The Lost World: Characterizing and Detecting Undiscovered Test Smells

YANMING YANG, Department of Computer Science and Technology, Zhejiang University, China
XING HU, School of Software Technology, Zhejiang University, China
XIN XIA, Software Engineering Application Technology Lab, Huawei, China
XIAOHU YANG, Department of Computer Science and Technology, Zhejiang University, China

Test smell refers to poor programming and design practices in testing and widely spreads throughout software projects. Considering test smells have negative impacts on the comprehension and maintenance of test code and even make code-under-test more defect-prone, it thus has great importance in mining, detecting, and refactoring them. Since Deursen et al. introduced the definition of "test smell", several studies worked on discovering new test smells from test specifications and software practitioners' experience. Indeed, many bad testing practices are "observed" by software developers during creating test scripts rather than through academic research and are widely discussed in the software engineering community (e.g., Stack Overflow) [70, 94]. However, no prior studies explored new bad testing practices from software practitioners' discussions, formally defined them as new test smell types, and analyzed their characteristics, which plays a bad role for developers in knowing these bad practices and avoiding using them during test code development. Therefore, we pick up those challenges and act by working on systematic methods to explore new test smell types from one of the most mainstream developers' Q&A platforms, i.e., Stack Overflow. We further investigate the harmfulness of new test smells and analyze possible solutions for eliminating them. We find that some test smells make it hard for developers to fix failed test cases and trace their failing reasons. To exacerbate matters, we have identified two types of test smells that pose a risk to the accuracy of test cases. Next, we develop a detector to detect test smells from software. The detector is composed of six detection methods for different smell types. These detection methods are both wrapped with a set of syntactic rules based on the code patterns extracted from different test smells and developers' code styles. We manually construct a test smell dataset from seven popular Java projects and evaluate the effectiveness of our detector on it. The experimental results show that our detector achieves high performance in precision, recall, and F1 score. Then, we utilize our detector to detect smells from 919 real-world Java projects to explore whether the six test smells are prevalent in practice. We observe that these test smells are widely spread in 722 out of 919 Java projects, which demonstrates that they are prevalent in real-world projects. Finally, to validate the usefulness of test smells in practice, we submit 56 issue reports to 53 real-world projects with different smells. Our issue reports achieve 76.4% acceptance by conducting sentiment analysis on developers' replies. These evaluations confirm the effectiveness of our detector and the prevalence and practicality of new test smell types on real-world projects.

CCS Concepts: • **Software and its engineering → Software maintenance tools;**

This research is supported by the Fundamental Research Funds for the Central Universities (No. 226-2022-00064) and the National Natural Science Foundation of China (No. 62141222).

Authors' addresses: Y. Yang and X. Yang, Department of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, 310027, China; e-mails: yanmingyang@zju.edu.cn, yangxh@zju.edu.cn; X. Hu (Corresponding author), School of Software Technology, Zhejiang University, Ningbo, Zhejiang, 315048, China; e-mail: xinghu@zju.edu.cn; X. Xia, Software Engineering Application Technology Lab, Huawei, Hangzhou, Zhejiang, 310007, China; e-mail: xin.xia@acm.org.

1049-331X/2024/03-ART59 $15.00
https://doi.org/10.1145/3631973

Additional Key Words and Phrases: Test smell, mining software repositories, test smell detection, empirical study

ACM Reference format:
Yanming Yang, Xing Hu, Xin Xia, and Xiaohu Yang. 2024. The Lost World: Characterizing and Detecting Undiscovered Test Smells. *ACM Trans. Softw. Eng. Methodol.* 33, 3, Article 59 (March 2024), 32 pages.
https://doi.org/10.1145/3631973

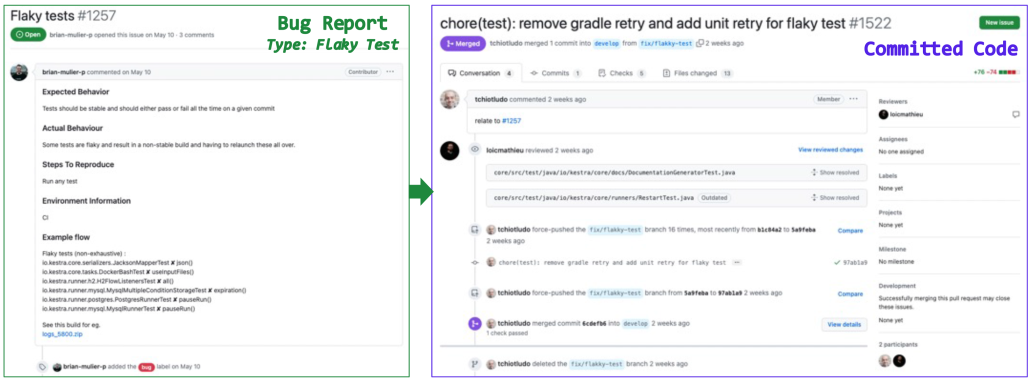
1 INTRODUCTION

Test code, as an important part of the software, can compete with production code in size [74]. The ratio between production code and test code could be anywhere between 1:1 and 1:3 [74]. Just like code smells in production code, test code also exists in many bad programming and design practices [56]. Van Deursen et al. [99] defined these bad testing practices as *test smells*. Similar to code smells, test smells make it harder for developers to maintain and comprehend test code and even may lower the effectiveness and correctness of test code [57]. Recent studies also demonstrated that test smells hamper the quality and maintainability of production code [97]. Here, we provide a descriptive illustration of the impact of a test smell on software quality by presenting a real-world test smell [1]. Figure 1 presents a bug report of a popular project called *kestra*[1] with over 3.6k stars. This bug report elucidates the issue of inconsistent and unreliable test results attributed to the presence of the flaky test, a specific test smell type. Subsequently, developers initiate a pull request to address this issue by removing the flaky test. Considering the negative impacts of test smells, it is of great importance to explore potential test smells in order to stop developers from producing smelly test scripts.

Since Van Deursen et al. [99] introduced the definition of "test smell" in 2001, only nine formally-published literatures [69] proposed new test smells based on the test specifications in the testing framework [84, 100] and developers' and researchers' experiences [99]. Indeed, most of the bad testing practices and problems are "observed" by practitioners who are actively developing test scripts and are communicated by them via the grey literature and discussions (e.g., blog sources and industry conference talks) [69] rather than via academic research. So far, software practitioners have noticed many controversial programming practices in test code along with the development of testing techniques. Many of these practices have even raised hotspot discussions on the Q&A platform, such as Stack Overflow. However, since no academic studies present these new test smell types by mining software practitioners' discussions, such potential test smells without formal definitions are not acknowledged and further researched by academia as well as not standardized in the industry. It leads to the majority of software practitioners not having access to recognize the existence of potential test smells and still applying bad testing practices to produce low-quality test scripts.

To fill the gap, we consider **Stack Overflow (SO)** – one of the largest and most famous discussion platforms for software practitioners – as the data source and conduct systematic research to (1) explore potential test smell types from SO posts that discuss controversial testing practices and provide formal definitions for them, and then (2) curate a detector to identify them from software. More specifically, to excavate new test smells, we first design a search strategy to collect the posts discussing the controversial testing practices. Then, we classify and analyze the voting preference of answers in collected posts to determine potential test smells. Finally, we summarize six new test smells, including one project-level, two statement-level, and three method-level smells. Furthermore, for each smell type, we provide a clear definition and investigate its harmfulness to software quality. Empirical evidence shows that some test smells violate the design principles

[1]https://github.com/kestra-io/kestra



Flaky tests #1257

Open brian-mulier-p opened this issue on May 10 · 3 comments

brian-mulier-p commented on May 10 Contributor

Expected Behavior

Tests should be stable and should either pass or fail all the time on a given commit

Actual Behaviour

Some tests are flaky and result in a non-stable build and having to relaunch these all over.

Steps To Reproduce

Run any test

Environment Information

CI

Example flow

Flaky tests (non-exhaustive) :
io.kestra.core.serializers.JacksonMapperTest ✘ json()
io.kestra.core.tasks.DockerBashTest ✘ useInputFiles()
io.kestra.runner.h2.H2FlowListenersTest ✘ all()
io.kestra.runner.mysql.MysqlMultipleConditionStorageTest ✘ expiration()
io.kestra.runner.postgres.PostgresRunnerTest ✘ pauseRun()
io.kestra.runner.mysql.MysqlRunnerTest ✘ pauseRun()

See this build for eg.
logs_5800.zip

brian-mulier-p added the bug label on May 10

chore(test): remove gradle retry and add unit retry for flaky test #1522

Merged tchiotludo merged 1 commit into `develop` from `fix/flakky-test` 2 weeks ago

Conversation 4 · Commits 1 · Checks 5 · Files changed 13 · +76 −74

tchiotludo commented 2 weeks ago Member

relate to #1257

loicmathieu reviewed 2 weeks ago View reviewed changes

`core/src/test/java/io/kestra/core/docs/DocumentationGeneratorTest.java` Show resolved

`core/src/test/java/io/kestra/core/runners/RestartTest.java` Outdated Show resolved

tchiotludo force-pushed the `fix/flakky-test` branch 16 times, most recently from **b1c84a2** to **5a9feba** 2 weeks ago Compare

`chore(test): remove gradle retry and add unit retry for flaky test` ✓ 97ab1a9

tchiotludo force-pushed the `fix/flakky-test` branch from **5a9feba** to **97ab1a9** 2 weeks ago Compare

tchiotludo merged commit **6cdefb6** into `develop` 2 weeks ago View details
1 check passed

tchiotludo deleted the `fix/flakky-test` branch 2 weeks ago

Reviewers: loicmathieu
Assignees: No one assigned
Labels: None yet
Projects: None yet
Milestone: No milestone
Development: Successfully merging this pull request may close these issues. None yet
2 participants

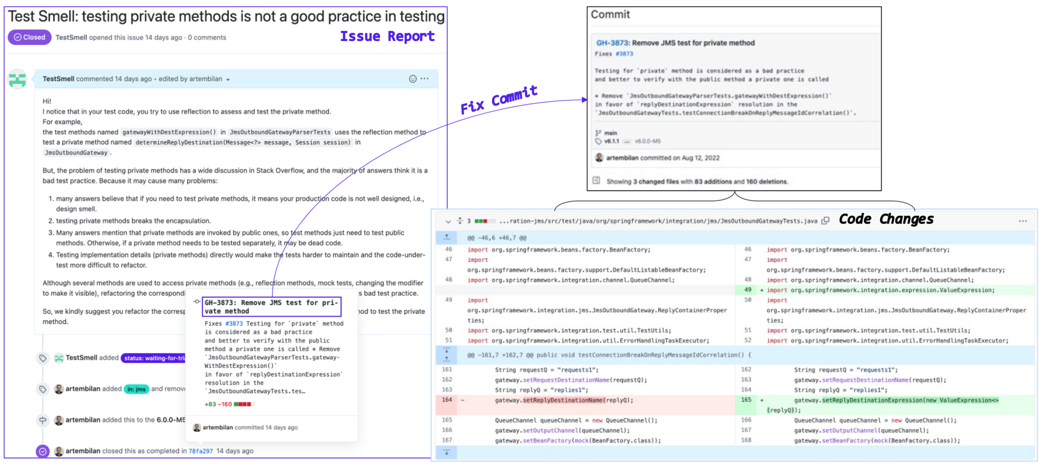
Fig. 1. An example of a real-world test smell.

of the test framework and some even prevent developers from finding the cause of failure once a test case fails. Hence, we provide possible solutions to eliminate them to reduce the negative impacts on software quality. Next, we develop a detector to identify the six test smells from software projects. To achieve this, we first extract code patterns of different smells by checking sample code in SO posts and GitHub projects. We then transform the test code for detection into **Abstract Syntax Trees (ASTs)**, and, for each smell, design corresponding syntactic rules of ASTs based on the summarized code pattern as its detection method.

To evaluate the correctness of our detector, we manually label 1,200 test smells from seven popular projects with more than 1.5k stars. Experimental results illustrate that our detector achieves 100% detection precision, 98% recall, and 99% F1-score, outperforming the baseline approaches. Some rare errors may happen during detection due to complex program logic and unusual code structures/styles in test code. To assess their prevalence in the real world, we detect each type of test smell from 919 real-world Java projects. The results show that these six test smells are detected from 722 projects and two out of six test smells widely exist in over 500 projects, which not only verifies their prevalence but also indicates that corresponding bad practices are still used by developers to create test scripts until now. Moreover, to explore the usefulness of test smells in practice, we randomly select ten smelly projects for each smell type and submit issue reports to report the corresponding test smells to developers. Finally, a total of 56 issue reports were submitted (one test smell type occurs in only six projects). As a result, we received 34 replies with 26 positive ones, achieving 76.4% acceptance. Among positive replies, many developers agree with us and even adopt our suggestions to refactor their smelly test scripts [18]. For example, a developer who works for the "*spring-integration*" project directly refactors smelly test code based on our issue report [2] in Figure 2.

In summary, this paper makes the following contributions:

(1) To the best of our knowledge, we are the first to unearth new test smell types from software practitioners' discussions and formally define them for serving further research in academia and specification development in the industry.
(2) We investigate the harmfulness of proposed test smells and analyze possible refactoring solutions to address them.
(3) We develop a test smell detector by extracting code patterns of the six new test smells as well as designing corresponding detection methods based on summarized code patterns.
(4) We evaluate the effectiveness of our detector and the prevalence and usefulness of the proposed test smell on 919 real-world Java projects.



Test Smell: testing private methods is not a good practice in testing

Closed TestSmell opened this issue 14 days ago · 0 comments

TestSmell commented 14 days ago • edited by artembilan

Hi!
I notice that in your test code, you try to use reflection to assess and test the private method.
For example,
the test methods named `gatewayWithDestExpression()` in `JmsOutboundGatewayParserTests` uses the reflection method to test a private method named `determineReplyDestination(Message<?> message, Session session)` in `JmsOutboundGateway`.

But, the problem of testing private methods has a wide discussion in Stack Overflow, and the majority of answers think it is a bad test practice. Because it may cause many problems:

1. many answers believe that if you need to test private methods, it means your production code is not well designed, i.e., design smell.
2. testing private methods breaks the encapsulation.
3. Many answers mention that private methods are invoked by public ones, so test methods just need to test public methods. Otherwise, if a private method needs to be tested separately, it may be dead code.
4. Testing implementation details (private methods) directly would make the tests harder to maintain and the code-under-test more difficult to refactor.

Although several methods are used to access private methods (e.g., reflection methods, mock tests, changing the modifier to make it visible), refactoring the correspondi… s bad test practice.

So, we kindly suggest you refactor the corresp… od to test the private method.

