Private—Keep Out? Understanding How Developers Account for Code Visibility in Unit Testing

Muhammad Firhard Roslan, José Miguel Rojas and Phil McMinn
University of Sheffield, UK

***Abstract*—Regression test maintenance costs can be reduced by striving to write tests that will require as few changes as possible in the future. Writing unit tests against behavior, as opposed to implementation, is one way to try to achieve this, because as long as the public API remains constant, units can be safely refactored without the need to also change the tests. However, in a study on 4,801 open-source Java projects reported in this paper, we found that 28% of projects contradict this advice, with tests that side-step the public API by directly calling non-public methods. We investigated why developers do not solely test public APIs—potentially increasing future test maintenance costs—by surveying 73 developers and conducting a systematic review of 60 StackOverflow posts dating from 2008–2023. Through numerical and thematic analyses, we uncover several findings, including (1) developers are disunited on whether to test only through public APIs or not; (2) those in favor of only testing through the public API tend to be more experienced and believe the need or desire to break with this is borne out of poor software design; while (3) those that test non-public methods directly are concerned about untested code complexity and overly intricate tests. Our findings provide multiple implications for future work, including automated developer support in the form of automated non-public method sequence replacement, and automated refactoring of production code using problematic public API-avoiding tests.**

***Index Terms*—Test Maintenance, Test Smells, Unit Testing, Access Modifiers, Developer Survey, Unit Testing Practices.**

I. Introduction

Regression test suites need to be maintained as the software applications they test evolve [57], [68]. While new tests need to be written to test new functionality, and old ones pruned for behaviors that no longer exist, existing tests can be brittle and may break due to small changes in the application code [55]. Ensuring these tests are updated and pass again can be a time-consuming process for developers [58]. Firstly, they have to understand each existing test and why it broke, so that they can then, secondly, update it so that it compiles and passes again with the new implementation. Often the effort involved is so great that developers simply decide to discard broken tests, potentially weakening the test suite in the long-term [55].

Since test maintenance is such a costly process that adds no immediate visible benefit to a software product, good software engineering practice suggests that developers should set out to write tests that, inasmuch as possible, are unlikely to need to change again in the future [57]. That is, they should focus on writing tests that are concentrated on the software's behaviors as opposed to its implementation details [41], [46], [57]. In practice, this means confining calls from unit tests to the externally visible API of the class under test; i.e., its public methods [63]. This means that developers can safely refactor application code without having to be concerned about breaking its tests and then having to fix them [57].

However, in this paper, we found that developers do not always follow this principle in practice. We collected data about public and non-public API calls made in 226,915 JUnit tests of 4,801 open-source Java projects collected from the Maven Central Repository [12]. While the tests of the majority of projects are restricted to the public API of the units they test, a not insignificant proportion of 28% involved at least one direct call to a *non-public* method. In these cases, developers write assertions against smaller parts of the internal implementation of a class, rather than its overall behavior. This raises the question as to what the original developer's intent and motivations for doing this were, given the potential future costs of maintaining those tests.

To address this, we present the first qualitative study of why developers choose to test through a unit's public API, or decide to write tests that directly test its non-public methods, at the risk of making the test suite harder to maintain in the future. We conducted a developer survey, with 73 participants and a systematic analysis of 60 threads dating from 2008 [25] to 2023 [30] related to the topic on StackOverflow. Through numerical and thematic analysis of questionnaire responses and StackOverflow posts, we identify a number of findings.

Approximately two-thirds of developers in our survey strictly test public APIs only. These developers, plus those posting on StackOverflow, believe the need or desire to test non-publics is borne out of poor code design in the first instance and suggest refactoring of production code is needed instead. Those willing to test non-public methods directly, in contrast, cite the need to ensure complex implementation details are well-tested, and the undesirable complexity involved in having to write tests solely using the unit's public API. They employ a variety of means to avoid this, including raising the visibility of methods so they become accessible to test frameworks for testing. Our study of open-source Java projects particularly supports this last finding, with a large number of non-publics being called directly from tests, including a disproportionate number designated as "package-private" — i.e., with just enough visibility that they can be invoked by JUnit tests, but not so much visibility that they actually become part of the unit's public API. Our findings provide several implications for future work, including how to provide automated support to developers to help them avoid the temptation of bypassing public APIs.

The contributions of this paper, therefore, are as follows:

1) An empirical analysis of the visibility of methods called directly from the tests of 4,801 Java projects taken from the Maven Central Repository (§V-A)
2) A numerical and thematic analysis of 73 responses to a developer survey and 60 StackOverflow threads to identify developer attitudes and approaches to testing public APIs versus testing non-public methods directly (§V-B–V-D);
3) A discussion of our findings, summarizing the current state of practice (§VI), and implications of our results that inform future research (§VII).

II. Background and Related Work

A. Encapsulation, Visibility and Access Modifiers

Object-oriented programming languages support the idea of encapsulation, which enables developers to provide interfaces so that their code can be used without the need for others to see or understand its underlying implementation [42]. The method or variables of some class A that some other class B can directly access are said to be *visible* to the class B. Developers specify these visibility rules by applying *access modifiers* defined by their programming language, which take the form of explicit keywords or other programming constructs.

The top level of visibility — i.e., all methods/variables are accessible to all other classes, including those in other applications or APIs — is referred to as the "public" level of access. While all programming languages facilitate public visibility so that developers can create interfaces for their units of code, each language tends to have its own rules pertaining to "non-public" levels of access. Many languages (e.g., C++, C#, Java, PHP and Ruby) have a "private" keyword to declare methods and variables that cannot be accessed from anywhere except the same class, together with a "protected" keyword for declaring methods and variables that can only be accessed by code residing in the same class or its subclasses. Moreover, some languages provide a level of access that is private to code in the same class *and* the logical grouping of classes or files that the class is organized into. For example, Java organizes groups of classes into "packages", with methods and variables having "package-private" visibility when they are declared without one of the public, protected or private access modifiers [4], [50]. Kotlin shares a similar concept in the form of the "internal" keyword, which developers use to declare code that is private to files organized in the same module [23]. In contrast to other languages, Go has no notion of a completely "private" level of access. Public member names are capitalized, while those that are uncapitalized are at the package-private level of access [6]. While JavaScript is to gain a private construct in its official standard [16], Python [17] does not have a built-in notion of non-public. Where mechanisms do not exist, developers may adopt conventions such as prefixing method and variable names with underscores to communicate intended privacy, though these are not enforced.

Despite the restrictions on non-public code by the languages that allow developers to set them, there is usually a means for the protection to be overridden, e.g., by using reflection APIs in languages such as Ruby [47], Java [59], and C# [34]; while in C++, special access can be granted to methods and variables of certain classes using the "friend" keyword [43].

