Viscount: A Direct Method Call Coverage Tool for Java

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

***Abstract*—Writing unit tests against implementation detail in production code, often embodied in non-public methods, is considered bad practice in formal and gray literature. This is because it leads to fragile tests that break easily when underlying implementation details change. For this reason, tests that focus on behavior are encouraged. One way to achieve this is to test units exclusively through their public API. However, our recent developer survey shows that this advice is not always followed in practice. Moreover, code coverage tools do not provide a way to determine which methods were called directly from tests, meaning there is no easy way to identify whether units make calls to non-public methods, other than through manual examination. To address this problem, we developed Viscount, a tool that can determine *direct method call coverage* for Java tests written in JUnit. Viscount reports the percentage of methods invoked *directly* from tests, according to their visibility — i.e., public or *non-public* (protected, package-private, or private). This can help developers and researchers identify tests that potentially need to be refactored or rewritten. In this paper, we describe Viscount's overall architecture, its core features, and how to use it. Viscount is also publicly available on GitHub: https://github.com/unittesting-nonpublic/viscount. A demo video of Viscount is available at: https://youtu.be/ZUyRtiUnbsU.**

***Index Terms*—Access Modifiers, Code Coverage, Test Smells, Unit Testing.**

I. Introduction

Writing high-quality, effective unit tests is a challenging task for software developers. When deciding which tests to write, developers must consider a number of factors, including what parts of the production code to test and how to test them. In doing this, developers often aim to achieve high code coverage and to capture implemented behavior through meaningful assertions. Guidance and advocacy for writing good unit tests are plentiful in both formal [15] and gray literature [1].

Among the vast amount of advice available, testing behavior (i.e., through public methods) instead of implementation details (often embodied in non-public methods) is a common recommendation. This is because directly testing the implementation details underpinning a class leads to tests that are fragile and prone to breaking when that implementation changes. Writing tests that call non-public methods that are tightly coupled to the implementation details of production code has been documented as a bad practice by researchers in the literature [16], and also reported as a test "smell" [9], [20], [21].

Despite the existing advice against testing non-public methods directly, in a previous study we found that 28% of 4,801 open-source Maven projects contained at least one test that directly calls a non-public production method (i.e., one that has protected, package-private, or private visibility) [19]. In the same study, we also performed a developer questionnaire survey, finding that approximately a third of the 73 participants are not against the practice of testing non-public methods directly. Figure 1 exemplifies what this anti-pattern can look like in practice. Here, the JUnit test `testResize` uses *reflection* to gain access to and execute a private method `resize` in an object of a Java class called `Wallet`. As shown by Figure 3, it is plausible that the implementation of the `Wallet` class could change in the future to use a resizable collection instead of an array (that the `Wallet` class needs to maintain the size of itself). In this scenario, the method `resize` would be removed and the test would break, requiring refactoring or removal. Figure 2 presents an alternative test, `testAddCard`, that also involves the wallet resizing beyond its initial capacity when new cards are added. Here, the method `resize` is still being exercised, but this time via the execution of the public method `addCard` instead. This test is more realistic because it forms an explicit contract: if it fails, it implies not only that the code is broken, but that what end users will receive as an output is incorrect.

```
@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());
}
```

Fig. 1. A JUnit test for the `resize` method of the `Wallet` class (Fig. 3).

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

Fig. 2. A JUnit test for the `addCard` method of the `Wallet` class (Fig. 3).

One of the motivations that leads developers to engage in the practice of testing non-public methods directly is the goal of trying to achieve higher coverage [19]. As the practice is more prevalent than expected, it poses serious threats to the maintainability of a test suite [13]. We argue that one of the reasons why this is not being addressed is the lack of tool support to analyze *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 [2] provide a way to measure method coverage. However, tools like JaCoCo cannot

```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 ...
}
```

Fig. 3. Example of a class named `Wallet` that can store multiple `Card` objects. The public method `addCard` adds `Card` objects into the `Wallet`. It will not add existing cards (checked via a call to `isCardPresent`), and updates the `size` field, queried to check whether the internal array used to store cards needs to be resized — implemented in the `resize` method.

