Hearing the voice of experts: Unveiling Stack Exchange communities' knowledge of test smells

Luana Martins
Institute of Computing
Federal University of Bahia (UFBA)
Salvador, Brazil
martins.luana@ufba.br

Denivan Campos
Institute of Computing
Federal University of Bahia (UFBA)
Salvador, Brazil
denivan.campos@ufba.br

Railana Santana
Institute of Computing
Federal University of Bahia (UFBA)
Salvador, Brazil
railana.santana@ufba.br

Joselito Mota Junior
Institute of Computing
Federal University of Bahia (UFBA)
Salvador, Brazil
joselito.mota@ufba.br

Heitor Costa
Department of Computer Science
Federal University of Lavras (UFLA)
Lavras, Brazil
heitor@ufla.br

Ivan Machado
Institute of Computing
Federal University of Bahia (UFBA)
Salvador, Brazil
ivan.machado@ufba.br

Abstract—Refactorings are transformations to improve the code design without changing overall functionality and observable behavior. During the refactoring process of smelly test code, practitioners may struggle to identify refactoring candidates and define and apply corrective strategies. This paper reports on an empirical study aimed at understanding how test smells and test refactorings are discussed on the Stack Exchange network. Developers commonly count on Stack Exchange to pick the brains of the wise, i.e., to 'look up' how others are completing similar tasks. Therefore, in light of data from the Stack Exchange discussion topics, we could examine how developers understand and perceive test smells, the corrective actions they take to handle them, and the challenges they face when refactoring test code aiming to fix test smells. We observed that developers are interested in others' perceptions and hands-on experience handling test code issues. Besides, there is a clear indication that developers often ask whether test smells or anti-patterns are either good or bad testing practices than code-based refactoring recommendations.

Index Terms—Developer expertise, stack exchange mining, refactoring, test smells.

I. INTRODUCTION

Software refactoring consists of changing the structure of the source code without compromising its overall functionality [1] and observable behavior [2]. We use software refactoring to enforce better design and coding practices through small code transformations [3]. In addition, software refactoring aims to facilitate readability and maintainability [4], especially in large code bases where multiple developers engage without a detailed view of the whole system [5].

Typically, the refactoring process takes place in the key phases [6], [7]: *(i)* identification of candidates for refactoring; *(ii)* determination of the appropriate refactoring technique; *(iii)* application of refactoring; and *(iv)* validation of the refactoring effect. Identifying a refactoring candidate requires an in-depth understanding of various parts of the system and knowledge of best practices [8]. Detecting problems in the code can be time-consuming and labor-intensive and often requires automated

tool support to be effective in practice [9]. Besides, deciding which refactoring technique best applies in each situation can be challenging and complex [10]. Such reasons lead software developers to understand how the software community has dealt with anti-patterns and their refactoring processes [11].

The Software Engineering research community has dedicated efforts to understanding anti-patterns and proposing solutions to assist developers in refactoring the code to cope with design issues [12]–[15]. However, the evolution of programming languages and frameworks creates new anti-patterns that require novel refactorings to fix them, making it challenging for the community to keep up with the actual problems developers face in practice [16]. For this reason, developers often post questions on Q&A platforms seeking help to solve a problem with their code. In a recent study, Peruma et al. [11] conducted quantitative and qualitative experiments to understand how developers discuss refactoring in a collaborative online discussion forum. The authors aimed to unveil the most explored and discussed topics concerning software refactoring.

From a broader perspective, refactoring is just as important for automated test code as production code as it supports the identification of issues caused by production code changes [17], [18]. Testing verifies whether the software functionality and observable behavior are kept the same after code refactorings [19], [20]. The test result should remain unchanged before and after the refactorings in the production code. However, the test code development is prone to human errors, harming the test code's ability to detect defects. Therefore, to effectively diagnose problems in production code, Beck et al. [21] stated the coding of tests should follow good design principles.

Inspired by those arguments, van Deursen et al. [22] defined test smells to indicate badly designed tests [22]. The presence of test smells can be interpreted as a symptom of poor software quality, harming the testing and maintenance activities [13], [23]–[25]. Garousi et al. [15] proposed a catalog of test smells

and a summary of existing techniques and tools resulting from a multivocal literature review. That catalog and summary are a good initial step towards advancing the field, but there is still a lack of understanding of which refactoring to apply and how to apply them, in practice, to fix test smells.

