Test Behaviors, Not Methods! Detecting Tests Obsessed by Methods

Andre Hora
Department of Computer Science, UFMG
Belo Horizonte, Brazil
andrehora@dcc.ufmg.br

Andy Zaidman
Delft University of Technology
Delft, The Netherlands
a.e.zaidman@tudelft.nl

Abstract

Best testing practices state that tests should verify a single functionality or behavior of the system. Tests that verify multiple behaviors are harder to understand, lack focus, and are more coupled to the production code. An attempt to identify this issue is the test smell *Eager Test*, which aims to capture tests that verify too much functionality based on the number of production method calls. Unfortunately, prior research suggests that counting production method calls is an inaccurate measure, as these calls do not reliably serve as a proxy for functionality. We envision a complementary solution based on runtime analysis: we hypothesize that some tests that verify multiple behaviors will likely cover multiple paths of the same production methods. Thus, we propose a novel test smell named *Test Obsessed by Method*, a test method that covers multiple paths of a single production method. We provide an initial empirical study to explore the presence of this smell in 2,054 tests provided by 12 test suites of the Python Standard Library. (1) We detect 44 *Tests Obsessed by Methods* in 11 of the 12 test suites. (2) Each smelly test verifies a median of two behaviors of the production method. (3) The 44 smelly tests could be split into 118 novel tests. (4) 23% of the smelly tests have code comments recognizing that distinct behaviors are being tested. We conclude by discussing benefits, limitations, and further research.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Software Testing, Test Smells, Test Comprehension, Runtime

ACM Reference Format:
Andre Hora and Andy Zaidman. 2026. Test Behaviors, Not Methods! Detecting Tests Obsessed by Methods. In *34th IEEE/ACM International Conference on Program Comprehension (ICPC '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3794763.3794791

1 Introduction

Best testing practices state that test methods should verify a single functionality or behavior of the system [1, 20, 21, 29, 31]. The Google Testing Blog refers to this practice as *"test behaviors, not methods"* [11]. This simple, but powerful recommendation brings several benefits to software development. First, tests are focused and easier to understand since each test contains code to exercise only one behavior [31]. Second, it reduces the coupling between the test and production code [31]. Third, when a new behavior is added to the system, a new test should be created for *that* behavior (i.e., existing tests are not changed). Thus, tests are more resilient to changes since adding new behaviors is unlikely to break the existing tests [13, 31].

Identifying test methods that violate this best practice is important for uncovering tests that verify multiple behaviors, that is, tests that are harder to understand, lack focus, are more coupled to the production code, and are less resilient to changes. However, this is not a trivial task because functionality (or behavior) is hard to define. An attempt at solving this is the test smell *Eager Test*, which aims to capture tests that verify too much functionality [20, 29]. Most solutions and tools to catch this test smell rely on rules that count the number of production method calls in test methods as a proxy for "the number of functionalities" [12, 22, 23, 30]. Unfortunately, it is not ideal to count production method calls to detect tests

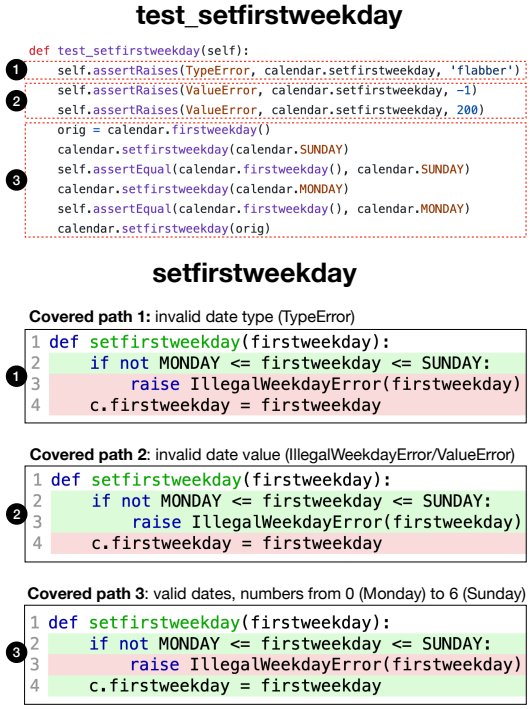

Figure 1: Example of a *Test Obsessed by Method* in CPython. Test `test_setfirstweekday` covers three paths of `setfirstweekday`. This test could be split into three tests.

