Fault-Proneness of Python Programs Tested By Smelled Test Code

Yuki Fushihara
Graduate School of Sc. & Eng.
Ehime University
Matsuyama, Japan

Hirohisa Aman
Center for Information Technology
Ehime University
Matsuyama, Japan
aman@ehime-u.ac.jp

Sousuke Amasaki
Faculty of Comp. Sc. & Systems Eng.
Okayama Prefectural University
Soja, Japan
amasaki@cse.oka-pu.ac.jp

Tomoyuki Yokogawa
Faculty of Comp. Sc. & Systems Eng.
Okayama Prefectural University
Soja, Japan
t-yokoga@cse.oka-pu.ac.jp

Minoru Kawahara
Center for Information Technology
Ehime University
Matsuyama, Japan
kawahara@ehime-u.ac.jp

***Abstract*—Software testing is one of the most crucial quality assurance activities, and test results are of great concern to software developers. However, the quality assurance of the test code (test case) itself also becomes critical because a poor-quality test case may fail to detect latent faults and give developers false comfort regarding the test result. A code smell threatening test code quality has been studied as "test smell." This paper conducts an investigation of test smells in 775 Python open-source programs and reports the results of a quantitative analysis regarding whether test smells impact the fault-proneness of the product code under test. The analysis results show the following two findings. (1) When a test code has one of the reported ten kinds of test smells, the production code under test is more fault-prone than the others. (2) The fault-proneness of a production code tends to get higher when the corresponding test code has two or more different kinds of test smells—over 75% of test smell combinations showed such a trend of increasing the risk of being faulty production code.**

***Index Terms*—unit testing, quality of test code, test smell, fault-proneness**

I. Introduction

Unit testing is a fundamental quality assurance activity for testing software modules. Because developers can perform unit testing as soon as they develop a new module or modify an existing one, it would contribute to detecting latent faults as early as possible during their coding and maintenance activities [1], [2]. To this end, developers often prepare test programs under a unit testing framework, e.g., xUnit [3], as their unit test cases. A unit testing framework provides an automated testing environment in which we can automatically run all test cases and obtain their testing results. Hence, by using various test cases under a unit testing framework, developers can develop and maintain their programs (production code) while testing them continuously. It is an effective software development method known as test-driven development [4].

However, it's crucial to recognize that test programs (test code) are also human-written source programs. This means that the test program itself carries a risk of being faulty and may improperly test its target production code. Test results from defective test programs are unreliable, as they may fail to detect latent faults in the production code. These unreliable test results can give developers a false sense of security, potentially leading to the release of a faulty product. Although the quality of the test code itself is likely to go unnoticed, it is also a noteworthy point of view for successful software development.

In code refactoring studies, the signs of potential problems we may refactor to improve code quality are called "code smells [5]." Code smells may also appear in test code and have been studied as "test smells [6]" in the past. For example, as Python has been becoming one of the most popular programming languages, Wang et al. recently proposed test smells in Python unit testing code and developed the detection tool, PyNose [7]. Although there are previous studies regarding test smells, their primary focus is on detecting smells [3], [8]–[11], [11], [12]. To the best of the authors' knowledge, there have been a few empirical studies of the relationship between the test smells and the production code quality in the past: Spadini et al. [13] analyzed Java production code's fault-proneness (defect-proneness) using six test smells. Therefore, we focus on test smells in another popular programming language, Python, and report a large-scale investigation of production code's fault-proneness using more kinds of test smells, i.e., 18 test smells detectable by PyNose, in this paper.

The remainder of this paper is organized as follows. Section II describes test smells in Python. Then, Section III reports our investigation and discusses the results. Finally, Section IV presents the conclusion of this paper and our future work.

II. Test Smells in Python

We briefly describe the previous work on test smells. Then, we present the Python test smells of interest in this study.

A unit testing framework offers an automated testing environment in which developers can test their production code whenever they add new code or modify an existing one. Such

an environment is helpful for effective quality assurance of production code under development. However, if the test case (test code) is defective, the test results are unreliable and deliver a false sense of security to the developers. Because the test code is also a program written by human developers, there is always a risk that test code threatens the reliability of test results.