By analyzing the literature on test smells [13], [15], [26], we may observe a few pieces of evidence of the challenges developers face during the automated test code development and maintenance and the commonly discussed issues, mainly unit testing. Furthermore, little is known about commonly used corrective strategies and strategies that ensure the preservation of test behavior after test code refactoring. These are important gaps to bridge.

This paper reports on the results of an empirical study aimed at understanding how the developers discuss test smells and test refactorings in the Stack Exchange, the leading software development collaboration network. By analyzing data from Stack Exchange, we could contribute with a synthesis of the community discussions about test smells, mainly regarding the key challenges, test design issues, test code refactoring, and post-refactoring test behavior.

II. RESEARCH METHODOLOGY

This study addressed the following Research Questions (RQ):

RQ$_1$ **What challenges do developers report for handling problems in the test code?** This RQ studies developers' main difficulties in refactoring test code by grouping the questions developers ask into *why-how-what* categories.

RQ$_2$ **What test smells do developers most actively discuss?** This RQ studies which test smells developers consider relevant to refactor and classifies them following Garousi et al.'s catalog [15].

RQ$_3$ **What preventive and corrective actions do developers suggest to handle test smells in the test code?** This RQ leverages the actions developers suggest to prevent test smell insertion and the refactoring operations from fixing test smells.

RQ$_4$ **Do developers discuss how to keep the test behavior after test code refactorings?** This RQ investigates whether and how developers care about test behavior during refactoring.

Fig. 1 shows the study design, which encompasses three main steps: (A) Identification of discussions, (B) Classification of discussions, and (C) Data analysis.

A. Identification of discussions

Developers' competence consists of a frequent lifelong quest for knowledge, and educational networks are excellent environments for spreading knowledge [27]. According to Tahir et al. [28], developers often gather in Q&A online forums to 'look up' how others complete similar tasks and cope with recurring issues, including how to get rid of anti-patterns. For example, *Stack Exchange* is a collaborative discussion forum network offering insightful resources on development issues [28]. Posnett et al. [27] mentioned participating in and sustaining learning communities is a durable and valuable aspect of professional life.

In this study, we used Internet Archive (IA)[1] to retrieve discussions on test code problems developers report and the corrective actions they suggest. IA keeps data dumps of discussions on the Stack Exchange network over time. We selected the following Stack Exchange sites:

- **Code Review**[2]**:** a site for peer programmer code reviews. It allows programmers to ask questions about specific code snippets and receive feedback from others;
- **Software Engineering**[3]**:** a site for developers and scholars interested in asking general questions on the systems development life cycle [28]. The site is adequate for opinion-based questions;
- **Stack Overflow**[4]**:** a site covering technical and general discussions on problems unique to software development [29], [30]. The discussions commonly focus on specific programming problems or tools.

The discussions on the Stack Exchange sites associate one question post with one or more answer posts given by different users. A question post consists of a title, body, and tags. Fig. 2 shows a sample StackOverflow question post [31]. It comes with three tags: `Java`, `JUnit`, and `refactoring`. In the post, the author asks if the practice she/he adopted in the `JUnit` test case consists of duplication and how she/he writes the test without duplication. Fig. 3 shows a response to the question post from Fig. 2. It reveals that the practice of the question refers to a test anti-pattern called `Ugly Mirror`. As a potential solution, the test code simplification (avoiding conditional structures and using assertions instead) and the test object creation to run the tests manually for complex data structures. The post's author rated the answer as accepted, and the answer got four votes.

We defined and applied a search string on the tags in each question post to select the discussions. It aimed at filtering out the discussion content based on selected tags (Fig. 1 - *Posts selection*). Table I shows four groups of tags composing our search string. The *TEST* group filters the discussions about test codes by specifying the types of tests, testing frameworks, and language constructs used in test codes. The *DESIGN* group filters the discussions containing smell-related problems concerning test code [15]. The *REFACTORING* group filters topics with discussions on how to refactor test code. Considering that the tagged topics rarely use the words of the *REFACTORING* group, we used a logical disjunction with the *REFACTORING* and *DESIGN* groups to create a less strict filter. Similarly, many topics are untagged with programming languages. Therefore, the group *LANGUAGE* removes those topics tagged with programming languages other than `Java`.