ICPC '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2482-4/2026/04
https://doi.org/10.1145/3794763.3794791

that verify multiple functionalities. Prior studies report it can be inaccurate [22], has limited predictive power [30], and developers tend to disagree with low threshold values of method calls [26].

Given the aforementioned limitations of properly detecting test methods that verify multiple behaviors, we envision a complementary solution based on runtime analysis. *We hypothesize that some test methods that verify multiple behaviors will likely cover multiple paths of the same production methods*. Thus, we propose to use the covered paths of the production methods as a proxy for behaviors (see Figure 1 for an example). Unlike *Eager Test* implementations that work with static analysis (i.e., count of method calls), this novel smell works with runtime analysis (i.e., count of covered paths).

Thus, we propose a novel test smell named *Test Obsessed by Method*, a test method that covers multiple paths of a single production method. A test smell can be seen as a symptom of a problem [20], and in this case, it manifests as the test method's greediness in trying to cover multiple paths of a production method, leading to a test that potentially verifies multiple behaviors. Figure 1 presents an example of a *Test Obsessed by Method* in CPython. The test `test_setfirstweekday`[1] calls two production methods (`firstweekday` and `setfirstweekday`), but the issue is that it verifies multiple behaviors of `setfirstweekday`. Specifically, the test executes three distinct paths of `setfirstweekday` (as detailed in Figure 1). Path 1 is executed due to an invalid date type, which raises the exception `TypeError`. Path 2 occurs due to the invalid date values, resulting in the exception `IllegalWeekdayError`. Path 3 is executed due to the valid dates (i.e., 0 to 6). This test could be fixed by splitting it into three test methods, one for each behavior, as recommended by best testing practices [31].

This paper has two contributions. First, we propose a novel test smell named *Test Obsessed by Method* (Section 3). Second, we provide an initial empirical study to explore the presence of *Tests Obsessed by Methods* in real-world test suites (Section 4). We analyze 2,054 test methods of 12 real-world test suites of the Python Standard Library. To detect the smells, we run an instrumented version of the test suites and collect information about the executed lines of code at runtime. We define two research questions:

- **RQ1: How prevalent are Tests Obsessed by Methods**? We detect 44 *Tests Obsessed by Methods* in 11 of 12 test suites. Each smelly test verifies two behaviors of the production method on the median.
- **RQ2: How can we fix Tests Obsessed by Methods**? The 44 smelly tests could be split into 118 novel tests. We find code comments in 10 out of 44 (23%) smelly tests, recognizing that distinct behaviors are being tested.

Finally, we conclude the paper by discussing the benefits and limitations of the proposed test smell and further research.

2 Related Work

Ideally, test suites should have good quality to catch bugs and protect against regressions [1, 7, 16–18]. Test smells indicate potential design problems in the test code [20, 29]. The presence of test smells in test suites may affect the test quality, maintainability, and extendability, reducing their effectiveness in finding bugs in production code [3–5, 10, 23, 25, 28]. Ideally, test methods should verify a single functionality or behavior of the system [31]. Tests that violate this best practice are considered *eager*. According to Meszaros [20], an *Eager Test* is a test that verifies too much functionality. Van Deursen *et al.* [21, 29] originally defined it as a test that checks several methods of the object to be tested. Both definitions [20, 29] address the *Eager Test* informally; the number of verified functionalities or methods is not clear.

To overcome this limitation, other studies define *Eager Test* more formally [12, 22, 23, 30]. For example, Van Rompaey *et al.* [30] proposed a metric-based approach that relies on the number of production method calls, but concluded that this metric has limited predictive power. Test smell detection tools also rely on the production method calls to detect *Eager Tests*, for example, setting a threshold of at least *two* calls to production methods [12, 23]. Spadini *et al.* [26] reported that developers disagree with such a low threshold, finding that *four* calls to production methods are better suited to detect *Eager Tests*. Recently, Panichella *et al.* [22] analyzed two test smell detection tools [12, 23] and concluded that existing tools simply rely on rules that count the number of production method calls in test methods as a proxy for "the number of functionalities", suggesting that such a simple heuristic is highly inaccurate. Another important conclusion of the research is that it is non-trivial to detect *Eager Tests* automatically, and it is fault-prone to assume a threshold of just two calls. Thus, detecting test methods that verify multiple functionalities requires more semantic awareness than is currently considered [22].

