To What Extent Can Code Quality be Improved by Eliminating Test Smells?

Haitao Wu*
Shanghai Normal University
Shanghai Shi, Fengxian Qu, China
ht.wu@163.com

Ruidi Yin
Shanghai Normal University
Shanghai Shi, Fengxian Qu, China
YRD9986@163.com

Jianhua Gao
Shanghai Normal University
Shanghai Shi, Fengxian Qu, China
jhgao@shnu.edu.cn

Zijie Huang
East China University of Science and Technology
Shanghai Shi, Xuhui Qu, China
hzj@mail.ecust.edu.cn

Huajun Huang
Shanghai Normal University
Shanghai Shi, Fengxian Qu, China
337248965@qq.com

Abstract

Software testing is a key activity to guarantee software reliability and maintainability. However, developers tend to ignore the maintenance of test code when performing a tradeoff between code quality and release deadlines. Moreover, the lack of research to quantify the relationship between test code and production code quality. As a result, test quality degrades due to the lack of appropriate refactoring plans. This paper fills the gap by evaluating to what extent can code quality be improved by eliminating test smells. First, we detect the presence of test smells in 119 historical releases of 10 open-source projects. Afterward, we evaluate code quality in 2 aspects, i.e., defect- and change-proneness. Finally, we exploit the odds ratio and Mann-Whitney test to quantify the extent of variation for the code quality. Results show that the OR values of the test code and production code are both much greater than 1, which proves that the test smell is indeed a risk factor to increase the defect-proneness of code. Moreover, the change-proneness of the test code and associated production code reduces significantly after their elimination. Experiment also reveals Assertion Roulette is the riskiest smell to degrade production code quality.

Keywords: test smell, code refactoring, error-proneness, change- proneness, empirical software engineering

1 Introduction

Software testing is a key activity to guarantee software quality by ensuring the robustness of production code under complex conditions [5, 7]. As a result, software contains a large number of test code [9], which is costly to implement and maintain it by hand. Meanwhile, developers tend to ignore the maintenance of test code when performing a tradeoff between code quality and release deadlines, resulting in the lack of appropriate refactoring plans [2]. Consequently, test code is more vulnerable to quality issues.

Automated test code is an important part of the software system, programming and maintaining have similar challenges as production code [4]. However, research[2] has found that developers tend to ignore the importance of test code. Therefore, the test code lacks a rigorous refactoring plan, and its quality is generally lower than the production code [3, 13, 14, 19]. The actual meaning of improving the quality of test code is mainly based on subjective experience, and it has not been quantitatively studied and discussed, leading to insufficient attention to the meaning of software testing. Therefore, it is imminent to choose a quantifiable perspective to analyze the significance of improving the quality of the test code.

Test smell is a derivative of code smells in the context of software testing [2], defined as the sign of poor design and implementation in software tests, which is detectable by heuristic and rule-based approaches [2, 10, 18]. However, the costs and benefits of test smell refactoring to the improvement of code quality have not been revealed quantitatively [2, 10], making it difficult for developers to decide whether they should remove test smells.

Therefore, with the purpose of understanding the impact of test smells on code quality from the granularity of class level, we also provide software developers a reference indicator to measure the cost and benefit of refactoring test code. In this paper, we tested 119

historical release versions of 10 open-source projects, involving 5 test smell types, including Mystery Guest, Resource Optimism, Eager Test, Assertion Roulette, and Sensitive Equality. The experimental results show that eliminating the test smells has a positive effect on code quality. At the same time, we also find that Assertion Roulette has the most significant effect on code quality.

The main contributions of this paper are as follows:

- To our knowledge, the first work to explore the impact of eliminating test smells on the quality of production code by capturing the existence of test smells in different versions.
- Our results show that comparing with the version without the test smells, the production code's defect- and change-proneness in the version with the test smells are significantly increased.
- We reveal the refactoring of test code is of practical significance for improving code quality, which is also a reference point for developers to weigh the cost and benefit of refactoring test smells.
- We provide an online replication package [12] for extending and validating our work.