B. Empirical Studies of Test Maintenance

Labuschagne et al. [58] studied 61 software projects and found that tests require maintenance for a variety of reasons, including invalid assumptions, dependencies on other tests, and changes in production code functionality, noting that test maintenance costs associated with these aspects could be avoided with better software development processes. Pinto et al. [68] studied the evolution of test suites for six programs, finding that of all changes to tests, 29% corresponded to modifications and 22% of these were repairs to broken tests. Of these repairs, 50% involved changes to method call sequences, i.e., additions or deletions of method calls and updates to parameters. However, neither of these works specifically investigated whether test suites required maintenance due to tests making direct calls to non-public API production code methods that were then refactored.

C. Test Smells

Test smells are poor testing practices that are observable in the source code of test suites and which hinder their maintenance [40], [75]. Various test smells have been identified relating to tests that prefer to test non-public implementation over the behavior of units. Firstly, Yang et al. identified the "Private Method Test" smell [76], which characterizes tests that access private methods in Java. Since private methods in Java are not visible outside of their class, they are not visible to JUnit tests either. This means that testers must resort to using Java Reflection to override the access restriction so that these methods can be invoked from test suites. The "Anal Probe" smell, referenced in a study by Martins et al. [60], more generally characterizes a test that "*has to use insane, illegal or otherwise unhealthy ways to perform its task*" [2] including accessing private methods and fields (using mechanisms such as Java Reflection to do, since they would not normally be visible), or that has to resort to "*extending a class to access protected fields/methods or having to put the test in a certain package to access package global fields/methods*" [2]. The "X-Ray Specs" smell [24], discussed by Garousi et al. [49], refers to tests that access or modify the internal states of units that they should not be able to access. Finally, Van Deursen et al. [75] identified the "For Testers Only" smell, where developers write production code methods specifically to support testing, and in doing so, allow testers to test a unit's implementation rather than behavior via its public API.

D. Studies into Public API vs Non-Public Method Testing

There is much literature advising that tests should strive to test behavior rather than implementation, citing the costs of test maintenance that may be incurred should production code be refactored (e.g., Bowes et al. [41], Google [57], and Microsoft [63]). However, there has been no work, hitherto, that investigates the issue of public API testing versus non-public method testing from a quantitative or qualitative perspective. Spadini et al. [74] interviewed 12 developers, in which participants mentioned not writing tests that are purely focussed on testing implementation details. However, this point arose as part of a larger study and discussion of test code quality during code review, rather than being focussed exclusively on the issue. Yang et al. [76] studied Java tests that specifically call private methods (e.g., using Java Reflection), having implemented a study of StackOverflow posts to identify the issue in the first instance. However, they do not address the issue of non-public methods in general, and do not perform qualitative studies with developers. Martins et al. [60] mined StackExchange posts to discover the challenges faced and the corrective actions taken by developers when they encounter test smells, but do not focus on public API vs non-public method testing. Several works perform developer surveys and/or mine StackOverflow, as we do, but focus on other topics; e.g., flaky tests [52], [65], continuous integration [54], regular expressions [62], and engineering test cases [37].

III. Research Questions

A unit will typically have a number of publicly accessible methods, constituting its "public API", that may be called from other units. Developers also write non-public methods into units, for implementation that is not to be freely invoked externally. Instead, these methods tend to contain useful routines that can be used in multiple places inside the bodies of other methods (public or otherwise). Unless the unit contains unreachable code, all non-public methods should be invoked at some point by a public method, meaning that all the methods of a unit should be testable through its public API.

In this paper, we focus on understanding how developers account for code with different visibility levels in unit testing. Given the associated long-term benefits of a maintainable test suite, do developers rigidly test units through their public API only? Or, do they side-step public APIs and directly test non-public methods, and if so, why?

We set out to answer the following 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 behavior-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?

IV. Methodology

A. Open-Source Study (RQ1)

In this RQ, we wrote a tool, named Viscount, to measure the number of times methods of different visibility levels are called from tests in open-source software.

We chose open-source Java projects as subjects to evaluate this, since the Java language has several different levels of access modifiers (i.e., public, protected, package-private and private; see §II-A) and is a mature language for which a large number of publicly available open-source projects exist.

Our dataset of open-source projects is derived from that of Gruber et al. [53], who collated the URLs of 38,841 Github-hosted Java projects listed in the index of the Maven Central Repository (as of 2023-04-13). The Maven Central Repository is one of the main official software repositories for Java [12], containing a diverse range of projects, from small to large. Using scripted automation, we found we could build 7,998 of these projects without error using Java 8 (the most-used version of the language in 2023 [31]) and Maven 3.9.6.

Viscount starts by statically extracting the access modifiers of each project's production methods using the Spoon framework [67]. It uses a dynamic approach to trace methods called directly from tests so that it can accurately account for calls made, for example, via Java Reflection or via a mocking library employing similar techniques. To this end, Viscount instruments each project's bytecode, using Javassist [7] to insert log statements at the points of constructor/method entry and exit (i.e., return statements and throw exceptions). It then executes the project's test suites to collect logging information and derive all direct calls made from any test. We set a budget of three hours for the execution of Viscount per each project and discarded projects for which it did not terminate within this budget. We also discarded projects that failed to produce any logs, which was usually due to a lack of test suites. Viscount further discards any tests that exercise multi-threaded code, due to the difficulty in capturing accurate traces, evident through interleaving entry and exit points of the same method in the collected logs (a frequent issue affecting other tools, e.g., [51]). Viscount then statically analyzes each project's source code to ensure that the production methods logged were indeed being called directly from tests, rather than via third-party libraries or the Java API itself (possible, for example, through Java's Serialization libraries). The first author manually inspected any cases of ambiguity that Viscount could not resolve automatically. For the interested reader, more detailed information about Viscount is available in reference [69].

Our final sample for RQ1 consisted of 226,915 tests from 4,801 Java projects. Statistics for the projects are shown in Figure 1, indicating a range of projects that are large to small in size. The mean lines of code for the sample was 5841.02, the mean number of tests was 47, and the mean number of

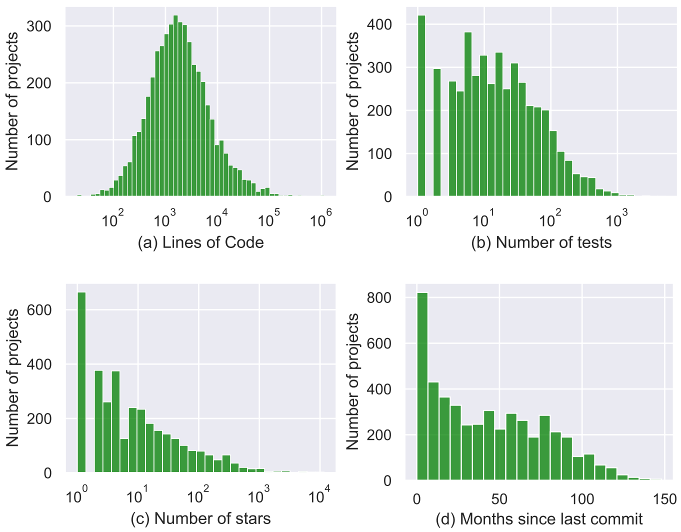


Fig. 1. Statistics of the Maven open-source projects studied in RQ1.