TestSmell added status: waiting-for-tri…

artembilan added in: jms and remov…

artembilan added this to the 6.0.0-M5

artembilan closed this as completed in 78fa297 14 days ago

```
GH-3873: Remove JMS test for pri-
vate method
Fixes #3873 Testing for `private` method
is considered as a bad practice
and better to verify with the public
method a private one is called * Remove
`JmsOutboundGatewayParserTests.gateway-
WithDestExpression()`
in favor of `replyDestinationExpression`
resolution in the
`JmsOutboundGatewayTests.tes…
+83 −160
```

artembilan committed 14 days ago

Commit

GH-3873: Remove JMS test for private method

```
Fixes #3873

Testing for `private` method is considered as a bad practice
and better to verify with the public method a private one is called

* Remove `JmsOutboundGatewayParserTests.gatewayWithDestExpression()`
in favor of `replyDestinationExpression` resolution in the
`JmsOutboundGatewayTests.testConnectionBreakOnReplyMessageIdCorrelation()`.
```

main
v6.1.1 … v6.0.0-M5

artembilan committed on Aug 12, 2022

Showing 3 changed files with 83 additions and 160 deletions.

```
3 ...ration-jms/src/test/java/org/springframework/integration/jms/JmsOutboundGatewayTests.java
@@ -46,6 +46,7 @@
  import org.springframework.beans.factory.BeanFactory;
  import
  org.springframework.beans.factory.support.DefaultListableBeanFactory;
  import org.springframework.integration.channel.QueueChannel;
+ import org.springframework.integration.expression.ValueExpression;
  import
  org.springframework.integration.jms.JmsOutboundGateway.ReplyContainerProper
  ties;
  import org.springframework.integration.test.util.TestUtils;
  import org.springframework.integration.util.ErrorHandlingTaskExecutor;
@@ -161,7 +162,7 @@ public void testConnectionBreakOnReplyMessageIdCorrelation() {
          String requestQ = "requests1";
          gateway.setRequestDestinationName(requestQ);
          String replyQ = "replies1";
-         gateway.setReplyDestinationName(replyQ);
+         gateway.setReplyDestinationExpression(new ValueExpression<>
  (replyQ));
          QueueChannel queueChannel = new QueueChannel();
          gateway.setOutputChannel(queueChannel);
          gateway.setBeanFactory(mock(BeanFactory.class));
```

Fig. 2. A commit for removing test smells based on our issue report [2].

(5) We have submitted a series of issue reports to apprise developers of the detected test smells in their projects. These reports were positively acknowledged and commended by GitHub developers, achieving a commendable 74% acceptance rate. The replication package is publicly accessible at: [3].

Paper Organization. The remainder of this paper is structured as follows. Section 2 presents a review of previous studies pertaining to test smells. In Section 3, a comprehensive overview of our study is provided. Section 4 elaborates on the empirical study conducted to identify potential test smells. Detailed information about the methodology for detecting the test smells identified in Section 4 is presented in Section 5. The evaluation process and the experimental results are discussed in Section 6. Sections 7 and 8 address the limitations of our study and highlight the valuable insights gained from this research. Finally, Section 9 concludes the paper.

2 RELATED WORK

In this section, we present an introduction to the background knowledge related to mining new test smell types, detecting test smells, and analyzing their impact on the software. Specifically, Table 1 provides a comprehensive overview of the main contributions and content of each related work.

Mining New Test Smell Types. Van Deursen et al. [99] was the first to introduce the concept of "test smell" and identified 11 types of test smells specifically in the Java language. Following their pioneering work, seven subsequent studies focused on proposing new test smell types in various programming languages. The research efforts, as depicted in Table 1, primarily revolved around mining and defining these novel test smell types based on Unit Testing Criteria [84, 100], developers' practical experience [99], and the observed behavior of test code [65]. Note that only one study [94] tapped into the knowledge derived from discussions within the Squeak community to introduce additional test smell types. Indeed, a previous study [69] has revealed that the majority of detrimental testing practices are primarily recognized and communicated by developers responsible for test scripts through informal channels such as gray literature (e.g., blogs) and discussions (e.g., conference talks and online forums), rather than formal academic publications. It is worth noting that there is a noticeable dearth of academic studies that summarize new test smell types from platforms like Stack Overflow, which serves as one of the largest and most

Table 1. A Comparison Analysis of Related Work

Category	Study	Main Contribution
Mining	Van Deursen et al. [99]	Proposed the definition of "test smell" and 11 Java test smell types.
	Hamill [73]	The literature on Unit testing frameworks proposed a fairly clear set of test smells.
	Van Rompaey et al. [100]	Proposed "General Fixture" and "Eager Test" based on Unit Testing Criteria.
	Reichhart et al. [94]	Introduced 27 abstract and fine-grained test smells from the Squeak community.
	Meszaros [84]	Proposed a number of test smells and classified them into three categories.
	Greiler et al. [70]	Proposed five new Java test smell types by interviewing Java developers.
	Delplanque et al. [65]	Defined the "Rotten Green Tests" based on Unit test code analysis.
	Peruma et al. [91]	Presented 12 new types of test smells inspired by bad test programming practices mentioned in unit testing-based literature.
Impact Analysis	Bavota et al. [57, 58]	Performed two empirical studies, which show that test smells are widely spread throughout software projects with a negative impact on the comprehensibility of test code.
	Palomba et al. [87]	The prevalence of test smells and the frequent co-occurrence of various design issues are notable observations.
	Spadini et al. [97]	Test smell is prone to be changed compared with other test scripts, and the production code tested by smelly test cases is comparatively more defect-prone.
	Campos et al. [63]	Test smells might negatively impact the project, particularly in test code maintainability and evolution.
	Aljedaani et al. [55]	The presence of test smells, particularly Assertion Roulette, leads to prolonged error correction time for developers in production code.
	Martins et al. [83]	Compared with discussions on test smell determination, few discussions focus on providing specific refactoring suggestions.
	Kim et al. [75]	Test smell metrics can offer enhanced explanatory power regarding post-release defects compared to traditional baseline metrics.
	Panichella et al. [89]	Test code, in reality, suffered from a host of problems not well-captured by current test smells.
	Soares et al. [96]	Many features are underutilized in the detection of test smells.
	Nagy and Abdalkareem [85]	The test code is still vulnerable to various smells that deter the overall quality of the tests.
Detection	Van Rompaey et al. [101]	Proposed a metric-based method to detect general fixture and eager tests.
	Breugelmans and Van Rompaey [62]	Present TESTQm a tool that allows developers to identify 12 different test smell types from C++ source code.
	Koochakzadeh and Garousi [76, 77]	Present an Eclipse plug-in, TeCReVis, to identify Test Redundancies.
	Bavota et al. [57]	Released an unnamed test smell detection tool, which detects nine test smell types in Java test suites.
	Greiler et al. [71]	Developed a detector, TestHound, to identify fixture-related test smells.
	Zhang et al. [110]	Developed DTDetector to identify dependent tests.
	Bell et al. [59]	Implemented ElectricTest to detect dependent tests.
	Gambi et al. [68]	Proposed PRADET to detect problematic dependencies in test code.
	Palomba et al. [88]	Devise a textual-based detector named TASTE to identify three test smell types, General Fixture, Eager Test, and Lack of Cohesion of Methods.
	Biagiola et al. [60]	Proposed a tool called TEDD to detect dependency tests using NLP.
	Delplanque et al. [65]	Proposed detector called DrTest to detect Rotten Green Tests.
	De Bleser et al. [64]	Developed an automated tool, SoCRATES, to identify six test smells in ScalaTest.
	Virgínio et al. [104]	Present JNose, which can identify 21 types of test smells.
	Lambiase et al. [78]	Introduced DARTS to identify three types of test smells (General Fixture, Eager Test, and Lack of Cohesion of Test Methods).
	Martinez et al. [82]	Designed a tool named RTi to detect and refactor rotten green test cases.
	Peruma et al. [91, 92]	Introduced, tsDetect, an automated test smell detector to locate 19 test smell types in Java test code.
	Virgínio et al. [103]	Implemented JNose-Core based on JNose to improve its performance and capability.
	Wang et al. [106]	Developed a plugin for PyCharm to detect 17 Python-specific test smells.
	Pontillo et al. [93]	Leveraged machine learning techniques to detect four test smell types from Java projects.
	Hadj-Kacem and Bouassida [72]	Adopted Multi-label classification techniques to identify five test smell types in Java projects.
	Bodea [61]	Proposed a new tool named Pytest-Smell to automatically detect 10 test smell types written with Pytest.
	Fulcini et al. [67]	Developed a static test suite analyzer to detect GUI test smells.
	Fernandes et al. [66]	Devised TEMPY to identify 10 different test smell types from Python code.
	Maier and Felderer. [80]	Created the tool named STest, which can detect five test smell types in JUnit tests.

popular communities for developers. Consequently, certain poor test programming practices may remain undocumented in the existing literature, preventing them from being established as industry specifications. This lack of visibility and recognition can result in many software practitioners remaining unaware that these practices are indeed considered test smells.

Impact Analysis of Test Smells. Several empirical studies have been conducted to analyze the impact of test smells on software quality [57, 58, 71, 81, 84, 87, 97, 104]. Four of the eleven studies specifically highlighted the correlation between test smells and increased susceptibility to defects or vulnerabilities in the test code [56, 75, 85, 97]. Additionally, three studies [57, 58, 63] indicated that test smells have negative impacts on software quality, particularly in terms of code comprehensibility, maintainability, and evolution. Other relevant studies have identified limitations in test smell detection [96] and test smell refactoring [83], as well as emphasized the correlation between test smells and design issues [87]. Furthermore, one study [89] concluded that the existing test smell types are insufficient to address the issues of poorly designed test code with inadequate testing practices, suggesting the existence of undiscovered poor programming practices in test code. Motivated by this finding, we aim to uncover and precisely define new test smell types by analyzing developers' discussions in Stack Overflow posts.

Test Smell Detection. Numerous studies have concentrated on test smell detection [70, 88, 91, 92, 101, 106]. Upon reviewing Table 1, it becomes apparent that while certain test smell detection tools are applicable to multiple **programming languages (PLs)**, the majority of them are primarily designed for Java (e.g., [72, 91–93, 102]), followed by Python [61, 66]. Conversely, only a limited number of studies have focused on other PLs, such as C++ [62] and Scala [65, 94]. Nonetheless, all of these detectors are solely capable of identifying smelly test scripts based on existing test smell types. Therefore, based on this observation, we firmly believe that the development of a novel test smell detector for potential test smell types is imperative.

Considering the aforementioned factors, which include the adverse impact of test smells on various software aspects, and the limited research on exploring and detecting new test smell types, our study aims to address these gaps. Specifically, we aim to identify and define new test smell types through extensive empirical investigation and subsequently develop corresponding detectors to accurately identify these smells within software code.

3 OVERVIEW

Given the intricate nature of this study, which encompasses both an empirical investigation into uncharted types of test smells and a design research endeavor aimed at creating corresponding detectors to identify them within the software, this section provides a comprehensive overview and high-level representation of our work. The primary objective is to establish a clear understanding of the interconnections between the empirical study section and the design research section.

To enhance readers' understanding, Figure 3 visually presents a high-level representation of the **research questions (RQs)** and their relationships with the inputs and outputs of each study section. Specifically, our study begins with a comprehensive empirical study that addresses two main RQs, i.e., RQ1 and RQ2. RQ1 involves a mixed analysis (i.e., a number of qualitative and quantitative analysis) aimed at identifying new types of test smells, while RQ2 employs a qualitative analysis to summarize the characteristics of these test smells. Building upon the findings of RQ1, we proceed to conduct design research to propose a method for detecting the test smells identified in RQ1 within real-world software projects. To ensure the effectiveness of our detector, we design detection methods for each test smell type based on their definitions and characteristics derived from RQ1 and RQ2. Within our design research section, we employ both quantitative analysis (RQ3) and qualitative analysis (RQ4) to validate the effectiveness and practicality of our test smell detector in real-world scenarios.

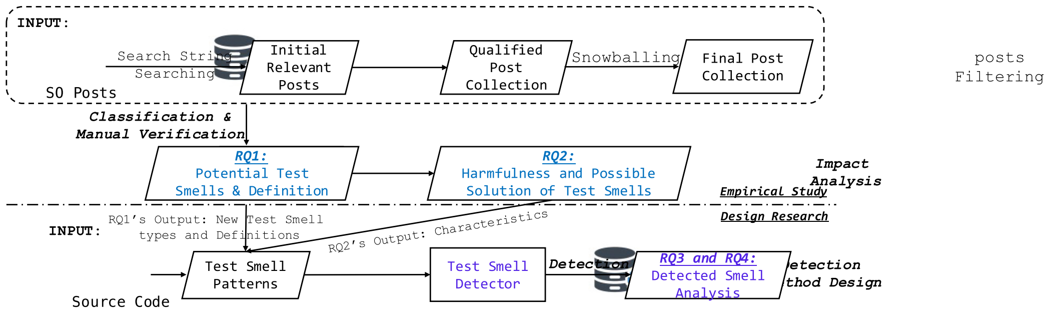

INPUT:
Search String Searching
SO Posts
Initial Relevant Posts
Qualified Post Collection
Snowballing
Final Post Collection
posts Filtering
Classification & Manual Verification
RQ1: Potential Test Smells & Definition
RQ2: Harmfulness and Possible Solution of Test Smells
Impact Analysis
Empirical Study
Design Research
INPUT:
RQ1's Output: New Test Smell types and Definitions
RQ2's Output: Characteristics
Test Smell Patterns
Source Code
Test Smell Detector
Detection
RQ3 and RQ4: Detected Smell Analysis
Detection Method Design

Fig. 3. The comprehensive overview of our study.

4 AN EXPLORATORY STUDY FOR NEW TEST SMELLS

In this section, we would like to explore new test smells and analyze their characteristics by answering the following two research questions (RQs).

RQ1: What potential test smells are explored from SO posts?

RQ2: What are the harmfulness and possible solutions of these test smells?

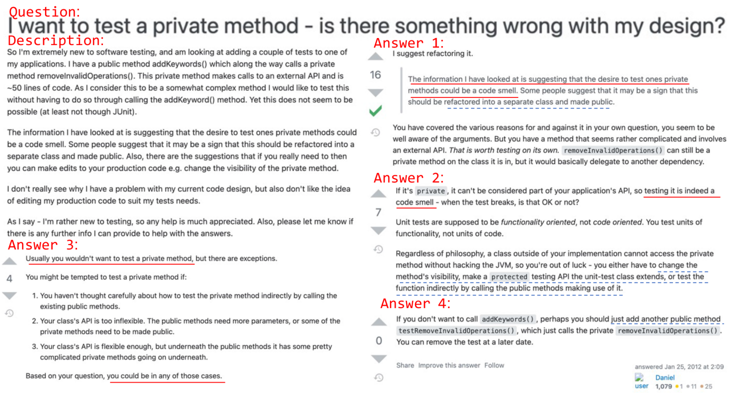
4.1 Methodology for Mining New Test Smells

Stack Overflow (SO) is one of the largest public platforms aiming to provide definitive answers for developers' technical issues, which encompasses a large number of discussions related to poor testing practices [4]. Therefore, we select the SO platform to explore controversial and new-introduced bad practices in the test code. The workflow of mining potential test smells mainly consists of three steps:

4.1.1 Search String Determination. To find as many relevant posts as possible, we conduct the search on the latest snapshot of SO. We first use the term "test smell" as a search string to collect relevant posts. To guarantee the completeness of the search, we refine the search string by checking the titles and descriptions of 48 relevant posts. The final search string, therefore, incorporates five search terms and is refined as:
"*test smell [java]*" and "*bad practice in test code [java]*" and "*poor practice in test code [java]*" and "*bad testing practice [java]*" and "*poor testing practice [java]*",
where "*[java]*" is the programming language studied in our study. Since the search string consists of five different terms, we use each one to search relevant SO posts, respectively, and then gather these results together as a search result collection. Besides, we adopt the snowballing strategy [98, 108, 109] to find out relevant posts but not found by the search string, i.e., similar post links provided by answers in collected posts. More specifically, suppose that an answer in one relevant post lists two additional links to similar posts. We include these two posts in the search result collection. Then, we go through the content of similar posts and stop the snowballing activity until new-introduced posts do not attach any other post links. The preliminary search result includes 231 posts.