Deursen et al. [6] introduced the notion of "test smells" and proposed eleven different smells and methods for refactoring them. Meszaros [3] and Greiler et al. [8] enriched the test smell studies by proposing other kinds of smells. Then, researchers developed test smell detection tools for some programming languages and their unit testing frameworks [9]–[12]. Although the basic concept of test smells is common to any programming language, there is variation in test smells among languages because different programming languages may have different characteristics. In other words, while a certain kind of test smell often appears in a programming language, the smell might not or cannot appear in another language. For example, we sometimes see a test smell called "Resource Optimism" [6] in Java test code, corresponding to the case that the test program uses a `File` object without checking its existence. Even if the file of interest did not exist, we could obtain an instance of `File` class, which might cause a defect in the test code. On the other hand, in Python, it is standard to open a file and associate it with a variable[1] before using the variable linked to the file resource. Hence, "Resource Optimism" may not become a noteworthy smell in Python [7].

From the perspective of language-specific characteristics related to test smells, Wang et al. [7] carefully selected 17 kinds of test smells from the conventional ones and added a new smell "Suboptimal Assert" for Python. Table I presents those 18 test smells. Furthermore, Wang et al. developed a test smell detection tool, PyNose. PyNose can detect those test smells from a test code under Python's `unittest` unit testing framework.

Because test smells may threaten the reliability of test code and increase the risk of overlooking latent faults, the fault-proneness of the production code tested by smelled test code might be higher than that of other production code. If we show such an increasing trend of fault-proneness and what kind of test smells are noteworthy, it will help drive successful software quality management from the perspective of test smells. Indeed, Spadini et al. [13] reported an empirical study showing the relationships between faulty Java production code and test smells. However, to the best of our knowledge, an empirical study regarding the fault-proneness of Python production code tested by smelled code has not been reported in the literature. Thus, we will utilize PyNose to detect test smells in Python test code and analyze the fault-proneness of production code under test in this paper.

III. Quantitative Investigation

We collected Python programs from 50 open-source software projects and conducted a quantitative investigation regarding the fault-proneness of production code tested by smelled test code. In this section, we report the results of our investigation and data analysis.

[1] `https://docs.python.org/3/library/filesys.html`

TABLE I
Test smells supported by PyNose

No.	Test Smell
S_1	**Assertion Roulette** Two or more assertion statements in a test case have no message for their failures [6].
S_2	**Conditional Test Logic** A conditional statement appears in a test case [10].
S_3	**Constructor Initialization** A test case is initialized without calling `setUp()` [7], [10].
S_4	**Default Test** The test suite's name is `MyTestCase` (the default one) [7], [10].
S_5	**Duplicate Assert** Two or more identical assertions appear in a test case [10].
S_6	**Empty Test** No executable statement appears in a test case [10].
S_7	**Exception Handling** An exception handling does not use `assertRaises()` [7], [10].
S_8	**General Fixture** Not all fields instantiated within `setUp()` are used by all test cases [7], [10].
S_9	**Ignored Test** A test case is ignored by `@unittest.skip` [7], [10].
S_{10}	**Lack of Cohesion of Test Cases** The mean cosine similarity of all test case pairs ≤ 0.4 [7], [8].
S_{11}	**Magic Number Test** A magic number (numerical literal) appears in a test case [10].
S_{12}	**Obscure In-Line Setup** A test case has ten or more local variables [8].
S_{13}	**Redundant Assertion** A test case has an assertion whose condition is always true [10].
S_{14}	**Redundant Print** A test case calls `print()` [7], [10].
S_{15}	**Sleepy Test** A test case calls `time.sleep()` [7], [10].
S_{16}	**Suboptimal Assert** An assertion statement could be replaced with another one that is more suitable for testing [7], e.g., `assertTrue()` is more suitable for testing a Boolean value than `assertEqual()`.
S_{17}	**Test Maverick** A test case does not use a class field from `setUp()` [7], [8].
S_{18}	**Unknown Test** No assertion statement appears in a test case [9].

A. Research Questions

We analyzed the collected data under the following two research questions (RQs):

RQ1: Does a test smell in the test code affect the fault-proneness of the production code under test?