3 Tests Obsessed by Methods

3.1 Overview

A *Test Obsessed by Method* is a test method that covers multiple paths of a single production method. The rationale is that each covered path in such a production method represents a behavior. Thus, a *Test Obsessed by Method* is a test that potentially verifies multiple behaviors. To fix a test method with this smell, we can create one test method for each tested behavior. In this case, behaviors can be conveniently identified by the covered paths of the production method. For example, the production method `setfirstweekday` presented in Figure 1 has three covered paths, thus, the test `test_setfirstweekday` could be split into three tests.

Figure 2 presents `test_constructor`[2] of the CPython argparse library. This test is problematic because it verifies two behaviors of the production method `Namespace`. These behaviors should ideally be tested in two distinct test methods to verify: (1) an exceptional case that raises the exception `AttributeError` and (2) valid cases.

```
def test_constructor(self):
    ns = argparse.Namespace()
1   self.assertRaises(AttributeError, getattr, ns, 'x')

    ns = argparse.Namespace(a=42, b='spam')
2   self.assertEqual(ns.a, 42)
    self.assertEqual(ns.b, 'spam')
```

Figure 2: *Test Obsessed by Method*: `test_constructor` tests two behaviors of `Namespace` (argparse).

[1] test_setfirstweekday: https://github.com/python/cpython/blob/2938c3d/Lib/test/test_calendar.py#L513

[2] test_constructor: https://github.com/python/cpython/blob/f474391b/Lib/test/test_argparse.py#L5673

Figure 3 shows the test `test_splitroot`[3] of the CPython pathlib library. This test method is problematic because it verifies three behaviors of the production method `splitroot`. These behaviors should ideally be tested in three distinct test methods, covering (1) basic paths, (2) POSIX paths (i.e., Unix-like), and (3) NT paths (i.e., Windows). Interestingly, the code comments on the test method highlight that distinct requirements are being tested.

```
def test_splitroot(self):
    f = self.flavour.splitroot
    self.assertEqual(f(''), ('', '', ''))
    self.assertEqual(f('a'), ('', '', 'a'))
1   self.assertEqual(f('a/b'), ('', '', 'a/b'))
    self.assertEqual(f('a/b/'), ('', '', 'a/b/'))
    self.assertEqual(f('/a'), ('', '/', 'a'))
    self.assertEqual(f('/a/b'), ('', '/', 'a/b'))
    self.assertEqual(f('/a/b/'), ('', '/', 'a/b/'))
    # The root is collapsed when there are redundant slashes
    # except when there are exactly two leading slashes, which
    # is a special case in POSIX.
2   self.assertEqual(f('//a'), ('', '//', 'a'))
    self.assertEqual(f('///a'), ('', '/', 'a'))
    self.assertEqual(f('///a/b'), ('', '/', 'a/b'))
    # Paths which look like NT paths aren't treated specially.
3   self.assertEqual(f('c:/a/b'), ('', '', 'c:/a/b'))
    self.assertEqual(f('\\/a/b'), ('', '', '\\/a/b'))
    self.assertEqual(f('\\a\\b'), ('', '', '\\a\\b'))
```

Figure 3: *Test Obsessed by Method*: `test_splitroot` tests three behaviors of `splitroot` (pathlib).

Lastly, Figure 4 presents `test_splitroot`[4] of the CPython configparser library. This test is problematic because it verifies multiple exceptional behaviors [7, 17, 18] of the production method `ParsingError`. These behaviors should ideally be tested in three distinct test methods, covering the distinct forms to use `ParsingError`.

```
def test_parsing_error(self):
    with self.assertRaises(ValueError) as cm:
1       configparser.ParsingError()
    self.assertEqual(str(cm.exception), "Required argument `source' not "
                                        "given.")
    with self.assertRaises(ValueError) as cm:
2       configparser.ParsingError(source='source', filename='filename')
    self.assertEqual(str(cm.exception), "Cannot specify both `filename' "
                                        "and `source'. Use `source'.")
    error = configparser.ParsingError(filename='source')
    self.assertEqual(error.source, 'source')
    with warnings.catch_warnings(record=True) as w:
        warnings.simplefilter("always", DeprecationWarning)
3       self.assertEqual(error.filename, 'source')
        error.filename = 'filename'
        self.assertEqual(error.source, 'filename')
    for warning in w:
        self.assertTrue(warning.category is DeprecationWarning)
```

Figure 4: *Test Obsessed by Method*: `test_parsing_error` tests three behaviors of `ParsingError` (configparser).