4.1.2 Posts Filtering. We then filter out the unqualified posts with the following steps:

(i) We discard duplicate posts and posts without answers.
(ii) The posts and answers are filtered based on the following inclusion and exclusion criteria:
✓ The title and answers in an SO post must be relevant to test smells or bad/controversial practices in test code.


Question:
I want to test a private method - is there something wrong with my design?

Description:
So I'm extremely new to software testing, and am looking at adding a couple of tests to one of my applications. I have a public method addKeywords() which along the way calls a private method removeInvalidOperations(). This private method makes calls to an external API and is ~50 lines of code. As I consider this to be a somewhat complex method I would like to test this without having to do so through calling the addKeyword() method. Yet this does not seem to be possible (at least not though JUnit).

The information I have looked at is suggesting that the desire to test ones private methods could be a code smell. Some people suggest that it may be a sign that this should be refactored into a separate class and made public. Also, there are the suggestions that if you really need to then you can make edits to your production code e.g. change the visibility of the private method.

I don't really see why I have a problem with my current code design, but also don't like the idea of editing my production code to suit my tests needs.

As I say - I'm rather new to testing, so any help is much appreciated. Also, please let me know if there is any further info I can provide to help with the answers.

Answer 1:
16

I suggest refactoring it.

> The information I have looked at is suggesting that the desire to test ones private methods could be a code smell. Some people suggest that it may be a sign that this should be refactored into a separate class and made public.

You have covered the various reasons for and against it in your own question, you seem to be well aware of the arguments. But you have a method that seems rather complicated and involves an external API. *That is worth testing on its own.* `removeInvalidOperations()` can still be a private method on the class it is in, but it would basically delegate to another dependency.

Answer 2:
7

If it's `private`, it can't be considered part of your application's API, so testing it is indeed a code smell - when the test breaks, is that OK or not?

Unit tests are supposed to be *functionality oriented*, not *code oriented*. You test units of functionality, not units of code.

Regardless of philosophy, a class outside of your implementation cannot access the private method without hacking the JVM, so you're out of luck - you either have to change the method's visibility, make a `protected` testing API the unit-test class extends, or test the function indirectly by calling the public methods making use of it.

Answer 3:
4

Usually you wouldn't want to test a private method, but there are exceptions.

You might be tempted to test a private method if:

1. You haven't thought carefully about how to test the private method indirectly by calling the existing public methods.
2. Your class's API is too inflexible. The public methods need more parameters, or some of the private methods need to be made public.
3. Your class's API is flexible enough, but underneath the public methods it has some pretty complicated private methods going on underneath.

Based on your question, you could be in any of those cases.

Answer 4:
0

If you don't want to call `addKeywords()`, perhaps you should just add another public method `testRemoveInvalidOperations()`, which just calls the private `removeInvalidOperations()`. You can remove the test at a later date.

Share Improve this answer Follow
answered Jan 25, 2012 at 2:09
user
Daniel
1,079 ●1 ●11 ●25

Fig. 4. Example of a card for a "PMT" test smell.

✓ The test code in a post must be written in Java.
✗ A post cannot be incorporated into our study if it asks for a specific solution to address a bug/defect/error that does not arise from any test smells in the test code or a detailed implementation for the test code.
✗ Answers using other programming languages to answer Java questions should be excluded.
✗ Posts in which the total number of vote casts is negative will be discarded.

(iii) We filter out the posts that discuss known test smells as presented by Aljedaani et al. [56].

Eventually, we collect 69 relevant posts from the SO.

4.1.3 Open Card Sorting. Following the card-sorting method [98, 108, 109], we conduct a mixed analysis to classify and analyze the filtered posts in order to determine new test smells. We create one card for each post. A card includes the information of the question title, description of the question, answers, and votes as shown in Figure 4. Two researchers who have more than five years of Java development experience and major in studying test smells work independently to (i) categorize the cards that discuss the same controversial practice together, (ii) analyze answers' preferences, and (iii) determine potential test smells. Note that the initial two steps primarily encompass qualitative analysis, while the final step predominantly involves quantitative analysis. The detailed steps are:

❶ *Card Classification:* In this step, two researchers perform independent assessments by carefully analyzing the provided tags, question title, and description on each card, and then summarize or select keywords that accurately depict the main purpose of the question. It is important to emphasize that the process of summarizing or selecting appropriate keywords is generally not a challenging task. This is due to the fact that numerous SO posts already include highly relevant keywords in their tags or question titles. Additionally, posts discussing similar controversial testing practices often employ similar terminology. As a result, in the majority of cases, the researchers implicitly choose or summarize similar words as the keywords for each post. For instance, the two posts [5, 6] both discuss the same testing practice, which is testing private methods. It is noteworthy that both of these posts utilize the words "test" and "private method" in their questions' titles. As to the post [6], the summarized keywords of this post are presented as "test private methods" and "test private methods, fields or inner classes" by the two researchers. These descriptions exhibit evident similarities in both form and meaning, facilitating the straightforward categorization of the summarized keywords into distinct semantic groups. After that, to preserve the experience and opinion of different researchers as much as possible and mitigate the potential bias that may arise from the influence of a single researcher's opinion on the final classification results, they independently classify the cards discussing the same controversial practice into one category based on their summarized keywords. To integrate their classification results, they discuss to solve their differences in each category and reach a consensus. Initially, they get 11 different categories. After that, they filter out the categories that only contain one card or no more than three answers. Finally, eight categories with 65 valid posts are left. Cohen's Kappa [108] is used to measure the agreement between them. Their overall Kappa value is 0.86, indicating a strong agreement.

❷ Preference Voting for Answers: Then, for each controversial practice category, the two researchers re-read each answer in cards to analyze its preference (i.e., whether an answer agrees this controversial practice is a bad practice or test smell). They classify answers' preferences into two categories: (i) The answers that regard this practice as test smell are categorized into "***Test Smell (TS)***" answers. (ii) Others with a different view are "***Not a Test Smell (NTS)***" ones. As for some answers that provide specific solutions rather than directly expressing their preferences, they analyze these answers to determine their potential preferences, respectively. Their decisions are highly consistent, with a 0.81 Kappa value.

❸ Determining Potential Test Smells: After analysis of the answers' preferences, for each practice category, in this quantitative analysis step, they count the total number of votes that answers with the same preference get, respectively. They design two rules to determine whether a practice is a test smell: (i) A controversial practice is a test smell if the majority of answers believe it is a bad practice or a test smell (i.e., *#TS >#NTS*) and (ii) the controversial practice is a test smell if its "*Test Smell*" answers win more supports (i.e., votes) than "*Not a Test Smell*" answers (i.e., *#Vote(TS) >#Vote(NTS)*). Based on the two rules, they discover six potential test smells from 62 posts and name smells based on their controversial practices. Also, they analyze the harmfulness of six test smells and possible solutions from relevant answers for addressing them.

✌ ***Example.*** Figure 4 illustrates an example of a card for a controversial practice: testing private methods. The card includes four parts: i.e., question, description, answers, and votes of answers. The two researchers first read the question and description to understand the topic of this practice (i.e., testing private methods). Then, they examine the corresponding answers. Different answers have different preferences towards this practice. The first three answers explicitly express they agree that testing private methods is a code smell (see red lines). Although the last answer does not directly express its view, it presents a specific solution (i.e., *refactor test code: adding a public method and testing it instead of testing private methods*). It indicates that this answer's potential preference believes it is not a good idea to test private methods. Thus, all four answers agree testing private methods is a bad practice and achieves 27 (16 + 7 + 4 + 0) support votes. While the opposite preference has 0 answers with 0 votes. Except for Answer 4, Answers 1 and 2 also provide some possible solutions for eliminating these smells (see blue lines).

4.2 RQ1: Potential Test Smells

Results. Based on the statistical data of posts in Table 2, six potential test smells we determine are illustrated in Table 3. Here we give a clear definition and a specific example test code in a real-world project for each smell type.

(1) **Private Method Test (PMT)** is a method-level test smell that refers to the test code used for accessing and testing private methods. Whether and how to test private methods is the most widely discussed controversial practice in SO, involving 33 posts and 181 relevant answers. 115 answers with 3,954 support votes perceive designing test cases for private methods can be regarded as a

Table 2. The Statistical Data of New Test Smells in SO Posts

Test smell	#SO post	#TS answer	#Vote(TS)	#NTS answer	#Vote(NTS)
PMT	33	115	3,954	61	2,830
POW	2	7	112	2	2
NDT	4	4	13	0	0
PTO	11	23	101	2	4
AL	10	26	39	5	7
RT	2	5	9	1	1
Total	62	180	4,228	71	2,844

test smell, much higher than the number of answers holding different views. Note that although this poor practice stimulates many discussions in SO, it has not been formally defined as one of the test smells, causing no research to further analyze or detect it. In our study, we formally formulate this bad testing practice as one of the test smell types based on experimental results.

✌ ***Example.*** An example PMT test smell (see Table 3) is detected from a real-world popular project named "*fastjson*" with 24.9k stars. The example adopts the reflection method to access and invoke a function (i.e., `getDeclaredMethod()`) to test a private method named "`checkTime`". For better understanding, we provide the source code of the private method beside this smelly test case in Table 3.

(2) **Parameter Order Wrong (POW)**, as a statement-level test smell, refers to the test case whose assertion holds the wrong parameter order. In testing frameworks, the specification for the parameter order in some assertions is easy to make developers confused. For instance, "`assertEquals()`", as one of the most common assertions, is used to determine whether the actual value obtained from the code-under-test is equal to the expected value. Thus, this assertion often has two parameters for actual and expected values. However, the API specification of this assertion in different testing frameworks (e.g., JUnit and TestNG) asks for different parameter orders. In JUnit, the first parameter is the expected value, while the expected value is at the second position in the parameter order in TestNG. This inconsistency confuses developers about the parameter order, leading to assertion misuse. Consider (i) smells are associated with bugs or errors [79, 106] and risk behaviors [86] in software, and (ii) developers who work for test scripts in many real-world projects often make such mistakes (i.e., create assertions in a wrong parameter order). We thus formulate this bad practice as a test smell for prompting existing projects to fix smelly test code and preventing similar problems from happening again. As shown in Table 3, two relevant posts with nine answers discuss this issue. Seven answers express that developers should stick to the specifications of testing frameworks and achieve 112 support votes. Two NTS answers believe it is unimportant for what parameter orders are in assertions as long as test cases can work. However, their view only gets two votes.

✌ ***Example.*** Table 3 shows a POW test smell detected from a project with 3.1k stars. The project applies the JUnit framework. According to the correct API usage (see the right of the second row), the first parameter of the "`assertEquals`" assertion should be the expected value. However, developers place the result of the code-under-test (i.e., the actual value) as the first parameter in the assertion, violating the corresponding specification in JUnit.

(3) **Non-deterministic Data Test (NDT)** refers to the test code that introduces non-deterministic randomness. Test code sometimes utilizes non-deterministic random data (e.g., unseeded random data) as inputs of the code-under-test. Since the non-deterministic random data can be produced in different ways and used at multiple places in a test case, NDT is regarded as a method-level smell. Generally, even in test scenarios where random data is often used (e.g., Property-based testing [7] such high-risk behavior may cause test data to be unable to reproduce

Table 3. The Definitions and Examples of Six New Test Smell Types

Test Smell	Definition
Private Method Test (PMT)	**A test method that tries to access and test private method.**

Example: *project name:* **fastjson**, *star number:* **24.9K**, ***file name:*** **JSONScannerTest.java**

Test code:

```
Class<?> c = Reflector.forName("com.alibaba.fastjson.
parser.JSONScanner");
Method m = c.getDeclaredMethod("checkDate", Reflector.
forName("char"), Reflector.forName("char"), Reflector.
forName("char"), Reflector.forName("char"), Reflector.
forName("char"), Reflector.forName("char"), Reflector.
forName("int"), Reflector.forName("int"));
m.setAccessible(true);
boolean retval = (Boolean)m.invoke(null, y0, y1,
y2, y3, M0, M1, d0, d1);
Assert.assertEquals(true, retval);
```

Production code:

```
private boolean checkTime(char h0, char h1, char m0,
char m1, char s0, char s1) {
        if (h0 == '0') {
            if (h1 < '0' || h1 > '9') {
                return false;
            }
            ...
                return true;
    }
```

Test Smell	Definition
Parameter Order Wrong (POW)	**A test case whose assertion holds the wrong parameter order.**

Example: *project name:* **librec**, *star number:* **3.1K**

Test code:

```
import static org.junit.Assert.assertEquals;
@Test public void test01ColumnFormatUIR() throws
    LibrecException { ...
        TextDataModel dataModel = new TextDataModel(conf);
        dataModel.buildDataModel();
        assertEquals(getDataSize(dataModel), 13); }
```

API usage:

```
import static org.junit.Assert.*;
...
static void assertEquals(double expected,
                                double actual)
```

Test Smell	Definition
Non-deterministic Data Test (NDT)	**A test method that introduces non-deterministic randomness in test data.**

Example 1 *(random number)*): *project name:* **apache/curator**, *star number:* **2.8K**, ***file name:*** **TestInterProcessReadWriteLock.java**

```
import java.util.Random;
@Test public void testBasic() throws Exception { ...
    final Random random = new Random(); ...
    if ( random.nextInt(100) < 10 ) {
        doLocking(lock.writeLock(), concurrentCount,
            maxConcurrentCount, random, 1);
        writeCount.incrementAndGet(); } }
```

Example 2 *(random string*): **flowable-engine**, *star number:* **4.5k**, ***file name:*** **JsonTest.java**

```
import org.apache.commons.lang3.RandomStringUtils;
@Test public void testUpdateJsonValueToLongValueDuringExecution()
{ ... String randomLongStreetName = RandomStringUtils.
randomAlphanumeric(processEngineConfiguration.
    getMaxLengthString() + 1); ...
assertThatJson(customerVar).isEqualTo("{name:
'Kermit',address:{street:'randomLongStreetName'}}");}
```

Test Smell	Definition
Poor Test Organization (PTO)	**Test code has poor code organization in the case of unit testing.**

Example 1 (*Test code in production code*): *Project name:* **docker-java**, *Star number:* **2.3K**, *File name:* **DockerHttpClientTCK.java**

```
/* File name: docker-java/docker-java-transport-tck
/src/main/java/com/github/dockerjava/transport
 /DockerHttpClientTCK.java */
// Test code:
@Test public void testHijacking();
@Test public final void testPath();
// production code:
private DockerHttpClient createDockerHttpClient();
private static class AttachContainerTestCallback;
private static class AttacheableContainer;
```

Example 2 (*Bad test code organization*): *Project name:* **basebox**, *Star number:* **801**

```
//The organization of test code:
    baasbox/test/core/TestCode.java
    baasbox/test/resources/TestCode.java
    baasbox/texst/unit/TestCode.java
    Testfiles.java ...
//The organization of production code:
    baasbox/app/com/baasbox/commands/ProductionCode.java
    baasbox/app/com/baasbox/do/ProductionCode.java
    baasbox/app/com/baasbox/db/ProductionCode.java
    ...
```

Test Smell	Definition
Assertion in Loop (AL)	**A test case that creates an assertion in the loop.**

Example: *Project name:* **hapi-fhir**, *Star number:* **1.5K**, *File name:* **DividerVisibilityManagerTest.java**

```
@Test public void testInvalidChar() {
String[] invalidPatterns = new String[]{"", "1", "ABC", "#x0001 - #x - #x0000", "#x0000 #x0022"};
for (String i : invalidPatterns){ assertThrows(IllegalArgumentException.class, () -> { myFilter.add(i);}); }
```

Test Smell	Definition
Return in Test (RT)	**A test case that returns a value.**

Example: *Project name:* **ninja**, *Star number:* **1.9K**, *File name:* **ApplicationController.java**

```
@Path("/mode/test") @GET @Test public Result testAnnotatedTestRoute() { return Results.text().render("test mode works."); }
```

and make it hard for developers to trace the reasons for failed test cases, thus decreasing the reliability of test code and even causing flaky tests [90]. Because this poor testing practice has not been included in the existing test smell list, we provide a formal definition of this bad testing practice for serving further research in the software engineering community. Four SO posts discuss this issue. All answers agree that using non-deterministic random data is not a good practice in testing.