distinguish direct or indirect calls to non-public methods, because they do not track call hierarchies from tests [3]. In a smaller-sized project, it is possible to manually analyze each test case to identify non-public methods being called directly. However, it is not feasible to analyze projects with thousands of test cases. To address this problem, we developed **Viscount**, a tool that can determine *direct method call coverage* — i.e., the percentage of methods of each level of visibility in Java that are directly invoked from JUnit tests. Our tool is designed to help developers identify tests that call non-public methods directly to facilitate the refactoring of tests and their ongoing maintenance. Viscount's core features include:

1) Retrieving every method's visibility in the production code;
2) Retrieving production methods that are called directly in test code; and
3) Summarizing direct method call coverage in a clear format.

II. 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 is invoked through a command-line interface.

Viscount works with Maven projects, parsing the project's source code to find every production method's visibility. It then installs the Surefire Report plugin to the Maven project (if the project does not already include it), and sets up an execution environment 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 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 [12].

TABLE I

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
⋮	⋮

The main entry point of Viscount is a script, `viscount.sh`. As part of the tool requirement, the user needs to include the Maven project's name, its 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 the tool would be:

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

III. Dependencies

Viscount is primarily built on top of two Java libraries. The first of these is Spoon [18], a meta-programming library to analyze and transform Java source code. Spoon parses the source code to build an abstract syntax tree (AST) meta-model for performing AST analyses and transformations. 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. Viscount uses Spoon to extract the visibility of production code methods and to distinguish between production code and test code. Secondly, we used Javassist [6], a Java-bytecode analysis library to transform bytecode at compile or load time. Javassist parses class files into objects representing the classes, methods, and 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 [10]. Viscount uses Javassist to instrument production methods and constructors, and test methods and helpers (inserting probes at entry and exit points) during test execution.

IV. Viscount's Architecture

Figure 4 depicts the overall architecture of Viscount. We discuss the tool's main features, and how it works, in the following sections. Viscount's main steps are to:

A. Extract production code method visibility;
B. Include the Surefire Report plugin in the Maven project;
C. Perform runtime instrumentation of the project during test execution; and
D. Analyze the test reports.

To illustrate the operation of Viscount, we will use the following two projects as running examples:

1) **square-javapoet** [5], a Java API to generate `.java` source files; and
2) **viscount-example**, an example project that we created to demonstrate the tool based around the `Wallet` class and its tests, partially shown in Figures 1–3, and which is located in Viscount repository.

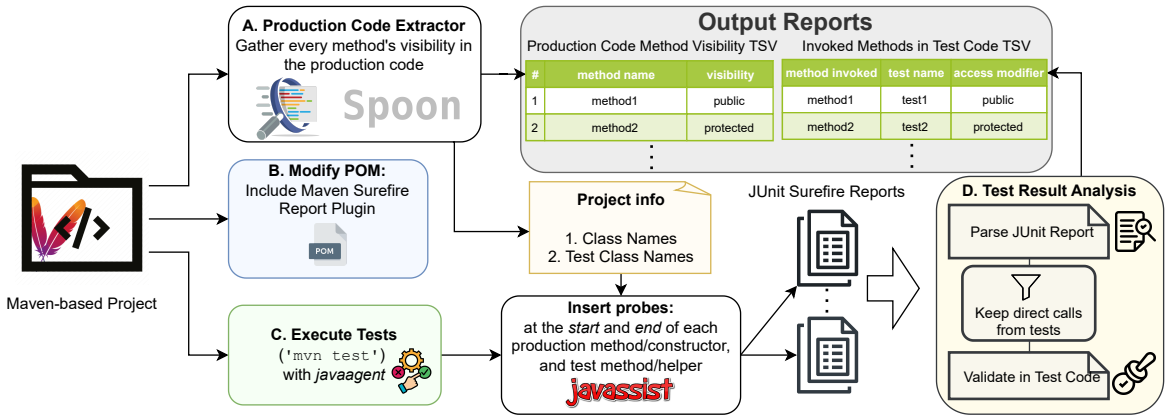

Fig. 4. Overall architecture of Viscount

A. Extracting Production Code Method Visibility

The first step investigates 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. This collects every method name and its visibility in all classes (including nested classes) in the production code (using Spoon's `CtMethod`). It also includes the parameters of each method to ensure method overloading is correctly handled. Table I shows an example of the TSV file output of this step. Since Spoon's MavenLauncher distinguishes between production classes and test classes, Viscount stores the names of each class and its type (production/test) in a temporary file. This is important information for later execution of the tests (see Section IV-C).