RQ1 is our fundamental question regarding the impact of test smells on the production code's quality. We examine whether the production code tested by smelled test code is more fault-prone. Moreover, we focus on the fault-proneness with respect to each smell to see the difference in their impacts. Although we can consider 18 kinds of test smells, their impacts would vary from smell to smell. We aim to reveal the importance level of each test smell to be alerted.

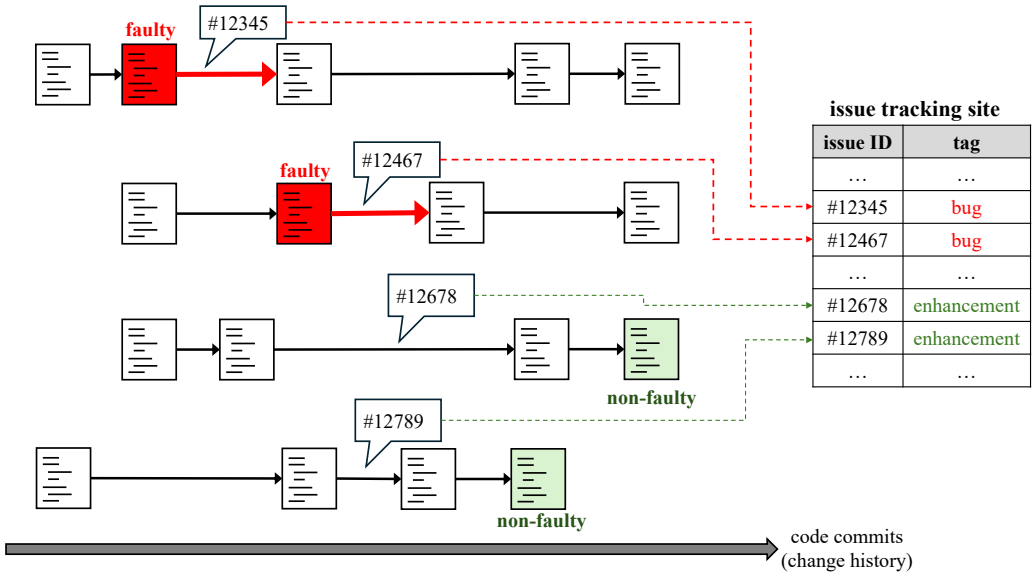

Fig. 1. Decision of "faulty" and "non-faulty" production code.

RQ2: How do combinations of test smells affect the fault-proneness of the production code?

RQ2 is a further question in which we expanded our focus to the co-occurrence of test smells rather than a single smell. Because we often see some different test smells simultaneously occur in a test case, we will examine if such a combination of smells increases the risk of being an unreliable test or not.

B. Dataset

We collected data on production and test code from 50 Python open-source software projects according to the previous study by Wang et al. [7]. We describe how we prepared our dataset in this subsection.

We collected 50 repositories of Python open-source projects available at GitHub (see our supplementary site[2] for the project list), satisfying the following conditions:

1) Their "stars" scores are higher than 50.
2) They have modified not only production code but test code through commits.
3) Their repositories have more than 1,000 commits.
4) They have more than ten contributors.
5) Their developments have lasted more than two years.
6) They have had one or more commits within the last twelve months.

We set the above criteria to pick up actively maintained projects based on the work by Kalliamvakou et al. [14] and our experience.

For each project, we made a local copy of the repository (cloned the repository). Then, we traced each production code's (source file's) change history and checked whether it had experienced a "bugfix" commit. If a source file was changed through a bugfix commit, we consider the previous revision (just before the bugfix) of the source file to be "faulty" production code (see Fig. 1). On the other hand, if a source file has not experienced any bugfix commit, we regarded its latest revision as a "non-faulty" one.

To decide whether a commit is a bugfix one, we focused on the issue ID described in the commit message. We considered the aim of a commit to be a bugfix if the corresponding issue ID was found at its issue management site and had a fault-related tag such as "bug," "defect," and "fault."

After extracting faulty or non-faulty production code, we checked if there is a test code that test the production code. Then, we examined each test code using PyNose and detected test smells. Notice that we had to focus only on the test code that was written under `unittest` unit testing framework to analyze its test smells by PyNose. In other words, we had to exclude the test code if they were written under another testing framework because we cannot evaluate their test smells in this study.

