Understanding Bug-Reproducing Tests: A First Empirical Study

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

Gordon Fraser
University of Passau
Passau, Germany
gordon.fraser@uni-passau.de

Abstract

Developers create bug-reproducing tests that support debugging by failing as long as the bug is present, and passing once the bug has been fixed. These tests are usually integrated into existing test suites and executed regularly alongside all other tests to ensure that future regressions are caught. Despite this co-existence with other types of tests, the properties of bug-reproducing tests are scarcely researched, and it remains unclear whether they differ fundamentally. In this short paper, we provide an initial empirical study to understand bug-reproducing tests better. We analyze 642 bug-reproducing tests of 15 real-world Python systems. Overall, we find that bug-reproducing tests are not (statistically significantly) different from other tests regarding LOC, number of assertions, and complexity. However, bug-reproducing tests contain slightly more try/except blocks and "weak assertions" (e.g., `assertNotEqual`). Lastly, we detect that the majority (95%) of the bug-reproducing tests reproduce a single bug, while 5% reproduce multiple bugs. We conclude by discussing implications and future research directions.

CCS Concepts

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

Keywords

Software Testing, Test Comprehension, Bugs, Reproducibility, Python

ACM Reference Format:
Andre Hora and Gordon Fraser. 2026. Understanding Bug-Reproducing Tests: A First Empirical Study. In *7th ACM/IEEE International Conference on Automation of Software Test (AST 2026) (AST '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3793654.3793752

1 Introduction

When fixing a bug, ideally, developers should create a corresponding automated test that reproduces the bug, ensuring that this test fails when the buggy code is present and passes once the bug has been fixed [29, 32]. This best practice ensures that future regressions are caught and is widely recommended: *"The best way to start fixing a bug is to make it reproducible. After all, if you can't reproduce it, how will you know if it is ever fixed?"* [29]. This practice is also common in open-source projects [2, 20], for example, the contribution guidelines of the Black project state: *"If you're fixing a bug, add a test. Run it first to confirm it fails, then fix the bug, run it again to confirm it's really fixed"* [2].

```python
def test_whitespaces_are_removed_from_url(self):
    # Test for issue #3696
    request = requests.Request("GET", " http://example.com").prepare()
    assert request.url == "http://example.com/"
```

(a) Test in Requests (bug #3696).

```python
def test_nonzero_multi_threading(self):
    # Test that MPS does not crash if nonzero called concurrently
    # See https://github.com/pytorch/pytorch/issues/100285
    x = torch.rand(3, 3, device="mps")
    t1 = threading.Thread(target=torch.nonzero, args=(x,))
    t2 = threading.Thread(target=torch.nonzero, args=(x,))
    t1.start()
    t2.start()
```

(b) Test in PyTorch (bug #100285).

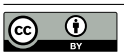
Figure 1: Examples of bug-reproducing tests.

Figure 1a shows a real example of a bug-reproducing test: it contains a bug ID that links it to a description of the underlying bug, which causes whitespaces to not be correctly removed from URLs (project "requests", bug #3696).[1] The test consists of a call to the API similar to an example given in the bug report. As another example, the test in Figure 1b additionally contains a short description, and then exercises concurrent usage on a certain GPU, though this time using an artificial scenario derived by the developer, rather than the actual user-reported call (PyTorch, bug #100285).[2] Note that this test has no assertions, meaning it will fail only if the production code raises an exception [5, 11, 18]. Both tests focus on testing individual bugs, but bug-reproducing tests may also test multiple bugs, as exemplified by the CPython test `test_strftime`,[3] which reproduces five bugs. There are also cases where multiple tests are needed to reproduce a single bug properly. For instance, four tests have been created in the Pandas project to reproduce bug #3490.[4]

Besides their role in software development, bug-reproducing tests are also frequently used in research, for example, in databases of real bugs [16, 30, 31], program repair [15, 19], test reduction [17, 23], and negative/exceptional tests [5, 10, 11, 18].