[3]test_splitroot: https://github.com/python/cpython/blob/0c5fc272/Lib/test/test_pathlib.py#L80
[4]test_parsing_error: https://github.com/python/cpython/blob/0c5fc272/Lib/test/test_configparser.py#L1604

It is important to note that a test method calling the same production method multiple times does not necessarily pose a problem. Consider, for example, a test method that calls method `add` of some data structure multiple times and then verifies the data structure size. In this case, calling the same method `add` multiple times is not an issue, and this test method is not a *Test Obsessed by Method*. Indeed, test methods like this one will be present in any test suite; thus, the challenge is to distinguish such valid tests from the ones that are *really* verifying multiple behaviors of a production method. To reduce the possibility of false positives, we rely on runtime analysis to detect *Tests Obsessed by Methods*.

3.2 Detecting Tests Obsessed by Methods

To detect *Tests Obsessed by Methods*, we perform runtime analysis by collecting data during test execution. We collect the covered paths of every production method executed by the test methods. A covered path refers to a set of input values that cause the production method to follow the same execution flow, resulting in the execution of identical lines of code. If a test covers two or more paths of a production method, it is classified as smelly. For example, `test_setfirstweekday` covers three paths of `setfirstweekday`: (1) line 2; (2) lines 2,3; and (3) lines 2,4; thus, it is smelly.

4 Preliminary Empirical Study

4.1 Design

Case Study: We aim to identify the presence of *Tests Obsessed by Methods* in real-world test suites. For this purpose, we analyze 2,054 test methods of 12 test suites of the Python Standard Library: gzip, email, calendar, ftplib, collections, os, tarfile, pathlib, logging, smtplib, argparse, and configparser. Our dataset is available at: https://doi.org/10.5281/zenodo.17469070.

Runtime Analysis: To detect the covered paths of the production methods, we run an instrumented version of the test suites and collect information about the executed lines at runtime. We rely on SpotFlow [15], a tool to ease runtime analysis in Python, which is implemented with the support of the standard trace function [27].

4.2 Results

RQ1: How prevalent are Tests Obsessed by Methods? We find 54 *Test Obsessed by Method* candidates in the 2,054 analyzed test methods. We manually analyzed the 54 tests and detected 44 true positives and 10 false positives, resulting in a precision of 81.5%. True positives are test methods that can be split into multiple tests, while false positives are harder or impossible to split. Examples of true positives are presented in Figures 1-4. False negatives occur in scenarios where the production method is invoked indirectly within the test or is executed as part of the test setup. The smelly tests are present in 11 of 12 test suites; on the median, they exercise two paths of the production method. Table 1 details the number of *Tests Obsessed by Methods* by covered paths.

Table 1: *Tests Obsessed by Methods* by covered paths.

	Total	Covered Paths				
		2	3	4	5	7
#Tests Obsessed by Methods	44	25	12	5	1	1

RQ2: How can we fix Tests Obsessed by Methods? In this RQ, we explore how the *Tests Obsessed by Methods* could be potentially fixed. Considering that each smelly test verifying *n* behaviors could be split into *n* tests (i.e., one for each covered path), the 44 smelly tests could be split into 118 novel tests (i.e., 25×2 + 12×3 + 5×4 + 1×5 + 1×7), as detailed in Table 1.

Interestingly, among the 44 smelly tests, 10 have code comments recognizing that distinct behaviors are, in fact, being tested. Due to the space limit, we briefly present three examples. The first example happens in the test `test_splitroot` (see Figure 3). In this case, the comments highlight that distinct behaviors of the production method `splitroot` are tested: (1) basic paths, (2) POSIX paths, and (3) NT paths. The second example is the test `test_is_tarfile_erroneous`[5] of the tarfile library, which tests two behaviors of `is_tarfile`. In this case, the comments suggest that two behaviors of `is_tarfile` are verified: (1) for string tar files and (2) for file-like object tar files. Finally, the third example happens in the test `test_is_absolute`[6] of the pathlib library. It tests two behaviors of the method `is_absolute`: (1) for NT files and (2) for UNC paths.