---

[1]Available at https://archive.org/download/stackexchange

[2]Available at https://codereview.stackexchange.com/

[3]Available at https://softwareengineering.stackexchange.com/

[4]Available at https://stackoverflow.com/

Fig. 1: Study Design

TABLE I: Groups of tags composing the search string.

ID	Group	Description	Tags	Criteria
1	TEST	Words related to testing, test type, or test structure	'unit-testing,' 'testing,' 'test-scenarios,' 'unit-test-data,' 'junit,' 'junit4,' 'junit5,' 'automated-tests,' 'test-automation,' 'tests,' 'testcase,' 'assertions,' 'assert,' 'assertion,' 'annotations'	Inclusion
2	DESIGN	Words related to programming practices and design	'code-smell,' 'code-smells,' 'anti-patterns,' 'programming-practices,' 'naming,' 'naming-conventions,' 'naming-standards,' 'coding-standards,' 'coding-style,' 'code-formatting,' 'format,' 'formatting,' 'bad-code,' 'technical-debt'	Inclusion
3	REFACTORING	Words related to refactoring	'refactoring,' 'test-refactoring,' 'automated-refactoring'	Inclusion
4	LANGUAGE	Words related to programming languages	'c++,' 'c#,' 'javascript,' 'vb6', 'python,' 'python-3', 'go,' 'c,' '.net,' 'php,' 'sql,' 'ruby'	Exclusion

Java: code duplication in classes and their Junit test cases

Asked 10 years, 6 months ago   Modified 10 years, 6 months ago   Viewed 942 times

▲
1   I've been writing code that processes certain fields of an object by modifying their values. To test it, I first wrote a JUnit test case that recursively traverses fields of an object and makes sure they're correctly modified. The CUT (Class Under Test) does something vary similar: it recursively traverses fields of an object and modifies them as required.
▼

So the code to recursively traverse the fields remains the same in test case and CUT, and is currently duplicated, which is against DRY. So I have two questions:

1) have you come across such situations in your project? If yes, did you apply DRY, or let such duplication remain as is?

2) if I put this common code in a util method, I will need to write a test case to test that, which would again involve traversing fields recursively. So how can this be solved without adding any duplication?

`java` `junit` `refactoring`

Fig. 2: Sample question post extracted from StackOverflow.

▲
4   You have just hit the ugly mirror testing anti-pattern. If your CUT has a bug, most likely you will copy it to your test case, essentially verifying that a bug is still there.
▼
You must show us some more code, but basically your test case should be much simpler, no `for` loops, no conditions - just assertions. If your production code does some fancy traversing, reflection, etc. on complicated data structures - create a test Java object and test every field manually in the unit test.
✓

Fig. 3: Accepted/useful answer from a StackOverflow post.

Although some studies automatically classify thousands of Stack Exchange topics [11], [28], [32], we fine-tuned our search string to reduce the number of non-relevant topics on test smells and test refactorings [33], [34]. We applied the search string "**TEST AND (DESIGN OR REFACTORING) NAND LANGUAGE**" on the data dumps of three StackExange sites from September 15th, 2008 to December 6th, 2022. As a result, we retrieved 303 potential posts.

TABLE II: Number or retrieved discussions.

Site	# Total posts	# Potential posts	# Selected posts
SoftwareEngineering	237,548	153	40
CodeReview	196,301	2	2
StackOverflow	+8million	158	59

B. Classification of discussions

We manually analyzed the discussions to select the ones related to test smells and the refactorings to solve them (Fig. 1 - *Manual analysis*). To align the analysis criteria, three coders performed a peer analysis on the CodeReview and SoftwareEngineering question topics. The coders read 155 question posts and applied the following inclusion (*IC*) and exclusion criteria (*EC*): (*IC_1*) question topics describing a problem in the test code related to bad design or implementation choices, (*EC_1*) question topics about the need for testing and how to create test code, and (*EC_2*) question topics without answers. We calculated the Kappa statistics [35] to assess the reliability of the manual classification (Fig. 1 - *Quality assessment*) and reached a substantial agreement level of 0.61.