In practice, bug-reproducing tests are usually integrated into existing test suites and executed regularly alongside all other tests

[1] Requests: https://github.com/psf/requests/blob/79b74ef/tests/test_requests.py#L187; https://github.com/psf/requests/issues/3696
[2] PyTorch: https://github.com/pytorch/pytorch/blob/d1f73fd/test/test_mps.py#L10786; https://github.com/pytorch/pytorch/issues/100285
[3] CPython: https://github.com/python/cpython/blob/d13ee0ae186f4704f3b6016dd52f7727b81f9194/Lib/test/datetimetester.py#L1586
[4] Pandas: https://github.com/pandas-dev/pandas/blob/a2fb11e/pandas/tests/plotting/test_datetimelike.py#L1445-L1508



AST '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2476-3/2026/04
https://doi.org/10.1145/3793654.3793752

to ensure that future regressions are caught. **Despite this co-existence with other types of tests, the properties of bug-reproducing tests are scarcely researched, and it remains unclear whether they differ fundamentally.**

In this paper, we provide an initial empirical study to understand bug-reproducing tests better, in order to provide a foundation for assessing whether such tests adhere to best testing practices and identifying potential areas for improvement. We analyze 642 bug-reproducing tests of 15 real-world Python systems, addressing two research questions:

RQ1: What are the code characteristics of bug-reproducing tests? Overall, we find that bug-reproducing tests are not (statistically significantly) different from other tests regarding LOC, number of assertions, and complexity. However, bug-reproducing tests have slightly more try/except blocks and "weak assertions".

RQ2: How are bugs mapped to bug-reproducing tests? The majority (95%) of the bug-reproducing tests reproduce a single bug, while a minority (5%) reproduce multiple bugs. Moreover, sometimes, multiple tests are needed to reproduce a bug. We find that 20% of the bug-reproducing tests reproduce shared bugs.

Contributions. The contributions of this study are twofold. First, we describe the first empirical study to understand bug-reproducing tests in the wild, in real-world systems. Second, we discuss actionable implications and future research direction.

2 Study Design

2.1 Case Studies

We aim to study bug-reproducing tests of real-world and relevant systems. For this purpose, we selected the top-15 most popular Python software systems hosted on GitHub according to the number of stars, a metric primarily adopted in the software mining literature as a proxy of popularity [3, 26]. We focus on Python because it is the most popular programming language nowadays, and it has a rich software ecosystem. The 15 selected systems are Transformers (132K stars), TheFuck (85K), PyTorch (82K), Django (80K), FastAPI (76K), Flask (68K), Ansible (62K), CPython (62K), Scikit-Learn (60K), Requests (52K), Scrapy (52K), Rich (50K), Pandas (43K), Black (39K), and Sentry (39K). In total, these systems have 121,447 test methods. Our dataset is available at: https://doi.org/10.5281/zenodo.17468629.

2.2 Detecting Bug-Reproducing Tests

Bugs are typically reported and stored in Issue Tracking Systems, such as GitHub Issues. These systems manage not only bugs but also other reported issues, such as new features, refactorings, and more. Labels such as *bug*, *new feature*, and *refactoring* can be used to categorize issues, but they are optional. Ideally, a bug-reproducing test should be linked to its corresponding bug-labeled issue. Links between code and issues can be found in various artifacts, such as pull requests (PRs), commits, and code comments. Using such indicators, starting from a bug-labeled issue, we can find its linked artifacts (such as commits or PRs) that actually contain the bug-reproducing tests. We can also perform the opposite operation: we can start from a commit or a PR, detect issue IDs in commit/PR messages (e.g., *"fixed issue #123"*), verify whether the issue is really a bug, and then identify bug-reproducing tests. However, both solutions present some drawbacks: (1) issues are not necessarily properly labeled; (2) commit messages may not include issue IDs; and (3) commits may include unrelated changes [7] (e.g., a bug fix and a new feature).

Table 1: Bug-reproducing tests by project.