GitHub stars was 54.9. The mean time elapsed since the last commit at the point of collection was 43 months.

B. Developer Survey (RQs 2–4).

We created a questionnaire featuring a mixture of multiple-choice and open-ended questions. We trialed our survey with professionals and academic colleagues, and undertook several trials with a small number of PhD students, refining its design following each iteration.

Our final survey included questions from four different angles. *Demographic* questions included the number of years of experience in development/testing the developer had and their main programming language. Questions related to *stance* explored the opinions the developer held on testing through public APIs only vs testing non-public methods directly, which we used to answer RQ2. Questions related to *rationale* focussed on the reasons behind why developers held the opinions they did, which we used to answer RQ3. Finally, we included a series of *practical* questions, aimed at finding out about the approaches developers took in everyday development practice. We used the responses to these questions to answer RQ4. While space does not permit us to reproduce the entire questionnaire as part of this paper, we have made it available in full as part of our replication package [32]. We provided participants with a background information sheet containing a language-agnostic definition for the notion of "non-public" methods, including details of access modifiers and common conventions to denote visibility in several languages, from C++, C#, Java, Go, Kotlin, and Rust, to dynamically typed languages such as JavaScript, Python, and Ruby, and those extended with enforced type annotations, e.g., TypeScript.

We kept the questionnaire open for three weeks using Google Forms, and distributed it via LinkedIn, X (formerly known as Twitter), regional technology forums, and personal industrial contacts — asking them to share it with their colleagues. To mitigate the risk of bot infiltration, we required each participant to sign in to their Google account, which also effectively prevented multiple submissions from the same person. We did not collect their account information or track their identity.

We received 73 responses in total. In terms of development experience, 10-15 years was the median response (mode 15+ years); for unit testing experience, the median and mode were 5–10 years. We asked developers what their main programming language was. The responses were Java (25), Python (15), TypeScript (9), C# (7), PHP (6), Kotlin (4), C++ (2), JavaScript (2), Ruby (2), and Scala (1). We analyzed the frequency of choices for questions with fixed responses, using Kendall's tau-b correlation coefficient to assess the strength of the relationship between different answers [9]. We applied inductive thematic analysis [44] to answers to our open-ended questions, assigning codes to each response that summarized its key concepts. Given that free-text responses to open-ended questions frequently contain multiple components, we divided the responses into distinct elements if they are using conjunctions such as *"and"*, *"or"*, and *"but"*. We conducted this analysis collaboratively to keep our coding as consistent as possible and to minimize any individual biases. We then grouped similar codes into a set of overarching themes. As part of our results (§V) we discuss the major themes to which we attributed more than two comments.

C. StackOverflow Analysis (RQs 2–4)

To further capture a range of different perspectives and develop a robust understanding of developers' perspectives towards testing non-publics, we employed triangulation [66] to complement the outcomes of our developer questionnaire with an analysis of online discussions on the question and answer website for software developers, StackOverflow [20].

We formulated the following four StackOverflow search queries: "*test non-public method*", "*test private method*", "*test protected method*", and "*test package-private method*". The intuition behind these queries was to retrieve threads discussing the testing of non-public methods in the general sense, as well as threads discussing the testing of specific types of non-public methods, hence the inclusion of the "private", "protected", and "package-private" keywords as search terms. To narrow down the scope of the search results, we appended the "*[unit-testing]*" tag to each query term. We purposefully refrained from tightening our search queries any further so as not to miss relevant threads in our search results. In the same spirit, we did not explicitly mention any programming language in our queries. The keywords "private" and "protected" are included because they signify different levels of visibility in different programming languages, e.g., C#, C++, Java, Kotlin, and PHP. While "package-private" is Java-specific, including it allows us to triangulate these results with those from our empirical study of Java projects for RQ1 (§IV-A).

We executed our search queries on November 16, 2023, and collected the top 25 threads for each query, sorted by relevance. We removed duplicates across queries and also removed insubstantial threads; i.e., threads where a genuine discussion did not materialize, resulting in fewer than two responses. This left us with 60 threads in total, including original posts from between 2008–2023.

All authors manually analyzed these 60 threads in collaborative in-person sessions. Firstly, we categorized each thread as either “Debate” or “Practical” depending on the style of the question. “Practical” threads, used to answer RQ4, mainly pertained to posters asking specific questions relating to testing non-public methods in the context of a specific programming language or testing framework. “Debate” threads, used to answer RQs 2–3 were more general and/or conceptual in nature, where the poster’s intent was mainly to ask the community’s opinions on testing non-publics, occasionally in the context of the particular scenario presented in the question. The categorizations we made were mostly obvious from the text of the thread, with some posters keen for their technical question not to devolve into a discussion about the pros and cons of directly testing non-publics (e.g., [27], *“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, we only focus on the top answer — i.e., the one with the highest number of up-votes, whereas for “Debate” threads, we extended this to the top three answers to take into account a variety of opinions and a more complete understanding of the discussion. We did not extend our analysis to further posts to both control the quality and the number of posts we had to manually examine. In the case of “debate” threads, some posts did not contain as many as three answers, in which case we analyzed just the one or two responses that it did have. We then applied inductive thematic analysis to the questions and answers in these threads, as we did for free-text responses in the developer questionnaire.

D. Threats to Validity

We discuss threats to validity and our mitigating strategies.

1) Questionnaire Participants: The sampling of participants may influence the *external validity* of our study, as with most questionnaires. Since we have no means to quantify the entire population of software developers, we used the *non-probability purposive sampling* method to recruit participants most likely to provide useful responses for our study [39]; we distributed our questionnaire widely via social media and professional networks to reach a large number of participants.

2) Questionnaire Evaluation: The *internal validity* of our study relies on the design of our questionnaire. Following empirical software engineering guidance [56], prior to releasing our questionnaire, we conducted a pilot study with a total of four iterations and ten respondents from different backgrounds (CS PhD students and professional developers). This pilot study was useful to mitigate potential researcher biases and to refine the questionnaire’s format, length, and clarity.

3) Thematic Analyses: For both the free-text responses in the questionnaire (§IV-B) and StackOverflow threads (§IV-C), we opted for a *collaborative* thematic analysis between three authors to ensure our interpretations are valid (*construct validity*) and mitigate any individual researcher bias.

4) Programming Language: In our open-source study, we only considered Java projects hosted on GitHub which use Maven as the build automation tool — a potential threat to *external validity*. While Java remains one of the most popular programming languages [29] and Maven is the main build automation tool for Java [31], further replications are needed to establish how our results would generalize to other languages. Furthermore, we only used projects that were we able to build with Java 8, the most popular version of the language in 2023 [29]. While we believe this to be a good representation of Java projects, further work is needed to establish whether our results generalize to more recent versions of the language.