As a result, we obtained a dataset consisting of data items of 775 production code (source files) tested by one or more test cases under `unittest` unit testing framework. Each data item gives information about whether the production code was faulty and which kinds of test smells occurred in the test code testing the production one. Our dataset is available from our supplementary site.

C. Procedure

We describe our experimental prodedure to answer RQ1 and RQ2 below.

1) RQ1: To answer RQ1, we estimate the probability that a production code is faulty, $P(Faulty)$, and the conditional probability that a production code is faulty when it was tested by a test code with test smell S_i, $P(Faulty|S_i)$, respectively (for $i = 1, \ldots, 18$), from our collected data. Because our dataset is a sample set of Python programs, we utilize the

[2] `https://se.cite.ehime-u.ac.jp/data/Fushihara_SEAA2024/`

Bayesian inference method [15] with the Jeffreys prior distribution [16] to obtain those probabilities rather than adopting the simple rates of faulty production code. We briefly describe the Bayesian inference method below.

Suppose we have a sample set consisting of n production code (source files) tested by test code having smell S_i, and a out of n production ones are faulty, and the remaining $n - a$ ones are non-faulty. Let x be the probability that a production program is faulty, and y be the fact that the above sample set is given. Bayesian statistics call $p(x|y)$ the posterior distribution and consider it a probability distribution rather than a single probability value. According to Bayes' theorem, the posterior distribution is proportional to the product of the prior distribution $p(x)$ and the likelihood $p(y|x)$:

$$p(x|y) \ \propto \ p(x)\ p(y|x)\ .$$

Because x corresponds to the binary data—faulty or non-faulty—in the context of our study, the likelihood $p(y|x)$ follows the binomial distribution:

$$p(y|x) \ \propto \ x^a(1-x)^{n-a} \quad .$$

To estimate the posterior distribution from the sample set, we need to decide the prior distribution $p(x)$. Because we have no special prior information regarding the distribution of the production code's fault-proneness, we adopted a non-informative prior distribution, and the following Jeffreys prior distribution $f(x)$ is a well-known one for binomial distribution context:

$$f(x) \ \propto \ x^{-0.5}(1-x)^{-0.5} \quad .$$

Hence, we obtain the following expression regarding the posterior distribution:

$$\begin{aligned} p(x|y) \ &\propto \ x^{-0.5}(1-x)^{-0.5}\ x^a(1-x)^{n-a} \\ &= \ x^{a-0.5}(1-x)^{n-a-0.5} \quad . \end{aligned}$$

Because a distribution whose probability density is proportional to $x^{s-1}(1-x)^{t-1}$ is a Beta distribution, $\mathrm{Beta}(s,t)$, we can express the posterior distribution as follows:

$$p(x|y) \ \propto \ \mathrm{Beta}(a+0.5, n-a+0.5) \quad .$$

Although the above expression is not equality ("=") but proportionality ("$\propto$"), we can compute the concrete probability values by scaling the values so as to hold $\int_0^1 p(x|y)dx = 1$. Fig. 2 shows an example of estimated posterior distribution under $n = 536$ and $a = 93$. The gray region in the figure corresponds to the 95% Bayesian credible interval whose lower and upper limits are 2.5% and 97.5% points of the distribution function. We use the 50% point (median) of the distribution function as an estimated probability in this study.

2) RQ2: To answer RQ2, we focus on the available combinations of two test smells and estimate the probability that a production code is faulty when it was tested by a test code in which two test smells S_i and S_j co-occur, $P(Faulty|S_i \wedge S_j)$ (for $i, j = 1, \ldots, 18;\ i < j$). Similar to answering RQ1, we estimate those probabilities using the Bayesian inference method.

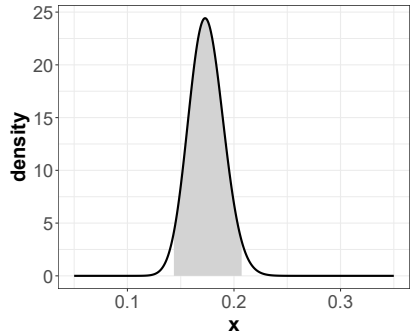

Fig. 2. An example of estimated posterior distribution.

D. Results

We show our results according to the RQs below.

