay in touch with participants who are interested in the results of the study, participants have the option to leave their email addresses in the Google Form.
We will create an anonymised copy of the raw responses with the email address removed. This is to carefully not include any personal information when we are analysing the data. My supervisor will only use the raw responses when a participant wants to withdraw their participation.

For the free text comments, we will not restrict the participants to what they can write, but my supervisor will redact any sentences that identify individuals or organisations from the raw data, and it will not be part of our final publication.

4. Data Storage and Security

In general terms, who will have access to the data generated at each stage of the research, and in what form

We will make sure that the raw responses will only be accessible to both of my supervisors. My supervisors will only access the raw responses if we receive an email from participants who want to withdraw from this survey effectively and the expiration of the withdrawal is one month following their submission. We will also not give the filtered data to other researchers, mitigating any concern risk about the raw/filtered data.

What steps will be taken to ensure the security of data processed during the project, including any identifiable personal data, other than those already described earlier in this form?

The raw responses specifically for the email addresses (optional to be shared by the participants) have been moved to another part of the Google Sheet and can only be accessible by my supervisors.

Will all identifiable personal data be destroyed once the project has ended?
Yes

Please outline when this will take place (this should take into account regulatory and funder requirements).

The personal data (participants' email addresses) will be destroyed right after we submit our results to a publication. The filtered responses (without the participants' email addresses) will be immediately destroyed after I finish my Ph.D. or 6 months after publication.

Section F: Supporting documentation

Information & Consent

Participant information sheets relevant to project?
Yes

Document 1122455 (Version 3) All versions

Consent forms relevant to project?
Yes

Document 1122457 (Version 5) All versions
The consent form will be the first section of the questionnaire survey in the Google Form.

Additional Documentation

External Documentation

Participant Information Sheet: https://docs.google.com/document/d/1BNj9WjrZr2-OCZripEmpcUOqeIhr16gpKilT2anbpBQ/edit?usp=sharing

Section G: Declaration

Signed by:
Muhammad Firhard Roslan
Date signed:
Fri 30 June 2023 at 14:40

Offical notes

- not entered -

Appendix C

Private — Keep Out? Developer Questionnaire and Information Sheet

Simplifying Access Modifiers Terminology

Taking into account the unique terminology used in various programming languages, we now present the formal definitions of the access modifiers that will be used in the questionnaire survey:

1. **Public:** Globally and externally visible, accessible from *any* part of the program.
2. **Non-Public:** Not globally visible, *limited* accessibility.
    - **Protected / Semi-visible:** Partially visible (to other Module / Package / Subclass / Class), *but* not globally.
    - **Private:** Visible *only* within the class / module / file; i.e., *inaccessible from outside* of the class / module / file.

Access modifier prefixes in certain programming languages (e.g. Python) are primarily based on **convention (C)** rather than strict enforcement by the language. This means that the prefixes used to indicate access modifiers are not mandated by the language itself.
The table provided below presents a comprehensive overview of the three access modifiers in different programming languages:

Programming Language	**Public**	**Non-Public**	
		Protected / Semi-visible	**Private**
C#	`public method/variable`	`protected / internal / protected internal / private protected method/variable`	`private method/variable`
C++	`public method/variable`	`protected method/variable`	`private method/variable`
Go	`Method/Variable (uppercase letter)`	`method/variable (lowercase letter)`	`Not available`
Java	`public method/variable`	`protected / package-private method/variable`	`private method/variable`
Javascript **(C)**	`[method/variable]`	`_[method/variable]`	`#[method/variable]`
Kotlin	`public method/variable`	`protected / internal method/variable`	`private method/variable`
Python **(C)**	`[method/variable]`	`_[method/variable]`	`__[method/variable]`
PHP	`public method/variable`	`protected method/variable`	`private method/variable`
Ruby	`public def method/variable`	`protected def method/variable`	`private def method/variable`
Rust	`pub method/variable`	`pub(crate) / pub(super) / trait method/variable`	`[method/variable]`
Typescript	`export / public method/variable`	`protected method/variable`	`private method/variable`

Industry Survey

* Indicates required question

Survey Questions

Please refer to the Simplifying Access Modifiers Terminology section in the Participant Information Sheet to get more information about the different access modifiers used in different programming languages covered by our questionnaire.

1. What is your main programming language? *

   Mark only one oval.

   - C#
   - C++
   - Go
   - Java
   - JavaScript
   - Kotlin
   - PHP
   - Python
   - Ruby
   - Rust
   - Typescript
   - Other: ____________________

2. If your main language **does not have access modifiers** (e.g., public, private, protected, etc.), do you follow any *conventions* to denote visibility of methods and instance variables? *

   (For example, some Python programmers elect to prefix private methods with an underscore.)

   Mark only one oval.

   - Yes
   - No
   - My main programming language has access modifiers
   - Other: ____________________

3. If you answered "yes", please tell us what conventions you follow.

   If you answered "no", please tell us your reasons as to why you do not follow any conventions.

   ____
   ____
   ____
   ____
   ____

Based on your main programming language chosen previously, we will ask specific questions about how do you do testing for public or non-public (e.g., private, protected) methods, following the syntax and conventions of your main programming language that you stated earlier in the questionnaire.