✌ ***Example.*** We take two NDT test smells as examples in Table 3. The two examples generate nondeterministic random numbers and strings as test data, respectively.

(4) **Poor Test Organization (PTO)** refers to test code with poor code organization in the case of the unit testing. As a project-level test smell, PTO smells can be mainly classified into two categories: i) ***Test code in production code (TCPC)*** denotes that test code is mixed with production code; ii) ***Bad test code organization (BTCO)*** denotes that production code and test code are not organized in a standard directory layout. 11 relevant posts seek a better solution for production and test code organization. 25 answers reply this question of which 23 answers believe that Java projects should be managed according to the standard directory layout [8] (i.e., parallel folder structure), where the src/main directory stores production code files and the src/test directory stores test code files. For instance, if the production code is placed in the "`src/main/a/b`" folder, developers should create a folder named "`src/test/a/b`" to place the corresponding test code.

✌ ***Example.*** Table 3 shows two types of PTO test smells. The first example is a TCPC test smell, because we can know from the method signature that the source file includes not only the production code but also two test methods. We take a project "*basebox*" as an example to illustrate the BTCO test smell. Folders for production code have no parallel structure to that for test code. Such poor code organization causes it to be time-consuming to find the corresponding test code to a production code fragment.

(5) **Assertion in Loop (AL)** refers to the test case that creates an assertion in the loop. Typically, a high-quality test case should be small, isolated, atomic, easy to understand, and executed fast [54]. On the one hand, the loop structure complicates test cases and increases the maintenance cost for test code. On the other hand, writing the assertion in the loop violates the principle that one test case should have only one assertion and run it once [9]. We search for 10 posts related to this issue from SO, including 31 answers. 25 answers oppose using loop statements in test code, while 6 answers express using the loop statement is acceptable as long as the test case works. For instance, an answer in the post [10] says that "*I'm generally OK with a loop in a test as long as there's only one occurrence of an assertion in the test.*". To ensure the correctness of proposed test smells, we examine five popular GitHub projects and also submit issue reports to discuss with developers. We find that although some believe that the loop, in some cases, is a necessary structure in order to test code-under-test in more conditions, they still admit a loop structure that has an assertion is indeed a poor testing practice, which can be refined by the "`@ParameteredTest/@Parametered`" annotation in Junit/TestNG or introducing deterministic randomness to break down such bad loops. Similarly, writing assertions in loops can be a poor testing practice from the perspective of a testing framework. This explains why they designed these annotations. Therefore, we regard test cases that create the assertion in the loop structure as AL test smells.

✌ ***Example.*** Table 3 describes a typical AL test smell. The value-in-loop is a new-created instance that consists of a limited number of strings. Creating an assertion in the loop violates the principle: one test case, one assertion. Indeed, such test cases can be refactored by the repeated test feature in JUnit or TestNG to decompose the bad loop structure.

(6) **Return in Test (RT)** refers to the test case that returns a value, which is a statement-level smell. The role of a test case is to determine whether the code-under-test performs as expected and confirm that the project satisfies related standards, guidelines, and customer requirements [11]. It means that a test case is used for judging, not returning a value. Five out of six answers in two relevant posts agree test cases should not return a value. Consider that writing the return statement in test cases is a bad implementation solution. Thus, RT should be a new test smell type according to the definition of the test smell (i.e., poor design and implementation choices [78, 95]). Note that some not all test cases that contain a return statement are treated as RT test smells. Some code-under-test is once in a while re-written in test cases to examine whether the code-under-test

Table 4. The Harmfulness of Test Smells

TS	Harmfulness	Description Examples	#M
PMT	Design smell	"*When you have private methods in a class and you want to Unit-test them, this is considered as a sign for a code smell, ...*" [12]	12
	Break the encapsulation	"*Testing private methods kind of defeats the point of encapsulation ...*" [13]	4
	Dead code	"*Private methods are consumed by public ones. Otherwise, they're dead code*" [14].	2
	Hard to maintenance	"*Testing implementation details (private methods) directly would make the tests harder to maintain and the code-under-test more difficult to refactor.*" [15]	2
	Burden the performance and execution of the application	"*If you change production code just for the sake of easier testing then you can burden the performance and the execution of the application in some way.*" [14]	1
POW	Inconsistency in stylistic	"*Another reason is stylistic. If you like lining things up, the expected parameter is better on the left because it tends to be shorter.*" [16]	3
	Poor readability	"*An ulterior purpose of "assertEqual()" is to demo code for human readers.*" [16]	1
	Wrong log information	"*You spend time/effort tracking down the wrong issue trying to trace the source of the actual value that wasn't actually the actual value.*" [17]	1
NDT	Nondeterministic test	"*Thus, in our unit test, we replace the random generator ..., the identifier used becomes deterministic.*" [18]	2
	Unexpected test failures	"*Using randomized data may lead to unexpected unit test failures.*" [19]	1
	Additional synchronization	"*Most random number generator classes are thread-safe and therefore introduce additional synchronization. It's the memory synchronization that may change the timing of your program.*" [20]	1
PTO	Poor code organization	"*Today's best practice is to use separate source roots but use parallel package structures.*" [21]	8
	Code mess	"*So what will happen if having all codes + tests together?*" [22]	5
	Increase software size	"*We create a build of source code for production and it shouldn't contain test code. An unnecessary increase in code size.*" [22]	3
AL	High complexity	"*Remember that adding more complexity to tests makes it harder to work with them.*" [10]	4
	Invalid testing	"*With lots of tests and wide parameter ranges it is just a waste of resources.*" [10]	2
	Poor traceability	"*When a test fails it should be immediately obvious what caused it. You shouldn't have to analyze the test code to figure it out.*" [10]	2
	High time cost	"*Running for loops in tests isn't terrible, but you definitely don't want to create a new test for every value. The overhead of starting a new test each time will slow you down much more than duplicating the for loops.*" [23]	1
RT	Wrong test case	"*If a TestNG "@Test" method is declared as returning a value, it is not treated as a test method.*" [24]	2
	Non-independent test	"*Test methods should be executed independently.*" [24]	1

Table 5. Possible Solutions for Eliminating Test Smells

TS	Possible Solution	Acceptance
PMT	Refactor test code	✓
	Reflection method	✗
	Change visibility modifiers	✗
	Mock tests	✗
POW	Correct the wrong parameter order in assertions	✓
NDT	Set a seed number for random generators to produce deterministic random data	✓
	Create a class and store n random data beforehand and then hand them out without synchronization	✓
	Adopt hashing algorithm to generate deterministic randomness	✓
PTO	Separate source and test code and use parallel package structures	✓
AL	Utilize repeated test feature	✓
	Create test cases for the edge cases and vulnerable values	✓
RT	Remove the return statement in test cases	✓

can work correctly in some conditions. If the code-under-test has a return statement, the test case that rewrites it also contains a return statement. Such test cases are not RT smells. A test case is an RT smell if it returns a value.

✌ ***Example.*** The test case"`testAnnotatedTestRoute()`" returns a value but without an assertion.

4.3 RQ2: Harmfulness and Possible Solutions of Test Smells

In this RQ, we analyze the negative impacts of six test smells on software quality and summarize possible solutions for addressing them.

Results. Table 4 lists the harmfulness of different test smells, excerpts relevant descriptions, and counts the frequency of negative impacts mentioned in SO posts (*#M*). Table 5 presents possible solutions for eliminating these test smells.

4.3.1 ***Private Method Test (PMT).*** ❶ *Harmfulness:* PMT test smells may influence software quality from five perspectives. 16 answers mention that testing the private methods not only breaks encapsulation but is also a sign of the design smell present in production code. Because private methods in well-design projects can be thoroughly tested through corresponding public methods. For the same reason, some believe that if a private method needs to be tested individually, it means that no public methods invoke it and it is a dead code. 3 answers hold that PMT smells make the tests harder to maintain and code-under-test more difficult to refactor and burden the performance and execution of projects. ❷ *Possible Solutions:* Refactoring the PMT test smell is a widespread acceptance solution among relevant answers. Although another three solutions (i.e., refection method, changing visibility modifiers, and mocking tests) are also widely mentioned in answers, these are not regarded as good solutions to address PMT smells. For example, the accepted answer in a post with 92 votes says "*If you are testing your own code, the fact that you need to use Reflection means your design is not testable, so you should fix that instead of resorting to Reflection*" [25]. Another answer also mentions that "*I can't really say more, but it is indeed regarded as a poor practice to use reflection*" [25]. Changing the "private" into other modifiers is another common option. But an answer states that "*If I ever need to make something public to test it, this usually hints that the system under test is not following the Single Responsibility Principle*" [5], which illustrates that changing modifiers visibility is not a good solution for testing private methods. It is also not a good idea to utilize the mocking technique to simulate the behavior of a private method for the sake of testing it. For instance, although an answer provides a specific solution that uses the mocking technique to solve the problem, it also points out that "*To give the answer you asked for (using JMockit's partial mocking). However, I would not recommend doing it like that. In general, private methods should not be mocked.*" [26].

4.3.2 Parameter Order Wrong (POW). ❶ *Harmfulness:* POW smells harm the quality of the test code from three perspectives: i) First, POW test smells are inconsistent in stylistics to the correct assertion usage, and ii) decrease the readability of test cases. iii) Another problem is that POW smells make log messages wrong. Here is the correct API usage of the "`assertEquals`" assertion in Junit: "`assertEquals(expected, actual)`", and the correct log message should be: "*expected [expected value] but found [actual value]*". However, once an assertion holds the wrong parameter order, the log message changes into: "*expected [actual value] but found [expected value]*", which makes developers confused while reading log information of failed test cases and makes it difficult to find the failure reason. ❷ *Possible Solutions:* Follow the correct API usage to fix the parameter order of assertions in POW test smells.

4.3.3 ***Non-deterministic Data Test (NDT).*** ❶ *Harmfulness:* NDT smell as one of the most common poor testing practices makes test code unreliable. Specifically, even if a test case with the NDT smell detects a bug, it is difficult for testers to know what test data causes it to fail and reproduce the failed test case. Besides, many random generators introduce additional synchronization to ensure thread-safe, which occurs error/bias when monitoring the running time of programs. ❷ *Possible Solutions:* A better solution for the removal of NRT test smells is to generate deterministic random data as test data. Two commonly used solutions are: employing (i) a simple random generator with a seed number or (ii) a hashing algorithm to generate deterministic random data can eliminate not only the additional synchronization but also the uncertainty in test cases.

4.3.4 ***Poor Test Organization (PTO).*** ❶ *Harmfulness:* Three negative impacts of PTO test smells are revealed in 11 SO posts. Eight answers mention that tests and production code should be separated and placed in folders with a parallel structure otherwise test scripts are in a poor code organization. In addition, mixing test and production code together is a kind of code mess. ❷ *Possible Solutions:* The best organization is to arrange test and production code into packages with a parallel structure.

4.3.5 ***Assertion in Loop (AL).*** ❶ *Harmfulness:* A general principle in software testing is to keep test cases small and simple [27]. AL smell increases the complexity of test code, reduces its readability, and makes it difficult for developers to track down the potential causes of failed test cases. Another negative effect is that loops in test cases may lead to invalid tests since some values in the loop may never identify errors but are just a waste of calculation resources and running time. Besides, if an assertion is in the loop, the test case violates the principle, i.e., a test case only has and runs an assertion once [56]. ❷ *Possible Solutions:* Two practical solutions can eradicate the negative impacts of the loop in the test. The most-mentioned solution is using the repeated test feature (i.e., with the annotation "`@ParameterizedTest`") to substitute the loop structure. For example, an answer says "*You are looking for Parameterized tests*" [28]. Some answers consider that creating test scripts for bordering and additional representatives (e.g., vulnerable values) is a more effective option to alleviate the bad influence of invalid testing than using the loop. For instance, an answer mentions "*You'd usually be fine with just testing the edge cases, and maybe one additional representative of one of the ranges*" [29].

4.3.6 ***Return in Test (RT).*** ❶ *Harmfulness:* A test case should not be declared as returning a value [4]. Some testing frameworks (e.g., TestNG) do not regard a test method with a returned value as a test case. Once a test case returns a value, it is likely to illustrate that this test case has dependencies with other test methods, breaching the principle that test cases should be independent. ❷ *Possible Solutions:* Remove the return statement in RT test smells.

5 TEST SMELL DETECTION

It is challenging to conduct test smell detection since one test smell type may occur in test code in multiple forms. These forms are closely related to many aspects, such as the differences in effect ranges of test smells on production code. Hence, we first perform source code classification to split test code from production code and analyze the influence zones of smelly tests on production code. To represent different forms of smells, we examine example test scripts in SO posts and test code in popular real-world projects to summarize common code patterns of different test smell types according to their influence zones and causes. After that, we design a detection method for each test smell type based on the summarized code patterns. A detection method comprises a set of syntactic rules derived from **Abstract Syntax Tree (AST)** analysis, specifically tailored to match the code patterns associated with a particular test smell type. Finally, we utilize these detection methods to identify test smells from projects. In this section, we detail the workflow of our detector. Figure 5 shows the overall framework of our test smell detector, which mainly includes three phases: (1) splitting test code and production code; (2) mining code patterns of test smells, and (3) designing detection methods for test smells.

5.1 Splitting Test Code and Production Code

We follow the convention (i.e., Java developers often name a test file with the word "Test".) used in Wang et al. [105] to distinguish test code from the production code. In addition to this convention, we observe another one when examining the code organization of the project. Developers often create a folder whose path usually includes the string "*/test/*" or "*/Test/*" to store test files (e.g., "*.../src/test/java/.../ServerClusterRemoteDocumentIT.java*" [30]). Therefore, to avoid misclassification, we refine the rule as: (1) The source file is regarded as a test file if the word "Test" in its

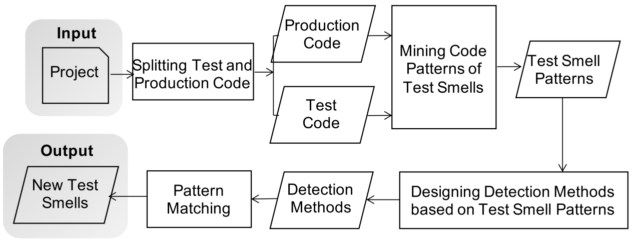

Input
Project
Splitting Test and Production Code
Production Code
Test Code
Mining Code Patterns of Test Smells
Test Smell Patterns
Output
New Test Smells
Pattern Matching
Detection Methods
Designing Detection Methods based on Test Smell Patterns

Fig. 5. The framework of our test smell detector.

file name; and (2) the source file is a test file if its folder path contains the string "*/test/*" or "*/Test/*" despite its file name does not comply with rule one. Then, the remaining files are regarded as files of production code.