Our paper is organized as follows. Section 2 introduces the background and related work. Section 3 describes the relevant tools for test smell detection and the quantitative analysis method of defect-proneness and change-proneness. In Section 4, we present the basic process and the result of the experiment. Section 5 illustrates potential threats to effectiveness. Section 6 concludes the paper and proposes future work.

2 Related Work

2.1 Test Smell Definition and Detection

Inspired by the conception of Code Smells, Bavota et al. [2] defined and proposed 11 test smells and their refactoring methods. Afterwards, Meszaros et al. [10] expanded Bavota et al. [2]'s work by involving another 18 test smells to capture actual problems they encountered during the development process.

Bavota et al. [2] designed an automatic detection tools for test smells, which is commonly applied in test smell detection literature. The research indicates that its accuracy rate can reach 88%, and the recall rate can reach 100%. However, the tool is not publicly available. Peruma et al. [15] developed tsDetect as an alternative to the tool of [2], which is applied in this paper. It can achieve at least 85% in accuracy and 90% in recall, with a F-Score of 96.5% [15].

2.2 The Impact of Test Smells

Bavota et al. [2] conducted the first large-scale quantification research of test smells, and they concluded that test smells harm the comprehensibility and maintainability of the test code. The impact of test smells is together with the size of software, and the intensity of Sensitive Equality and Mystery Guest is positively correlated to the life of the software system.

In terms of the impact of test smells on production code, Spadini et al. [17] found that the test smells will increase the error-proneness of the associated production code. However, the authors did not assess the influence of eliminating test smells. Although test smells are indicators of potential software defects, refactoring all test smells may be costly but ineffective. Due to the lack of research on the gain of refactoring test smells, it is difficult for developers to determine whether to refactor the test code. Therefore, we are still ignorant about the impact of test smells on production code quality in terms of test code refactoring.

2.3 Definition and Research of Defect-proneness

Software quality assurance activities (for example, source code inspection and unit testing) play an important role in the production of high-quality software. Software defects in released products can have costly consequences for the company and affect the company's reputation. Therefore, predicting and reducing the defect-proneness of software has important practical significance.

Nagappan and Ball [11] use relative code change metrics, which measure the number of code changes to predict the density of defect at the file level.

2.4 Definition and Research of Change-proneness

Some files of the software are easier to change because they may contain bugs, poor code structure, or inflexible design patterns. These files consume more maintenance costs than other files, which are called change-prone files. If these files are recognized that must be changed after a long period, then developers must be familiar with the source code of these files again , which further increases the cost of maintenance.

Bieman et al. [6] analyzed the impact of using design patterns on changes in earlier versions of the software. They regard changed times as a proxy of the change-proneness. They tested whether class size has an effect

on change-proneness, and explored the relationship between design patterns and change-proneness. The results show that, on the one hand, large classes are more likely to change in two systems, and on the other hand, in four of the five systems, pattern classes are more likely to change.

3 Approach

3.1 Data Preprocessing

Table 1 shows the 5 test smells considered by this paper derived from [2], including Mystery Guest, Resource Optimism, Eager Test, Assertion Roulette, and Sensitive Equality.

We choose these 5 kinds of smells because they are commonly assessed in related research [17] and open-source software system (OSS) projects [2]. Moreover, these test smells have different origins and causes, i.e., related to different characteristics of the test code. Meanwhile, they do not co-occur with each other.

The goal of our study is to evaluate to what extent the elimination of test smells improves the quality of the test code and its associated production code. With the purpose of understanding the impact of test smells on code quality from the granularity of class level, we also provide software developers a reference indicator to measure the cost and benefit of refactoring test code. Figure 1 depicts the process of our experiment, which will be described in the following paragraphs. Figure 2 shows the pretreatment process.

First, we use the test smell detection tool tsDetect [15] to obtain the distribution of test smells in 119 historical releases. The CSV files obtained include the distribution of 5 test smells, that is, the presence of some smell is 'TRUE' or 'FALSE' and the detected test classes and their associated code path. Then, according to the existence of the test smells in the files, we programmed to capture the process of removing the test smells.

