A Trend Analysis of Test Smells in Python Test Code Over Commit History

Yuki Fushihara*, Hirohisa Aman†, Sousuke Amasaki‡, Tomoyuki Yokogawa‡ and Minoru Kawahara†

*Department of Computer Science, Faculty of Engineering, Ehime University
Matsuyama, Ehime, 790–8577 Japan
†Center for Information Technology, Ehime University
Matsuyama, Ehime, 790–8577 Japan
‡Faculty of Computer Science and Systems Engineering, Okayama Prefectural University
Soja, Okayama, 719–1197 Japan

***Abstract*—Software testing is an essential activity for developing and maintaining high-quality software. Unit testing with test code (test cases) is a fundamental testing activity, and developers can test their production code whenever they create or modify the code. However, such quality assurance relies on the correctness of the test code. If a test code had a flaw, it would mislead the developers about the hidden faults and prevent early detection of the faults. This paper focuses on "test smells," which may cause test code flaws in Python programs, and analyzes their changing trends over commit history (code changes) toward better Python test code management. Through an empirical data analysis of 100 open-source projects, the paper reports the following findings: (1) a few kinds of test smells constitute the majority of smells detected in the studied projects, and (2) most kinds of smells tend to increase over commits, i.e., many test smells are likely to have remained in test code as technical debt.**

***Index Terms*—unit testing, test smell, technical debt, commit history, trend analysis**

I. Introduction

Unit testing is the most fundamental software testing activity that can detect latent faults as early as possible during software development and maintenance. The unit testing framework (e.g., xUnit [1]) provides a helpful mechanism for automatically testing software modules. Under a unit test framework, developers prepare test code for their production code and can quickly test their products whenever they newly create or modify their production code [2]. Such an automated test environment can assist developers in developing and maintaining their programs successfully.

However, the quality assurance of software modules using an automated testing environment with unit test code depends on the reliability of the prepared test code. A test code is also a source code written by a human being and has a risk of testing the production code improperly. In other words, some test code might overlook latent faults in the production code and mislead the developers with erroneous test results. For example, if a test code has a conditional branch statement, there is a risk that the test code does not test the production code properly because the test engineer might write a flawed condition in the branch statement. Such a risky feature is a kind of "code smells" [3] in the test code.

Although such smells in test code may not directly cause improper testing, they can become a technical debt for the software product. Thus, it is ideal for detecting problematic test code and performing refactorings to reduce the risk of unreliable tests. Some undesirable features of test code have been studied as "test smells" [4] in the past. Recently, Wang et al. [5] developed a tool for detecting 18 different kinds of test smells in Python test code and reported that a few kinds of test smells are likely to appear in many test cases. While their tool is helpful and their reports are noteworthy, the changing trend of test smells has not been analyzed and discussed well. In other words, although their reports showed which kind of test smells are frequently detected, they missed analyzing the trend of test smells through the development and maintenance activities.

For example, suppose we observe a test smell that tends to increase through code changes. It would be a kind of test smell that developers and test engineers are likely to overlook, and the smelled test code would survive for an extended period as a potential risk of causing unreliable tests. We conducted a large-scale data analysis of test smell trends in 100 open-source Python projects to explore test smells from the time series perspective. In this paper, we report our data analysis and discuss the results to present fundamental data of test smell changing trends toward adequate test code management.

II. Test Smells in Python

In this section, we briefly describe the Python unit testing framework of interest and the concept of test smells. Then we discuss the related work and our research motivation.

A. Unit Testing Framework and Test Smells

There have been some unit testing frameworks to automatically perform unit testing of Python programs and report the testing results. We focus on *unittest* in this paper because it is one of the most popular and fundamental testing frameworks for Python. Fig. 1 presents a simple example of Python test code based on *unittest*, which tests a function "`fizz_buzz`." A test case is a subclass of "`unittest.TestCase`," and it has some test methods whose names begin with "`test`."

```python
import unittest
class TestFizzBuzz(unittest.TestCase):
    def test_fizz_buzz(self):
        for value in range(1,31):
            if value in [3,6,9,12,18,21,24,27]:
                expected = "Fizz"
            elif value in [5,10,20,25]:
                expected = "Buzz"
            elif value in [15,30]:
                expected = "Fizz Buzz"
            else:
                expected = str(value)
            actual = fizz_buzz(value)
            self.assertEqual(expected, actual)
```

Fig. 1. An example of test case having test smells.