5) Test Instrumentation: Gathering 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 the risk of implementation errors affecting the results which may have an impact on *internal validity*. We mitigate this by careful reviewing, debugging and testing of the implementation and thorough discussions of the results. We also acknowledge certain limitations in our current instrumentation: it cannot handle multi-threading tests (similar to java-callgraph [51]) or some obscure method invocation mechanisms used in a small number of projects. These features are orthogonal to the use of access modifiers, therefore we do not expect them to affect our analysis. We mitigate this further by using a sufficiently large number of projects and tests.

6) Triangulation: The nature of the problem under investigation in this paper led us to use three complementary methods: a questionnaire, an analysis of online discussions, and a quantitative experiment on open-source code. This *triangulation* allows us to mitigate threats to validity across research methods, e.g., self-reporting biases, to develop a comprehensive understanding of the problem and provide more confident answers to our research questions [66].

7) Reproducibility and Replicability: We make our research materials available online to foster reproducibility and replicability [32], including our questionnaire, its responses, analysis spreadsheets, and experiment scripts for open-source projects.

V. Results

A. RQ1: Open-Source Testing

Figure 2 breaks down the proportion of open-source Java projects in terms of how often unit tests directly call production code methods with different access modifiers.

Out of 4,801 projects, 3,455 (72%) have tests which call public methods exclusively, meaning the remaining 1,346 projects (28%) have tests which make at least one call to a non-public method. Among these, projects with tests directly calling package-private methods were most frequent (828 projects), followed by those with direct calls to protected (698 projects) and private (42 projects) methods. The tests in 64 projects (1.3%) call non-public methods exclusively, of which 37, 23, and 1 projects’ tests make *exclusive* calls to package-private, protected, and private methods respectively

TABLE I
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	116	132,701	0.1%

and 3 projects have calls to both package-private and protected methods. The one project calling only private methods contains only one test [14], which exclusively tests a private method directly, using Java Reflection [13] to achieve this. In terms of tests directly calling private methods, as shown by the data in Table II, 68 used Java Reflection to override visibility restrictions to make the method accessible for invocation. The remaining 48 used third-party utilities to achieve the same effect, including mechanisms in mocking frameworks such as PowerMock Whitebox [15] and EasyMock ReflectionUtils [5].

While the Venn diagram gives a useful overview of method testing on a per-project basis, varying project sizes could give a false impression of overall frequencies. We therefore tracked the number of production methods being called by access modifier type across all projects, as shown in Table I. We counted a production method as executed if directly called at least once, by at least one test. The vast majority of direct calls tests make are to public methods. The number of calls to package-private is greater than that of protected and private, and overall the proportion of package-private directly-called methods is greater than that for public.

Since package-private methods may be accessed by JUnit tests without breaking the encapsulation of the method outside of its package[1], this might suggest that developers deliberately make methods package-private to test them, as opposed to leaving them private. However, we do not know the original intentions of the developers of these projects, so we return to this issue as part of our developer survey and StackOverflow analysis in the next research questions.

Conclusion (RQ1: Open-Source Testing). Open-source developers call both public and non-public methods directly in tests. 72% of the projects studied have tests that call public methods only, meaning the remaining 28% contain at least one call to a non-public method. The raw number of non-public methods invoked from tests is small compared to the number of public methods, but this is due to the number of public methods being an order of magnitude higher in the first place. Proportionally, package-private methods are the most frequently called type of method by access modifier.

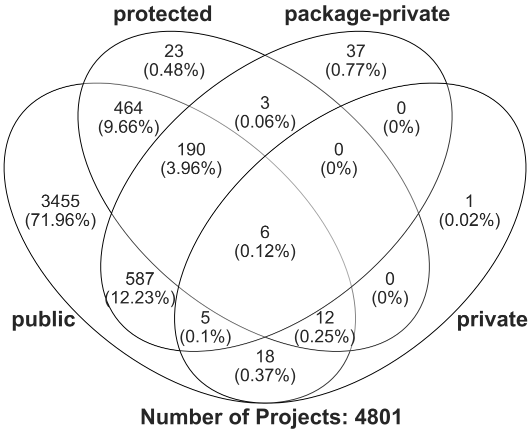


Fig. 2. Venn diagram of open-source Java projects, grouped by visibility of production code methods called directly from their tests.

TABLE II
Mechanisms used to directly invoke private methods in tests, by numbers of tests ("# Tests") and projects ("# Projects"), in our dataset of Java projects.

Framework	# Tests	# Projects
Java Reflection [13]	68	31
JMockit Deencapsulation [8]	14	1
PowerMock Whitebox [15]	13	5
Apache Commons Lang3 MethodUtils [3]	12	4
Manifold Jailbreak [10]	6	1
EasyMock ReflectionUtils [5]	1	1
Spring Framework ReflectionTestUtils [19]	1	1
tvd12 test-util MethodInvoker [22]	1	1

B. RQ2: Stance

We asked developers if 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 3 summarizes their responses. Almost two-thirds (64%) of participants either *agreed* or *strongly agreed*; 30% *disagreed* or *strongly disagreed*, while the remaining 6% were unsure.

We also asked developers *"How often do you write tests that directly invoke non-public methods?"* "Never" and "Rarely" were selected most by participants, as shown in Figure 4. This figure also shows that the frequencies of answers ("Always", "Often", "Sometimes", "Rarely", "Never") are consistent with the different levels of agreement observed earlier in Figure 3, with Kendall's tau-b indicating a significant strong association between these sets of answers (τ_b = 0.463, $p < 0.01$).

We also analyzed the 18 StackOverflow threads we classed as being "Debate"-based (§IV-C). We categorized each of the top three responses in each thread as being either in favor of testing public APIs only (19; ~40%), in favor of testing non-public methods directly (17; 35%), or neutral in stance (12; ~25%). Similar to the developer survey, we observe more advocacy for testing public APIs only, albeit by a smaller margin. These posts were also more likely to receive more upvotes by users of the site, and consequently be ranked first in the list of answers to the original poster's question.

[1] Although Maven organizes 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 test despite being in different directories on the file system [11], [21].

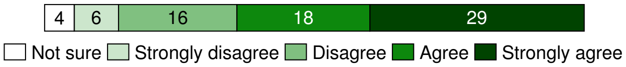


Fig. 3. Developer responses to the question *“To what extent do you agree with the following statement? ‘In general, developers should write unit tests that only invoke public methods, avoiding direct calls to non-public methods.’”*

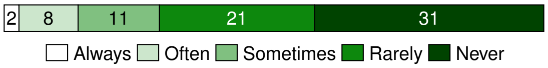


Fig. 4. Developer responses to the question: *“How often do you write tests that directly invoke non-public methods?”*

Conclusion (RQ2: Stance). Among our developer survey participants, almost two-thirds prefer to test public APIs only, while the remaining third will test non-public methods directly. We observed a similar pattern in the answers in StackOverflow “Debate” threads, where responses arguing to test only via public APIs were more often the top answer than those in favor of testing non-publics directly.