After, independent coders analyzed the StackOverflow question posts and accepted 59 pots (Fig. 1 - *Selected posts*, Table II - *#Selected posts*). Next, we analyzed 101 selected posts to extract the data items listed in Table III. The data extraction followed the same steps described in this section. We performed the data extraction on the CodeReview and SoftwareEngineering in peers. The coders discussed divergences in data extraction in daily meetings and performed individual data extraction on StackOverflow.

TABLE III: Data items extracted from discussions.

#	Data Item	Description	RQ
D1	Challenges for refactoring test smells	Challenges that developers face for refactoring the test code to fix test smells	RQ_1
D2	Description of test smells	Descriptions of test smells by developers based on their understanding	RQ_2
D3	Cause of test smells	Causes that lead to test smells	RQ_2
D4	Actions for preventing test smells and refactoring the test code	Actions suggested by developers to prevent the insertion of test smells and refactor the test code to fix test smells	RQ_3
D5	Tools for refactoring test smells	Tools and sources that support developers fixing test smells	RQ_3
D6	Strategies to keep the test code behavior	Strategies to verify whether the test code behavior is kept the same after the test code refactorings	RQ_4

(a) Time between asking a question and receiving a first answer.

(b) Time between asking a question and closing the discussion.

Fig. 4: Timeline of answer time and closing time of discussion topics in days

C. Data Analysis

To answer RQ_1, we analyzed the question topics in two steps. First, we applied a Thematic Content Analysis (TCA) [36] to classify the types of questions asked by developers into *why-how-what* questions (Golden-circle theory) [29]: (1) *why* is a type of question that seeks to understand the reason or the cause of a problem, (2) *how* is the type of question that seeks approaches or better ways to achieve a result, and (3) *what* is the type of question to get the information related to the problem. Then, we performed an open coding [37] to interpret the question topics and extract the main challenges developers face when applying refactorings to fix problems in the test code.

To answer RQ_2, we applied a TCA to classify the discussions into (1) *specific discussion* about a specific test smell and (2) *general discussion* that does not explicitly ask about a test smell. After, we interpreted the description of the test smell and classified it according to Garousi et al.'s catalog [15].

To answer RQ_3, we applied a TCA in the top answers (top-rated answer, accepted answer, or unique answer) to classify their actions into [28]: (1) *Fix* to recommend test code refactorings for fixing problems in the test code, (2) *Capture* to explain the test problems but does not recommend test code refactoring to solve the problems, (3) *Ignore* to recommend ignoring taking any action to fix the test code, and (4) *Explain* why something is considered a test problem. In addition, we extracted the tools and applied open coding to list the test code refactorings suggested in the answers.

To answer RQ_4, we analyzed whether the developers asked for strategies to verify the test code behavior after performing test code refactorings. We listed the strategies and tools suggested in the answers. Data is publicly available in an online open data repository [38].

III. RESULTS

A. Discussions characterization

We characterized 101 discussion topics on test smells and their solutions found on the Stack Exchange network. There is nearly no repetition of users asking questions. For those users who asked more than one question, the author rephrased the question in a more specific way or on a different site.
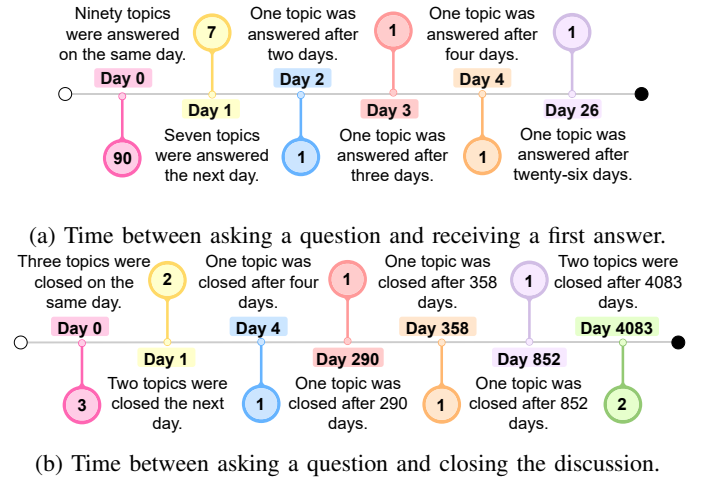
We collected 111 top answers, of which we accepted 61 answers, and 38 answers were top-rated. We found another 12 answers, and although not accepted, we deemed relevant for our investigation. Most users answered only one question, and only seven answered more than one. It is worth noting four out of these seven users are top-ten users when considering users' reputation score, number of questions, and number of answers. Next, we collected the time between asking a question and receiving the first answer (Fig. 4a) and the time to close the discussion topic (Fig. 4b). We observed that 90 out of 101 questions received an answer on the same day, and seven received an answer the next day. The remainder received an answer from the second to the twenty-sixth day. Regarding the time to close, only three topics were closed the same day they were posted, and the other 11 were closed within a year. Curiously, some took 4,083 days to close, and many never closed.