Summary: (1) We detect 44 *Tests Obsessed by Methods* in 11 of 12 test suites. (2) Each smelly test verifies a median of two behaviors of the production method. (3) The 44 smelly tests could be split into 118 novel tests. (4) 10 in 44 (23%) smelly tests have comments recognizing that distinct behaviors are being tested.

5 Discussion

5.1 Fixing Tests Obsessed by Methods

We identified *Tests Obsessed by Methods* in 11 of the 12 analyzed test suites, indicating that the problem is spread over multiple projects rather than isolated. One important aspect of *Tests Obsessed by Methods* is that behaviors can be conveniently identified by the covered paths of the production method. This is why the 44 smelly tests could be refactored into 118 novel, focused tests. In contrast, the 10 false positives refer to cases harder to refactor, such as production methods part of the test setup.

Thus, the proposed test smell not only identifies the problem (i.e., testing multiple behaviors within a single test), but also provides guidance for resolution (i.e., create one test per covered path).

5.2 Runtime Analysis

Most test smell detection techniques rely on static analysis. However, identifying *Tests Obsessed by Methods* requires runtime (dynamic) analysis, which involves executing the test suite and gathering runtime data. Other test smells also rely on runtime analysis, for example, *Rotten Green Tests*, which are passing tests with at least one assertion not executed [2, 9, 19, 24]. While runtime analysis is more expensive than static, it can identify problems that are only "visible" when code is executed, such as a test with an assertion not executed or that covers multiple paths of a production method.

[5]test_is_tarfile_erroneous: https://github.com/python/cpython/blob/850189a6/Lib/test/test_tarfile.py#L359
[6]test_is_absolute: https://github.com/python/cpython/blob/850189a6/Lib/test/test_pathlib.py#L1222

Table 2: *Eager Test* vs. *Test Obsessed by Method*.

Test Method	Eager Test		Test Obsessed by Method
	2 calls	4 calls	
`test_setfirstweekday`	✔	✘	✔
`test_constructor`	✘	✘	✔
`test_splitroot`	✘	✘	✔
`test_parsing_error`	✘	✘	✔
`test_is_tarfile_erroneous`	✘	✘	✔
`test_is_absolute`	✘	✘	✔

5.3 Comparison with Eager Test

Eager Test is defined as a test method that contains multiple calls to multiple production methods [23]. It is unclear what should be the minimum number of production method calls: some studies suggest at least two calls [12, 23], while others recommend at least four calls [26]. Moreover, constructor calls are typically excluded [22]; otherwise, any test that instantiates a class and calls a method could be an *Eager Test*. Considering the six *Tests Obsessed by Methods* discussed in this paper, only `test_setfirstweekday` with a threshold of 2 calls would be classified as *Eager Test* as well. All the other tests would not be classified as *Eager Test*, as detailed in Table 2. Therefore, *Tests Obsessed by Methods* can detect smelly tests that *Eager Test* does not identify.

6 Limitations

The proposed test smell is not intended to detect *all* test methods that verify multiple behaviors. Instead, we aim to identify *some* test methods that verify multiple behaviors, particularly those focused on testing multiple behaviors of a single production method. Therefore, it should be used in complement to other test smells, such as *Eager Test*, rather than a substitute. In fact, *Tests Obsessed by Methods* may identify smelly tests that *Eager Test* misses, and vice versa. Further analysis is needed to better compare both smells.

7 Conclusion and Future Work

We proposed a novel test smell named *Test Obsessed by Method*, a test method that covers multiple paths of a single production method. We conducted an initial study to explore the presence of this smell and found it in 11 of 12 test suites.

Future Work: First, we plan to conduct a qualitative study with experts to better understand the limitations of tests that verify multiple behaviors. Second, we plan to expand the empirical study by including more real-world test suites, not only from the Python Standard Library. Lastly, we intend to conduct a contribution study [6, 8, 14] in which we submit pull requests containing refactorings that remove *Tests Obsessed by Methods* in open-source projects. We hope to better understand software engineers' ideas about the problem and the proposed refactoring.

Acknowledgments

This research was supported by CNPq (process 403304/2025-3), CAPES, and FAPEMIG. This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence), www.ines.org.br, CNPq grant 408817/2024-0.