Project	Total	Bug ID in...	
		Test Name	Test Comment
CPython	264	99	165
Django	149	28	121
Scikit-Learn	81	0	81
PyTorch	64	1	63
Pandas	55	21	34
Transformers	16	0	16
Rich	5	0	5
Sentry	5	1	4
Black	2	0	2
Scrapy	1	0	1
Total	642	150 (23%)	492 (77%)

To avoid these possible limitations, this study focuses on detecting tests that the developers themselves label in the source code as bug-reproducing. This is a conservative method where we prioritize precision over recall. This solution is inspired by the rich literature on self-admitted technical debt, which has successively used a similar motivation to detect technical debt [22, 25]. For this purpose, we mine test methods that directly include the words *"bug"* or *"regression"* and refer to an issue ID in test comments or test names. For example, test methods with comments like "*# Regression #123*" or with test names like `test_bug_123`.

Following this method identifies 642 bug-reproducing tests in 10 out of the 15 analyzed systems, as detailed in Table 1. We note that five projects have a higher number of bug-reproducing tests: CPython (264), Django (149), Scikit-Learn (81), PyTorch (64), and Pandas (55). Other projects have lower numbers: Transformers (16), Rich (5), Sentry (5), Black (2), and Scrapy (1). Also, 23% (150) of the bug-reproducing tests include the bug ID in their names, e.g., `test_bug_3061`.[5]

2.3 Research Questions

2.3.1 RQ1: What are the code characteristics of bug-reproducing tests? We compute four metrics from the bug-reproducing tests: LOC (lines of code), number of assertions, complexity, and number of try/except blocks. Complexity is measured by the count of control flow structures in the test code, such as `if`, `for`, `while`, and `try`. For comparison, we also compute the same metrics for all 121K tests of the 15 selected systems. We apply the Mann-Whitney U-test and Cohen effect size to verify whether the metrics of both groups of tests differ. We further explore differences in assertions by extracting the most used assertion commands in each group of tests. We analyze the usage of "weak assertions", that is, assertions that are potentially less effective [33], such as `assertNotEqual` and `assertContains`. **Rationale:** We aim to understand better whether bug-reproducing tests differ from ordinary tests regarding

[5] CPython: https://github.com/python/cpython/blob/d13ee0ae186f4704f3b6016dd52f7727b81f9194/Lib/test/test_time.py#L657

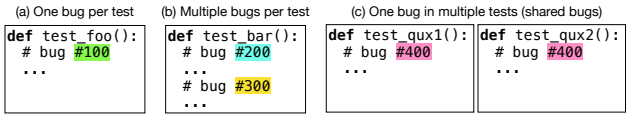


Figure 2: Mapping between bugs and bug-reproducing tests.

Table 2: Code characteristics of bug-reproducing tests. Q1, Q2, Q3: first, second, and third quartiles. ES: effect-size

Metric	Quartile	All Tests	Bug-Reproducing Tests	p-value	ES
LOC	Q1	5	6	<0.01	+0.064 negligible
	Q2	10	11		
	Q3	20	19		
	Mean	16	17.8		
Assertions	Q1	1	1	0.32	+0.103 negligible
	Q2	2	2		
	Q3	4	3		
	Mean	3	3.5		
Complexity	Q1	0	0	0.16	−0.003 negligible
	Q2	0	0		
	Q3	0	0		
	Mean	0.39	0.38		
Try/except	Q1	0	0	<0.01	+0.265 small
	Q2	0	0		
	Q3	0	0		
	Mean	0.029	0.088		

the analyzed metrics and whether they tend to use the same or distinct assertion commands. Moreover, using "weak assertions" may indicate that the tested code is somehow harder to test.

2.3.2 RQ2: How are bugs mapped to bug-reproducing tests? As detailed in Figure 2, we focus on three possible scenarios: (a) one bug per test, (b) multiple bugs per test, and (c) one bug in multiple tests (shared bugs, for short). **Rationale:** Our goal is to reveal how developers map bugs to tests. Ideally, a test is focused and reproduces a single bug, as represented by scenario (a). In this case, a failing test will clearly indicate the problem [32]. In contrast, cases like (b) are less desirable because the test reproduces multiple bugs (like a test verifying multiple functionalities). In this case, the test may be harder to understand, and a failing test will not clearly address the problem. Lastly, cases like (c) may indicate that a bug is too complex to be revealed or checked by a single test, like a larger functionality that is broken into smaller, testable ones.