4. Does the code that you are usually testing involve **non-public** methods? *

   Mark only one oval.

   - Yes
   - No
   - Not sure

5. To what extent do you agree with the following statement? *

"In general, developers should write unit tests that only invoke *public methods*, avoiding direct calls to *non-public methods*."

Mark only one oval.

- Strongly disagree
- Disagree
- Agree
- Strongly agree
- Not sure

6. How often do you write tests that **directly invoke** non-public methods? *

Mark only one oval.

- Never
- Rarely
- Sometimes
- Often
- Always

7. How do you go about testing **non-public** methods? *

Mark only one oval per row.

	Not a feature in my language	Never	Rarely	Sometimes	Often	Mostly	Not sure
Via public method that invokes the non-public method							
Directly invoking the non-public method							
Using Reflection / Mocks							
Adding test code (e.g., print statements or assertions) in production code							
Temporarily changing non-public methods to public							
Changing the visibility of the method to public permanently							

8. Are there any other approaches or strategies that you use to test **non-public** methods?

9. Do you take a different approach for testing **different levels of visibility of non-public methods?** (For example, is your treatment of *private* methods different compared to *protected* methods?) *

Mark only one oval.

- Yes
- No

10. If you answered "yes", please tell us how and why.

11. Are there any guidelines, best practices, or specific rules you follow when testing **non-public** methods?

If so, please tell us about them here.

12. To what extent do you value the following aspects when writing unit tests? *

Mark only one oval per row.

	Not important	Somewhat important	Important	Very Important	No opinion
Coverage of production code	⬭	⬭	⬭	⬭	⬭
Capturing the behaviour of the production code (through assertions)	⬭	⬭	⬭	⬭	⬭
Ease of debugging following production code failures	⬭	⬭	⬭	⬭	⬭
Robustness following refactoring of production code	⬭	⬭	⬭	⬭	⬭
Sensitivity to behavioural changes of production code	⬭	⬭	⬭	⬭	⬭
Realistic exercising of the unit by the test in a similar way to its usage in production	⬭	⬭	⬭	⬭	⬭
Tests that instill confidence in the production code	⬭	⬭	⬭	⬭	⬭
Writing concise unit tests to test the production code	⬭	⬭	⬭	⬭	⬭

13. To what extent do you agree with the following statement? *

"Testing non-public methods leads to more tests failing spuriously when modifications are made to those methods."

Mark only one oval.

⬭ Strongly Disagree

⬭ Disagree

⬭ Agree

⬭ Strongly Agree

⬭ Not sure

14. If you have anything else to say about your thoughts and/or processes when writing unit tests with respect to testing public and non-public methods, please let us know here:

15. Are there features you'd like to see in unit testing or mocking tools in the future to better accommodate the testing of non-public methods?

16. How many years of experience do you have in software development? *

Mark only one oval.

- ◯ 0–2
- ◯ 2–5
- ◯ 5–10
- ◯ 10–15
- ◯ More than 15

17. How many years of experience do you have in **writing unit tests**? *

Mark only one oval.

- ◯ 0–2
- ◯ 2–5
- ◯ 5–10
- ◯ 10–15
- ◯ More than 15

18. In which industry are you currently working in? *

Tick all that apply.

- ☐ Information Technology
- ☐ University / Education
- ☐ Electronics
- ☐ Enterprise / Business Software
- ☐ Fintech (Finance)
- ☐ Hospitality / Leisure Industry
- ☐ Healthcare
- ☐ Mass Media / Entertainment
- ☐ Public Sector / Government / Defense
- ☐ Retail Industry
- ☐ Sports Industry
- ☐ Transportation / Automotive
- ☐ Other: ____________________

19. Do you employ any of the following software development methodologies and/or programming practices? Please tick all that apply.

Tick all that apply.

- ☐ Test-driven development
- ☐ Behaviour-driven development
- ☐ Acceptance test-driven development
- ☐ Test-last development
- ☐ Feature-driven development
- ☐ Agile methodology, e.g. Scrum
- ☐ Waterfall method
- ☐ Other: ______________________

Bibliography

[1] Access modifiers — C# programming guide. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers. Online; Accessed: 9/2024.

[2] Access specifiers — cppreference.com. https://en.cppreference.com/w/cpp/language/access. Online; Accessed: 9/2024.

[3] Agitar One. http://www.agitar.com/solutions/products/automated_junit_generation.html. Online; Accessed: 9/2024.

[4] Anal probe — test smells catalog. https://test-smell-catalog.readthedocs.io/en/latest/Test%20semantic-logic/Other%20test%20logic%20related/Anal%20Probe.html. Online; Accessed: 9/2024.

[5] Apache commons lang3 — Class MethodUtils. https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/reflect/MethodUtils.html. Online; Accessed: 9/2024.

[6] Blue screen of death: Microsoft says turn it off and on again and again and again. https://www.forbes.com/sites/daveywinder/2024/07/20/blue-screen-of-death-microsoft-says-turn-it-off-and-on-again-and-again-and-again/. Online; Accessed: 3/2025.