For different release versions under each project, we compare the existence of the test smells of each test class Ci corresponding to each version. If any test smell always exists before a certain release version reln and disappears after the release, we record the test class and the corresponding version number.

3.2 Calculation of defect-proneness

To quantify the defect-proneness, we refer to the SZZ algorithm proposed by Sliwerski, Zimmermann, and Zeller [16]. The idea of quantifying the defect-proneness is to estimate bug- fixing or bug-inducing commits according to the version control system's record of the class associated with the bug which introduced the code. In this paper, we define defect- proneness as the number of bug-introduced commits between the two versions of the java files where the test class and its associated production class are located. We determine whether the submission fixes the defect according to the method of Fischer et al. [8].

This approach is implemented based on the message recorded by the version control system when the code is submitted. If the submission matches an ID in the issue tracker, or contains keywords such as '*bug*', '*fix*', or '*defect*', we consider it as a bug fixing activity. The calculation process of defect-proneness is shown in Figure 2.

In this experiment, we used RepoDriller [1] to mine all code submissions to obtain bug-fixing submissions containing keywords such as '*bug*', '*fix*' or '*defect*' and the change information, '*diff*', of the code lines associated with the bug.

We use Git's blame feature to track the last time of the source code line included in the '*diff*' information list which is associated with the test class and its related production class between the two versions corresponding to the elimination of test smells.

According to the idea of the SZZ algorithm, we count the tracked changes of a certain class' code lines as the introduction of defect-proneness. If the introduction is before the version of the test smells elimination, it will be counted as the defect-proneness before the removal, Defect_B. If the introduction is after the version of the test smells elimination, then it is counted as the defect-proneness after the removal, Defect_A.

We use the odds ratio to evaluate whether the test smell is a risk factor that affects the defect-proneness of the code. The calculation formula is shown in (1). When the OR value is equal to 1, the test smell does not affect the defect-proneness of the code. When the OR value is less than 1, it means that the test smell is a safety factor that affects the code's defect- proneness. When the OR value is greater than 1, it means that the test smell is a risk factor that affects the defect-proneness of the code.

$$OR = \frac{p/(1-p)}{q/(1-q)} \tag{1}$$

In this paper, p represents the odds of being defective in the classes before the test smells are eliminated and q represents the odds of being defective in the classes after the elimination.

Test Smell	Description	Influence
Mystery Guest	When a test method uses external resources,the test is no longer self contained.	There is not enough information to understand the tested functionality and make it hard to use.
Eager Test	Test code that makes optimistic assumptions about the existence and state of external resources	It can cause non-deterministic behavior in test outcomes.
Assertion Roulette	A test method checks several methods of the object to be tested	It makes tests more dependent on each other and harder to maintain.
Sensitive Equality	A test method has many assertions in a test method that have no explanation.	If one of the assertions fails, it is not possible know which one it is.
Resource Optimism	A test method use toString in the test method	When the toString method for an object is changed, tests start failing.