B. Trends and Challenges (RQ1)

In RQ_1, we aimed to understand the trends and challenges around developers' discussions on test code refactoring concepts and activities. We classified the discussions into 30 categories representing the questions asked by developers while evolving the test code. Fig. 5 presents the number of discussions in each category and their relationship with the *why-how-what* questions.

In the *why* questions, we classified the discussions into ten categories to understand the test code problems and bring insights into the decision-making of whether to refactor the test code for fixing such problems. The *Understanding which code structures to test*, *Understanding coupling in unit tests and production code*, and *Understanding how to evolve legacy code* categories refer to problems with origin in the production code. The *Convincing people to follow good practices* category relates to problems faced by newcomers joining a team that does not follow good practices for testing the

WHY

Understanding test smells (12)
Understanding which code structures to test (12)
Understanding coupling in unit tests and production code (9)
Understanding the best way to implement the test code (8)
Understanding test code duplication (4)
Convincing people to follow good practices (3)
Understanding naming conventions (2)
Understanding how to evolve legacy code (2)
Understanding mocking anti-patterns (1)
Understanding the test results (1)

HOW

How to handle dependencies (with mock) (12)
How to assure that the refactoring does not break the tests (8)
How to refactor test code to remove duplication (8)
How to organize the test code (6)
How to name test methods and classes following naming conventions (4)
How to refactor test code to improve coverage (2)
How to deal with async (2)
How to refactor test smells (2)
How to refactor to evolve legacy code (2)
How to refactor test smells (pointed by tools) (2)
How to organize the documentation of test code (1)
How to refactor mocks (1)

WHAT

What are the best practices to name test classes, methods, mocks, or paths (8)
What are the best practices for unit testing (5)
What are the best practices for mocking (3)
What are the strategies and tools to find unused/dead code (2)
What are the tools to rename and organize tests (2)
What are the mocking anti-patterns (1)
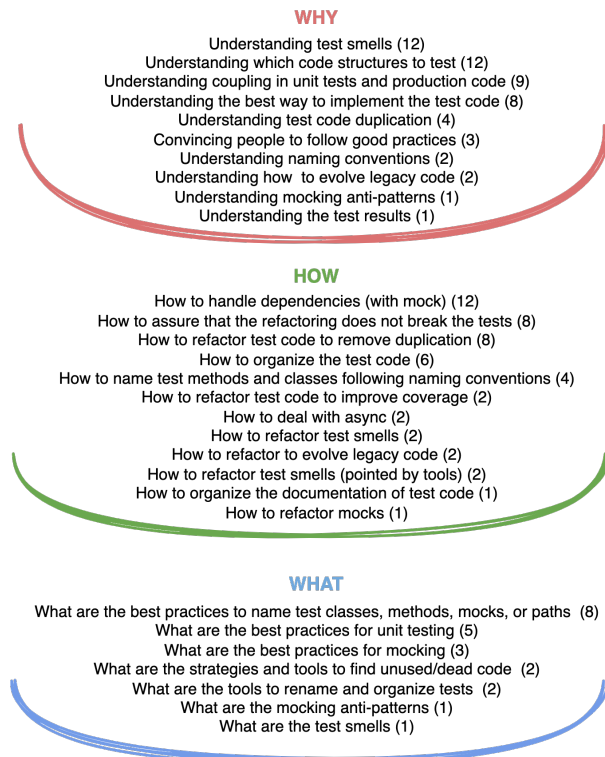What are the test smells (1)

Fig. 5: Classification of types of questions and challenges

code due to organizational decisions or team conventions. The *Understanding test smells*, *Understanding the best way to implement the test code*, *Understanding test code duplication*, *Understanding naming conventions*, *Understanding mocking anti-patterns*, and *Understanding the test results* categories refer to problems related to the comprehension of the testing frameworks constructs, the importance of following good practices and conventions for developing test code, and the problems caused by the wrong usage of such constructs, practices, and conventions.