[7] Controlling access to members of a class, the Java tutorials. https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html. Online; Accessed: 9/2023.

[8] Crowdstrike incident root cause analysis. https://www.crowdstrike.com/content/dam/crowdstrike/www/en-us/wp/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf. Online; Accessed: 3/2025.

[9] Crowdstrike it outage affected 8.5 million windows devices, microsoft says. https://www.bbc.co.uk/news/articles/cpe3zgznwjno. Online; Accessed: 3/2025.

[10] Crowdstrike: More testing, staged rollouts now in place for updates. https://www.crn.com/news/security/2024/crowdstrike-more-testing-staged-rollouts-now-in-place-for-updates. Online; Accessed: 3/2025.

[11] Crowdstrike reveals what happened, why, and what's changed. `https://www.forbes.com/sites/kateoflahertyuk/2024/08/07/crowdstrike-reveals-what-happened-why-and-whats-changed/`. Online; Accessed: 3/2025.

[12] Crowdstrike: What was the impact of the global it outage. `https://www.bbc.co.uk/news/articles/cr54m92ermgo`. Online; Accessed: 3/2025.

[13] Delta airlines hits out at crowdstrike, alleging $500m loss. `https://www.bbc.co.uk/news/articles/c6284e7r7d7o`. Online; Accessed: 3/2025.

[14] DSpot. `https://github.com/STAMP-project/dspot`. Online; Accessed: 9/2024.

[15] EasyMock — Class ReflectionUtils. `https://easymock.org/api/org/easymock/internal/ReflectionUtils.html`. Online; Accessed: 9/2024.

[16] Google Testing Blog — The advantages of unit testing early. `https://testing.googleblog.com/2009/07/by-shyam-seshadri-nowadays-when-i-talk.html`. Online; Accessed: 9/2024.

[17] JaCoCo Coverage of methods being invoked directly from a test case. `https://groups.google.com/g/jacoco/c/x4OGEGPyi3E`. Online; Accessed: 9/2024.

[18] JaCoCo Java Code Coverage Library. `https://www.eclemma.org/jacoco/`. Online; Accessed: 11/2024.

[19] JMockit — Class Deencapsulation. `https://javadoc.io/doc/com.googlecode.jmockit/jmockit/latest/mockit/Deencapsulation.html`. Online; Accessed: 9/2024.

[20] Kendall's Tau – Simple Introduction. `https://www.spss-tutorials.com/kendalls-tau/`. Online; Accessed: 9/2024.

[21] Log4j vulnerability - what everyone needs to know. `https://www.ncsc.gov.uk/information/log4j-vulnerability-what-everyone-needs-to-know`. Online; Accessed: 11/2024.

[22] Manifold Systems — Annotation Type Jailbreak. `https://javadoc.io/static/systems.manifold/manifold-ext-rt/2020.1.41/manifold/ext/rt/api/Jailbreak.html`. Online; Accessed: 9/2024.

[23] Maven — introduction to the standard directory layout. `https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout`. Online; Accessed: 9/2024.

[24] Maven central index. `https://maven.apache.org/repository/central-index.html`. Online; Accessed: 9/2024.

[25] Maven central repository. `https://repo.maven.apache.org/maven2/`. Online; Accessed: 9/2024.

[26] Maven Surefire plugin. `https://maven.apache.org/surefire/maven-surefire-plugin/`. Online; Accessed: 9/2024.

[27] Maven Surefire plugin — rerun failing tests. `https://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html`. Online; Accessed: 9/2024.

[28] The Open Catalog of Test Smells. `https://test-smell-catalog.readthedocs.io`. Online; Accessed: 9/2024.

[29] OW2. 2024. ASM. `https://asm.ow2.io/`. Online; Accessed: 9/2024.

[30] Package java.lang.instrument. `https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html`. Online; Accessed: 9/2024.

[31] Package java.lang.reflect. `https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html`. Online; Accessed: 9/2024.

[32] password-generator GitHub project. `https://github.com/javadev/password-generator`. Online; Accessed: 9/2024.

[33] Powermock — Class Whitebox. `https://www.javadoc.io/doc/org.powermock/powermock-reflect/1.6.4/org/powermock/reflect/Whitebox.html`. Online; Accessed: 9/2024.

[34] PyTest — Flaky Tests. `https://docs.pytest.org/en/stable/explanation/flaky.html`. Online; Accessed: 11/2024.

[35] Reddit: Stop using Code Coverage as a Quality metric. `https://www.reddit.com/r/programming/comments/194htrz/stop_using_code_coverage_as_a_quality_metric`. Online; Accessed: 9/2024.

[36] Spring Framework — Class ReflectionTestUtils. `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/util/ReflectionTestUtils.html`. Online; Accessed: 9/2024.

[37] Stackoverflow. `https://stackoverflow.com`. Online; Accessed: 9/2024.

[38] Technical details: Falcon update for windows hosts: Crowdstrike. `https://www.crowdstrike.com/en-us/blog/falcon-update-for-windows-hosts-technical-details/`. Online; Accessed: 3/2025.

[39] Testing with Maven — Organizing unit and integration tests. `https://dev.to/rodnan-sol/testing-with-maven-organizing-unit-and-integration-tests-35oh`. Online; Accessed: 9/2024.