B. Including Surefire Report Plugin

Before executing the tests, Viscount automatically includes the Surefire Report plugin [8] as part of the project. This plugin 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 generates test reports, in `.xml` format, which Viscount uses to analyze test suites.

C. Runtime Instrumentation and Test Execution

Next, Viscount runs the command "`mvn test`" to execute the project's tests. It attaches a Java agent [11] that dynamically inserts instrumentation (probes) into the production methods and test code using Javassist [6] before the Java Virtual Machine loads the class. Since Viscount is only interested in direct method call coverage of the production code, it will only insert probes into methods and constructors for the classes that are part of the project, and not imported libraries or Java's own APIs. The agent uses the information collected in the previous step 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 Javaassist's `CtMethod` class (representing a method in Java) and `CtConstructor` (representing a constructor), and their corresponding `insertBefore` / `insertAfter` methods. For each

```
108 public String newName(String suggestion, Object tag) {
      logStartMethod("newName(String,Object)");
109   checkNotNull(suggestion, "suggestion");
110   checkNotNull(tag, "tag");
111   // ... omitted ...
123   return suggestion;
      // This is conceptually after the return, but
         actually runs before the return in bytecode
      logEndMethod("newName(String,Object)");
124 }
```

Fig. 5. Example of added probes at entry/exit points in a production method.

```
52  @Test
    public void characterMappingSubstitute() throws
        Exception {
      logStartTest("characterMappingSubstitute()");
53    NameAllocator nameAllocator = new NameAllocator();
54    assertThat(nameAllocator.newName("a-b", 1)).isEqualTo
        ("a_b");
      logEndTest("characterMappingSubstitute()");
55  }
```

Fig. 6. Example of added probes at entry/exit points in a test method.

production constructor and method, the probe logs its name, the types of its parameters, and its visibility modifier, on its entry and exit points (Figure 5). Since a constructor/method may exit abnormally upon an exception, Viscount further logs an exit when an exception is thrown (via the `addCatch` method in Javassist's `CtMethod` and `CtConstructor` classes). In the test code, the agent inserts a probe at the beginning and end of each test method and helper (Figure 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 final step (discussed in Section IV-D).

The overall process of instrumentation is similar to that employed by java-callgraph [4] when generating a call graph dynamically. The current limitation of Viscount, similar to that of java-callgraph, is that it does not work reliably for multi-threaded and concurrent programs. This is because the logs made by probes to denote the start and end of each method can interleave between threads, causing inaccuracies in later processing. Therefore, Viscount skips any tests involving concurrent execution in the analysis, as discussed next.

D. Analyzing Test Reports

The final step for Viscount is to analyze the JUnit XML test reports generated by the Surefire Report plugin. Since

TABLE II

Production methods directly called by tests for **square-javapoet**, as outputted by Viscount in a TSV file.

PROJECT	TEST CASE (TC)	METHOD NAME	VISIBILITY	...
javapoet	com.squareup.javapoet.TypeNameTest.isPrimitive()	com.squareup.javapoet.TypeName.isPrimitive()	public	...
javapoet	com.squareup.javapoet.UtilTest.characterLiteral()	com.squareup.javapoet.Util.characterLiteralWithoutSingleQuotes(char)	package-private	...
javapoet	com.squareup.javapoet.ClassNameTest.peerClass()	com.squareup.javapoet.ClassName.get(java.lang.Class)	public	...
⋮	⋮	⋮	⋮	⋮

```
<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>
```

Fig. 7. Output of `TypeNameTest.isPrimitive()` test from `square-javapoet` in Surefire Report