We could sort most topics into *Understanding which code structures to test* and *Understanding of test smells* categories. In the former, the discussions usually present a code excerpt asking whether the developers should test a structure of the production code (e.g., overloaded/private/public methods). In the latter, the discussions often refer to whether a particular test code is either smelly or not and why someone should be concerned about the impacts of test smells.

For example, we classified the following question in the *Understanding test smells* category. That category shows a barrier developers face in learning specific constructs for creating test cases. Although they perceive the code as smelly, they need to understand the pros/cons of improving the test code.

> (...) I always tell myself as long as the "real" code is "good," that's all that matters. Plus, unit testing usually requires various "smelly hacks" like stubbing functions. How concerned should I be over poorly designed ("smelly") unit tests? [39]

In the *how* questions, we classified the discussions into

twelve categories. These encompass strategies and techniques to handle problems reported in the *why* questions through other developers' experience. The *How to handle dependencies (with mock)* and *How to refactor mocks* categories deal with the test code evolution to use mocks and stubs for testing external dependencies with APIs, databases, web services, and files. The *How to refactor test code to remove duplication* category discusses how to handle duplication of setup methods, annotations, and test cases covering overloaded production methods. The *How to deal with async* category discusses testing framework constructs to avoid non-deterministic tests given the asynchronicity of the production code. The *How to refactor test code to improve coverage* and *How to refactor to evolve legacy code* categories involve understanding the production code to co-evolve the production and test codes, improving test coverage. The *How to organize the test code*, *How to name test methods and classes following naming conventions*, and *How to organize the documentation of test code* categories seek to standardize the naming, documentation, and organization of test codes. The *How to refactor test smells* and *How to refactor test smells (pointed by tools)* categories discuss strategies to fix test smells identified by developers or tools (e.g., SonarQube). Lastly, the *How to assure that the refactoring does not break the tests* category presents discussions on techniques to verify whether the behavior of the test code keeps the same after test code refactorings.

For example, we classified the following question into two categories *Understanding which code structures to test* and *How to handle dependencies (with mock)*. The question shows an example of a helper method containing some external dependency. The first category aims to understand whether the developer should test the helper method, and the second asks for strategies to implement the test code handling the dependencies of the helper method.

> So I have a helper method [..] for which I can call to grab a particular object rather than to remember which dependencies I need to hook up to get the object I require.
> My first question here is: should methods like these be tested? The only reason I can think of to test these methods would be to ensure that the correct dependencies are used and set up correctly. If the answer to the first question is yes, my second is: how? [40]

In the *what* questions, we classified the discussions into seven categories. The *What are the best practices to name test classes, methods, mocks, or paths*, *What are the best practices for unit testing*, and *What are the best practices for mocking* categories ask for coding standards, patterns, and guidelines for constructs in the test code. The *What are the strategies and tools to find unused/dead code* and *What are the tools to rename and organize tests* categories present tools for refactoring the test code to match the naming and structure of the production classes and improve the test code readability by removing unused code. The *What are the mocking anti-patterns* and *What are the test smells* categories ask for the anti-patterns and test smells definitions and catalogs that can occur in the test code.

For example, we classified the following question in the

What are the good practices for unit tests category. With this question, the developer received guidance on which patterns to use while developing the test code, e.g., the *Arrange, Act, Assert* (AAA) pattern for arranging and formatting the code.

> Usually, when talking about coding standards, we refer to the code of the program itself, but what about the unit tests? Are there certain coding standards guidelines that are unique to unit tests? [41]

Conversely, we classified the following question [42] in the *What are the test smells* category. The answers to this question provided more than 31 test code anti-patterns.