1) RQ1: For each test smell S_i, we estimated $\widehat{P}(Faulty|S_i)$ to see the impact of S_i on the fault-proneness of the production code under test (for $i = 1, \ldots, 18$). Table II presents the estimated probabilities and their 95% credible intervals (95% CI). In the table, we emphasize the higher probabilities than the estimated probability that a production code is faulty (without any test smell condition), $\widehat{P}(Faulty)$, by boldface. Because "S_4: Default Test" did not appear in our dataset, we omit the data regarding S_4 below.

The highest probability of being faulty was $\widehat{P}(Faulty|S_3)$, i.e., the cases that the production code was tested by a test code having "S_3: Constructor Initialization" (see Table II). The estimated probability is about two times higher than the overall probability (without any test smell condition), and its 95% CI was significantly higher. As a result, 10 out of 18 test smells showed higher estimated probabilities of being faulty than the overall one. Interestingly, the more frequently appearing test

TABLE II
ESTIMATED PROBABILITY OF BEING FAULTY WHEN TESTED BY A SMELLED TEST CODE

Test Smell	$\widehat{P}(Faulty\|S_i)$	95% CI
S_1: **Assertion Roulette**	**0.174**	[0.143, 0.207]
S_2: **Conditional Test Logic**	**0.173**	[0.138, 0.211]
S_3: **Constructor Initialization**	**0.419**	[0.180, 0.688]
S_4: Default Test	—	—
S_5: **Duplicate Assert**	**0.153**	[0.114, 0.198]
S_6: Empty Test	0.125	[0.048, 0.247]
S_7: **Exception Handling**	**0.155**	[0.108, 0.211]
S_8: **General Fixture**	**0.165**	[0.128, 0.208]
S_9: Ignored Test	0.141	[0.076, 0.228]
S_{10}: Lack of Cohesion of Test Cases	0.144	[0.097, 0.200]
S_{11}: **Magic Number Test**	**0.190**	[0.150, 0.234]
S_{12}: Obscure In-Line Setup	0.134	[0.082, 0.200]
S_{13}: Redundant Assertion	0.068	[0.006, 0.244]
S_{14}: **Redundant Print**	**0.182**	[0.124, 0.252]
S_{15}: **Sleepy Test**	**0.171**	[0.059, 0.349]
S_{16}: **Suboptimal Assert**	**0.216**	[0.165, 0.273]
S_{17} **Test Maverick**	**0.168**	[0.125, 0.219]
S_{18}: Unknown Test	0.124	[0.091, 0.164]
Overall probability $\widehat{P}(Faulty)$	0.147	[0.123, 0.173]

TABLE III
ESTIMATED PROBABILITIES THAT THE PRODUCTION CODE IS FAULTY WHEN IT WAS TESTED BY TEST CODE HAVING TWO SMELLS

No.	Combination	$\widehat{P}(Faulty \mid S_i \wedge S_j)$	95% CI
1	$S_3 \wedge S_5$	**0.948**	[0.555, 1.000]
2	$S_3 \wedge S_6$	**0.933**	[0.464, 1.000]
3	$S_3 \wedge S_9$	**0.933**	[0.464, 1.000]
4	$S_3 \wedge S_{10}$	**0.933**	[0.464, 1.000]
5	$S_3 \wedge S_{12}$	**0.933**	[0.464, 1.000]
⋮	⋮	⋮	⋮
101	$S_2 \wedge S_7$	**0.148**	[0.101, 0.205]
102	$S_1 \wedge S_{18}$	0.146	[0.104, 0.196]
⋮	⋮	⋮	⋮
130	$S_2 \wedge S_{13}$	0.072	[0.007, 0.257]
131	$S_5 \wedge S_{13}$	0.072	[0.007, 0.257]
132	$S_7 \wedge S_{13}$	0.072	[0.007, 0.257]
133	$S_{12} \wedge S_{13}$	0.027	[0.000, 0.262]
134	$S_{13} \wedge S_{14}$	0.018	[0.000, 0.185]

smells are not always associated with a higher probability of being faulty. For example, although S_1 occurs most frequently in the test code [7], the probability of being faulty when it appears in the test code ranked 5th within 18 smells.