The unit test runner executes those test methods and reports the results. Each test method runs the production code under test and checks if it behaves as expected using the assertion methods provided by the framework. In the example of Fig. 1, the "`assertEqual`" method checks if the actual value equals the expected one.

Whenever developers create a new Python program or modify an existing one, they can automatically test those new or modified programs using the test cases under the unit testing framework. Such an automated testing environment helps developers detect latent faults quickly. However, it should be noted that the successful cycle of coding and testing depends on the test cases' reliability. Suppose some of the prepared test cases are incorrect and overlook some faults. In that case, the program under test may pass all tests even if it is buggy, and the test results would mislead the developers while giving a false sense of reassurance. Because a unit test case is also a program written by a human being, there is always a risk of producing an erroneous test code [6].

Some problematic characteristics of source code to be refactored have been studied as "code smells" [3] in the past. Although source code with a smell does not always cause trouble in development and maintenance, it has a higher risk of becoming buggy or hard-to-understand code and can be a technical debt. Because test code is also source code, a similar concept is applicable to test code and is referred to as "test smell" [4]. When a test code has a test smell, it is better to refactor it to remove the smell to enhance the test's reliability toward the successful development of high-quality production code.

B. Related Work and Research Motivation

Deursen et al. [4] first introduced the concept of test smells. They focused on the difference in refactoring between production code and test code. Through an investigation of the test code refactoring, Deursen et al. discovered that test code also tends to have code smells that differ from the well-known code smells in production code, and then they presented 11 test smells and the refactorings for those smells. After that, Meszaros et al. [1] and Greiler et al. [7] ramped up the kind of test smells. Spandini et al. [8] reported that test smells might adversely affect both the test and production codes, as follows: (1) the test code with a test smell is more change-prone and fault-prone than the other test code, and (2) the production code tested by a smelled test code is also more fault-prone than the other production code. Hence, detecting test smells and resolving their problems as early as possible is crucial for successful software development and maintenance.

TABLE I
TEST SMELLS SUPPORTED BY PYNOSE

Test Smell	Description
Assertion Roulette	An AS has no explanation/message in a TC.
Conditional Test Logic	A control statement—if, for, while—is used.
Constructor Initialization	A constructor is declared in a TS.
Default Test	A TS is called `MyTestCase`.
Duplicate Assert	Two or more ASs have the same parameters.
Empty Test	No executable statement appears in a TC.
Exception Handling	A `try/except` or `raise` statement is used.
General Fixture	Not all fields instantiated within `setUp()` of a TS are utilized by all TCs.
Ignored Test	A `@unittest.skip` decorator is used.
Lack of Cohesion of Test Cases	The mean of cosine similarities of all TC pairs ≤ 0.4.
Magic Number Test	A numeric literal is used as an argument of an AS.
Obscure In-Line Setup	A TC declares ten or more local variables.
Redundant Assertion	An AS has a condition that is always true, e.g., `assertEqual(X,X)`.
Redundant Print	`print()` is used in a TC.
Sleepy Test	`time.sleep()` is used with no comments.
Suboptimal Assert	A suboptimal assert is used in a TC.
Test Maverick	A TC does not use any field from `setUp()`.
Unknown Test	No AS is used in a TC.

(AS: assertion statement, TC: test case, TS: test suite)

There have been studies of test smell detection and automated detection tools [9]–[12]. Recently, to support Python programs, Wang et at. [5] developed a test smell detection tool, *PyNose*, to distill 18 kinds of test smells (see Table I) from test code automatically. Through a case study examining open-source projects, they empirically showed PyNose successfully detects test smells and reported some test smells, such as "Assertion Roulette," "Conditional Test Logic," and "Magic Number Test," tend to appear in many Python test code. For instance, the test case shown in Fig. 1 has a smell categorized as "Conditional Test Logic" because it contains conditional branches. Although its risk of making mistakes is low because of the code simpleness, it would be better to describe assertion methods corresponding to straightforward test cases such as `(3, "Fizz")`, `(4, "4")` and `(5, "Buzz")`, for making the test case more readable and inerrable.