> There must be at least two key elements present to formally distinguish an actual anti-pattern from a simple bad habit, bad practice, or bad idea [...] Vote for the TDD anti-pattern that you have seen "in the wild" one time too many. [42]

> **Finding 1:** Developers are interested in the practical experience of other developers to understand test smells and decide whether and how to refactor the test code to fix them.

C. Test code problems (RQ2)

In RQ$_2$, we investigated the test smells discussed by developers in the Stack Exchange network. We followed Garousi et al.'s catalog [15] to classify the posts. Although most question topics did not explicitly ask about a test smell, we could establish a link between the description of the test code problem and the test smell categorization. Therefore, we labeled the discussions without explicit test smells as **General Discussion (GD)** and the discussions with explicit test smell as **Specific Discussion (SD)** on test smells. We analyzed 36 SD with 46 test smells and 64 GD with 125 test smells.

Fig. 6 presents the categorization of discussions into test smells. We found test smells composing seven of the eight top categories proposed by Garousi et al. [15]. The *Code related* category refers to test smells related to the test code duplication, long, complex, and hard-to-understand tests, and tests that do not follow coding best practices regarding naming conventions and code organization. The *Dependencies* category refers to test smells related to dependencies within the test code or with external resources. The *In association with production code* category refers to test smells related to coupling and dependencies between test and production code, making the tests hard to evolve. The *Mock and stub related* category refers to test smells related to misusing mock objects and mocking verification. The *Issues in test steps* category refers to occurring test smells in specific language constructs such as assertions and setup methods. The *Test execution/behavior* category refers to test smells that can lead to unexpected results as non-determinism. The *Test semantic/logic* category refers to test smells related to test logic and several responsibilities per test. The only category we did not find was the *Design related* category, which mainly presents test smells related to page-object patterns in Selenium tests.

The seven categories from Fig. 6 group 54 test smells. The *Issues in test steps* category is the most diverse regarding test smells, comprising 18 test smells. Although the category is diverse, most of its test smells were discussed only once. Conversely, the *Code related* category is diverse and comprises two of the most recurrent test smells in the discussions. The *Bad Naming* test smell was the most frequent, with 14 occurrences. It describes the non-compliance with naming conventions for test code structures such as variables, methods, and classes. The *Code Organization* test smell was the second most frequent, with 13 occurrences. It describes the non-compliance with test code organization as following the same production and test classes package hierarchy.

In addition, the *Code related* category has the most test smells gathered from specific topics. The developers explicitly discussed ten out of 17 test smells of this category. Differently, we gathered the most test smells of other categories from the GD. While the *In association with production code*, *Mock and stub related* only contain GD, the *Dependencies* and *Issues in test steps* categories have three SD each, and the *Test execution/behavior* and *Test semantic/logic* categories have one SD each. It can indicate that developers face problems in different categories of test smells. Still, they know more about naming the test smells in the *Code related* category.

> **Finding 2:** Developers usually ask whether something is a test smell or an anti-pattern, rather than referring to a particular one.

D. Test code refactorings (RQ3)

In RQ$_3$, we investigated the solutions for handling test smells that developers suggest in the Stack Exchange network. We analyzed the actions suggested in the top answers of each discussion, resulting in 33 answers in the *Fix* category, seven answers in the *Capture* category, 70 answers in the *Explain* category, and one answer in the *Ignore* category.

In addition, we extracted the code-based answers, tools, and documents suggested in the answer topics to support developers in fixing test smells. The answers categorized into the *Fix* category suggested 40 test code refactorings for fixing test smells. Less than half of the answers (13; 39.4%) in this category presented code-based refactoring recommendations, i.e., the answers included code samples. In comparison, 32 answers (41.5%) categorized into the *Explain* and *Capture* categories presented 64 patterns for organizing the test code and good practices for preventing test smells. Some answers (10; 12.9%) presented examples of using patterns and good practices. The only answer in the *Ignore* category discussed the trade-offs of refactoring a test method with many assertions to fix a test smell or keeping the assertions as documentation.