5.2 Abstracting Code Patterns of Test Smells

Since one test smell type may exist in various patterns in test code, two main challenges for abstracting code patterns of different test smells lay in that:

Challenge 1: Different test smell types in the test code have different effects on the source code, and thus we need to analyze different scopes of source code for abstracting various patterns of test smells in the same type from test scripts. For example, according to the definition, RT smells only occur in test methods and produce no effect on the production code, which allows us to only focus on test files to detect all of them without considering other source files. In contrast, to detect a PMT smell, we need not only to examine the syntax of the test code to determine whether its syntax satisfies one of the code patterns for this smell type but also to confirm whether the corresponding code-under-test is a private method in production code. In other words, RT smells affect only test code, but the code pattern of PMT smells has relationships to both test methods and production code. Thus, we need to tailor the analysis scope for each smell type to summarize its code patterns.

Challenge 2: It is necessary to combine various aspects, including potential solutions, API usages, and the syntax of test code, to comprehensively abstract code patterns for different test smell types due to their different characteristics. Take the PMT and RDT smells as two illustrating examples. Code patterns of PMT smells are derived from three poor solutions present in Table 5. In detail, we summarize the possible packages and APIs for implementing these three solutions. These APIs are abstracted into specific syntaxes, which together with package information consist of code patterns of PMT test smells as shown in Table 6. As for the RDT test smell, its code patterns are the syntax structures of common implementations for generating non-deterministic random data in test code. Similar to the PMT smell, the abstraction for RDT code patterns also involves the analysis of packages and APIs.

To address these challenges, we examine example test scripts in both SO posts and five popular large-scale projects in GitHub (i.e., *openj9*, *platform_frameworks_support*, *aws-sdk-java*, *junit4*, and *CoreNLP*) to summarize the common code patterns of each type of test smells. To summarize as many code patterns as possible, we also search for other implementation solutions (e.g., the involved packages and corresponding APIs) and code examples for each smell type on the Internet and abstract them as new code patterns. Finally, the code patterns of different test smell types are listed in Table 6, which provide the basis for designing corresponding detection methods. We can observe that code patterns of PMT, POW, and NDT smells are highly related to package information and specific API usages, while code patterns of NDT, PTO, AL, and RT smells are abstracted into syntax structures of Java.

Table 6. Code Patterns for Test Smells

TS	Pattern Name (PM)	Code Pattern (CP)	Abbreviation&Description (AD)
PMT	***ReflectionPattern(RP) 1:***	**import** $java.lang.reflect$ $getDeclaredMethod(PrivMethod, Params)$	$RPs = \{RP\ 1, 2, 3\}$ $RP\ i =< RPImport\ i, RPAPISet\ i >, i \in [1, 3]$
	ReflectionPattern(RP) 2:	**import** $ReflectionTestUtils$ $invokeMethod(PrivMethod, Params)$	$RPImports = \{RPImport\ 1, 2, 3\}$ $RPAPISets = \{RPAPISet\ 1, 2, 3\}$
	ReflectionPattern(RP) 3:	**import** $ReflectionUtils$ $invokeMethod(PrivMethod, Params)$	$RPAPISet\ i = \{RPAPI\ i\_1\}$
	MockPattern(MP) 1:	**import** $PowerMockito$ $PowerMockito.when(PrivMethod, Params)$	$MPs = \{MP\ 1, 2\}; MPAPISet\ i = MPAPI\ i\_1$ $MP\ i =< MPImport\ i, MPAPI\ i >, i \in [1, 2]$
	MockPattern(MP) 2:	**import** $powermock$ $Whitebox.invokeMethod(PrivMethod, Params)$	$MPImports = \{MPImport\ 1, 2\}$ $MPAPISets = \{MPAPISet\ 1, 2\}$
	VisAnnoPattern(VAP):	***VAPAnnotation:*** $@VisibleForTesting$	
POW	***CommonUsePattern(CUP) 1:***	**import** $org.testng.Assert$ $assertEquals(actual, expected)$ $/assertNotSame(actual, expected)$ $/assertEqualsNoOrder(actual, expected)$	$CUPs = \{CUP\ 1, 2\}$ $CUP\ i =< CUPImport\ i, CUPAssertSet\ i >$ $i \in [1, 2]$ $CUPImports = \{CUPImport\ 1, 2\}$
	CommonUsePattern(CUP) 2:	**import** $org.junit.Assert$ $assertEquals(expected, actual)$ $/assertSame(expected, actual)$ $/assertArrayEquals(expected, actual)\}$	$CUPAsserSets = \{CUPAssertSet\ 1, 2\}$ $CUPAssrtSet\ i = \{CUPAssrt\ i\_1, i\_2, i\_3\}$
	SpecialUsePattern(SUP) 1:	**import:** $org.testng.Assert$ $assertEquals(actual, expected, msg)$ $/assertNotSame(actual, expected, msg)$ $/assertEqualsNoOrder(actual, expected, msg)$	$SUPs = \{SUP\ 1, 2\}$ $SUP\ i =< SUPImport\ i, SUPAssertSet\ i >,$ $i \in [1, 2]$ $SUPImports = \{SUPImport\ 1, 2\}$
	SpecialUsePattern(SUP) 2:	**import** $org.junit.Assert$ $assertEquals(msg, expected, actual)$ $/assertSame(msg, expected, actual)$ $/assertArrayEquals(msg, expected, actual)$	$SUPAssertSets = \{SUPAssertSet\ 1, 2\}$ $SUPAssrtSet\ i = \{SUPAssrt\ i\_1, i\_2, i\_3\}$
NDT	***RandomDataPattern(RDP) 1:***	**import** $java.util.Random$ $/*create\ a\ random\ generator*/$ $Random\ rnd = new\ Random()$ $/*generate\ a\ random\ number*/$ $rnd\_data = rnd.invokeAPIs()$ $/*use\ the\ random\ number*/$ $Assign/Express/ConditionStmt(rnd\_data)$	$RDPs = \{RDP\ 1, 2, 3\}$ $RDP\ i =< RDPImport\ i, RDPSyntaxSet\ i >,$ $i \in [1, 2, 3]$ $RDPImports = \{RDPImport\ 1, 2, 3\}$ $RDPSyntaxSets = \{RDPSyntaxSet\ 1, 2, 3\}$ $RDPSyntaxSet\ i =$ $\{RDPSyntax\ i\_1, i\_2, i\_3\},$ $when\ i == 1$
	RandomDataPattern(RDP) 2:	**import** $Math$ $/*generate\ a\ random\ number*/$ $rnd\_data = Math.random()$ $/*use\ the\ random\ number*/$ $Assign/Express/ConditionStmt(rnd\_data)$	$RDPSyntaxSet\ i = \{RDPSyntax\ i\_1, i\_2\},$ $when\ i \in [2, 3]$
	RandomDataPattern(RDP) 3:	**import** $RandomStringUtils$ $/*generate\ a\ random\ string*/$ $rndStr = RandomStringUtils.func(para)$ $/*use\ the\ random\ string*/$ $Assign/Express/ConditionStmt(rndStr)$	
PTO	***TCPCPattern(TCPCP):***	***TCPCPAlgorithm:*** $productionFile\ i \in productionFileSet$ $producMethodSet\ i =$ $\{Methods\}\ in\ productionFile\ i$ $GetAnnota(Method\ k).contain(@Test),$ $whereMethod\ k \in producMethodSet\ i$	$productionFileSet$: a file set for all production files. $producMethodSet\ i$: a method set for all methods in the ith production file. $GetAnnota()$: a function for getting annotations for a method, and it returns an annotation list.
	BTCOPattern(BTCOP):	***BTCOPAlgorithm:*** $ParallelMapping(TestFileFolder\ i,$ $ProduFileFolderSet) \in \{True, False\}$ $GoodTestFolderSet.add(TestFileFolder\ i),$ $when\ ParalleMapping() == True$ $parallel\_ratio = testFilesInGTFS/allTestFiles$	$ParallelMapping()$: a function for determining whether exists a production code folder that has a parallel structure to the ith folder of test files. $GoodTestFolderSet(GTFS)$: a set for gathering test folders that are parallel to source code folders. $parallel\_ratio$: a ratio between the number of test files in GTFS and the total number of files.
AL	***ForLoopPattern(FLP):***	***FLPSyntax:*** $MemberReference\ MR = new\ MemberReference()$ $stmt.control.condition.operandr ==$ $len(MR)/Integer$ $ForBlockStmts = ForStatement.body.statements$ $Type(ForBlockStmts[i]) == AssertionStatement$	MR: is a class instance with multiple elements. The condition in a loop should be an integer number.
	EnForLoopPattern(EFLP):	***EFLPSyntax:*** $MemberReference\ MR = new\ MemberReference()$ $ForStatement.control.iterable == MR$ $ForBlockStmts = ForStatement.body.statements$ $Type(ForBlockStmts[i]) == AssertionStatement$	The iterable variable in a loop should be the elements in a class instance.
RT	***RTPattern(RTP):***	***RTSyntax:*** $Type(Stmts[i]) == ReturnStatement$	$Stmts[i]$: the ith statement in a test method

5.2.1 Code Patterns for PMT Smells. Code patterns of PMT smells can be concluded into three categories (i.e., RPs, MPs, and VAP), including six specific patterns. They are abstracted from three poor specific implementations (i.e., the reflection method, mock tests, and visible annotation) in Table 5. Specifically, we summarize three **reflection patterns** (i.e., **RPs**), two **mock patterns** (i.e., **MPs**), and one **visible annotation pattern** (i.e., **VAP**) by examining the example test code. As shown in Table 6, each pattern in RPs and MPs incorporates the necessary package information and the corresponding API. For instance, developers can adopt the reflection method to test private methods by invoking the function "`getDeclaredMethod()`" in the package "`java.lang.reflect`" or utilize the API named "`PowerMockito.when()`" in the package "`PowerMockito`" by mock tests to access private methods. The first parameter of summarized APIs in RPs and MPs, i.e., "`PrivMethod`", denotes the name of the method-under-test. The second parameter "`Params`" can be filled into a set of values, denoting parameters of the method-under-test. VAP is the third pattern for PMT smells. It refers to making the test code able to access and test production code by changing the modifier of the code-under-test from private to others. In general, developers leverage the "`@VisibleForTesting`" annotation to achieve this. Besides, in order to simplify the representation of code patterns, we provide some abbreviations and descriptions in the last column of Table 6. These abbreviations and descriptions are also beneficial for formulating the syntactic rules in detection methods for different smells. For example, we use `RPs={RP 1, 2, 3}` to represent three reflection patterns for PMT smells and use `MPs={MP 1, 2}` to represent two all mock patterns. Considering that each RP/MP can be abstracted into a tuple: `<package info, corresponding APIs>`, we thus use `RP/MP i = <RPimport/MPimport i, RPAPISet/MPAPISet i)>` to describe a reflection/mock pattern. Where "`RPimport/MPimport i`" represents the package info of the ith RP/MP, and "`RPAPISet/MPAPISet i`" represents a set that contains related APIs in the ith RP/MP's package. Since only one related API is summarized from each RP/MP's package, the length of the set "`RPAPISet/MPAPISet i`" equals 1, i.e., `RPAPISet/MPAPISet i = {RPAPI/MPAPI i_1}`.

5.2.2 Code Patterns for POW Smells. After examining all APIs in two mainstream testing frameworks, i.e., JUnit and TestNG, we summarize three assertion APIs (i.e., `assertEquals()`, `assertNotSame()`, and `assertEuqalsNoOrder()`) that may have POW smells. These three easy-to-confusing assertions have two different usages in both JUnit and TestNG. The difference between the two usages lay in the number of parameters. In general, creating these three assertions with two parameters (i.e., the expected value and actual value) is the most common usage in test scripts. Whereas developers can also use the three assertions in a special way, i.e., by adding the third parameter in assertions for logging some messages. Based on the difference in usage, we classify the code patterns of POW smells into two categories: **common usage patterns (CUPs)** and **special usage patterns (SUPs)**. We can notice from Table 6 that assertions in SUPs have one more "`msg`" parameter than that in CUPs. Besides, each category has two specific code patterns designed for different testing frameworks, respectively. Similar to reflection/mock patterns for PMT smells, each code pattern of POW smells includes an import package and corresponding assertion APIs. But the difference is that each POW code pattern contains three assertion APIs rather than one. Here we also give some abbreviations and descriptions for POW code patterns. We take the common usage patterns (CUPs) for POW smells as an example. We use "`CUPAssertSet i`" to represent a collection that includes assertion APIs in the ith CUP and use "`CUPAsset i_n`" to represent the nth assertions in this collection. Since each CUP has three specific assertions, "`CUPAssertSet i`" can be equal to the collection "`{CUPAssert i_1, i_2, i_3}`".

5.2.3 Code Patterns for NDT Smells. Considering that random numbers and random strings are the two most common non-deterministic random data in test scripts, we thus aim to summarize code patterns for generating non-deterministic random numbers/strings in test code. As a result, we abstract three code patterns, two for generating non-deterministic random numbers and one for producing random strings. To help readers understand, we use pseudocode to describe the syntax and usage of these code patterns and provide a specific comment (see green words) for each line of pseudocode. As illustrated in the first code pattern of NDT smells (i.e., RDP 1), except for the first code line for clarifying the import package, another three lines of code are employed for creating a random generator, generating a non-deterministic random number through the generator, and using this random number in test scripts, respectively. Different from the first code pattern, the second one can generate a random number in the test method without the help of a random generator due to the difference in the imported packages. The third pattern outlines the key steps for generating non-deterministic random strings in test code. Similar to the second code pattern, it generates random data without the need to declare a random generator. As shown in Table 6, we use "`RDPSyntaxSet i`" to denote the pseudocode of the ith code pattern for NDT smells. Hence, the integer variable `i` ranges from 1 to 3. Consider that the pseudocode in RDP 1 has three lines, which has one more line than the other two code patterns, and thus "`RDPSyntaxSet i`" equals `{RDPSyntax i_1, i_2, i_3}` when `i==1`, otherwise `RDPSyntaxSet i={RDPSyntax i_1, i_2}`.

5.2.4 Code Patterns for PTO Smells. Since PTO smells are classified into two categories (i.e., TCPC and BTCO), we abstract the code pattern for TCPC smells and BTCO smells, respectively. The pattern for the TCPC smell is to abstract the production files that contain at least one method annotated by the "`@Test`" annotation. More specifically, "`productionFileSet`" signifies a collection of production code files in a project, and "`productionFile i`" is one of the source files in the collection. We utilize the "`producMethodSet i`" to collect the declared methods in "`productionFile i`". Once a method in "`producMethodSet i`" has the "`@Test`" annotation, the source file "`productionFile i`" is a smelly test file, and the corresponding project will be considered as a TCPC smell. In general, the best practice for code organization should (a) separate test code and production code and (b) organize the folders for test code and production code into a parallel structure. We regard the projects not following the best practice of code organization as the second type of PTO smells, i.e., BTCO smells. Hence, the pattern for BTCO smells aims to describe the projects whose test code is not well-organized in a parallel structure with its production code. In particular, the function "`ParallelMapping()`" can be used to determine whether a test file folder has a parallel structure to one of the production code folders and if yes, the test file folder will be appended in the '**`GoodTestFolderSet(GTFS)`**" set. The variable "`parallel_ratio`" calculates the ratio between the number of test files in the GTFS set and the total number of test files. Once the "`parallel_ratio`" has a low value, the project can be regarded as a BTCO smell.