Viscount inserts the probes to log methods called in both production and test code, as described previously, it can now extract the methods that are directly called in the test code from the logs produced. Figure 7 shows an example of the output from one test case in the JUnit XML report. To analyze the report, Viscount parses the XML file and extracts method names immediately following tests. It discards any methods or constructors called from other production methods, as these are not directly invoked. The highlighted methods from Figure 7 are examples of those directly invoked from tests. Finally, to verify that each method is directly invoked from a test, Viscount performs a textual search of the tests's source code to find a match for the method name. This cross-checks both regular method calls as well as those made using reflection, while also revealing any potential unsoundness caused by concurrent thread execution [4]. The reason Viscount cannot just employ this simple text check on its own (i.e., without performing dynamic analysis) is because we cannot reliably statically determine method calls made using reflection — performed either directly or via third-party libraries [17]. Finally, Viscount does not include any test cases that failed or were skipped. It also discards any test for which it could not find matching entry and exit points or where the points are interleaved between methods (due to concurrent threads).

Table II shows an example Viscount's output. Using 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, grouped by each type of access modifier, as shown in Figure 8 for **square-javapoet** and Figure 9 for **viscount-example**. In the ideal case, direct method call coverage for non-public methods will be zero.

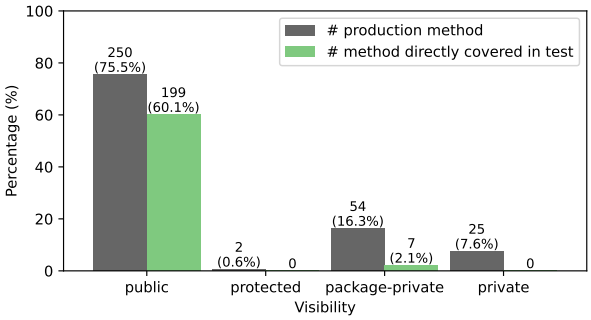

Fig. 8. *Direct method call coverage* of **square-javapoet**

TABLE III

The visibility of each production method 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

V. Applying the Tool

To demonstrate the capability of the tool, we replicate the analysis on **viscount-example**. In this example project, there are two test cases, `testResize` (Figure 1) and `testAddCard` (Figure 2). The production methods and their visibility are shown in Table III and the methods directly invoked from the tests are shown in Table IV. As shown by Figure 9, there are two non-public methods being called directly in the test suite, indicating to developers that they have tests that are coupled to implementation details.

We have also applied Viscount on a larger scale (4,801 projects) to evaluate the tool [19]. These projects are Maven-built projects from the Maven Central Repository [7] that contain at least one passing test with source code available on GitHub. We extracted the visibility of each method in the production code and identified the methods directly called in the test code. We were able to analyze 226,915 tests from 4,801 projects. We found that 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 in the tests across all projects are non-public methods.

VI. Current Limitations

As discussed in Section IV-C, Viscount cannot compute direct method call coverage for test cases that execute multi-threaded code. This is because it cannot guarantee the entry

TABLE IV

Production methods directly called by tests for **viscount-example**, as outputted by Viscount in a TSV file.

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

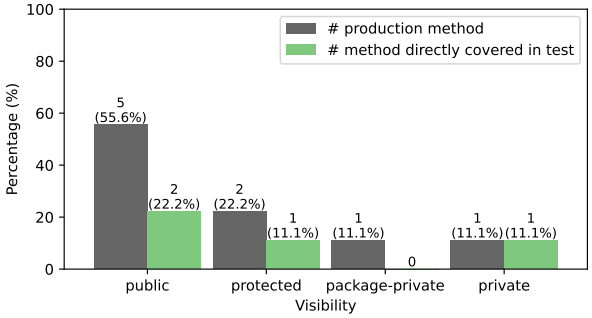

Fig. 9. *Direct method call coverage* of **viscount-example**

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 take a long time for projects that include recursive calls. One potential solution to each of these limitations is to statically analyze the call graph of each test method/helper to determine the direct method calls [14]. However, as already noted, most techniques based on static call graphs cannot identify methods invoked through reflection [17]. Finally, as we used Spoon's MavenLauncher to analyze the source code, the tool does not support other build systems, such as Gradle or Ant. We leave these as items for future work.