Although previous studies on test smells have presented valuable suggestions and practical tools, their main focuses were on detecting and refactoring test smells, and "the changes in test smell trends" have not been well-analyzed to the best of our knowledge. For example, suppose the number of test cases with a specific test smell tends to increase in a software development project through its code changes (commits on the repository). Such a test smell may keep staying and growing in the test code as a technical debt for the project without the developers being aware of it. Hence, we consider the changes in test smell trends are also worthwhile to analyze for successful technical debt management, and this is our research motivation in this paper.

III. Data Analysis

As mentioned in Section II, previous studies on test smells have mainly focused on detecting and refactoring the test smells but have not analyzed the changes in smell trends well. Thus, we conducted a large-scale investigation of test smells in 100 Python open-source projects and quantitatively analyzed their trends over commit histories. We report our investigation and the analysis results in this section.

A. Aim

We tackle the following two research questions (RQs).

- RQ1: What test smells will likely appear in Python test code? Do the trends differ among projects?
- RQ2: How does each test smell's trend change over commits? Do the trends differ among projects?

RQ1 is a fundamental question regarding test smells. To answer RQ1, we collect test smell data from various open-source projects. The investigation results would present basic data for discussing effective preventive measures against quality deterioration caused by test smells. RQ2 is a further and our primary question by introducing a time perspective: although RQ1 focuses on test smells detected in a snapshot of test code, RQ2 considers their changes over time (commit histories). As mentioned above, we will analyze the changes in test smell trends toward successful technical debt management.

B. Studied Projects and Data Analysis Procedure

We collected test smell data from 100 Python open-source projects[1], satisfying the following six conditions to examine active projects [13]: (i) Their "stars" scores are higher than 50; (ii) They have modified not only production code but test code through commits; (iii) Their repositories have more than 1000 commits; (iv) They have more than ten contributors; (v) Their developments have lasted more than two years; (vi) They have had one or more commits within the last twelve months.

We performed the following steps for each project to collect and analyze test smell data.

(1) Make a list of commits. Clone the Git repository and list all commits by checking the commit logs.

(2) Detect test smells. Check out each commit from the repository and run PyNose to detect test smells in that version of the test programs.

(3) Analyze the collected data for answering RQ1. Count the detected test smells by smell type (see Table I) in the latest version and compare their appearance patterns among projects.

(4) Analyze the collected data for answering RQ2. Count the detected test smells by smell type in each version (commit) and make their moving averages (window size = 50 commits) to capture their change trends. Then compute the correlation coefficient (Spearman's ρ) between the appearance counts and commit times for each smell type to understand its increasing/decreasing trend. Moreover, compare the correlation coefficients among projects.

[1]The project list is available from our supplemental data site, https://se.cite.ehime-u.ac.jp/data/Fushihara_SEAA2023/ .

TABLE II
RELATIVE FREQUENCIES OF TEST SMELLS IN ALL PROJECTS

Rank	Test Smell	Relative Frequency
1	Assertion Roulette	23.6%
2	Conditional Test Logic	13.0%
3	Magic Number Test	11.7%
4	General Fixture	10.1%
5	Unknown Test	8.6%
6	Duplicate Assertion	7.4%
7	Test Maverick	5.0%
8	Exception Handling	5.0%
9	Lack Cohesion	4.9%
10	Suboptimal Assert	4.7%
11	Obscure In Line Setup	1.9%
12	Redundant Print	1.7%
13	Ignored Test	0.9%
14	Sleepy Test	0.8%
15	Redundant Assertion	0.4%
16	Empty Test	0.2%
17	Constructor Initialization	0.1%
18	Default Test	0.0%

Test smell counts may fluctuate after adding new test code because some test engineers notice and refactor test smells. We adopted the moving average method because we aim to obverse the rough increasing/decreasing trends in commit histories. We empirically set the window size to 50 to filter out short-term fluctuations.

C. Results and Discussions on RQ1

We begin with reporting the results regarding RQ1. Table II presents the relative frequencies of test smell appearance counts in all the latest test programs of all studied projects. The table shows that "Assertion Roulette" is the most frequently appearing smell, and "Default Test" did not appear at all. The distribution of test smells shown in Table II is close to the results reported in the previous study [5]. The Spearman's ρ between our result and the previous one is 0.98, and we consider that we could replicate the empirical results successfully.