5.2.5 Code Patterns for AL Smells. Considering that rare while-loops occur in test scripts, we mainly focus on abstracting for-loop structures in the test code.[2] In general, for-loop structures can be classified into two categories: common for-loop (e.g., `for(int i = 0; i <= 10; i++)`) and enhanced for-loop (e.g., `for(int i: ArrayList al)`). Hence, we tailor a code pattern for the common for-loop type and enhanced for-loop type, respectively, and use `FLP` and `EFLP` to denote them. As shown in Table 6, the two patterns are both composed of four specific syntaxes with few differences. Generally, when a test method has a for-loop structure, this test method would contain a statement for creating an instance of a class with a limited number of elements (i.e., `MemberReference MR = new MemberReference()`). For the code pattern of the common **for-loop (FLP)**, the number of elements in this instance (i.e., an integer number) is treated as the iteration

[2]Note that we only abstract the code patterns of for-loops from test methods that have no external dependencies on other test scripts.

times of the for-loop in the loop condition, (i.e., `stmt.control.condition.operandr==len(MR)`). For the code pattern of the **enhanced for-loop (EFLP)**, the created instance is directly regarded as an iterator of the loop (i.e., `ForStatement.control.iterable == MR`). Finally, once one of the statements in the for-loop block is an assertion, the test method with this loop is an AL test smell (see the third and fourth syntaxes in AL code patterns). Note that the reason for logging the created instance `MR` is to make it easier to refactor AL smells using the repeat test features in JUnit or TestNG.

5.2.6 *Code Patterns for RT Smells.* Of all the code patterns for test smells, the RT code pattern is the most concise one. It depicts the character of a test method that contains a statement with the "`ReturnStatement`" type.

5.3 Designing Detection Methods for Test Smells

To identify test smells from projects, we develop detection methods for different test smells based on summarized code patterns. Table 7 illustrates detection methods for different test smells. One test smell type may have multiple detection methods due to different types of code patterns for its smell type. Finally, once a test method matches one of the code patterns for a certain smell type, we regard this method as a test smell in this type.

5.3.1 *Detection Methods for PMT Test Smells.* The main idea of detecting the PMT smell is to determine whether the code-under-test in a test case is a private method. To achieve this, we first gather all private methods from the production code into a set (denoted as "`PrivMethodSet`") by checking their modifiers. Following that, we design corresponding detection methods for three PMT code patterns. Due to the high similarity between the first two PMT code patterns (i.e., RPs and MPs), their detection methods are similar. Specific syntactic rules of these two detection methods are as shown in Table 7: (i) For a test method, the import information of its file should contain at least one package in RPs and MPs (i.e., `∃R/MPImports[i] ∈ TMImports`). (ii) Consider a mapping between import information and APIs, in each RP/MP code pattern, the test method should invoke at least one API of the packages summarized in code patterns (i.e., `∃R/MPAPISet[i] ∈ TMStmts`). (iii) The summarized APIs in RP/MPs should have one more parameter (i.e., the first parameter) than the private method-under-test (see SR4). The first parameter of these APIs should be the name of a private method-under-test (see SR3 and AD9), and (iv) the remaining parameters of APIs in code patterns should be consistent with the parameters of the method-under-test in amount and type (see SR5 and AD10). Making modifiers visible is another alternative to accessing and testing private methods. According to its syntactic rule in Table 7, the third PMT detection method is to identify the private methods annotated by the "`@VisibleForTesting`" annotation in the production code as smells.

5.3.2 *Detection Methods for POW Test Smells.* POW test smells are abstracted into two code patterns because of different usages in summarized assertions. Hence, we follow POW patterns to create syntactic rules for designing corresponding detection methods. The syntactic rules in both detection methods are: (i) In JUnit or TestNG, a test method should use one of the assertions summarized in POW code patterns (see SR 1 and 2 in POW). (ii) The parameter order of the assertions in the test method is different from the correct API usage in Table 6 (see SR 3). Since the special usage of assertions allows the assertions to add a third parameter for message logging, the detection method for CUPs has a few differences from SUPs. More specifically, when detecting POW smells in a special usage, the method "`ParamOrderDiff()`" needs to eliminate the effect of the position of the message parameter on the other two parameters and then execute the above two rules to identify POW smells.

Table 7. Detection Methods for Code Patterns of Different Test Smells

TS	Detection Method (DM)	Syntactic Rule (SR)	Abbreviation&Description (AD)
PMT	***DM for* RPs**	1.$\exists RPImports[i] \in TMImports \wedge$ 2. $\exists RPAPISet[i] \in TMStmts[j] \wedge$ 3.$RPAPISet[i].PrivMethod \in PrivMethodSet \wedge$ 4.$len(RPAPISet[i].Params) == len(PrivMethodSet[j].Params) + 1 \wedge$ 5.$\forall Type(PrivMethodSet[j].Param[k]) == Type(RPAPISet[i].Params[k+1])$	1.$R/MPImports[i]$: $R/MPImport\ i$ in Table 6. 2.$R/MPAPISet[i]$: $R/MPAPISet\ i$ in Table 6. 3.$R/MPAPISet[i].PrivMethod$: the invoked method by $R/MPAPISet\ i$, i.e., the second parameter in this API. 4.$R/MPAPISet[i].Params$: the parameter list of the invoked method of $R/MPAPISet[i]$, i.e., its second parameter value. 5.$PrivMethodSet$: a private method collection in production code. 6.$PrivMethodSet[j]$: the jth method in $PrivMethodSet$. 7.TM: a test method. 8.$VisAnno$: @VisibleForTesting; $VisAnno.ProducMethod$: a production method annotated by $VisAnno$. 9.$TMStmts[j].Params[0] == R/MPAPISet[i].PrivMethod$ 10.$TMStmts[j].Params[m] == R/MPAPISet[i].Params[m]$ $, m \in [0, len(R/MPAPISet[i].Params)]$
	***DM for* MPs**	1.$\exists MPImports[i] \in TMImports \wedge$ 2.$\exists MPAPISet[i] \in TMStmts[j] \wedge$ 3.$MPAPISet[i].PrivMethod \in PrivMethodSet \wedge$ 4.$len(MPAPISet[i].Params) == len(PrivMethodSet[j].Params) + 1 \wedge$ 5.$\forall Type(PrivMethodSet[j].Param[k]) == Type(MPAPISet[i].Params[k+1])$	
	***DM for* VAP**	1.$VisAnno.ProducMethod \in PrivMethodSet$	
POW	***DM for* CUPs**	1.$\exists CUPImports[i] \in TMImports \wedge$ 2.$\exists CUPAssertSets[i] \in TMStmts[j] \wedge$ 3.$ParamOrderDiff(TMStmts[j].Assertion, CUPAssertSets[i][n]), n \in [1,3]$	1.$C/SUPImports[i]$: $C/SUPImport\ i$ in Table 6. 2.$C/SUPAssertSets[i]$: $C/SUPAssertSet\ i$ in Table 6. 3.$TMImports[i]$: the ith import package in TM. 4.$TMStmts[j]$: the jth statement in TM. 5.$ParamOrderDiff()$: a method for determining whether the two methods/APIs differ in the order of the parameters. (return a boolean value)
	***DM for* SUPs**	1.$\exists SUPImports[i] \in TMImports \wedge$ 2.$\exists SUPAssertSets[i] \in TMStmts[j] \wedge$ 3.$ParamOrderDiff(TMStmts[j].Assertion, SUPAssertSets[i][n]), n \in [1,3]$	
NDT	***DM for* RDPs**	1.$RDPImports[i] \in TMImports \wedge$ 2.$RDPSyntaxSets[i] \in MTStmts$	1.$RDPImports[i]$: $RDPImport\ i$ in Table 6 2.$RDPSyntaxSets[i]$: $RDPSyntaxSet\ i$ in Table 6.
PTO	***DM for* TCPCP**	$\exists PrivMethod = producMethodSet[i], GetAnnota(PrivMethod).contain(@Test)$	1.$GetAnnota()$: a function for getting annotations of a method.
	***DM for* BTCOP**	$parallel\_ratio < 0.5$	
AL	***DM for* (E)FLP**	1.$\exists Type(TMStmts[i]) == ForStatement \wedge$ 2.$\exists ALSyntaxSet[k] \in TMStmts, k \in [1,2]$ 3.$\exists ForBlockStmts[j],\ TMStmts[i].ForBlockStmts[j] == AssertionStatement$	1.$ForBlockStmts$: the statements in a for-loop block 2.$ALSyntaxSet = \{FLPSyntax, EFLPSyntax\}$ 3.$ALSyntaxSet$: a set containing code patterns for AL test smells
RT	***DM for* RTP**	$Type(TMstmts[i]) = ReturnStatement$	

5.3.3 Detection Method for NDT Test Smells. Following the code patterns for generating random data, we design a method for detecting NDT smells. The syntactic rules of this detection method are present in Table 7: (i) The source file to which a test method belongs should import at least one of the three packages that are used for generating non-deterministic random data, and (ii) the statements of the test method should satisfy one of the common code patterns in Table 6.

5.3.4 Detection Methods for PTO Test Smells. Since PTO smells involve two types of confusing code organizations, we tailor a detection method for each PTO type. The syntactic rule of the detection method for the first PTO code pattern type (i.e., TCPCP) is concise and comprehensive, i.e., identifying the production code files that declare methods with the "`@Test`" annotation. The detection method for the BTCO test smell is designed based on the corresponding code pattern. Its code pattern calculates the ratio between well-organized test files and the total number of test files to describe the degree of the orderliness of test code. This ratio is referred to as "`parallel_ratio`". Hence, the syntactic rule of the detection method is to identify the project whose majority of test files are poorly organized, i.e., the value of "`parallel_ratio`" is lower than a classification threshold. Here the threshold is set to 0.5, which means if more than half of the test files in a project are not in the standard directory layout [8], this project would be treated as a PTO smell.

5.3.5 Detection Method for AL Test Smells. Based on the code patterns of common and enhanced for-loop, detailed syntactic rules for the AL test smell detection method are: (i) A test method should contain a for-loop structure; (ii) The test method should satisfy at least one of the code patterns of AL test smells; and (iii) An assertion should be one of the statements in the code block of the for-loop.

5.3.6 Detection Method for RT Test Smells. Compared with other smells, the detection method for RT smells has the simplest syntactic rule. We identify the test methods with return values as RT smells. Note that our detector only detects RT smells from test cases, not from test fixtures as test fixtures are allowed to have return values for use by other test cases [31, 32].

Table 8. Real-world Projects in our Dataset

Project Name	#Stars	#Test Methods	Test Smell Type (#Num)
fastjson	24.9k	10,895	PMT (36), POW (33), RDT (63)
orientdb	4.5k	9,979	PMT (1), RDT (246), POW (769)
hapi-fhir	1.5k	13,671	POW (5), RDT(4), LT (4)
functionaljava	1.5k	1,157	POW (4), PTO(1), RT (7)
ninja	1.9k	1,327	POW (18), RT (3)
JSqlParser	3.9k	1,664	RDT(2), LT (3)
zipkin	15.6k	1,429	PTO (1)

6 EVALUATION

In this section, we first evaluate the correctness of our detector and then validate the prevalence and usefulness of detected test smells in practice. Therefore, we aim to answer the following research questions (RQs):

RQ3: What is the performance of our detector?
RQ4: Do new test smells have practical values for real-world projects?

6.1 Correctness of Test Smell Detection

6.1.1 Motivation. Detection methods for different test smells involve complex program logic and syntactic rules. Hence, we would like to confirm the design and implementation of our detector are effective in real-world projects.

6.1.2 Method. To evaluate the correctness of our detector, we select seven popular real-world projects as our datasets based on the following principles: (i) Each project should win over 1,500 stars, which proves its popularity; (ii) these projects should involve all six test smell types and ensure that each smell type occurs in at least two different items; and (iii) this group of projects should contain as many smells as possible. More specifically, two researchers who have more than five years of Java development experience and major in test smells independently review each test method and determine whether a test method is a smell, and if yes, they further decide what smell type it belongs to based on the code patterns. They discuss to resolve their disagreement and reach a consensus. Their overall Kappa value is 0.83, indicating substantial agreement between them. After that, we manually label 1,200 test smells from these seven projects. Table 8 describes our dataset in detail.

6.1.3 Baseline. Random Guess is a straw-man baseline that randomly detect test smells from the dataset. For each test smell type, we run the baseline method five times and report the best result as the final detection performance.

6.1.4 Metrics. We adopt three common evaluation metrics (i.e., precision (P), recall (R), and F1-score (F1)) to evaluate the performance of our detector.

6.1.5 Results. Table 9 presents the performance of our detector and the baseline. *#TrueTS* is the number of test smells labeled by researchers and *#DetectedTS* is the number of test smells our detector detects. In total, we successfully identify 1,177 test smells from these seven projects. Our detector achieves high performance in precision (100%), recall (98%), and F1-score (99%), while the baseline method cannot correctly detect any types of smells from the dataset and gets poor performance among all metrics. Although our detector misdetects some test smells from the dataset, its performance substantially outperforms the baseline, which indicates that the code patterns we abstract can accurately depict the characteristics of different test smell types and corresponding detector methods are able to correctly identify different smells from projects based on these code patterns. Overall, our detector is effective to identify test smells in real-world projects.

Table 9. The Performance of our Detector

Approach	TS	#TrueTS	#DetectedTS	#TruePos	P	R	F1
Random Guess	PMT	37	6846	8	0.00	0.22	0.00
	POW	829	6715	169	0.03	0.20	0.05
	NDT	315	6704	67	0.01	0.21	0.02
	PTO	2	1	1	1	0.50	0.67
	AL	7	6746	2	0.00	0.29	0.00
	RT	10	6696	1	0.00	0.20	0.00
	Avg.	–	–	–	**0.17**	**0.27**	**0.12**
Our Detector	PMT	37	37	37	1	1	1
	POW	829	829	829	1	1	1
	NDT	315	292	292	1	0.93	0.96
	PTO	2	2	2	1	1	1
	AL	7	7	7	1	1	1
	RT	10	10	10	1	1	1
	Avg.	–	–	–	**1**	**0.98**	**0.99**

Detection failure analysis. Our evaluation results show that 23 test smells are misdetected by our detector. After a manual check, the reasons for misdetection are: (1) Developers sometimes create some statements to declare some instances as class members. These class members are placed outside any test methods in this class. It may cause some pattern-related statements/instances to be outside smelly methods, and thus statements only in these smelly methods cannot contain all syntaxes of a code pattern, being unable to be detected by the corresponding detection method. 17 out of 23 misdetected smells involve such a statement which is outside the scope of any methods as a member variable of the whole class for declaring a random generator, such as a random generator declared as a class variable in [33]. (2) Some developers prefer to create an inner class (i.e., a nested class) for performing specific/complex operators to variables or instances in test methods. Since our study considers a single test method as a basic test unit, code patterns and detection methods are both abstracted at the method level. Therefore, our detector cannot cover such smells once some bad testing practices occur in the code block of a nested class. For example, six undetected smells occur in the methods of a nested class [34]. Therefore, these 23 test smells are all under the above two cases. They are either out of our detection scope or unable to meet all syntaxes of a code pattern and syntactic rules in the corresponding detection method. In the future, we will improve the related code patterns and detection methods to cover such cases.