C. RQ3: Rationale

To understand developer stances more deeply, let us cluster participants into two groups, namely “agree” and “disagree”, based on their answers to the statement *“In general, developers should write unit tests that only invoke public methods, avoiding direct calls to non-public methods”* (Figure 3). The “agree” group comprises participants *agreeing* or *strongly agreeing* with the statement — i.e., those who prefer to test through public methods only — while the “disagree” group includes those who *disagreed* or *strongly disagreed.*

We asked developers how much they valued eight different qualities of test suites, listed in Table IV. The table shows some interesting differences between the two groups. Participants in the “agree” group most frequently marked all the possibilities as “Very Important”, except code coverage which was just “Important”. This was in contrast to the “disagree” group, who marked code coverage as “Very Important”. This seems to suggest that testers in the “disagree” group place more emphasis on exercising implementation — to obtain higher coverage, one might need to directly invoke a non-public from a test to execute a statement that is hard to cover otherwise; whereas members of the public-only (“agree”) group are more concerned with testing behavior. Some participants made this point explicitly in free-text responses in our survey. Participant P_{49} commented, for example, *“It’s important that unit tests test the behaviour and not the implementation”*, while P_{72} said *“I think that testing public rather than non-public methods leans more towards testing behaviour rather than implementation which is usually preferable”*.

We also asked developers the extent to which they agreed with the statement “*Testing non-public methods leads to more tests failing spuriously when modifications are made to those methods.*” We found those in the “agree” group also tended to agree or strongly agree with this statement. Kendall’s tau-b test indicated a strong and significant association ($\tau_b = 0.53$, $p < 0.01$). There was no such correlation for the “disagree” group. Public-only testers are concerned therefore about tests breaking during an internal refactoring if they are highly coupled to non-public methods via direct invocations, further evidencing that this group prefer to test behavior rather than implementation. This was backed up by a handful of comments made by these participants in free-text responses in our survey. P_{65} said, for example, *“Writing tests that are coupled to the private implementation of a class hurts the ability to refactor in the future”*. The “agree” group also pointed to likely process or design problems with the code, e.g., P_{21} said *“If you have to do it, that means you probably have a code smell. . . ”*; P_{43} agreed with this, stating *“. . . that may be a sign that there is a design/decomposition problem”*, while P_{58} commented it could be *“a sign of a leaky abstraction”*. P_{25} said 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”*.

TABLE III

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 on 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

TABLE IV

Developer responses to the question *“To what extent do you value the following aspects when writing unit tests?”*, where participants 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 behavior 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

In contrast, developers who disagree with the public-only approach cited the complexity of logic in non-public methods, and the need to thoroughly test them, e.g., 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 highlighted that testing non-publics directly is often less verbose and potentially 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, we performed a thematic analysis on the original poster questions that started the 18 “debate” threads we retrieved from StackOverflow. These gave further 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 who — in contrast to developer survey respondents who said that the need to directly test non-publics was a code design problem — felt they would have to compromise code design to *avoid* direct testing of non-public methods. One poster wrote *"If I wrote my class this way, I could have unit tests [...] but I feel like this pattern is not correct. Is there a better way?"* [28].

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

– Complex logic: Echoing participants in our developer survey, this theme characterized posters who wanted to test non-publics directly due to their inherent complexity.

– Complexity of testing through the public interface: Posters described a further reason that the developer survey did not surface as an explicit theme: they felt 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"* [26]; or just too complex in its own right: *"I want to test logic used in synchronous threads without having to worry about threading problems"* [26].

Conclusion (RQ3: Rationale). Public API testers believe testing *behavior* is important, while developers who test non-public methods directly are more concerned with the correctness of *implementation*. Public API testers believe the urge to directly test non-publics is down to code design problems that will lead to maintainability and refactoring difficulties in the future. Implementation-driven testers are concerned about complex parts of non-public code going untested and about the complexity of test code, since testing non-public methods through the public API is a more complex task that may require a lot of shared setup between tests.

D. RQ4: Practice

1) Access Modifiers in Programming Languages: We asked developers *"If your main language does not have access modifiers, do you follow any conventions to denote visibility of methods and instance variables?"* 54 participants responded that their programming language has access modifiers, 19 said theirs did not, and 16 of those 19 said they followed conventions to denote visibility instead. We asked, as a free-text follow-up question to those who said they followed conventions, which ones they used. Our thematic analysis of responses revealed the following major themes:

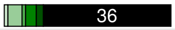
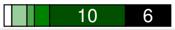
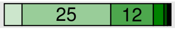
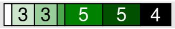
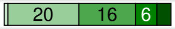
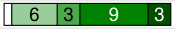
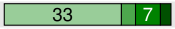
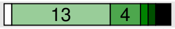
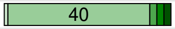
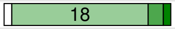
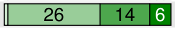
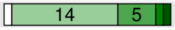
– Underscore prefixes: Almost all respondents said they use underscores to indicate non-public methods.

– Project-specific: Participants use different conventions depending on the project.

– Visibility is irrelevant for understanding: Some participants questioned the need for such mechanisms at all, e.g., P_{50}: *"Visibility is usually irrelevant for understanding the code"*.

TABLE V

Responses to the question: *"How do you go about testing non-public methods?"*, where 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	 36	 10 6
By directly invoking	 25 12	 3 3 5 5 4
Using reflection / mocks	 20 16 6	 6 3 9 3
Adding test code in production	 33 7	 13 4
Temporary switch to public	 40	 18
Permanent change to public	 26 14 6	 14 5

– Conventions increase maintenance: Participants observed that encoding visibility in the name of methods (e.g., by using underscores) increases maintenance/refactoring effort — P_{50}: *"encoding it in the name increases the work required to change visibility."*

2) Approaches to Testing Non-Public Methods: We asked developers *"How do you go about testing non-public methods?"*, and analyzed the responses in the two groups defined in the last RQ. Unsurprisingly, those in the "agree" group (i.e., those with the stance of testing through public APIs only) respond most frequently to "mostly" for "via public methods only", and "never" to the other possibilities (Table V). The "disagree" group most frequently only ranked "via public methods only" as "often" and "sometimes" for testing by direct invocation and using reflection or mocks.

We asked developers about any other means they use to test non-public methods. In our thematic analysis of their free-text responses, the major themes, in order of prevalence, were:

– Refactor: Developers think that directly testing non-publics is a *code smell* related to its design and that it needs refactoring (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: Developers elevate the visibility of a method so that it is accessible and therefore callable directly from a unit test. P_{39} said *"Mostly what I do (in Java) is have the method be 'package private' (the default visibility)"*), optionally using test framework annotations to mark the reason for the visibility (P_{39}: *"...and use an annotation (@VisibleForTesting) to mark the reason for the visibility"*).

– Via public methods: Developers underlined that non-publics should be tested indirectly by calling public methods.