[40] The Python Language Reference — Private name mangling. `https://docs.python.org/3/reference/expressions.html#private-name-mangling`. Online; Accessed: 9/2024.

[41] tvd12 test-util repository. `https://github.com/tvd12/test-util`. Online; Accessed: 9/2024.

[42] Visibility — php.net. `https://www.php.net/manual/en/language.oop5.visibility.php`. Online; Accessed: 9/2024.

[43] We finally know what caused the global tech outage - and how much it cost. `https://edition.cnn.com/2024/07/24/tech/crowdstrike-outage-cost-cause/index.html`. Online; Accessed: 3/2025.

[44] X-Ray Specs — Test Smells Catalog. `https://test-smell-catalog.readthedocs.io/en/latest/Test%20semantic-logic/Other%20test%20logic%20related/X-Ray%20Specs.html`. Online; Accessed: 9/2024.

[45] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[46] Stackoverflow: How do I test a class that has private methods, fields or inner classes? `https://stackoverflow.com/questions/34571/`, 2008. Online; Accessed: 9/2024.

[47] Stackoverflow: Making a private method public to unit test it...good idea? `https://stackoverflow.com/questions/7075938/`, 2011. Online; Accessed: 9/2024.

[48] Regression testing minimization, selection and prioritization: a survey. *Journal of Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[49] Stackoverflow: Unit testing private method - objective C. `https://stackoverflow.com/questions/18354788/`, 2013. Online; Accessed: 9/2024.

[50] Stackoverflow: Should a concrete class that implements an interface have extra public methods for testing? `https://stackoverflow.com/questions/57632038/`, 2019. Online; Accessed: 9/2024.

[51] Replication package - an empirical comparison of evosuite and dspot for improving developer-written test suites with respect to mutation score. `https://github.com/test-amplification/EvoSuiteAmp-framework`, 2022.

[52] Do automatic test generation tools generate flaky tests? [dataset]. `https://doi.org/10.6084/m9.figshare.22344706`, 2023.

[53] Stackoverflow: Unit testing a overridden protected method from a class that does not have default constructors. `https://stackoverflow.com/questions/76868236/`, 2023. Online; Accessed: 9/2024.

[54] The state of developer ecosystem 2023. `https://www.jetbrains.com/lp/devecosystem-2023/java/`, 2023. Online; Accessed: 9/2024.

[55] Evosuite test runner - do automatic test generation tools generate flaky tests? `https://github.com/firhard/evosuite-test-generation`, 2024.

[56] Replication package - private — keep out? understanding how developers account for code visibility in unit testing. `https://github.com/unittesting-nonpublic/private-keep-out_replication-package`, 2024.

[57] Replication package - replacing developer-written tests that call non-public methods directly. `https://anonymous.4open.science/r/evosuite-utopia`, 2024.

[58] Mehrdad Abdi, Henrique Rocha, Serge Demeyer, and Alexandre Bergel. Small-amp: Test amplification in a dynamically typed language. *Empirical Software Engineering*, 27(6):128, 2022.

[59] Allen T Acree, Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Mutation analysis. Technical report, Georgia Institute of Technology, Atlanta, School of Information And Computer Science, 1979.

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

[61] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. Extending source code pre-trained language models to summarise decompiled binaries. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 260–271. IEEE, 2023.

[62] Joseph Albahari. Reflection and Metadata. In *C# 12 in a Nutshell*, chapter 18. O'Reilly Media, 2023.

[63] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D. Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 170–180, 2021.

[64] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272, 2017.

[65] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at Meta. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, To Appear, 2024.

[66] Abdulrahman Alshammari, Paul Ammann, Michael Hilton, and Jonathan Bell. A study of flaky failure de-duplication to identify unreliably killed mutants. In *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 257–262. IEEE, 2024.

[67] Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, 2016.

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

[69] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering (ICSE)*, pages 402–411, 2005.

[70] Maurício Aniche, Christoph Treude, and Andy Zaidman. How developers engineer test cases: An observational study. *IEEE Transactions on Software Engineering*, 48:4925–4946, 2022.

[71] Andrea Arcuri. It does matter how you normalise the branch distance in search based software testing. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 205–214. IEEE, 2010.

[72] Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness? In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 265–275, 2011.

[73] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *International Conference on Software Engineering (ICSE)*, pages 1–10, 2011.

[74] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Automated unit test generation for classes with environment dependencies. In *International Conference on Automated Software Engineering (ASE)*, pages 79–89, 2014.

[75] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. Generating tcp/udp network data for automated unit test generation. In *International Symposium on Foundations of Software Engineering (FSE)*, page 155–165, 2015.

[76] Andrea Arcuri, Gordon Fraser, and René Just. Private API access and functional mocking in automated unit test generation. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[77] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):1–39, 2018.

[78] Sebastian Baltes and Paul Ralph. Sampling in software engineering research: a critical review and guidelines. *Empirical Software Engineering*, 27, 2022.

[79] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[80] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *International Conference on Software Maintenance (ICSM)*, pages 56–65, 2012.

[81] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry—a study at facebook. In *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 268–277, 2021.