6.2 Prevalence and Usefulness of Test Smells

6.2.1 Motivation. Our work is the first to investigate and define new test smell types by a comprehensive exploratory study. We would like to know whether Java developers agree with us, how well they accept our detection results, and what opinions they have toward these smells.

6.2.2 Method. We use the dataset collected by Wen et al. [107], including the top 1,500 Java projects from GitHub, and then filter out failed-to-clone projects, toy projects, and tutorials. Finally, 919 Java projects are left, including 3,022,308 test methods (see our replication package). Then, we apply our detector to identify six types of test smells from these projects. We select 10 projects from the ones with every smell type (PTO test smells only occur in six projects) and submit an issue report to each project for reporting test smells. As shown in Figure 2, an issue report consists of three parts: test smell description, negative impacts of test smells, and potential solutions. In total, we submit 56 issue reports. We collect all replies to our issue reports, filter out irrelevant ones, and count the number of the remaining developers' replies. Following that, we classify the remaining replies based on their preferences and count the number of positive replies or negative ones, respectively. We do not find neutral ones among all replies. We believe such a result is reasonable. Because, after checking the replies for other issue reports, especially for the

Table 10. Results of Prevalence and Usefulness of Test Smells

	PMT	POW	NDT	PTO	AL	RT	Total
#Projects	87	509	561	6	123	11	722
#Replies	7	4	6	2	11	4	34
#Positive	6	4	3	1	8	4	26
#Negative	1	0	3	1	3	0	8

replies of the issue reports proposed by users, we notice that developers' replies often show two types of attitudes. One is to present their views to illustrate why they do not tend to address the proposed problem. Another attitude is to express that they agree with you and invite the proposer to open a pull request to contribute to their projects or directly fix the issue by themselves.

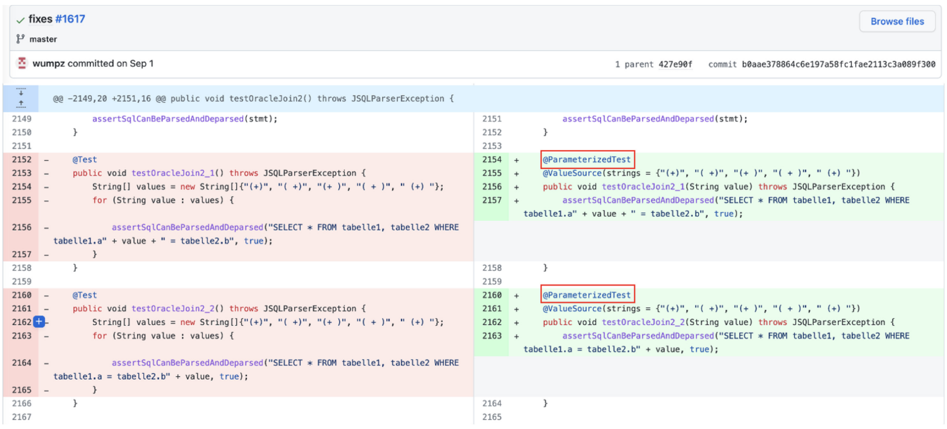
6.2.3 Results. Table 10 presents our experimental results.

Prevalence. The first row in Table 10 lists the number of projects in which each test smells type occurs. Among 919 projects, test smells occur in 720 ones, demonstrating that these new test smell types are prevalent in real-world Java projects. In addition, we find that NDT and POW are the two most common test smell types in real-world projects, appearing in 561 and 509 projects, respectively. It demonstrates that many developers have no awareness of the correct way to use random data and some assertions, still introducing and employing them following their poor programming habits. Therefore, it is necessary for developers, even those who contribute to high-quality projects, to know these new test smell types. This is also side-by-side evidence for stating the importance to explore and formally define potential test smells through academic studies.

Usefulness. As illustrated in Table 10, we receive 34 valid replies among 56 issue reports, including 26 positive and 8 negative replies. The 76.4% acceptance rate provides the initial evidence for test smells' practicality and usefulness. We analyze developers' replies for each smell type as follows:

For POW and RT test smells, all eight replies express the test scripts we submit in issue reports are indeed poor testing practices. <u>*RT test smells.*</u> For example, the reply for the RT smell [35] is agree with us. They tag our issue report as a "`good first issue`" and also mention that "*there is not yet a warning against this practice*", which also validates the importance of our study. <u>*POW test smells.*</u> A developer who works for a popular Android project "*android-beacon-library*" accepts our suggestion and strongly invites us to open a pull request for eliminating POW test smells. They say "*Contributions to this library are welcome! Please feel free to open a pull request to address this issue.*" [36]. In addition, the other two issue reports also encourage us to open a pull request for fixing POW test smells [37, 38].

Among the replies for PMT and AL test smells, the number of positives is far more than the negatives. <u>*PMT test smells.*</u> More specifically, most positive replies accept our opinion and clearly express that they will fix related smelly test scripts. For example, an Alibaba developer says "*You are right!*" [39] to voice their support for eliminating PMT smells in their project, which again implies that testing private methods is a bad practice in test code. Another developer of a spring project "*spring-integration*" directly refactors their test cases to fix PMT test smells (see commit: "*GH-3873: Remove JMS test for private method*" [2]). One negative reply mentions it is intentional to introduce a PMT smell, and their test method is a special case. Because this test script utilizes the reflection method to access private methods to match an exact signature in order for implementing Java serialization. However, we believe that the occurrence of a PMT smell in the test code precisely demonstrates a design problem in their related production code. <u>*AL test smells.*</u> We receive 11 replies for the issue reports on AL smells, including eight positives and three negatives. Among positive replies, we find that many developers agreed with us, but they only learn of the


fixes #1617

master

wumpz committed on Sep 1

1 parent 427e90f commit b0aae378864c6e197a58fc1fae2113c3a089f300

```
@@ -2149,20 +2151,16 @@ public void testOracleJoin2() throws JSQLParserException {
            assertSqlCanBeParsedAndDeparsed(stmt);
        }

-    @Test
-    public void testOracleJoin2_1() throws JSQLParserException {
-        String[] values = new String[]{"(+)", "( +)", "(+ )", "( + )", " (+) "};
-        for (String value : values) {
-            assertSqlCanBeParsedAndDeparsed("SELECT * FROM tabelle1, tabelle2 WHERE
tabelle1.a" + value + " = tabelle2.b", true);
-        }
    }

-    @Test
-    public void testOracleJoin2_2() throws JSQLParserException {
-        String[] values = new String[]{"(+)", "( +)", "(+ )", "( + )", " (+) "};
-        for (String value : values) {
-            assertSqlCanBeParsedAndDeparsed("SELECT * FROM tabelle1, tabelle2 WHERE
tabelle1.a = tabelle2.b" + value, true);
-        }
    }
```

```
            assertSqlCanBeParsedAndDeparsed(stmt);
        }

+    @ParameterizedTest
+    @ValueSource(strings = {"(+)", "( +)", "(+ )", "( + )", " (+) "})
+    public void testOracleJoin2_1(String value) throws JSQLParserException {
+        assertSqlCanBeParsedAndDeparsed("SELECT * FROM tabelle1, tabelle2 WHERE
tabelle1.a" + value + " = tabelle2.b", true);
    }

+    @ParameterizedTest
+    @ValueSource(strings = {"(+)", "( +)", "(+ )", "( + )", " (+) "})
+    public void testOracleJoin2_2(String value) throws JSQLParserException {
+        assertSqlCanBeParsedAndDeparsed("SELECT * FROM tabelle1, tabelle2 WHERE
tabelle1.a = tabelle2.b" + value, true);
    }
```

Fig. 6. A commit for fixing AL test smells.

specific solution for addressing AL smells after our suggestion [40]. For example, a developer who works for a popular project with 4.1k starts, replies "*You are right. Since we moved to JUnit 5 this annotated parameter possibility is a way to go*" [41], and then they eliminate AL smells by adopting our suggested solution as shown in Figure 6. We can notice that they use the repeated test feature in JUnit (i.e., @ParameterizeTest) to deconstruct for-loops in two test cases. Besides, some other replies also accept our suggestion (e.g., Replies: [42, 43]) and fix AL smells, such as Commit: "*Commit: Change issue 1945 test to paramaterized* [44]". Except for the positives, several negatives express their point of view. They feel their smelly test methods are so specific that the for-loops cannot be removed from them [45, 46]. One reply agrees to the potential problems caused by AL test smells but it thinks the solution we provide is hard to implement in their case, and they say: "*There are about 65,000 MIDs and for me, it doesn't make sense to put them in a "compiled list"*" [45]. We carefully analyze this reply to solve the mentioned problem. We think they can test their code by generating deterministic random numbers between 0 and 65,000 instead of using a loop structure and creating the assertion within the code block of the loop.

As shown in Table 10, both the PTO smell and NDT smell receive the same number of positive replies and negative ones. *PTO test smells.* We only receive two replies among all PTO issue reports, including one positive reply and one negative reply. The positive one accepts our view, i.e., organizing production code and test scripts in a parallel package structure is a good practice and is sure that their project can be improved in organization [47]. They also mention that the best practice may not work when a project has instead unit tests of a more broad nature that tests behavior/features. This problem is also mentioned in the negative reply [48]. In fact, the PTO test smell and the corresponding best practice are both targeted at the unit test. These two projects are detected as PTO smells by our detector, which illustrates that the two projects both mix unit tests with integration tests. Therefore, as the positive one says, their code organization can be indeed enhanced. *NDT test smells.* We find that the NDT smell receives the same number of positive and negative replies, i.e., three negative replies and three positive replies. Among positive replies, two replies say that they *agree with our opinion and will fix the smelly test code* [49, 50]. One reason for the negatives is that some developers believe that the test cases in question are reliable [51]. They mention using a random sequence in each test run increases the space of test data the code-under-test has to pass but using a fixed seed can't achieve it. After discussion, we regard there is a solution that can address the problem they propose. Specifically, developers can generate test data randomly by setting different random seeds. However, it is hard to reproduce failed test cases and trace the reason for the failures if rejecting to utilize deterministic test data. Overall, the prevailing opinion is that employing non-deterministic data as test data remains a poor testing practice. Another reason for negative replies is that developers are used to the inertia way to generate and employ non-deterministic random data and tend to ignore the potential threats NDT may cause. For example, many developers do not care about the potential threats we mention. They just express "*the test is fine for us as is*" or "*we do not care about the randomness of the test. Therefore, this test is doing its job properly*" [52, 53].

To sum up, our issue reports achieve high acceptance by developers. Most developers strongly accept our suggestions and refactor smelly test scripts based on our suggestions. However, some replies are used to their inertia programming practices and raise concerns about possible solutions of certain test smells in special cases, deserving further study.

7 THREATS TO VALIDITY

Internal Validity. During the process of mining SO posts, we design and strictly adhere to rigorous search steps to minimize the risk of overlooking relevant posts. Specifically, to minimize the possibility of missing any relevant posts, we have formulated a comprehensive search string that encompasses a broader scope. This approach aims to maximize the identification of new test smell types. Although our study focuses exclusively on the Java programming language, it is worth noting that four out of the six new test smell types we discover are applicable across multiple programming languages. Furthermore, we conduct a pilot experiment in which we observe that targeting the Java language enables us to gather the most pertinent SO posts compared to other programming languages. To ensure the accuracy of card classification during the open card sorting process for the collected posts, we engage two researchers with extensive expertise in the field of test smells to independently label and categorize the posts. For manual verification, the validity of the evaluation can be affected by human errors. To minimize potential human errors, any disagreements are resolved through discussions until a consensus is reached. Additionally, we employ Cohen's Kappa coefficient to demonstrate a high level of agreement between the researchers. In the determination of new test smell types, we establish strict rules that consider the majority's consensus for each potential test smell type, thus mitigating human errors. Our detector has tried its best to consider all types of syntax in Java and as many coding styles as possible for reducing the risk of ignoring some real smells with rare code styles. Detecting TC smells depends on specific thresholds that are picked from common thresholds with similar meanings in prior studies. Although we fully test our detector, the detector may have some unnoticed errors in its implementation.

External Validity. When splitting production code and test code, we not only adopt the common naming convention [105] but also refine it to improve classification accuracy. To validate the effectiveness of our test file identification method, we conduct a pilot experiment to compare our test file identification method with the method based on checking import information. We discovered that our test file identification method achieved a notable accuracy of 99.83% on our evaluation dataset. This performance surpasses that of the test file identification method based on checking import information, exhibiting a notable improvement of 10%. This arises because a considerable proportion of source files, despite not explicitly importing testing framework-related packages (e.g., Junit), can still be classified as test files. These files, called test fixtures [31], play a crucial role in constructing the necessary environment to consistently test specific items. Although, we encountered few instances where certain code files were erroneously identified as test files by our method, such as Stream_Test.java in the "functionaljava" project, PerformanceTest.java in the "JSqlParser" project, and Test.java in the "ninja" project. One test file was erroneously identified as a code file, i.e., LensPerson.java in the "functionaljava" project. However, these few instances hardly impact the performance of our test smell detection tool. The results of evaluating the performance of our detector and analyzing the prevalence of test smells rely on a specific range of open-source projects and may not generalize to all projects. However, we have widely adopted 919 real-world projects and many of them are high-star GitHub projects.

Construct Validity. When evaluating the effectiveness of our test smell detector, we painstakingly construct a carefully curated ground-truth test smell dataset derived from seven extensively acknowledged Java projects. Our diligent efforts are dedicated to achieving a comprehensive identification of as many test smells as possible within these projects, while simultaneously minimizing human errors. To accomplish this, we adopt a similar approach employed in the process of mining new test smell types by engaging two highly experienced researchers from the test smell field. They meticulously examine each function in the evaluation dataset encompassing the seven Java projects, independently labeling test smells based on the provided definitions of the six new test smell types. To ensure the correctness of the constructed dataset and further reduce human errors, in cases of disagreement, two researchers engage in discussions to reach a consensus on the labeling of each instance.

8 DISCUSSION

8.1 The Benefits and Opportunities for Industry and Academia

The exploration, operationalization, and detection of new test smell types in our study offer considerable value to both the industry and academic domains. In the industry domain, the introduction of these innovative test smells serves as a valuable enhancement to test case specifications and unit testing procedures. This, in turn, facilitates the development of a robust test strategy that ensures high-quality outcomes within the DevOps framework. In the academic domain, our study serves as an inspiration for researchers, motivating them to propose innovative methodologies for the automated refactoring of these newly identified test smells in software. It also encourages them to conduct empirical studies aimed at uncovering the underlying causes behind the creation of these test smells. The introduction of these research directions presents notable challenges, particularly due to the presence of design smells within some of the newly identified test smell types, such as PMT smells. Considering that different design smells frequently stem from inadequate design practices encompassing a substantial scope of source code, their identification by automated tools poses a great challenge, let alone the automated refactoring of these smells. In light of these factors, we believe that these research directions hold significant potential for further advancements and should be pursued.