We also asked developers *"Are there any guidelines, best practices, or specific rules you follow when testing non-public methods?"*, from which we identified two new themes not present in responses to the previous question:

– Apply good software engineering process: In this theme, developers additionally pointed out that if a good process or design had been followed, one shouldn't need to directly test non-publics. These sentiments are similar to the previously identified "refactor" theme.

– Test non-public methods directly if necessary: This theme captured developers who believe non-publics need to be tested directly when they are complex.

We asked participants if they took a different approach for different levels of visibility for non-public methods, with the majority (59) answering “No”, and the remaining 14 replying “Yes”. In a follow-up free-text question we gave participants the opportunity to explain further. From these responses, we identified the following themes:

– Test what the tests have access to: Respondents tested non-publics that were visible to tests. For example, with 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)”*; 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 further underlined developers should test 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 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, we analyzed StackOverflow answers in “Practical” threads (§IV-C) where posters specifically ask how to test a non-public method in a given scenario. Our 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-publics, 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-publics normally inaccessible for testing, including using reflection (e.g., in Java) to override access controls, and C++’s “friend” construct. We also identified themes that we had previously seen in responses to our 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-publics directly.

– Elevate access modifier: Posters in favor of side-stepping the public API advising raising the access level of the non-public so that it is visible for testing.

3) Tooling: Finally, we asked developers if they would like to see features or improvements to unit testing frameworks to facilitate testing non-public methods, from which we identified the following themes:

– None: Respondents did not believe any new features were necessary. This was the most prevalent response.

– Less support in current tools: Respondents believed current test frameworks made it too convenient to test non-publics directly (using the mechanisms featured in Table II, for example), and should instead do more to discourage or even prevent the practice.

– Support for inaccessible methods: Respondents wanted more test framework support for testing inaccessible methods (e.g., private methods in Java) without resorting to tricks like reflection to override access controls.

Conclusion (RQ4: Practice). The majority of developers whose languages do not have visibility modifiers adopt conventions, such as underscores in method names, to denote they are intended to be non-public. Testers preferring to test through a unit’s public API only see the desire to test non-public methods as indicative of a code smell indicating poor process or the need to refactor production code. Developers who do test non-public methods use different “back-door” means, one of the most popular being to elevate the method’s visibility to make it accessible to the tests.

VI. Discussion

1) Further Analyses: We investigate possible correlations between developer experience and our survey results, and between the size of a project and the number of direct calls to non-publics in tests studied in RQ1.

As part of our demographic survey questions, we asked developers how many years of experience of development and testing they had. Figure 5 shows a stacked bar chart of *testing experience*, where the bars are split according to whether they are in the group that “agree” with testing through public APIs only or the “disagree” group from RQ3, or were not sure — based on their answers shown to the question in Figure 3. The chart shows an increase in the proportion of those in the “agree” group as the number of years of testing experience increases. Kendall’s tau-b showed a moderate positive association between these two traits, which is significant at $\alpha = 0.05$ ($\tau_b = 0.240$, $p = 0.014$). Overall this implies that the more experience a tester has, the more likely they are to be opposed to directly testing non-publics, and suggests that more organizational guidance is needed for more junior members if development teams are to avoid side-stepping the public API and directly testing non-public methods. We found less evidence that *development experience* has a similar relationship; Kendall’s tau-b test shows only a weak association that is not significant ($\tau_b = 0.135$, $p = 0.17$).

We also looked into whether the size of the project was somehow correlated with specific practices regarding testing of

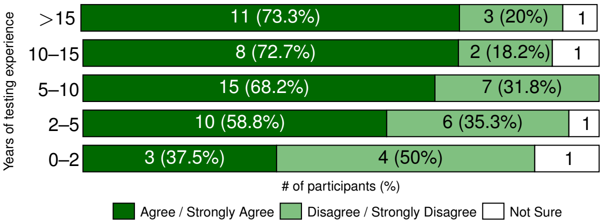


Fig. 5. Participants’ years of testing experience, grouped by stance on testing through public methods only (distribution shown in Figure 3).

non-publics. It would be conceivable that directly testing non-publics is less prevalent in larger projects due to having larger development teams with potentially more development/testing experience and more rigorously defined software engineering processes. However, Kendall's tau-b showed only a weak association between the number of tests and the proportion of non-publics directly called from them ($\tau_b = 0.179$, $p < 0.01$). While the tests of both projects considered big and small relative to our dataset tended to focus on exercising public methods (in accordance with the data presented in Table I), they also tended to include direct calls to non-public methods.

2) Summary of Findings: Our results show that the majority of developers agree with testing solely through public APIs, but not overwhelmingly so, with a third disagreeing and in favor of testing through non-public methods — albeit depending on the circumstances (RQ2). In summary, developers in the "agree" group, oppose directly testing non-publics, commenting that it will make code harder to refactor and maintain in the future, and that the temptation to test non-publics stems from poor code design, suggesting refactoring as the solution instead (RQ3, RQ4). However, this needs to be done with care, since it could lead to repetition in tests that need to keep calling the same public methods — another topic of debate in the literature suggested to be a code smell in its own right [40]. Some literature suggests such tests themselves would need refactoring [75] in keeping with the DRY principle ("Don't Repeat Yourself"), with others preferring to keep tests "DAMP" ("Descriptive And Meaningful Phrases"); e.g., [57]. Another issue in refactoring might be the temptation to make parts of the implementation visible solely for the purposes of testing, also considered a test smell (referred to as "For Testers Only" [35], [40], [75], discussed in §II-C).

Some of the drawbacks of the alternatives to testing non-publics directly are exactly those expressed by participants in our survey and those on StackOverflow who are in favor of testing non-publics, often depending on the circumstances. Others further cited the complexity of the code in the non-public method itself — perhaps with code coverage of tests in mind — and the difficulty of exercising it indirectly, citing the resultant complexity of the tests and/or the problem of isolating the aspects of the code they want to exercise (RQ3). Participants in our survey and posters on StackOverflow proposed a variety of remedies, some of the most prevalent being *using language-specific mechanisms* to bypass access modifiers or *elevating access modifiers* — a change equivalent to making internals more visible for testing. Our results to RQ1 suggest that developers do make methods package-private in Java for the purposes of testing, thereby not rigidly testing exclusively through the public API only.

Finally, the majority of developers responding to our questionnaire did not believe further tooling support is needed to address the problem (RQ4), but perhaps were unaware of potential future research techniques that could help convert direct calls to non-publics into those that solely exercise the public API, or be used to manage other perceived negatives of testing non-publics over the public API, such as repetition in tests. We discuss this implication as part of our ideas for future work in the next section.

VII. Implications and Future Work

A. Software Testing Education

Our survey results indicated that different attitudes were prevalent based on level of experience. Instead of learning the "hard way", we propose that educators are made aware of some of these opinions and seek to ensure that they emphasize the best software engineering practices in their teaching.

1) Behavioral-Driven Testing: Our survey results show that less experienced developers and testers tend to be in favor of testing non-publics directly, suggesting that better education and materials are needed around behavioral-driven approaches to development and testing. This would give developers clearer incentives to avoid the practice by studying the longer-term downsides, such as less maintainable test suites.