[82] Leonardo Bottaci. A genetic algorithm fitness function for mutation testing. In *Proceedings of the SEMINALL-workshop at the 23rd international conference on software engineering, Toronto, Canada*. Citeseer, 2001.

[83] David Bowes, Tracy Hall, Jean Petric, Thomas Shippey, and Burak Turhan. How good are my tests? In *Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 9–14, 2017.

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

[85] Timothy Budd. *Introduction to object-oriented programming*. Addison-Wesley, 2008.

[86] Timothy A Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta informatica*, 18(1):31–45, 1982.

[87] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, page 209–224, 2008.

[88] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[89] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. Continuous test generation: Enhancing continuous integration with automated test generation. In *International Conference on Automated Software Engineering (ASE)*, pages 55–66, 2014.

[90] Santo Carino, James H Andrews, Sheldon Goulding, Pradeepan Arunthavarajah, and Jakub Hertyk. Blackhorse: creating smart test cases from brittle recorded tests. *Software Quality Journal*, 22:293–310, 2014.

[91] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *International Conference on Software Engineering (ICSE)*, pages 597–608. IEEE, 2017.

[92] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[93] Tsong Yueh Chen, Hing Leung, and Ieng Kei Mak. Adaptive random testing. In *Asian Computing Science Conference*, pages 320–329. Springer, 2005.

[94] Yang Chen and Reyhaneh Jabbarvand. Neurosymbolic repair of test flakiness. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 1402–1414, 2024.

[95] Jianlei Chi, Xiaotian Wang, Yuhan Huang, Lechen Yu, Di Cui, Jianguo Sun, and Jun Sun. Reaccept: Automated co-evolution of production and test code based on dynamic validation and large language models. *arXiv preprint arXiv:2411.11033*, 2024.

[96] Shigeru Chiba. Load-time structural reflection in Java. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 313–336. Springer, 2000.

[97] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: a practical mutation testing tool for java. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 449–452, 2016.

[98] William J Conover. *Practical nonparametric statistics*, volume 350. john wiley & sons, 1999.

[99] Steve Counsell and Peter Newson. Use of friends in C++ software: An empirical investigation. *Journal of Systems and Software*, 53:15–21, 2000.

[100] Daniela S. Cruzes and Tore Dyba. Recommended steps for thematic synthesis in software engineering. In *International Symposium on Empirical Software Engineering and Measurement*, pages 275–284, 2011.

[101] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[102] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming.* Academic Press Ltd., 1972.

[103] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: would you name your children thing1 and thing2? In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 57–67, 2017.

[104] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 85–96, 2010.

[105] 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.

[106] 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.

[107] Brett Daniel, Tihomir Gvero, and Darko Marinov. On test repair using symbolic execution. In *International Symposium on Software Testing and Analysis (ISSTA)*, page 207–218, 2010.

[108] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. Reassert: Suggesting repairs for broken unit tests. In *International Conference on Automated Software Engineering (ASE)*, pages 433–444, 2009.

[109] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. Leveraging large language models for enhancing the understandability of generated unit tests. In *International Conference on Software Engineering (ICSE)*, 2025.

[110] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[111] Jens Dietrich, Shawn Rasheed, and Amjed Tahir. Flaky test sanitisation via on-the-fly assumption inference for tests with network dependencies. In *IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 264–275, 2022.

[112] Edsger W. Dijkstra. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1398–1404, 1989.

[113] Hang Du, Vijay Krishna Palepu, and James A Jones. Ripples of a mutation—an empirical study of propagation effects in mutation testing. In *International Conference on Software Engineering (ICSE)*, pages 1–13, 2024.

[114] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Internet Measurement Conference*, pages 475–488, 2014.

[115] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: The developer's perspective. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 830–840, 2019.

[116] Zhiyu Fan. A systematic evaluation of problematic tests generated by evosuite. In *International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*, pages 165–167, 2019.

[117] Michael Feathers. *Working effectively with legacy code.* Prentice Hall Professional, 2004.

[118] David Flanagan and Yukihiro Matsumoto. Reflection and Metaprogramming. In *The Ruby Programming Language: Everything You Need to Know*, chapter 8. O'Reilly Media, 2008.

[119] Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, page 378, 1971.

[120] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[121] Gordon Fraser. A tutorial on using and extending the evosuite search-based test generator. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 106–130, 2018.

[122] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2011.

[123] Gordon Fraser and Andrea Arcuri. The seed is strong: Seeding strategies in search-based software testing. pages 121–130. IEEE, 2012.

[124] Gordon Fraser and Andrea Arcuri. Evosuite: On the challenges of test case generation in the real world. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 362–369, 2013.

[125] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology*, 2014.

[126] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology*, 24(4), 2015.

[127] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 147–158, 2010.

[128] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2011.

[129] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *2013 ieee 24th international symposium on software reliability engineering (issre)*, pages 360–369. IEEE, 2013.

[130] Vahid Garousi and Barış Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81, 2018.

[131] Atul Gawande. *The Checklist Manifesto: How to Get Things Right*. Metropolitan Books, 2010.

[132] Gregory Gay. To call, or not to call: Contrasting direct and indirect branch coverage in test generation. In *International Workshop on Search-Based Software Testing (SBST@ICSE)*, pages 43–50, 2018.