3 Results

3.1 RQ1: Code Characteristics

Table 2 details the code metrics extracted from the bug-reproducing tests, as well as for all 121K test methods of the 15 analyzed projects (column *All Tests*). We find that the differences are statistically significant (*p-value* < *0.01*) for LOC, but the difference has a *negligible* effect size, and the difference is significant for try/excepts with a *small* effect size. This means that there are no practical differences between bug-reproducing and all tests in LOC, assertions, and complexity, while there are significant/small differences for try/excepts.

Observation 1: Overall, bug-reproducing tests are not (statistically significantly) different from other tests regarding the number of lines of code, number of assertions, and complexity. However, bug-reproducing tests tend to have slightly more try/except blocks.

Table 3: Top-25 most commonly used assertions (legend: weak assertions and exclusive weak assertions).

All Tests (8 weak assertions, 2 exclusives)
assertEqual, assert, assertTrue, assert_frame_equal, assertFalse, assertIn, assertIs, assert_series_equal, assertIsInstance, assert_index_equal, assertIsNone, assertListEqual, assert_numpy_array_equal, assertNotEqual, assertSequenceEqual, assertNotIn, assertIsNotNone, assertExpectedInline, assertContains, assert_equal, assert_called_with, assert_produces_warning, assertAlmostEqual, assertNumQueries, assertQuerySetEqual
Bug-Reproducing Tests (11 weak assertions, 4 exclusives)
assertEqual, assert, assertTrue, assertIs, assertQuerySetEqual, assertIsInstance, assertSequenceEqual, assertFalse, assertNumQueries, assert_index_equal, assertIsNone, assertIn, assert_series_equal, assert_frame_equal, assertContains, assertNotHasAttr, assertNotIn, assertRedirects, assert_numpy_array_equal, assertNotEqual, assertNotContains, assertIsNot, assert_called_with, assertCountEqual, assertLess

Table 4: Mapping between bugs and bug-reproducing tests.

		#	%
One/Multiple	Tests reproducing one bug	613	95%
	Tests reproducing multiple bugs	29	5%
Exclusive/Shared	Tests reproducing exclusive bugs	508	80%
	Tests reproducing shared bugs	134	20%

Despite the similarities, considering the overall comparison, we also find an important difference between the groups. First, 6% of the bug-reproducing tests have at least one try/except block, while this value for all tests is only 2%. Second, despite having an equivalent number of assertions, they are not necessarily the same. To further investigate this potential difference, Table 3 details the top-25 most commonly used assertion commands in both all tests and bug-reproducing tests. We paid special attention to the usage of "weak assertions", that is, assertions that are potentially less effective [33], such as `assertNotEqual`, `assertAlmostEqual`, and `assertContains`. We find 8/25 weak assertions in all tests (from which 2 are exclusives), while 11/25 in bug-reproducing tests (from which 4 are exclusives).

Observation 2: Bug-reproducing tests contain slightly more "weak assertions" (e.g., `assertNotEqual` and `assertContains`) than other tests. 6% of the bug-reproducing tests have try/except blocks.

3.2 RQ2: Bug to Test Mapping

Table 4 shows that 95% of the bug-reproducing tests handle a single bug, while 5% handle multiple bugs. For example, the two bug-reproducing tests presented in Figure 1 focus on a single bug. In contrast, the Django test `test_more_more_more`[6] reproduces six bugs (#10199, #10248, #10290, #10425, #10666, and #10766).

Observation 3: The majority (95%) of the bug-reproducing tests reproduce a single bug; a minority (5%) reproduce multiple bugs.