Next, we compared the test smell distributions among projects: For each project, we regarded the project's relative frequencies of test smells as the 18-dimensional vector and computed their cosine similarities. Fig. 2 shows the histogram of cosine similarities; the median and mean values were 0.8990 and 0.8287. Hence, we can say that the test smell distributions are similar among most projects. We also examined six projects with the lowest similarities, i.e., those with different smell distributions. As a result, five out of six

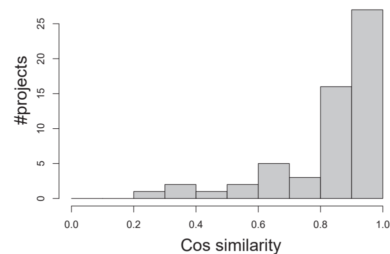


Fig. 2. Histogram of cosine similarities among projects.

TABLE III
PERCENTAGES OF PROJECTS CORRESPONDING TO TRENDS OF TEST SMELL CHANGES THROUGH COMMITS

Test Smell	$\rho < -0.6$	$-0.6 \leq \rho < -0.2$	$-0.2 \leq \rho < 0.2$	$0.2 \leq \rho < 0.6$	$\rho \geq 0.6$	NA
Assertion Roulette	9.2%	10.5%	6.6%	7.9%	**57.9%**	7.9%
Conditional Test Logic	7.9%	10.5%	6.6%	10.5%	**56.6%**	7.9%
Magic Number Test	7.9%	7.9%	7.9%	13.2%	**46.1%**	17.1%
General Fixture	3.9%	15.8%	5.3%	9.2%	**47.4%**	18.4%
Unknown Test	7.9%	7.9%	13.2%	14.5%	**44.7%**	11.8%
Duplicate Assertion	5.3%	10.5%	5.3%	5.3%	**55.3%**	18.4%
Test Maverick	6.6%	13.2%	5.3%	7.9%	**43.4%**	23.7%
Exception Handling	7.9%	7.9%	11.8%	11.8%	**38.2%**	22.4%
Lack Cohesion	2.6%	17.1%	10.5%	13.2%	**39.5%**	17.1%
Suboptimal Assert	5.3%	10.5%	6.6%	7.9%	**43.4%**	26.3%
Obscure In Line Setup	3.9%	7.9%	11.8%	10.5%	22.4%	**43.4%**
Redundant Print	0.0%	2.6%	11.8%	14.5%	22.4%	**48.7%**
Ignored Test	3.9%	2.6%	6.6%	3.9%	15.8%	**67.1%**
Sleepy Test	2.6%	5.3%	10.5%	10.5%	15.8%	**55.3%**
Redundant Assertion	2.6%	6.6%	13.2%	5.3%	14.5%	**57.9%**
Empty Test	5.3%	3.9%	21.1%	3.9%	11.8%	**53.9%**
Constructor Initialization	1.3%	5.3%	1.3%	3.9%	2.6%	**85.5%**
Default Test	0.0%	0.0%	1.3%	0.0%	0.0%	**98.7%**
Average	4.7%	8.1%	8.7%	8.6%	32.1%	37.9%

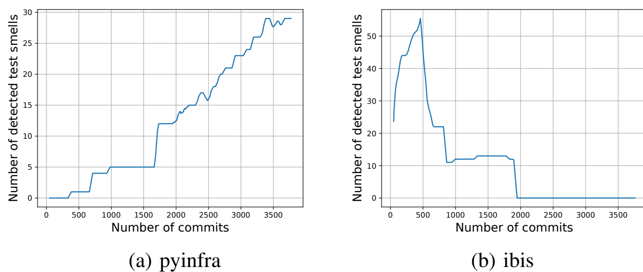
(NA: the projects have no test smell corresponding to the row.)

projects (workload-automation, scvi-tools, pudb, rosdistro, and magic-wormhole) had fewer test smells, which may be the reason why they showed dissimilar distributions. The remaining project—remi—had a project-specific trend: "General Fixture" appeared three times as many as "Assertion Roulette." No other projects showed such a project-specific difference.