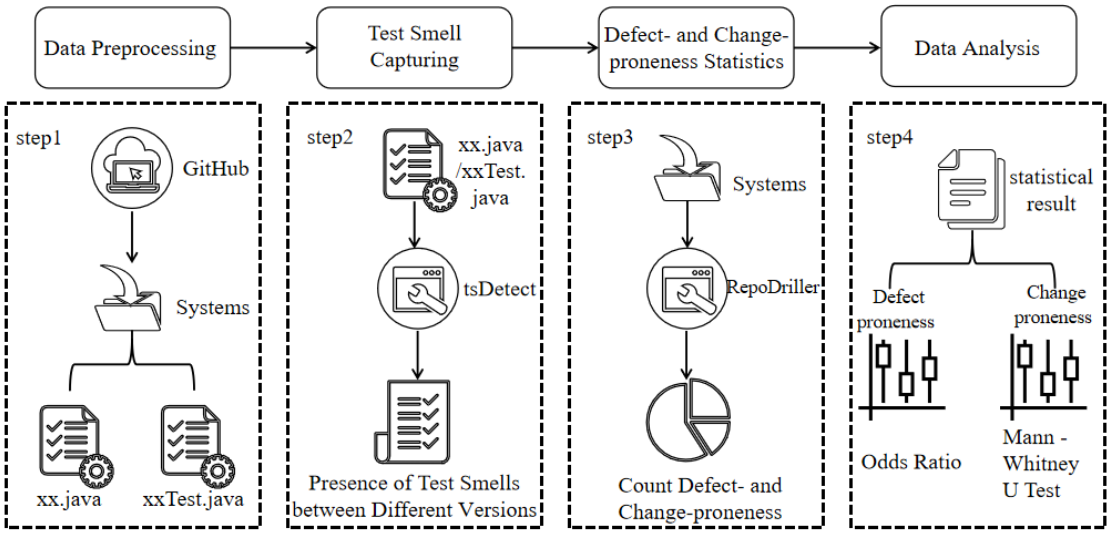

Figure 1. Experimental Steps

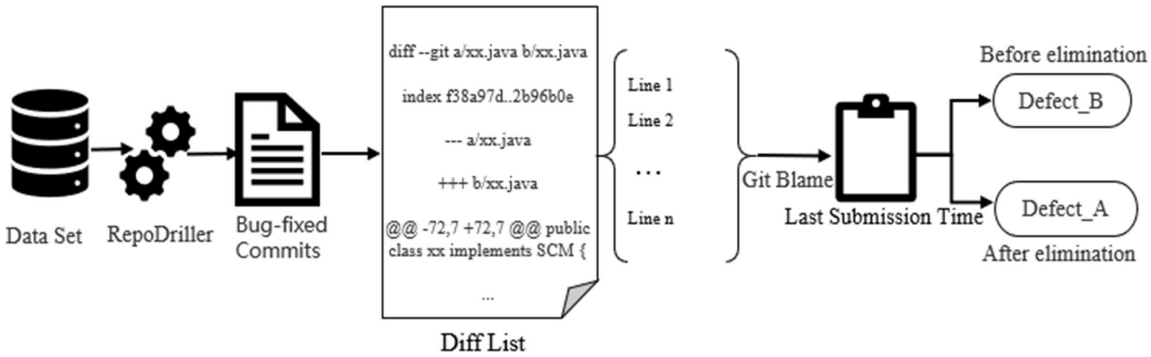

Figure 2. The Acquisition Process of Code Defect-proneness

3.3 Calculation of change-pronenes

We define change-proneness as the density of changes between the two versions of the Java file where the test classes and their associated production classes are located. According to the version number obtained, we have formulated a corresponding time-division interval

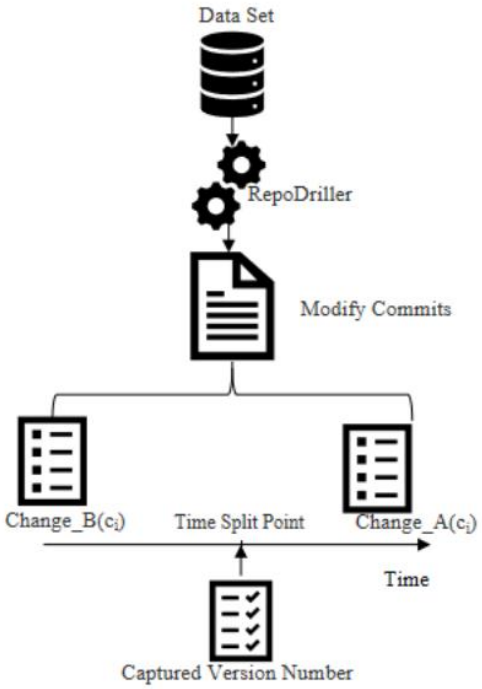

Figure 3. The Acquisition Process of Code Change-proneness

for each project. The calculation process of change-proneness is shown in Figure 3.

For the captured test class c_i and its associated production class, set the number of modification commits in the period before the test smells removal is Change_B(c_i), and the number of modification commits in the period after the test smells removal is Change_A(c_i). Among all code commits mined by RepoDriller [1], all change commits include additions, deletions, and modifications. Considering that the addition and deletion operations are mainly for the functional level. In this experiment, we only count the number of modification commits within the related time range of each class as the change-proneness. Finally, we use the Mann-Whitney test (with confidence level 95%) to analyze if there is a significant difference in the change-proneness of the test code and the associated production code before and after the elimination of the smells.