2) Attitudes to Coverage: While there is healthy skepticism amongst developers regarding coverage in general (e.g., [18]), coverage was correlated with favoring testing non-publics directly in our study. Increasing coverage may therefore be a motive behind testing implementation as opposed to behavior (as also observed by Bowes et al. [41]). Ideally, coverage should be used by developers to identify untested behaviors instead of individual lines that tests execute in their production code. As such, testers should be less concerned with coverage as a quality metric of their test suites, as tests that are tightly coupled to implementation are not of high quality since they will be more costly to maintain in the future.

3) Identifying and Discouraging Testing Anti-Patterns: Our findings indicate ways in which developers sometimes take shortcuts to test non-public implementation or to make it accessible, including elevating access levels of methods, using language-specific mechanisms like the "friend" keyword in C++, or other means to access code normally inaccessible to tests like using Reflection in Java or using functionality in mocking libraries to achieve the same effect. These practices should be identified in education materials as anti-patterns to be avoided in software tests.

B. Automated Techniques

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

1) Automated Support To Replace Direct Non-Public Calls: In our survey, some developers described the complexity or laborious nature of testing exclusively through public interfaces. Future research therefore could concentrate on techniques that refactor existing developer tests to remove direct calls to non-public methods and replace them with call sequences that rely on public methods only. This could be done with the help of existing test generation tools, such as EvoSuite [48]. For example, once a non-public call is found, a search-based technique [61] could be used to find the equivalent set of public calls that lead to the non-public method being invoked in the same way. However, the resulting generated tests might suffer from readability issues [33], [45], [70]. For this reason, applying a large language model to assist may therefore also prove fruitful, in a similar style to recent work by Alshahwan et al. [36] and Yaraghi et al. [77].

2) Refactoring Production Code Based on Problematic Tests: In our survey, several developers noted that the desire to side-step the public API and test a non-public method directly is because the production code is poorly designed in the first place. This means that tests that call non-public methods could indicate production code that needs refactoring. These refactorings would of course be more intricate than simply making non-public methods visible to tests, as that would defeat the purpose. While there has been work on refactoring *tests* to remove test smells (e.g., [73]), to our knowledge, poorly written tests have not been used as the basis of work in automated refactoring *production code* before. Calls to non-public methods in tests could be one direction in which to drive automated refactoring tools towards better code design so that tests only need to invoke the public API to better effect.

3) Improved Automated Test Smell Detection Tools: Existing test smell detection tools focus on private methods only [76], however direct calls to all non-public methods couple tests to implementation that will make them hard to maintain in the future. We therefore argue that these tools need to be extended to detect calls to other non-public methods, e.g., package-private. Furthermore, detecting when non-public methods are called by tests is a surprisingly non-trivial task. Static methods miss cases when mocking libraries are used to bypass visibility restrictions to call private methods. In our study, we resorted to dynamic methods, and even then still, we found cases that were hard to analyze due to the tests calling threaded code or starting threads themselves (both indicative of further test smells). Furthermore, callbacks are possible through external APIs. We checked these cases by hand or excluded them entirely from our study. Future work needs to focus in these areas.

4) Automatic Test Generation and Non-Public Methods: The developers in our survey or those using StackOverflow did not comment on automatically generated test suites (such as those produced by EvoSuite [48], Randoop [64], or AgitarOne [1]). These tools retain the option to call non-publics with the goal of increasing coverage [38], [72], meaning tests generated by these tools will be hard to maintain. 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 [71].

C. Developer Support

The answers to our survey imply different ways in which existing tools may be improved to assist developers in ensuring their tests avoid making calls to non-publics directly.

1) Stricter Test Frameworks / Build Tools: As already mentioned, developers often employ shortcuts in their tests to exercise non-public methods, but these practices could be stopped entirely by stricter test frameworks. JUnit, for example, encouraged by build tools like Maven and Gradle, organizes unit tests so that they exist in the same package as the classes they test, making certain non-public methods, like those declared package-private visible to tests. We found these types of calls particularly to be prevalent in our study of open-source Java projects. Placing test suites *outside* of the package of classes they are designed to test, however, would render these direct calls impossible by virtue of the visibility rules of the Java language. This suggestion comes directly from developer responses to our survey in RQ4 (§V-D3). Similarly, mocking tools should deprecate or remove functionality allowing developers to access private methods through Java Reflection.

2) IDE support: Integrated Development Environments (IDEs) could build in support for some of the suggestions for future work on automated techniques outlined above. Whenever a developer is tempted to call a non-public method directly, the IDE could auto-suggest a sequence of calls that call the public interface of the class instead. Similarly, these tools could implement bad smell detectors to give developers more useful indicators of test suite quality rather than just code-based metrics like coverage.

VIII. Conclusions

Test suites that directly test internal implementation require maintenance when that implementation changes — a cost that could be avoided if testers stuck to testing public APIs only. In this paper, through a developer survey and an analysis of relevant StackOverflow threads, we presented 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. We support our qualitative study with a quantitative analysis of 4,801 open-source Java projects, which we obtained from the Maven Central Repository. Among several findings, our work revealed that while the majority of testers prefer to test the public API of a unit only, a significant proportion are willing to side-step it and directly test a non-public method, particularly if it is complex, including raising its visibility level if needed. While many developers suggested that such code needs to be refactored so that it can be tested via its public methods, others employ various means to access non-public methods directly if needed, including making these methods visible to tests. This behavior was supported by the results of our open-source Java code study, where a seemingly disproportionate number of methods directly tested were declared “package-private” — i.e., not fully private but with just “enough” visibility to make them accessible to unit tests. We proposed several lines of research and ways to support developers in reducing the need to test non-publics directly, with the aim of improving test maintainability in the future.

Acknowledgements

Muhammad Firhard Roslan receives PhD funding from the Majlis Amanah Rakyat (MARA). Phil McMinn is supported, in part, by the EPSRC grant “Test FLARE” (EP/X024539/1).

REFERENCES

[1] Agitar One. http://www.agitar.com/solutions/products/automated_junit_generation.html. Accessed: 4/2024.

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

[3] Apache commons lang3 — Class MethodUtils. https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/reflect/MethodUtils.html. Accessed: 7/2024.

[4] Controlling access to members of a class, the Java tutorials. https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html. Accessed: 4/2023.

[5] EasyMock — Class ReflectionUtils. https://easymock.org/api/org/easymock/internal/ReflectionUtils.html. Accessed: 7/2024.

[6] Exported identifiers — Go: The Go programming language. https://go.dev/ref/spec#Exported_identifiers. Accessed: 4/2024.

[7] Javassist. https://www.javassist.org. Accessed: 4/2024.

[8] JMockit — Class Deencapsulation. https://javadoc.io/doc/com.googlecode.jmockit/jmockit/latest/mockit/Deencapsulation.html. Accessed: 7/2024.