2) RQ2: Although it is ideal to consider all combinations of 18 different test smells, the number of combinations becomes too many to examine. Thus, we considered combinations of two smells S_i and S_j (co-occurrences of them) in a test code and estimated the probabilities that the production code is faulty when the corresponding test code has those smells: $\widehat{P}(Faulty|S_i \wedge S_j)$ (for $i, j = 1, \ldots, 18$; $i < j$). As a result, we could calculate the probabilities for 134 combinations, and Table III presents some of them in descending order. All results are available on our supplementary site.

Since the estimated probability of being faulty without any test smell condition is 0.147 (see Table II), we observed that 101 out of 134 combinations (75.4%) of test smells tend to be related to higher fault-proneness of production code when those test smells co-occur in the test code.

E. Discussions

We collected and analyzed data on Python test smell under two RQs. We discuss the results regarding the RQs below.

1) RQ1: As shown in Table II, the estimated probability that the production code is faulty when the smelled test code tests is higher than the overall estimated probability (0.147) for 10 out of 18 test smells. "S_3: Constructor Initialization" and "S_{16}: Suboptimal Assert" showed trends that they are significantly high; While the 95% credible interval of the overall probability of being faulty $\widehat{P}(Faulty)$ is [0.123, 0.173], those of the conditional probabilities $\widehat{P}(Faulty|S_3)$ and $\widehat{P}(Faulty|S_{16})$ are [0.180, 0.688] and [0.165, 0.273].

"S_3: Constructor Initialization" is the test smell that the test code's way of initializing is anti-recommendation and would result in overlooking latent faults. Although it is hard to firmly conclude that such an anti-recommended way of initializing always contributes to raising the risk of being faulty, the smell only sometimes occurs in the test code (only 12 out of 775 samples), and those test cases had disparate characteristics. On the other hand, "S_{16}: Suboptimal Assert" relatively frequently occurs in the test code (223 out of 775 samples). That smell is a warning of improper usage of assertion methods and might be related to inappropriate test cases.

We answer **RQ1 (Does a test smell in the test code affect the fault-proneness of the production code under test?)** as:

> We revealed that 10 out of 18 test smells detectable by PyNose would have risks of increasing the fault-proneness of the production code under test. Especially, "S_3: Constructor Initialization" and "S_{16}: Suboptimal Assert" would be worth detecting and alerting the test engineers to refactor them to prevent overlooking latent faults through the test.

2) RQ2: By focusing on the co-occurrence of test smells, we observed that 101 out of 134 combinations would increase the fault-proneness of the production code under test (see Table III). Although individual test smells may adversely affect the quality of test code (e.g., readability), their co-occurrences might have negative synergies on the testing. For example, although the test smell S_3 itself showed a high impact on the fault-proneness of the production code under test (see Table II), the combinations of S_3 with another test smell showed still higher impacts on the fault-proneness (see Table III). Although it is hard to make a strong conclusion regarding those combinations' harmfulness due to the limited number of samples, we can say that the production code tested by test code having two different test smells is more fault-prone. Thus, those test smells may affect testing quality.

We answer **RQ2 (How do combinations of test smells affect the fault-proneness of the production code?)** as:

> When different test smells co-occur in the test code, the production code under test is likely to be more fault-prone. Test smells may have negative synergies, and it is better to preferentially review and refactor those test programs toward more reliable and successful testing activity.

F. Threats to Validity

We describe our threats to the validity of this study below.

1) Internal Validity: We have quantitatively shown that some test smells are likely to affect the fault-proneness of the production code tested by the smelled test code in this study. However, we did not investigate how those test smells related to the overlooked faults, and this is a threat to internal validity. Although we discussed trends of test smells that might impact the production code's faults, we need to conduct a further and deeper analysis of their impacts as our important future work.

Furthermore, we did not examine the co-occurrence and interaction between test smells in this study. It can also threaten the internal validity of the study. Some test smells may occur together with other specific smells or some smells may be hard to co-occur with specific ones. Understanding the relationships between test smells is better for building a useful test smell-based warning system. We plan to conduct a data

analysis regarding the relationships and study a reasonable way of considering their effects in the future.