8.2 Programming Language Selection

Our study primarily focuses on the Java language, aiming to explore new types of Java test smells and detect them in Java projects. This choice is driven by three main reasons: (1) The majority of relevant studies focus primarily on the Java language. (2) There is a significantly higher number of relevant SO posts available for discussing test smells or poor testing practices in the Java language compared to other **programming languages (PLs)**. We conduct a pilot experiment to validate this finding. We utilize the search strings "test smell [Java/Python/C++/C/Javascript/Scala]" and "bad practice in test code [Java/Python/C++/C/Javascript/Scala]" to obtain relevant posts for different PLs, respectively. The search results yield 454, 132, 112, 40, 132, and 30 for the first search string, while the second search string produces results of 500, 254, 272, 171, 327, and 22. It is evident that the search term "test smell [Java]" yields the most relevant SO posts. Therefore, we choose Java as the targeted PL for our study. (3) To ensure the utmost practicality of our test smell detector, we focus our study on Java, given its widespread adoption on GitHub, with a source code volume reaching 117 million by July 2023. This surpasses the code volume of other frequently utilized PLs in the industry, such as Python (50.1M), C++ (76.5M), C (35.1M), JavaScript (80.5M), and Scala (7.8M).

Indeed, out of the six newly identified test smell types, four of them, namely PMT, NDT, AL, and RT, are considered to be PL-agnostic. These test smells are applicable across different PLs. This is attributed to the fact that test cases involving non-deterministic data, loop structures, return statements, and the testing of private methods are generally regarded as poor choices across all PLs. On the other hand, the remaining two test smells, POW and PTO, are specifically associated with poor programming practices within certain testing frameworks' APIs and Java's project organization specifications.

9 CONCLUSION AND FUTURE WORK

In this paper, we aim to explore potential test smell types and develop a tool for detecting them from projects. More specifically, six new test smell types are determined through a comprehensive exploratory study. Then, we analyze their negative impacts on software quality and summarize possible solutions to address them. To perform test smell detection, we abstract code patterns for each test smell type and design corresponding detection methods based on these patterns. Experiments show that our detector can automatically detect test smells from real-world Java projects with 100% precision and 98% recall. To assess the prevalence of test smells, we perform test smell detection on 919 real-world projects. The experimental result shows that the six test smells occur in 722 out of 919 projects, and *the POW and NDT test smell* are the two most common smell types in software. In addition, to verify the usefulness of test smells, we submit 56 issue reports to report test smells detected by our detector. The issue reports have been well received and praised by Java developers, with 76.4% acceptance. In the future, we will integrate our detector as a plugin for IDE and build a large-scale dataset based on the test smells detected by our detector.

REFERENCES

[1] https://github.com/kestra-io/kestra/issues/1257
[2] https://github.com/spring-projects/spring-integration/commit/78fa2970fa9f7b64c6bf65db886f78b5dc1e4596
[3] https://github.com/Yanming-Yang/TestSmellPackage
[4] https://www.techtarget.com/searchsoftwarequality/definition/test-case
[5] https://stackoverflow.com/questions/7075938/making-a-private-method-public-to-unit-test-it-good-idea
[6] https://stackoverflow.com/questions/34571/how-do-i-test-a-class-that-has-private-methods-fields-or-inner-classes/34586#34586
[7] https://www.codemotion.com/magazine/devops/qa-testing/how-to-challenge-your-code-with-property-based-testing-part-3
[8] http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html
[9] https://stackoverflow.blog/2022/11/03/multiple-assertions-per-test-are-fine
[10] https://stackoverflow.com/questions/1789834/is-it-okay-to-run-for-loops-in-functional-test-methods
[11] https://www.techtarget.com/searchsoftwarequality/definition/test-case
[12] https://stackoverflow.com/questions/28083624/how-to-test-private-classes-in-junit-whitout-changing-accessible-to-true
[13] https://stackoverflow.com/questions/28818735/is-it-bad-practice-to-make-a-method-package-or-private-for-the-sake-of-testing
[14] https://stackoverflow.com/questions/34571/how-do-i-test-a-class-that-has-private-methods-fields-or-inner-classes
[15] https://stackoverflow.com/questions/46207409/unit-test-private-and-static-methods-in-java
[16] https://stackoverflow.com/questions/2404978/why-are-assertequals-parameters-in-the-order-expected-actual
[17] https://stackoverflow.com/questions/16267660/assert-assertequals-junit-parameters-order
[18] https://github.com/JSQLParser/JSqlParser/commit/b0aae378864c6e197a58fc1fae2113c3a089f300
[19] https://stackoverflow.com/questions/42633279/are-test-data-factory-methods-dangerous-or-beneficial

[20] https://stackoverflow.com/questions/18747811/why-not-use-a-pseudo-random-number-generator-to-produce-test-data
[21] https://stackoverflow.com/questions/16180485/gae-separating-tests-from-app-code
[22] https://stackoverflow.com/questions/42273490/is-it-a-bad-practice-to-put-junit-test-method-into-the-tested-class
[23] https://stackoverflow.com/questions/59292626/how-to-execute-for-loop-for-all-test-cases-using-the-base-class-beforemethod-in
[24] https://stackoverflow.com/questions/60654124/junit-test-methods-cant-return-a-value
[25] https://stackoverflow.com/questions/2811141/is-it-bad-practice-to-use-reflection-in-unit-testing
[26] https://stackoverflow.com/questions/19699122/mock-private-method-in-the-same-class-that-is-being-tested
[27] https://act-rules.github.io/pages/design/test-cases/
[28] https://stackoverflow.com/questions/44606798/how-to-iterate-the-different-user-ids-given-under-before-for-same-test-case-un
[29] https://stackoverflow.com/questions/53597020/best-practice-for-looped-junit-test
[30] https://github.com/orientechnologies/orientdb/blob/6e94047200f6b7a15c02902eea21a26a85351fe6/distributed/src/test/\java/com/orientechnologies/orient/server/distributed/ServerClusterRemoteDocumentIT.java
[31] https://en.wikipedia.org/wiki/Test_fixture
[32] https://www.boost.org/doc/libs/1_60_0/libs/test/doc/html/boost_test/tests_organization/fixtures/case.html
[33] https://github.com/alibaba/fastjson/blob/bcd0505019425b2b8a7146828bec51739b933318/src/test/java/com/\alibaba/json/test/FNVHashTest.java{#}L15
[34] https://github.com/orientechnologies/orientdb/blob/6e94047200f6b7a15c02902eea21a26a85351fe6/core/src/test/java/com/orientechnologies/common/collection/closabledictionary/OClosableLinkedContainerTest.java{#}L186
[35] https://github.com/cbeust/testng/issues/2797
[36] https://github.com/AltBeacon/android-beacon-library/issues/1101
[37] https://github.com/pippo-java/pippo/issues/607
[38] https://github.com/docker-java/docker-java/issues/1923
[39] https://github.com/alibaba/fastjson/issues/4261{#}issuecomment-1295173220
[40] https://github.com/JSQLParser/JSqlParser/commit/b0aae378864c6e197a58fc1fae2113c3a089f300
[41] https://github.com/JSQLParser/JSqlParser/issues/1617
[42] https://github.com/hapifhir/hapi-fhir/issues/3928{#}issuecomment-1217922279
[43] https://github.com/biojava/biojava/issues/1038{#}issuecomment-1228993250
[44] https://github.com/javaparser/javaparser/pull/3739
[45] https://github.com/eclipse/californium/issues/2056
[46] https://github.com/AntennaPod/AntennaPod/issues/6021{#}event-7204518302
[47] https://github.com/apache/cordova-android/issues/1476
[48] https://github.com/eclipse-openj9/openj9/issues/15708
[49] https://github.com/apache/datasketches-java/issues/413{#}issuecomment-1218401059
[50] https://github.com/guoguibing/librec/issues/350{#}issuecomment-1235063757
[51] https://github.com/stefan-zobel/streamsupport/issues/20
[52] https://github.com/ReactiveX/RxJava/issues/7467
[53] https://github.com/flowable/flowable-engine/issues/3433{#}issuecomment-1216538228
[54] [n. d.]. How To Write Effective Test Cases? https://www.softwaretestingclass.com/how-to-write-good-test-cases/
[55] Wajdi Aljedaani, Mohamed Wiem Mkaouer, Anthony Peruma, and Stephanie Ludi. 2023. Do the test smells assertion roulette and eager test impact students' troubleshooting and debugging capabilities? *arXiv preprint arXiv:2303.04234* (2023).
[56] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D. Newman, Abdullatif Ghallab, and Stephanie Ludi. 2021. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering* (2021), 170–180.
[57] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 56–65.
[58] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? An empirical study. *Empirical Software Engineering* 20, 4 (2015), 1052–1094.
[59] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 770–781.
[60] Matteo Biagiola, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. 2019. Web test dependency detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 154–164.

[61] Alexandru Bodea. 2022. Pytest-smell: A smell detection tool for Python unit tests. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 793–796.

[62] Manuel Breugelmans and Bart Van Rompaey. 2008. TestQ: Exploring structural and maintenance characteristics of unit test suites. In *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. Citeseer.

[63] Denivan Campos, Larissa Rocha, and Ivan Machado. 2021. Developers perception on the severity of test smells: An empirical study. *arXiv preprint arXiv:2107.13902* (2021).

[64] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2019. SoCRATES: Scala radar for test smells. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*. 22–26.

[65] Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew P. Black, and Anne Etien. 2019. Rotten green tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 500–511.

[66] Daniel Fernandes, Ivan Machado, and Rita Maciel. 2022. TEMPY: Test smell detector for Python. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*. 214–219.

[67] Tommaso Fulcini, Giacomo Garaccione, Riccardo Coppola, Luca Ardito, and Marco Torchiano. 2022. Guidelines for GUI testing maintenance: A linter for test smell detection. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*. 17–24.

[68] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–11.

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

[70] Michaela Greiler, Arie Van Deursen, and Margaret-Anne Storey. 2013. Automated detection of test fixture strategies and smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 322–331.

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

[72] Mouna Hadj-Kacem and Nadia Bouassida. 2022. A multi-label classification approach for detecting test smells over Java projects. *Journal of King Saud University-Computer and Information Sciences* 34, 10 (2022), 8692–8701.

[73] Paul Hamill. 2004. *Unit Test Frameworks: Tools for High-quality Software Development*. O'Reilly Media, Inc.

[74] Vladimir Khorikov. 2020. *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster.

[75] Dong Jae Kim, Tse-Hsun Chen, and Jinqiu Yang. 2021. The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empirical Software Engineering* 26 (2021), 1–47.

[76] Negar Koochakzadeh and Vahid Garousi. 2010. TeCReVis: A tool for test coverage and test redundancy visualization. In *Testing–Practice and Research Techniques: 5th International Academic and Industrial Conference, TAIC PART 2010, Windsor, UK, September 3-5, 2010. Proceedings*. Springer, 129–136.

[77] Negar Koochakzadeh and Vahid Garousi. 2010. A tester-assisted methodology for test redundancy detection. *Advances in Software Engineering* 2010 (2010).

[78] Stefano Lambiase, Andrea Cupito, Fabiano Pecorelli, Andrea De Lucia, and Fabio Palomba. 2020. Just-in-time test smell detection and refactoring: The DARTS project. In *Proceedings of the 28th International Conference on Program Comprehension*. 441–445.

[79] Wei Li and Raed Shatnawi. 2007. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software* 80, 7 (2007), 1120–1128.

[80] Florian Maier and Michael Felderer. 2023. Detection of test smells with basic language analysis methods and its evaluation. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 897–904.

[81] Mika V. Mäntylä and Casper Lassenius. 2006. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering* 11, 3 (2006), 395–431.

[82] Matias Martinez, Anne Etien, Stéphane Ducasse, and Christopher Fuhrman. 2020. RTj: A Java framework for detecting and refactoring rotten green test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 69–72.

[83] Luana Martins, Denivan Campos, Railana Santana, Joselito Mota Junior, Heitor Costa, and Ivan Machado. 2023. Hearing the voice of experts: Unveiling Stack Exchange communities' knowledge of test smells. *arXiv preprint arXiv:2305.03431* (2023).

[84] Gerard Meszaros. 2007. *xUnit Test Patterns: Refactoring Test Code*. Pearson Education.

[85] Nicholas Alexandre Nagy and Rabe Abdalkareem. 2022. On the co-occurrence of refactoring of test and source code. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 122–126.

[86] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. 2009. The evolution and impact of code smells: A case study of two open source systems. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 390–400.

[87] 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 *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 5–14.

[88] 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)*. IEEE, 311–322.

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

[90] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–74.

[91] 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 *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1650–1654.

[92] Anthony Peruma, Khalid Saeed Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the distribution of test smells in open source Android applications: An exploratory study. (2019).

[93] Valeria Pontillo, Dario Amoroso d'Aragona, Fabiano Pecorelli, Dario Di Nucci, Filomena Ferrucci, and Fabio Palomba. 2022. Machine learning-based test smell detection. *arXiv preprint arXiv:2208.07574* (2022).

[94] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. 2007. Rule-based assessment of test quality. *J. Object Technol.* 6, 9 (2007), 231–251.

[95] Renaud Rwemalika, Sarra Habchi, Mike Papadakis, Yves Le Traon, and Marie-Claude Brasseur. 2021. Smells in system user interactive tests. *arXiv preprint arXiv:2111.02317* (2021).

[96] Elvys Soares, Marcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André Santos. 2022. Refactoring test smells with JUnit 5: Why should developers keep up-to-date? *IEEE Transactions on Software Engineering* 49, 3 (2022), 1152–1170.

[97] 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)*. IEEE, 1–12.

[98] Rosemarie Streeton, Mary Cooke, and Jackie Campbell. 2004. Researching the researchers: Using a snowballing technique. *Nurse Researcher* 12, 1 (2004), 35–47.

[99] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*. Citeseer, 92–95.

[100] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. 2006. Characterizing the relative significance of a test smell. In *2006 22nd IEEE International Conference on Software Maintenance*. IEEE, 391–400.

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

[102] Tássio Virgínio, Luana Martins, Larissa Rocha, Railana Santana, Adriana Cruz, Heitor Costa, and Ivan Machado. 2020. JNose: Java test smell detector. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*. 564–569.

[103] Tássio Virgínio, Luana Martins, Railana Santana, Adriana Cruz, Larissa Rocha, Heitor Costa, and Ivan Machado. 2021. On the test smells detection: An empirical study on the JNose test accuracy. *Journal of Software Engineering Research and Development* 9 (2021), 8–1.

[104] Tássio Virgínio, Railana Santana, Luana Almeida Martins, Larissa Rocha Soares, Heitor Costa, and Ivan Machado. 2019. On the influence of test smells on test coverage. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. 467–471.

[105] Sinan Wang, Ming Wen, Yepang Liu, Ying Wang, and Rongxin Wu. 2021. Understanding and facilitating the co-evolution of production and test code. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 272–283.

[106] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. 2021. PyNose: A test smell detector for Python. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 593–605.

[107] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 53–64.

[108] Yanming Yang, Xin Xia, David Lo, Tingting Bi, John Grundy, and Xiaohu Yang. 2022. Predictive models in software engineering: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–72.

[109] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2020. A survey on deep learning for software engineering. *ACM Computing Surveys (CSUR)* (2020).

[110] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 385–396.

Received 28 December 2022; revised 18 July 2023; accepted 9 October 2023