We also explore whether the bugs are exclusive or shared among the bug-reproducing tests (Table 4). We find that 80% of the tests reproduce exclusive bugs, that is, bugs that are tested only by a

[6]Django: https://github.com/django/django/blob/790f0f8/tests/aggregation_regress/tests.py#L1026

single test. In contrast, 20% of tests reproduce shared bugs, that is, bugs that are tested by multiple tests. For example, the bug #11371 in Django is tested by `test_post` and `test_put`.[7]

Observation 4: Sometimes, multiple tests are needed to reproduce a single bug. We find that 20% of the bug-reproducing tests reproduce shared bugs.

4 Discussion

Improve tests with weak assertions and try/except blocks. Overall, we detected that bug-reproducing tests contain slightly more try/except blocks and "weak assertions" than other tests. Verifying exception-raising via `assertRaises` commands is a better choice than try/except blocks to properly assert that an exception is raised. Moreover, using strong assertions is a better choice to verify the program's output more effectively. Several reasons may lead to the use of such weak assertions and try/except blocks, such as developer inattention or vague user-written bug reports. Another possible explanation is that such tests suffer from poor observability [28]. In this context, tests without assertions may happen due to this lack of observability: when it is hard to observe the program's output, developers cannot write assertions [28]. We hypothesize that some bug-reproducing tests face similar limitations, lacking straightforward methods for observing outputs, which leads to the use of weak assertions or try/except blocks as a workaround. *Future research could explore such tests, proposing solutions to enhance observability and recommending the usage of strong assertions.*

Split multi-bug test into multiple single-bug tests. Ideally, a test should reproduce a single bug so that a failing test indicates the problem [32]. A test that reproduces multiple bugs is harder to understand, and its failure will not clearly address the problem. We found that the majority of tests target a single bug (95%), but some tests target multiple bugs (5%). *This can be an opportunity to support developers (e.g., via contribution study [4, 6, 9]) by investigating the possibility of turning one multi-bug test into multiple single-bug tests.*

Avoid naively replicating bug-reproducing scenarios. While performing our study, we noticed multiple bug-reproducing tests that originated almost entirely from bug-reproducing scenarios reported by users in the issue tracker. For instance, the PyTorch bug #116095[8] presents a bug scenario that is replicated in test `test_cross_entropy_loss`.[9] Similarly, the CPython bug #620179[10] includes a bug scenario that ends up in `test_ipow`.[11] When creating test cases, ideally, developers should strive to minimize tests for bug reproduction, a technique known as *test reduction* [13, 14, 17, 23, 27]. A reduced test can help identify the central problem, allowing developers to spend more time determining the solution [27]. In this context, one potential problem happens when developers copy and paste bug-reproducing scenarios directly into tests. For example, in the two examples provided, neither test included assertions; instead, they simply replicated the bug scenario as reported. This approach misses the opportunity to validate expected outputs and confirm that the bug has been fixed. *Future research could investigate methods to identify this practice and warn developers about potentially less effective bug-reproducing tests.*

[7] Django: https://github.com/django/django/blob/790f0f8/tests/test_client_regress/tests.py#L1083-L1101
[8] https://github.com/pytorch/pytorch/issues/116095
[9] https://github.com/pytorch/pytorch/blob/d1f73fd844/test/test_mps.py#L4860
[10] https://bugs.python.org/issue620179
[11] https://github.com/python/cpython/blob/4767a6e31c055/Lib/test/test_descr.py#L3973

5 Limitations

We analyzed a large set of bug-reproducing tests, but they do not represent all possible bug-reproducing tests in the selected systems. Our heuristic detected bug-reproducing tests that are explicitly admitted by developers in the source code. Therefore, further studies could increase recall and expand the set of bug-reproducing tests.

6 Related Work

Bug reproducing tests are related to multiple aspects of testing research. For example, *databases of real bugs* (e.g., Defects4J [16], BugsInPy [31], and BugSwarm [30]) are typically accompanied by bug-reproducing tests. *Program repair techniques* may rely on failing tests to create patches that make the test suite pass [15, 19]. Creating a bug-reproducing test is also part of the process of *test reduction* [17, 23], a technique that aims to minimize test cases for bug reproduction. The importance of test reduction is highlighted in projects such GCC [13], LLVM [14], and WebKit [27]. In those studies, the main focus is on the buggy/fixed code, while bug-reproducing tests are present to support that the fixed code works properly. Lastly, bug reproducing tests are also related to a set of studies in the context of testing negative and exceptional tests [5, 10, 11, 18] and may support a better understanding of them.