[9] Kendall's Tau — Simple Introduction. https://www.spss-tutorials.com/kendalls-tau/. Accessed: 7/2024.

[10] Manifold Systems — Annotation Type Jailbreak. https://javadoc.io/static/systems.manifold/manifold-ext-rt/2020.1.41/manifold/ext/rt/api/Jailbreak.html. Accessed: 7/2024.

[11] Maven — Introduction to the standard directory layout. https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout. Accessed: 7/2024.

[12] Maven Central Repository. https://repo.maven.apache.org/maven2/. Accessed: 4/2024.

[13] Package java.lang.reflect. https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html. Accessed: 7/2024.

[14] password-generator GitHub project. https://github.com/javadev/password-generator. Accessed: 4/2024.

[15] Powermock — Class Whitebox. https://www.javadoc.io/doc/org.powermock/powermock-reflect/1.6.4/org/powermock/reflect/Whitebox.html. Accessed: 7/2024.

[16] Private properties — the modern JavaScript tutorial. https://javascript.info/private-protected-properties-methods. Accessed: 4/2024.

[17] The Python Language Reference — Private name mangling. https://docs.python.org/3/reference/expressions.html#private-name-mangling. Accessed: 4/2024.

[18] 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. Accessed: 4/2024.

[19] Spring Framework — Class ReflectionTestUtils. https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/util/ReflectionTestUtils.html. Accessed: 7/2024.

[20] Stack Overflow. https://stackoverflow.com. Accessed: 4/2024.

[21] Testing with Maven — Organizing unit and integration tests. https://dev.to/rodnan-sol/testing-with-maven-organizing-unit-and-integration-tests-35oh. Accessed: 7/2024.

[22] tvd12 test-util repository. https://github.com/tvd12/test-util. Accessed: 7/2024.

[23] Visibility modifiers — Kotlin programming language. https://kotlinlang.org/docs/visibility-modifiers.html. Accessed: 4/2024.

[24] 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. Accessed: 4/2024.

[25] Stack Overflow: How do I test a class that has private methods, fields or inner classes? https://stackoverflow.com/questions/34571/, 2008. Accessed: 4/2024.

[26] Stack Overflow: Making a private method public to unit test it...good idea? https://stackoverflow.com/questions/7075938/, 2011. Accessed: 4/2024.

[27] Stack Overflow: Unit testing private method — Objective-C. https://stackoverflow.com/questions/18354788/, 2013. Accessed: 4/2024.

[28] Stack Overflow: Should a concrete class that implements an interface have extra public methods for testing? https://stackoverflow.com/questions/57632038/, 2019. Accessed: 4/2024.

[29] Stack Overflow: Developer survey results. https://survey.stackoverflow.co/2023, 2023. Accessed: 4/2024.

[30] Stack Overflow: Unit testing a overridden protected method from a class that does not have default constructors. https://stackoverflow.com/questions/76868236/, 2023. Accessed: 4/2024.

[31] The state of developer ecosystem 2023. https://www.jetbrains.com/lp/devecosystem-2023/java/, 2023. Accessed: 4/2024.

[32] Replication package. https://github.com/unittesting-nonpublic/private-keep-out_replication-package, 2024.

[33] 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.

[34] Joseph Albahari. Reflection and Metadata. In *C# 12 in a Nutshell*, chapter 18. O'Reilly Media, 2023.

[35] 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.

[36] 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 *International Symposium on the Foundations of Software Engineering (FSE)*, 2024.

[37] Maurício Aniche, Christoph Treude, and Andy Zaidman. How developers engineer test cases: An observational study. *IEEE Transactions on Software Engineering*, 48:4925–4946, 2022.

[38] 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.

[39] Sebastian Baltes and Paul Ralph. Sampling in software engineering research: a critical review and guidelines. *Empirical Software Engineering*, 27, 2022.

[40] 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.

[41] 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.

[42] Timothy Budd. *Introduction to object-oriented programming*. Addison-Wesley, 2008.

[43] Steve Counsell and Peter Newson. Use of friends in C++ software: An empirical investigation. *Journal of Systems and Software*, 53:15–21, 2000.

[44] Daniela S. Cruzes and Tore Dyba. Recommended steps for thematic synthesis in software engineering. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 275–284, 2011.

[45] 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.

[46] Michael Feathers. *Working effectively with legacy code*. Prentice Hall Professional, 2004.

[47] David Flanagan and Yukihiro Matsumoto. Reflection and Metaprogramming. In *The Ruby Programming Language: Everything You Need to Know*, chapter 8. O'Reilly Media, 2008.

[48] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011.

[49] 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.

[50] James Gosling. *The Java language specification*. Addison Wesley, 2000.

[51] Georgios Gousios. Java-callgraph: Programs for producing static and dynamic (runtime) call graphs for Java programs. https://github.com/gousiosg/java-callgraph. Accessed: 4/2024.

[52] Martin Gruber and Gordon Fraser. A survey on how test flakiness affects developers and what support they need to address it. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 82–92, 2022.

[53] 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.

[54] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *International Conference on Automated Software Engineering (ASE)*, 2016.

[55] Javaria Imtiaz, Salman Sherin, Muhammad Uzair Khan, and Muhammad Zohaib Iqbal. A systematic literature review of test breakage prevention and repair techniques. *Information and Software Technology*, 113:1–19, 2019.

[56] 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.

[57] 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.

[58] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 821–830, 2017.

[59] Yue Li, Tian Tan, and Jingling Xue. Understanding and analyzing Java reflection. *ACM Transactions on Software Engineering and Methodology*, 28(2):1–50, 2019.

[60] 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.

[61] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[62] Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *International Conference on Automated Software Engineering (ASE)*, pages 415–426, 2019.

[63] Roy Osherove. Unit Testing Tips — Write Maintainable Unit Tests That Will Save You Time And Tears. *MSDN Magazine*, pages 107–118, 2006. Available online: https://learn.microsoft.com/en-us/archive/msdn-magazine/2006/january/unit-testing-writing-maintainable-unit-tests-save-time-and-tears.

[64] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for Java. In *OOPSLA Companion: Object-Oriented Programming Systems, Languages, and Applications*, 2007.

[65] Owain Parry, Michael Hilton, Gregory M. Kapfhammer, and Phil McMinn. Surveying the developer experience of flaky tests. In *International Conference on Software Engineering — Software Engineering In Practice track (ICSE-SEIP)*, 2022.

[66] Michael Quinn Patton. Enhancing the quality and credibility of qualitative analysis. *Health services research*, 34, 1999.

[67] 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.

[68] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *International Symposium on the Foundations of Software Engineering (FSE)*, pages 1–11, 2012.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. When testing meets code review: Why and how developers review tests. In *International Conference on Software Engineering*, pages 677–687, 2018.

[75] 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.

[76] 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.

[77] Ahmadreza Saboor Yaraghi, Darren Holden, Nafiseh Kahani, and Lionel Briand. Automated test case repair using language models. *arXiv preprint arXiv:2401.06765*, 2024.