[133] Milos Gligoric, Vilas Jagannath, Qingzhou Luo, and Darko Marinov. Efficient mutation testing of multithreaded code. *Journal of Software Testing, Verification and Reliability*, 23(5):375–403, 2013.

[134] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[135] David E Golberg. Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989(102):36, 1989.

[136] John B Goodenough and Susan L Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510, 1975.

[137] James Gosling. *The Java language specification*. Addison Wesley, 2000.

[138] Georgios Gousios. Java-callgraph: Programs for producing static and dynamic (runtime) call graphs for Java programs. `https://github.com/gousiosg/java-callgraph`. Online; Accessed: 9/2024.

[139] Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C Gall. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software*, 156:312–327, 2019.

[140] Martin Gruber and Gordon Fraser. Flapy: Mining flaky python tests at scale. In *International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*, pages 127–131, 2023.

[141] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of flaky tests in Python. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 148–158, 2021.

[142] Martin Gruber, Muhammad Firhard Roslan, Owain Parry, Fabian Scharnböck, Phil McMinn, and Gordon Fraser. Do Automatic Test Generation Tools Generate Flaky Tests? In *International Conference on Software Engineering (ICSE)*, 2024.

[143] Mark Harman. Search based software engineering. In *International Conference on Computational Science*, pages 740–747. Springer, 2006.

[144] Mark Harman and Bryan F Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, 2001.

[145] Mark Harman and Phil McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[146] Mark Harman and Peter O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23. IEEE, 2018.

[147] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. Seed selection for successful fuzzing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 230–243, 2021.

[148] Michael Hilton, Jonathan Bell, and Darko Marinov. A large-scale study of test coverage evolution. In *International Conference on Automated Software Engineering (ASE)*, pages 53–63, 2018.

[149] Rubing Huang, Weifeng Sun, Yinyin Xu, Haibo Chen, Dave Towey, and Xin Xia. A survey on adaptive random testing. *IEEE Transactions on Software Engineering*, 47(10):2052–2083, 2019.

[150] Chen Huo and James Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 621–631, 2014.

[151] Chen Huo and James Clause. Interpreting coverage information using direct and indirect coverage. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 234–243, 2016.

[152] Javaria Imtiaz, Salman Sherin, Muhammad Uzair Khan, and Muhammad Zohaib Iqbal. A systematic literature review of test breakage prevention and repair techniques. *Information & Software Technology*, 113:1–19, 2019.

[153] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. Code coverage at google. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 955–963, 2019.

[154] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[155] Ricardo Job and Andre Hora. How and why developers implement os-specific tests. *Empirical Software Engineering*, 30(1):1–33, 2025.

[156] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 433–436, 2014.

[157] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, 2014.

[158] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. Sootup: A redesign of the soot static analysis framework. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 229–247, 2024.

[159] Mehdi Keshani, Tudor-Gabriel Velican, Gideon Bot, and Sebastian Proksch. Aroma: Automatic reproduction of maven artifacts. *Proceedings of the ACM on Software Engineering*, 2024.

[160] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[161] Barbara A. Kitchenham and Shari Lawrence Pfleeger. Personal opinion surveys. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 63–92. Springer, 2008.

[162] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[163] L. Koskela. *Effective Unit Testing: A guide for Java developers*. Manning, 2013.

[164] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot™ client compiler for java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, pages 1–32, 2008.

[165] Erik Kuefler. Unit Testing. In Titus Winters, Tom Manshreck, and Hyrum Wright, editors, *Software Engineering at Google: Lessons Learned from Programming Over Time*, chapter 12. O'Reilly Media, 2020.

[166] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 821–830, 2017.

[167] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta. A study on the lifecycle of flaky tests. In *International Conference on Software Engineering (ICSE)*, pages 1471–1482, 2020.

[168] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. iDFlakies: A framework for detecting and partially classifying flaky tests. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 312–322, 2019.

[169] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 403–413, 2020.

[170] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *Biometrics*, pages 159–174, 1977.

[171] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. Challenges for static analysis of java reflection-literature review and empirical study. In *International Conference on Software Engineering (ICSE)*, pages 507–518, 2017.

[172] Jeff Lawson. *Ask Your Developer: How to Harness the Power of Software Developers and Win in the 21st Century*. Harper Business, 2021.

[173] Meir M Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1979.

[174] Zalán Lévai and Phil McMinn. Batching non-conflicting mutations for efficient, safe, parallel mutation analysis in rust. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2023.

[175] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. Repairing order-dependent flaky tests via test generation. In *International Conference on Software Engineering (ICSE)*, pages 1881–1892, 2022.

[176] Nan Li and Jeff Offutt. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, 2016.

[177] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. Intent-preserving test repair. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 217–227. IEEE, 2019.

[178] Yue Li, Tian Tan, and Jingling Xue. Understanding and analyzing Java reflection. *ACM Transactions on Software Engineering and Methodology*, 28(2):1–50, 2019.

[179] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*, pages 168–172, 2022.

[180] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 9–24, 2020.