4 Experiment

We select 119 historical releases of 10 open-source software systems used in [6] as the dataset because there are a large number of JUnit test cases in all versions, as shown in Table 2. In order to explore the extent to how

Table 1. Experimental Dataset

System	Versions	Classes
Sonarqube	7	2447
Apache Ant	8	282
Apache Cassandra	11	494
Apache Wicket	33	699
ElasticSearch	12	3627
Spring Framework	10	2061
VRaptor4	19	125
Mybatis-3	8	327
Apache Hadoop	5	2717
Hibernate-orm	6	2609

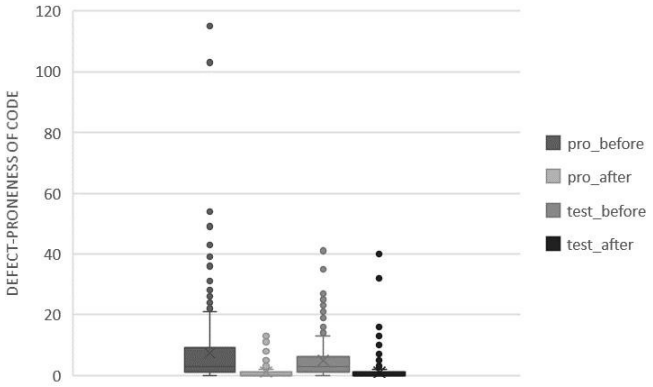

Figure 4. Defect-proneness Comparison Box Plot

the elimination of test smells can affect the quality of the code, this experiment will answer the following 3 questions:

- RQ1: Before and after the test smells are eliminated, how does the defect-proneness of the test code and associated production code change?
- RQ2: Does the elimination of test smells affect the change-proneness of the test code and related production code?
- RQ3: Is there any difference in the impact of the 5 test smells on the quality of the production code?

4.1 Experimental Analysis

To answer RQ1, we analyze the defect-proneness from two perspectives, namely: 1) the overall change trend; 2) whether the test smell is a risk factor that affects the code's defect-proneness. The results are shown in Figure 4 and Table 3.

Figure 4 depicts that the number of defect-proneness contained in the test code and its associated production

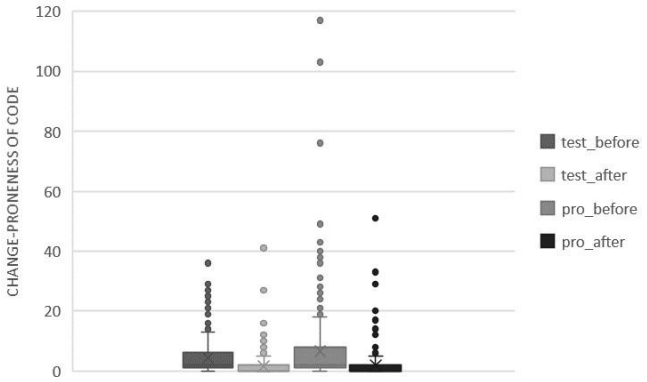

Figure 5. Change-proneness Comparison Box Plot

Table 2. The OR Value of Test Code Group

		Test Coed		Defect-proneness Probability	OR
Has Test Smells		Y	N		
Test Smells Elimination	Before	336	99	p=0.77	7.11
	After	141	296	q=0.32	

Table 3. The OR Value of Production Code Group

		Production Coed		Defect-proneness Probability	OR
Has Test Smells		Y	N		
Test Smells Elimination	Before	336	99	p=0.61	3.17
	After	141	296	q=0.33	