From the above results, we answer RQ1 as follows. The most frequently appearing (top 3) test smells in Python would be "Assertion Roulette," "Conditional Test Logic," and "Magic Number Test," and they share about half of all test smells. The trend of test smell seems to be common to most projects. Because these smells are related to how to write assertions or issues on test code readability, it is efficient to consider the preventive measures for them to control technical debt.

D. Results and Discussions on RQ2

We focused on the changes in test smell appearances over commits and collected the trend data (moving averages) from the studied projects. Fig. 3 presents the changing trends of "Assertion Roulette" in pyinfra and ibis projects. While we can see a monotonically increasing trend of the test smell over commits in pyinfra project (see Fig. 3(a)), we also observe an opposite trend in ibis project (see Fig. 3(b)). We obtained those



(a) pyinfra (b) ibis

Fig. 3. Change trends of number of detected test smells corresponding to "Assertion Roulette" in pyinfra and ibis projects.

trend data for all combinations of the available test smells and the studied projects. Then we computed Spearman's ρ to capture their increasing/decreasing trends quantitatively. For instance, we obtained $\rho = 0.995$ and $\rho = -0.899$ for the trend data of "Assertion Roulette" in pyinfra and ibis projects (Fig. 3(a) and (b)), respectively; these correlation coefficients indicate a strong increasing trend and a strong decreasing one. Table III presents the results of all test smells. In the table, the highest percentage within a test smell (a row) is emphasized by boldface. For example, "Assertion Roulette" had a strong increasing trend ($\rho \geq 0.6$) in 57.8% of projects[2]. On the other hand, there are also projects having decreasing trends of test smells ($\rho < -0.2$), although they are a minority ($4.7 + 8.1 = 12.8\%$). We can see that the highest percentage categories of projects are "$\rho > 0.6$," i.e., increasing trends for the most appearing ten kinds of test smells—Assertion Roulette, Conditional Test Logic, . . . , Suboptimal Assert—in Table III. Although the remaining eight kinds of test smells tend not to appear in projects, six out of eight showed that "$\rho > 0.6$" are the highest percentage categories within the projects where those smells appeared. Hence, we can say most test smells tend to increase over commits. It would be adequate to detect and refactor those frequently appearing test smells as early as possible to avoid the risk of performing unreliable tests.

Interestingly, 7.9–9.2% of projects also had decreasing trends ($\rho < 0.6$) for the most appearing three kinds of test smells—Assertion Roulette, Conditional Test Logic, and Magic Number Test. Although those projects are the minority, some developers or test engineers of the projects might pay attention to test smells. On average, while 40.7% (= 8.6% + 32.1%) of projects showed increasing trends of test smells,

[2]Since no test smell was detected by PyNose throughout the commits in 24 out of 100 projects, the presented values are the percentages in the 76 projects where at least one test smell had appeared.

12.8% of projects had decreasing trends, so that the test smell changing trend may differ among projects.

From the above results, we answer RQ2 as follows. Test smells tend to increase through commits in Python projects. The potential risks caused by test smells would grow without preventive measures, and it is vital to detect and resolve test smells as early as possible. Although those trends may differ among projects, the most frequently appearing ten kinds of test smells have increasing trends in 39.5–57.9% of the studied projects.

Notice that the above analysis has focused on the number of detected test smells but not their ratios to the number of test cases or the lines of test code. In this study, we have paid attention to the existence of test smells because their presence can cause a risk of unreliable tests. On the other hand, it would also be worthwhile to analyze "the rate" of each kind of test smells in a project, like a bug (fault) rate discussed in the fault-prone module analysis studies, because the test code would also evolve with the production code through commits. We plan to do a further analysis focusing on the test smell rate to obtain a deeper insight into the test smell trends as our significant future work.

E. Threats to Validity

We empirically set the window size of the moving average to 50 in our data analysis. Since a different window size can present a different trend data of test smell, the window size selection can threaten our construct validity. Although we have used the window size to filter out short-term fluctuations, it would be better to utilize an advanced time series analysis method for mitigating the threat, and we also plan to perform it in our future work.