2) Construct Validity: Although there are several test smell detection tools (e.g., [17]), we have used only PyNose in this study. Hence, the tool may cause a bias regarding smell detection, which threatens the construct validity. Because PyNose works only for the test code under Python's `unittest` unit testing framework, we had to exclude the test code that adopted another testing framework from our experimental subjects. A comparative study with other smell detection tools is our significant future work.

Another threat to the construct validity is our way of marking "faulty" programs. In this study, when a program is changed at a commit whose aim is "bugfix," we regarded the source file before the commit as a faulty program. Although it is a common way of extracting faulty programs, two or more source files are simultaneously changed at the commit, and all of their older revisions may not always be faulty ones; there might be a source file that was accidentally changed at the same time with another defective program. Such a mislabeled faulty program might threaten the construct validity.

3) External Validity: We have examined 50 Python open-source projects to analyze the relationship between the production code's fault-proneness and the test smells appearing in the corresponding test code. We were not selective about which project we studied to capture the general trend of the impact of test smells on the production code's fault-proneness. However, because all our subject projects were actively developed and maintained by more than ten contributors and have attracted more developers (whose star scores are higher than 50), they might be well-tested and well-reviewed projects. Therefore, minor projects with fewer contributors might have different fault-proneness trends than our results. It can threaten the external validity of this study.

Another threat to the external validity is that we have analyzed only 50 Python open-source projects using only PyNose. We cannot say with absolute certainty that those Python program samples reflect the general trends of fault-proneness and test smells. We need to conduct further data analysis with more projects, including open-source and commercial ones, using various smell detection tools, not only PyNose, to mitigate the threat to external validity.

IV. Conclusion and Future Work

We focused on 18 kinds of test smells in Python test code and conducted a quantitative investigation regarding the fault-proneness of the production code tested by smelled test code in this paper. As a result, for 10 out of 18 test smells, we observed that the production code tested by the test code having those smells tend to be more fault-prone. Moreover, we also examine the impact of co-occurrence of test smells in terms of fault-proneness and showed that 101 out of 134 test smell combinations adversely affect the quality of production code under test. Especially, the test smells "Constructor Initialization" and "Suboptimal Assert" showed stronger trends that the Python production code would be more fault-prone when those test smells occur in the corresponding test code. Through our data collection and analysis, we could confirm the importance of focusing on the quality of not only the production code but the test code, resulting in the usefulness of detecting and refactoring test smells as early as possible.

Our future work is a further analysis of test smell data from various software domains using not only PyNose but other smell detection tools to enhance the generalizability of our findings. Moreover, the comparative study of the fault-proneness of the production code tested by smelled test code in various language (not only Python) is also our future work.

Acknowledgment

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

References

[1] A. P. Mathur, *Foundations of Software Testing*. Delhi: Pearson Education, 2008.
[2] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 1999.
[3] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.
[4] K. Beck, *Test Driven Development: By Example*. Boston: Addison-Wesley Professional, 2002.
[5] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd ed. Hoboken, NJ: Addison-Wesley Professional, 2018.
[6] A. V. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proc. 2nd Int'l Conf. Extreme Prog. & Flexible Processes in Softw. Eng*, May 2001, pp. 92–95.
[7] T. Wang, Y. Golubev, O. Smirnov, J. Li, T. Bryksin, and I. Ahmed, "Pynose: A test smell detector for python," in *Proc. 36th IEEE/ACM Int'l Conf. Automated Softw. Eng.*, Nov. 2021, pp. 593–605.
[8] M. Greiler, A. Van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *Proc. IEEE 6th Int'l Conf. Softw. Testing, V. & V.*, Mar. 2013, pp. 322–331.
[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 Annual Int'l Conf. Comp. Sc. & Softw. Eng.*, 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 ACM Joint Europ. Softw. Eng. Conf. & Symp. Foundations of 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'l 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] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *Proc. IEEE Int'l Conf. Softw. Maintenance & Evol.*, Sep. 2018, pp. 1–12.
[14] 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.
[15] W. M. Bolstad and J. M. Curran, *Introduction to Bayesian Statistics*, 3rd ed. Hoboken, NJ: Wiley, 2016.
[16] H. Jeffreys, "An invariant form for the prior probability in estimation problems," *Proc. Royal Society of London. Series A*, vol. 186, no. 1007, pp. 453–461, Sep. 1946.
[17] 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.