[181] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of automated unit test generation for Python. *International Symposium on Empirical Software Engineering and Measurement*, page 36, 2023.

[182] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 643–653, 2014.

[183] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, 2013.

[184] L. Martins, D. Campos, R. Santana, J. Junior, H. Costa, and I. Machado. Hearing the voice of experts: Unveiling stack exchange communities' knowledge of test smells. In *International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 80–91, 2023.

[185] M. Maton, G. M. Kapfhammer, and P. McMinn. Exploring pseudo-testedness: Empirically evaluating extreme mutation testing at the statement level. In *International Conference on Software Maintenance and Evolution (ICSME)*, 2024.

[186] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[187] Phil McMinn, Muhammad Firhard Roslan, and Gregory M. Kapfhammer. Beyond Test Flakiness: A Manifesto for a Holistic Approach to Test Suite Health. In *Proceedings of the 2nd International Workshop on Flaky Tests (FTW)*, 2025.

[188] Phil McMinn, Mark Stevenson, and Mark Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *International Workshop on Software Test Output Validation (STOV)*, pages 1–4, 2010.

[189] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242. IEEE, 2017.

[190] Gerard Meszaros. *xUnit test patterns: Refactoring test code.* Pearson Education, 2007.

[191] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[192] John Micco. The state of continuous integration testing @google, 2017.

[193] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1995.

[194] Amir M Mir, Mehdi Keshani, and Sebastian Proksch. On the effect of transitivity and granularity on vulnerability propagation in the maven ecosystem. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 201–211, 2023.

[195] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. Supporting test suite evolution through test case adaptation. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 231–240. IEEE, 2012.

[196] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[197] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. Will my tests tell me if i break this code? In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, pages 23–29, 2016.

[198] Mitchell Olsthoorn, Dimitri Stallenberg, and Annibale Panichella. Syntest-javascript: Automated unit-level test case generation for javascript. In *International Workshop on Search-Based & Fuzz Testing (SBFT@ICSE)*, pages 21–24, 2024.

[199] Roy Osherove. Unit Testing Tips–Write Maintainable Unit Tests That Will Save You Time And Tears. *MSDN Magazine*, pages 107–118, 2006.

[200] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for Java. In *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[201] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding errors in .net with feedback-directed random testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[202] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[203] Michael Paleczny, Christopher Vick, and Cliff Click. The java HotSpot™ server compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, pages 1–12, 2001.

[204] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017.

[205] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 523–533. IEEE, 2020.

[206] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering*, 27(7):170, 2022.

[207] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. Sbst tool competition 2021. In *International Workshop on Search-Based Software Testing (SBST@ICSE)*, pages 20–27. IEEE, 2021.

[208] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in computers*, volume 112, pages 275–378. Elsevier, 2019.

[209] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[210] Owain Parry, Michael Hilton, Gregory M. Kapfhammer, and Phil McMinn. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology*, 2022.

[211] Owain Parry, Michael Hilton, Gregory M. Kapfhammer, and Phil McMinn. What do developer-repaired flaky tests tell us about the effectiveness of automated flaky test detection? In *International Conference on Automation of Software Test (AST)*, 2022.

[212] Michael Quinn Patton. Enhancing the quality and credibility of qualitative analysis. *Health services research*, 34, 1999.

[213] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016.

[214] Goran Petrovic and Marko Ivankovic. State of mutation testing at google. In *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2018.

[215] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Practical mutation testing at scale: A view from google. *IEEE Transactions on Software Engineering*, 48(10):3900–3912, 2022.

[216] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 1–11, 2012.

[217] Md Tajmilur Rahman and Peter C Rigby. The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 857–862, 2018.

[218] Shanto Rahman and August Shi. Flakesync: Automatically repairing async flaky tests. In *International Conference on Software Engineering (ICSE)*, pages 1–12, 2024.

[219] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security Symposium*, pages 861–875, 2014.

[220] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[221] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 93–108, 2015.

[222] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 338–349, 2015.

[223] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Seeding strategies in search-based unit test generation. *Journal of Software Testing, Verification and Reliability*, 26(5):366–401, 2016.

[224] Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. An Empirical Comparison of EvoSuite and DSpot for Improving developer-written test suites with respect to mutation score. In *International Symposium on Search-Based Software Engineering (SSBSE)*, 2022.

[225] Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. Private — Keep Out? Understanding How Developers Account for Code Visibility in Unit Testing. In *International Conference on Software Maintenance and Evolution (ICSME)*, 2024.

[226] Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. Viscount: A Direct Method Call Coverage Tool for Java. In *International Conference on Software Maintenance and Evolution (ICSME): Tool Demo Track*, 2024.

[227] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. Deeptc-enhancer: Improving the readability of automatically generated tests. In *International Conference on Automated Software Engineering (ASE)*, pages 287–298, 2020.

[228] Abdelilah Sakti, Gilles Pesant, and Yann-Gaël Guéhéneuc. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering*, 41(3):294–313, 2014.

[229] Alireza Salahirad, Hussein Almulla, and Gregory Gay. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Journal of Software Testing, Verification and Reliability*, 29(4-5):e1701, 2019.

[230] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 2023.