Another threat to our construct validity is the test smell detection tool we used in this study because we have used only the PyNose to detect smells. There might be a tool-specific potential bias in the test smell detection. For example, there might be a kind of test smells that PyNose is likely to miss, but another tool can detect it. We also need to use other state-of-the-art tools [14] to mitigate the threat for further smell data analysis.

Our data analysis results are derived from only Python open-source projects, which can threaten the external validity of this study. Although we can expect that similar trends of test smells may be observed in other language projects because the concept of unit testing is common to other languages, we need to conduct further data collection and analysis to generalize our results to test programs in other languages.

IV. Conclusion and Future Work

We have focused on "test smells," i.e., code smells in test code, which may adversely affect both production and test code. Although the previous studies presented valuable test smell detection and refactoring methods, they missed analysis of their changing trends through commits. In this paper, we investigated 100 open-source Python projects and analyzed trend data of test smells. As a result, we obtained the following findings: (1) a few kinds of test smells—Assertion Roulette, Conditional Test Logic, and Magic Number Test—constitute the majority of smells detected in the studied projects, and (2) most kinds of smells tend to increase over commits, i.e., many test smells are likely to have remained in test code as technical debt. Since developers and test engineers may overlook those kinds of test smells, detecting and refactoring them as early as possible is helpful for successful test code management. In particular, detecting the above frequently appearing smells and giving a warning to test engineers would be an efficient preventive measure against test-related technical debt.

Our future work includes (i) detailed analyses of what causes the test smell growth and of how the test smell growth affects the quality of production code; (ii) an analysis of human aspects of the test smells, e.g., on the question "Is a test smell specific to a particular test engineer?"; (iii) further data analysis using another state-of-the-art detection tool [14] that can detect other kinds of test smells, such as the "Rotten Green Test."

Acknowledgment

This work was supported by JSPS KAKENHI #20H04184, #21K11831, #21K11833, and #23K11382.

References

[1] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Boston, MA: Addison-Wesley Professional, 2007.

[2] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production & test code," in *Proc. 1st Int. Conf. Softw. Testing, V. & V.*, Apr. 2008, pp. 220–229.

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[4] A. V. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proc. 2nd Int. Conf. Extreme Programming & Flexible Processes in Softw. Eng.*, May 2001, pp. 92–95.

[5] T. Wang, Y. Golubev, O. Smirnov, J. Li, T. Bryksin, and I. Ahmed, "Pynose: A test smell detector for python," in *Proc. 36th Int. Conf. Automated Softw. Eng.*, Nov. 2021, pp. 593–605.

[6] C. Jones, *Applied Software Measurement: Global Analysis of Productivity and Quality*, 3rd ed. New York: McGraw-Hill, 2008.

[7] M. Greiler, A. Van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *Proc. 6th Int. Conf. Softw. Testing, V. & V.*, Mar. 2013, pp. 322–331.

[8] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *Proc. Int. Conf. Softw. Maintenance & Evolution*, Sep. 2018, pp. 1–12.

[9] A. Peruma, K. S. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory study," in *Proc. 29th Int. Conf. Comp. Sc. & Softw. Eng.*, Nov. 2019, pp. 193–202.

[10] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "tsDetect: an open source test smells detection tool," in *Proc. 28th Joint European Softw. Eng. Conf. & Symp. Foundations Softw. Eng.*, Nov. 2020, pp. 1650–1654.

[11] S. Lambiase, A. Cupito, F. Pecorelli, A. De Lucia, and F. Palomba, "Just-in-time test smell detection and refactoring: The darts project," in *Proc. 28th Int. Conf. Program Comprehension*, May 2020, pp. 441–445.

[12] T. Virgínio, L. Martins, L. Rocha, R. Santana, A. Cruz, H. Costa, and I. Machado, "Jnose: Java test smell detector," in *Proc. 34th Brazilian Symp. Softw. Eng.*, Oct. 2020, pp. 564–569.

[13] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proc. 11th Working Conf. Mining Softw. Repositories*, May 2014, pp. 92–101.

[14] F. Maier and M. Felderer, "Detection of test smells with basic language analysis methods and its evaluation," in *Proc. IEEE Int. Conf. Softw. Analysis, Evolution & Reeng.*, Mar. 2023, pp. 897–904.