7 Conclusions and Future Work

We provided an initial empirical study to understand bug-reproducing tests better. Overall, we found that bug-reproducing tests are not different from other tests regarding LOC, number of assertions, and complexity, but they contain slightly more try/except blocks and "weak assertions". We also found that the majority (95%) of the bug-reproducing tests reproduce a single bug, while 5% reproduce multiple bugs. We concluded by discussing multiple possibilities to improve bug-reproducing tests.

Future Work: First, this study assesses syntactic differences between bug-reproducing tests and others. Future work could look at semantic properties, e.g., do they differ in terms of coverage achieved per test or mutants killed? Second, understanding the differences between bug reproducing and "regular" tests is also important to automatically generate such tests, e.g., by developing adequate prompts that reflect these properties when using LLMs to generate tests [1, 8, 12, 21, 24]. Finally, as testing is generally well adopted in the industry, some companies invest huge amounts of resources into test execution infrastructure. In this context, one important question is whether bug-reproducing tests and other tests need to be executed at the same frequency, which may lead to a potential reduction of cost.

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] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated unit test improvement using large language models at meta. In *International Conference on the Foundations of Software Engineering*. 185–196.

[2] Black Contribution Guidelines. October, 2025. https://black.readthedocs.io/en/latest/contributing/the_basics.html.

[3] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the Factors that Impact the Popularity of GitHub Repositories. In *International Conference on Software Maintenance and Evolution*.

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

[5] 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.

[6] 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.

[7] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 341–350.

[8] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. *arXiv preprint arXiv:2310.03533* (2023).

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

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

[11] 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.

[12] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* (2023).

[13] How to Minimize Test Cases for Bugs. October, 2025. https://gcc.gnu.org/bugs/minimize.html.

[14] How to submit an LLVM bug report. October, 2025. https://llvm.org/docs/HowToSubmitABug.html.

[15] Yang Hu, Umair Z Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-factoring based program repair applied to programming assignments. In *International Conference on Automated Software Engineering*. 388–398.

[16] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis*. 437–440.

[17] Patrick Kreutzer, Tom Kunze, and Michael Philippsen. 2021. Test Case Reduction: A Framework, Benchmark, and Comparative Study. In *International Conference on Software Maintenance and Evolution (ICSME)*. 58–69.

[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] Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.

[20] Next.js Contribution Guidelines. October, 2025. https://github.com/vercel/next.js/blob/canary/contributing/core/testing.md.

[21] Wendkûuni C Ouédraogo, Yinghua Li, Kader Kaboré, Xunzhu Tang, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F Bissyandé. 2024. Test smells in LLM-Generated Unit Tests. *arXiv preprint arXiv:2410.10628* (2024).

[22] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *International Conference on Software Maintenance and Evolution*. IEEE, 91–100.

[23] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *SIGPLAN conference on Programming Language Design and Implementation*. 335–346.

[24] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[25] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. 2019. A survey of self-admitted technical debt. *Journal of Systems and Software* 152 (2019), 70–82.

[26] Hudson Silva and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129.

[27] Test Case Reduction. October, 2025. https://webkit.org/test-case-reduction/.

[28] Tests without assertions: why do they happen? October, 2025. https://www.effective-software-testing.com/tests-without-assertions-why-do-they-happen.

[29] David Thomas and Andrew Hunt. 2019. *The Pragmatic Programmer: your journey to mastery*. Addison-Wesley Professional.

[30] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *International Conference on Software Engineering (ICSE)*. 339–349.

[31] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. BugsInPy: A database of existing bugs in Python programs to enable controlled testing and debugging studies. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1556–1560.

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

[33] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2023. Mutation Analysis. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/html/MutationAnalysis.html