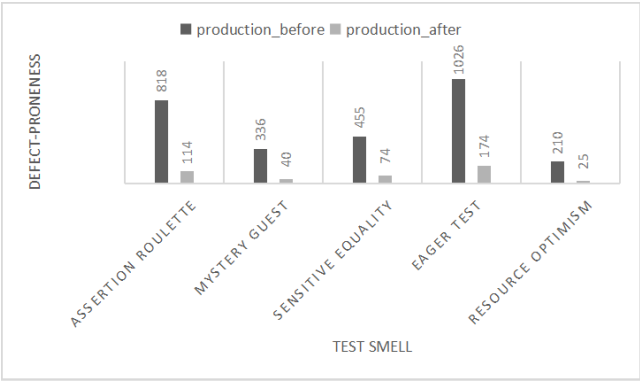

Figure 6. Defect-proneness Histogram of Different Test Smells

code reduces significantly after the elimination.Table 3 and Table 4 show that the OR value of the test code group and the associated production code group are

Figure 7. Change-proneness Histogram of Different Test Smells

Table 4. The Result of MANN -WHITNEY Test

Mann -Whitney U Test of Change-proneness		
Table Analyzed	Test Code	Production Code
P value	<0.0001	<0.0001
Exact or approximate P value?	Approximate	Approximate
Significantly different (P<0.05)?	Yes	Yes
One- or two-tailed P value?	Two-tailed	Two-tailed
Mann-Whitney U	36442	31708

greater than 1, which shows that the test smell is a potential risk factor for the test code and related production code.

To answer RQ2, we analyze the change-proneness from two perspectives, namely: 1) the overall change trend; 2) whether the change-proneness of the test code and the associated production code is significantly different before and after the removal of the test smells.The results are shown in Figure 5 and Table 4.

Figure 5 depicts that the number of change-proneness in the test code and associated production code after the removal of the test smells reduces significantly. Table 5 shows the Mann- Whitney rank-sum test results of the statistical data of code's change-proneness before and after the elimination of the smells. There are significant differences between the two sets of data. To answer RQ3, we analyze it from two perspectives: 1) The variation of the production code's defect-proneness before and after the removal of the different test smells; 2) The variation of the production code's change-proneness

before and after the removal of the different test smells. The results are shown in Figure 6 and Figure 7.

Figure 6 and Figure 7 depict that after the removal, the defect- and change-proneness of the production code decrease significantly. Among them, Assertion Roulette has the most significant impact on the defect- and change-proneness of the production code compared to the other four test smells.

After performing case studies, we conclude that high cognitive complexity of understanding the smelly code may be the cause of such outliers, which make the developers more likely to introduce defects in turn. Similarly, judging from the OR of the test code and related production code being greater than 1, the test smell is indeed a risk factor that increases the defect-proneness of the code. Moreover, the OR value of the associated production code is 3.17, and the OR value of the test code is 7.11, which is twice the OR value of the production code. We speculate that the existence of the test smells greatly increases the possibility of being defective in the test code and indirectly affects the production code's reliability.

For RQ2, according to the result of the Mann-Whitney rank-sum test with confidence level 95% is $p<0.0001$, it can be concluded that the elimination of test smells has a significant impact on the reduction of code's change-proneness. As for the outliers in Figure 5, such as, the test class before the elimination has a maximum of 36 change-proneness, but after the elimination, there are 41. We speculate that this situation may be due to the Eager Test needs to call multiple methods of the production object and involves many files, which leads to a larger amount of change to eliminate the smell.

For RQ3, the elimination of different types of test smells all reduce the production code's defect- and change-proneness obviously. Among them, the elimination of Assertion Roulette has the most significant effect. After it was eliminated, the defect-proneness of the production code was reduced by 86.1%, and the change-proneness was reduced by 75.8%.

According to the research of Bavota et al. [2], the Assertion Roulette appears most frequently in the test code and because the feature that Assertion Roulette includes more than one type of assertion to check different behaviors, it is more difficult to be understood during software testing, which leads developers to introduce more errors and requires more modifications to eliminate it. Therefore, eliminating this smell is most useful for reducing code's defect- and change-proneness.

5 Threats to Validity

A threat to internal validity is that software systems in different scales, code styles and development habits of different development teams may cause uneven distribution of test smells, which affects the experimental results.

Threats to external validity is the conclusion of this paper is the reliability of the test smell detection tool. The quality of the detection tool may affect the validity of the experiment. However, the reliability of the tool is validated in [15] based on experimental researches.