VII. Related Tools

JaCoCo [2], 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 [3], and therefore cannot be used to find direct calls to non-public methods in tests or be used to compute direct method call coverage. Yang et. al. [21] developed a test smell detector that can statically detect direct invocation of a private method in test code. Unlike Viscount, their test smell detector could only detect private methods that are being directly called in the test code, whereas Viscount can detect all levels of visibility in the Java language.

VIII. Conclusions and Future Work

This paper presents Viscount, a tool to help developers analyze which methods are being directly called from tests. It can aid in identifying tests that exercise non-public methods directly (i.e., protected, package-private, and private methods in Java), which is considered a bad practice in unit testing. The general aim is to help developers identify tests that focus on implementation details as opposed to behavior when maintaining tests. In summary, Viscount:

1) Retrieves every production method and its visibility;
2) Identifies directly called methods in the test code; and
3) Calculates *direct method call coverage* of the production code by the tests for each visibility modifier.

Future work could improve Viscount's report generation and enhance its handling of different types of members (which include fields, constructors, and classes) that are directly called from the test code. It could also extend the tool by incorporating static analysis (Spoon [18]) and bytecode analysis (SootUp [14]) to identify methods directly invoked by the tests, complementing or replacing aspects of dynamic test execution. This could improve the tool's efficiency and reduce the time taken to analyze test code.

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] Google Testing Blog — The advantages of unit testing early. https://testing.googleblog.com/2009/07/by-shyam-seshadri-nowadays-when-i-talk.html. Accessed: 8/2024.
[2] JaCoCo code coverage tool. https://www.jacoco.org/jacoco/. Accessed: 8/2024.
[3] JaCoCo Coverage of methods being invoked directly from a test case. https://groups.google.com/g/jacoco/c/x4OGEGPyi3E. Accessed: 8/2024.
[4] Java-callgraph: Programs for producing static and dynamic (runtime) call graphs for Java programs. https://github.com/gousiosg/java-callgraph. Accessed: 8/2024.
[5] JavaPoet. https://github.com/square/javapoet. Accessed: 8/2024.
[6] Javassist. https://www.javassist.org. Accessed: 8/2024.
[7] Maven Central Repository. https://repo.maven.apache.org/maven2/. Accessed: 8/2024.
[8] Maven Surefire Plugin. https://maven.apache.org/surefire/maven-surefire-plugin/. Accessed: 8/2024.
[9] The Open Catalog of Test Smells. https://test-smell-catalog.readthedocs.io. Accessed: 8/2024.
[10] OW2. 2024. ASM. https://asm.ow2.io/. Accessed: 8/2024.
[11] Package java.lang.instrument. https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html. Accessed: 8/2024.
[12] Viscount. https://github.com/unittesting-nonpublic/viscount. Accessed: 8/2024.
[13] Javaria Imtiaz, Salman Sherin, Muhammad Uzair Khan, and Muhammad Zohaib Iqbal. A systematic literature review of test breakage prevention and repair techniques. *In IST*, 113:1–19, 2019.
[14] 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 *TACAS*, pages 229–247, 2024.
[15] Lasse Koskela. *Effective Unit Testing: A guide for Java developers*. Manning, 2013.
[16] 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, 2020.
[17] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. Challenges for static analysis of Java reflection — literature review and empirical study. In *Proc. of ICSE*, pages 507–518, 2017.
[18] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of Java source code. *In Software: Practice and Experience*, 46(9):1155–1179, 2016.
[19] Muhammad Firhard Roslan, José Miguel Rojas, and Phil McMinn. Private — keep out? Understanding how developers account for code visibility in unit testing. In *Proc. of ICSME*, 2024.
[20] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. Refactoring test code. In *XP*, pages 92–95, 2001.
[21] Yanming Yang, Xing Hu, Xin Xia, and Xiaohu Yang. The lost world: Characterizing and detecting undiscovered test smells. *In TOSEM*, 2023.