References

[1] Mauricio Aniche, Christoph Treude, and Andy Zaidman. 2022. How Developers Engineer Test Cases: An Observational Study. *IEEE Trans. Software Eng.* 48, 12 (2022), 4925–4946.

[2] Vincent Aranega, Julien Delplanque, Matias Martinez, Andrew P Black, Stéphane Ducasse, Anne Etien, Christopher Fuhrman, and Guillermo Polito. 2021. Rotten green tests in Java, Pharo and Python: An empirical study. *Empirical Software Engineering* 26 (2021), 1–41.

[3] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. 2014. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Trans. Software Eng.* 40, 11 (2014), 1100–1125.

[4] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *International Conference on Software Maintenance (ICSM)*. 56–65.

[5] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? an empirical study. *Empirical Software Engineering* 20 (2015), 1052–1094.

[6] Carolin Brandt, Ali Khatami, Mairieli Wessel, and Andy Zaidman. 2024. Shaken, Not Stirred. How Developers Like Their Amplified Tests. *IEEE Transactions on Software Engineering* 50, 5 (2024), 1264–1280.

[7] Francisco Dalton, Márcio Ribeiro, Gustavo Pinto, Leo Fernandes, Rohit Gheyi, and Baldoino Fonseca. 2020. Is exceptional behavior testing an exception? an empirical assessment using java automated tests. In *International Conference on Evaluation and Assessment in Software Engineering*. 170–179.

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

[9] Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew P. Black, and Anne Etien. 2019. Rotten Green Tests. In *International Conference on Software Engineering ICSE*. IEEE, 500–511.

[10] Vahid Garousi and Barış Küçük. 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software* 138 (2018), 52–81.

[11] Google Testing Blog - Test Behaviors, Not Methods. October, 2024. https://testing.googleblog.com/2014/04/testing-on-toilet-test-behaviors-not.html.

[12] Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C Gall. 2019. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software* 156 (2019), 312–327.

[13] Michaela Greiler, Andy Zaidman, Arie van Deursen, and Margaret-Anne D. Storey. 2013. Strategies for avoiding text fixture smells during software evolution. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, 387–396.

[14] Andre Hora. 2024. PathSpotter: Exploring Tested Paths to Discover Missing Tests. In *International Conference on the Foundations of Software Engineering (ICSE)*. ACM, 647–651.

[15] Andre Hora. 2024. SpotFlow: Tracking Method Calls and States at Runtime. In *International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*. IEEE, 35–39.

[16] Andre Hora. 2024. Test polarity: detecting positive and negative tests. In *International Conference on the Foundations of Software Engineering (FSE)*. 537–541.

[17] Andre Hora and Gordon Fraser. 2025. Exceptional Behaviors: How Frequently Are They Tested?. In *International Conference on Automation of Software Test (AST)*. IEEE, 70–79.

[18] Diego Marcilio and Carlo A Furia. 2021. How Java programmers test exceptional behavior. In *International Conference on Mining Software Repositories (MSR)*. IEEE, 207–218.

[19] Matias Martinez, Anne Etien, Stéphane Ducasse, and Christopher Fuhrman. 2020. Rtj: a Java framework for detecting and refactoring rotten green test cases. In *International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*. IEEE, 69–72.

[20] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[21] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. 2008. On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension. In *Software Evolution*, Tom Mens and Serge Demeyer (Eds.). Springer, 173–202.

[22] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. 2022. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering* 27, 7 (2022), 170.

[23] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. Tsdetect: An open source test smells detection tool. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1650–1654.

[24] Paul T Robinson. 2023. Rotten Green Tests in Google Test. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2020–2025.

[25] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the relation of test smells to software code quality. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 1–12.

[26] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. 2020. Investigating severity thresholds for test smells. In *International Conference on Mining Software Repositories*. 311–321.

[27] sys.settrace. October, 2025. https://docs.python.org/3/library/sys.html#sys.settrace.

[28] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *International Conference on Automated Software Engineering*. ASE, 4–15.

[29] Arie van Deursen, Leon Moonen, Alex van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *International Conference on Extreme Programming and Flexible Processes in Software Engineering*. 92–95.

[30] Bart van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. 2007. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on software engineering* 33, 12 (2007), 800–817.

[31] Titus Winters, Hyrum Wright, and Tom Manshreck. 2020. Software Engineering at Google: Lessons Learned from Programming over Time.