6 Summary And Future Work

Measuring the impact of test smells on the code quality can provide guidance for code refactoring and maintenance. Correct refactoring choices can effectively improve code quality. This paper focuses on 119 historical releases of 10 open-source systems to quantitatively analyze the changes in the defect- and change-proneness of the test code and its associated production code before and after the removal of the test smells. According to the experimental results, we find that refactoring the test smells is beneficial to improve the code quality. At the same time, the refactoring of Assertion Roulette may be of greater help to the improvement of production code quality.

In the future, we can consider open-source projects based on other programming languages to conduct research, and we can also further refine it to the method level to explore the situation where developers are coding.

References

[1] Maurïcio Aniche. 2018. *RepoDriller.* Retrieved Dec 15, 2021 from https://github.com/mauricioaniche/repodriller

[2] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave W. Binkley. 2015. Are test smells really harmful? An empirical study. *Empir. Softw. Eng.* 20, 4 (2015), 1052–1094. https://doi.org/10.1007/s10664-014-9313-0

[3] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 179–190. https://doi.org/10.1145/2786805.2786843

[4] Stefan Berner, Roland Weber, and Rudolf K. Keller. 2005. Observations and lessons learned from automated testing. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh (Eds.). ACM, 571–579. https://doi.org/10.1145/1062455.1062556

[5] Antonia Bertolino. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, Lionel C. Briand and Alexander L. Wolf (Eds.). IEEE Computer Society, 85–103. https://doi.org/10.1109/FOSE.2007.25

[6] James M. Bieman, Greg Straw, Huxia Wang, P. Willard Munger, and Roger T. Alexander. 2003. Design Patterns and Change Proneness: An Examination of Five Evolving Systems. In *9th IEEE International Software Metrics Symposium (METRICS 2003), 3-5 September 2003, Sydney, Australia*. IEEE Computer Society, 40–49. https://doi.org/10.1109/METRIC.2003.1232454

[7] George Candea, Stefan Bucur, and Cristian Zamfir. 2010. Automated software testing as a service. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 155–160. https://doi.org/10.1145/1807128.1807153

[8] Michael Fischer, Martin Pinzger, and Harald C. Gall. 2003. Populating a Release History Database from Version Control and Bug Tracking Systems. In *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*. IEEE Computer Society, 23. https://doi.org/10.1109/ICSM.2003.1235403

[9] Vahid Garousi, Baris Küçük, and Michael Felderer. 2019. What We Know About Smells in Software Test Code. *IEEE Softw.* 36, 3 (2019), 61–73. https://doi.org/10.1109/MS.2018.2875843

[10] Gerard Meszaros. 2010. XUnit test patterns and smells: improving the ROI of test code. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 299–300. https://doi.org/10.1145/1869542.1869622

[11] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh (Eds.). ACM, 284–292. https://doi.org/10.1145/1062455.1062514

[12] Results Package. 2021. *Replication Package*. Retrieved Dec 15, 2021 from :https://github.com/vincent950602/defect-proneness-and-change-proneness

[13] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2016. On the diffusion of test smells in automatically generated test code: an empirical study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST@ICSE 2016, Austin, Texas, USA, May 14-22, 2016*. ACM, 5–14. https://doi.org/10.1145/2897010.2897016

[14] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. 2018. Automatic Test Smell Detection Using Information Retrieval Techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 311–322. https://doi.org/10.1109/ICSME.2018.00040

[15] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. tsDetect: an open source test smells detection tool. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1650–1654. https://doi.org/10.1145/3368089.3417921

[16] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM SIGSOFT Softw. Eng. Notes* 30, 4 (Sep 2005), 1–5. https://doi.org/10.1145/1082983.1083147

[17] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the Relation of Test Smells to Software Code Quality. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 1–12. https://doi.org/10.1109/ICSME.2018.00010

[18] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 4–15. https://doi.org/10.1145/2970276.2970340

[19] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empir. Softw. Eng.* 16, 3 (2011), 325–364. https://doi.org/10.1007/s10664-010-9143-7