[231] Herbert Schildt. *Java: The Complete Reference, Twelfth Edition*. McGraw-Hill Education Group, 2021.

[232] Ebert Schoofs, Mehrdad Abdi, and Serge Demeyer. Ampyfier: Test amplification in python. *Journal of Software: Evolution and Process*, 34(11):e2490, 2022.

[233] Sebastian Schweikl, Gordon Fraser, and Andrea Arcuri. Evosuite at the sbst 2022 tool competition. In *International Workshop on Search-Based Software Testing (SBST@ICSE)*, pages 33–34, 2022.

[234] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[235] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C Gall, and Alberto Bacchelli. On the effectiveness of manual and automatic unit test generation: ten years later. In *International Conference on Mining Software Repositories (MSR)*, pages 121–125. IEEE, 2019.

[236] Sina Shamshiri, José Campos, Gordon Fraser, and Phil McMinn. Disposable testing: Avoiding maintenance of generated unit tests by throwing them away. In *International Conference on Software Engineering (ICSE)*, pages 207–209, 2017.

[237] Sina Shamshiri, Rene Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an

empirical study of effectiveness and challenges. In *International Conference on Automated Software Engineering (ASE)*, pages 201–211, 2015.

[238] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, pages 591–611, 1965.

[239] August Shi, Jonathan Bell, and Darko Marinov. Mitigating the effects of flaky tests on mutation testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 112–122, 2019.

[240] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. ifixflakies: A framework for automatically fixing order-dependent flaky tests. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 545–555, 2019.

[241] Denini Silva, Martin Gruber, Satyajit Gokhale, Ellen Arteca, Alexi Turcotte, Marcelo d'Amorim, Wing Lam, Stefan Winter, and Jonathan Bell. The effects of computational resources on flaky tests. *IEEE Transactions on Software Engineering*, 2024.

[242] Dan Simon. *Evolutionary optimization algorithms.* John Wiley & Sons, 2013.

[243] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 38–45, 1986.

[244] Elvys Soares, Márcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André Santos. Refactoring test smells with JUnit 5: Why should developers keep up-to-date? *IEEE Transactions on Software Engineering*, 49(3):1152–1170, 2023.

[245] StackOverflow. Developer survey results. `https://survey.stackoverflow.co/2023`, 2023. Online; Accessed: 9/2024.

[246] Paolo Tonella. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes*, 29(4):119–128, 2004.

[247] A. Vahabzadeh, A. A. Fard, and A. Mesbah. An empirical study of bugs in test code. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110, 2015.

[248] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. Refactoring test code. In *International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP)*, pages 92–95, 2001.

[249] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. A comprehensive study of pseudo-tested methods. *Empirical Software Engineering*, 2019.

[250] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, Jose Campos, and Annibale Panichella. Evosuite at the sbst 2021 tool competition. In *International Workshop on Search-Based Software Testing (SBST@ICSE)*, pages 28–29, 2021.

[251] Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. Preempting flaky tests via non-idempotent-outcome tests. In *International Conference on Software Engineering (ICSE)*, pages 1730–1742, 2022.

[252] Elaine J Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, (12):1128–1138, 1986.

[253] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, pages 80–83, 1945.

[254] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 380–403. Springer, 2006.

[255] Yanming Yang, Xing Hu, Xin Xia, and Xiaohu Yang. The lost world: Characterizing and detecting undiscovered test smells. *ACM Transactions on Software Engineering and Methodology*, 2023.

[256] Ahmadreza Saboor Yaraghi, Darren Holden, Nafiseh Kahani, and Lionel Briand. Automated test case repair using language models. *arXiv preprint arXiv:2401.06765*, 2024.

[257] Vahid Garousi Yusifoğlu, Yasaman Amannejad, and Aysu Betin Can. Software test-code engineering: A systematic mapping. *Information & Software Technology*, 58:123–147, 2015.

[258] Andy Zaidman, Bart Van Rompaey, Arie Van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16:325–364, 2011.

[259] Michal Zalewski. American fuzzy lop (afl) technical whitepaper. `http://lcamtuf.coredump.cx/afl/technical_details.txt`, 2021.

[260] Lucas Zamprogno, Braxton Hall, Reid Holmes, and Joanne M Atlee. Dynamic human-in-the-loop assertion generation. *IEEE Transactions on Software Engineering*, 49(4):2337–2351, 2022.

[261] Peilun Zhang, Yanjie Jiang, Anjiang Wei, Victoria Stodden, Darko Marinov, and August Shi. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In *International Conference on Software Engineering (ICSE)*, pages 50–61. IEEE, 2021.

[262] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *International Symposium on Software Testing and Analysis (ISSTA)*, page 385–396, 2014.

[263] Wei Zheng, Guoliang Liu, Manqing Zhang, Xiang Chen, and Wenqiao Zhao. Research progress of flaky tests. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 639–646, 2021.

[264] Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, Xiapu Luo, Jingzhu He, and Yutian Tang. Coverage goal selector for combining multiple criteria in search-based unit test generation. *IEEE Transactions on Software Engineering*, 2024.

[265] Hong Zhu. Test data adequacy measurement. *Software Engineering Journal*, 8(1):21–30, 1993.

[266] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 1997.