Next, we analyzed the solutions proposed in the answers to remove duplicates and group similar answers according to the developers' definitions. Fig. 7 presents 54 solutions classified into three categories: i) *Good practices for preventing*
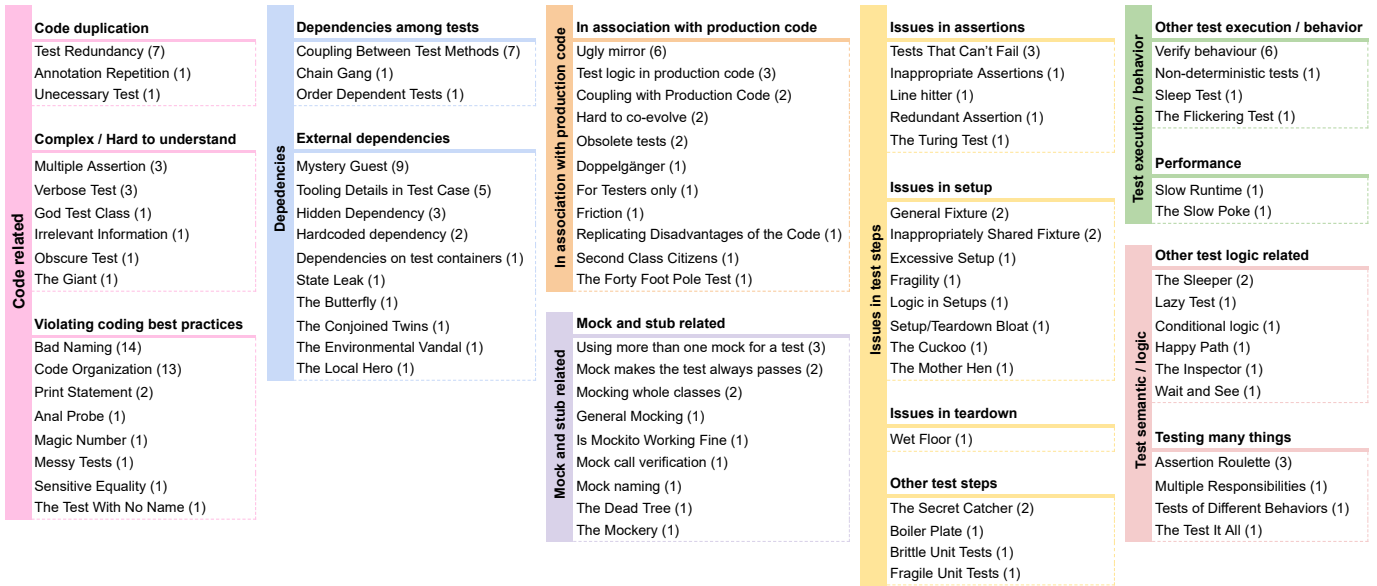
Fig. 6: Classification of GD and SD on test smells according to Garousi's catalog of test smells [15].

Code related

Code duplication
Test Redundancy (7)
Annotation Repetition (1)
Unecessary Test (1)

Complex / Hard to understand
Multiple Assertion (3)
Verbose Test (3)
God Test Class (1)
Irrelevant Information (1)
Obscure Test (1)
The Giant (1)

Violating coding best practices
Bad Naming (14)
Code Organization (13)
Print Statement (2)
Anal Probe (1)
Magic Number (1)
Messy Tests (1)
Sensitive Equality (1)
The Test With No Name (1)

Dependencies

Dependencies among tests
Coupling Between Test Methods (7)
Chain Gang (1)
Order Dependent Tests (1)

External dependencies
Mystery Guest (9)
Tooling Details in Test Case (5)
Hidden Dependency (3)
Hardcoded dependency (2)
Dependencies on test containers (1)
State Leak (1)
The Butterfly (1)
The Conjoined Twins (1)
The Environmental Vandal (1)
The Local Hero (1)

In association with production code
Ugly mirror (6)
Test logic in production code (3)
Coupling with Production Code (2)
Hard to co-evolve (2)
Obsolete tests (2)
Doppelgänger (1)
For Testers only (1)
Friction (1)
Replicating Disadvantages of the Code (1)
Second Class Citizens (1)
The Forty Foot Pole Test (1)

Mock and stub related
Using more than one mock for a test (3)
Mock makes the test always passes (2)
Mocking whole classes (2)
General Mocking (1)
Is Mockito Working Fine (1)
Mock call verification (1)
Mock naming (1)
The Dead Tree (1)
The Mockery (1)

Issues in test steps

Issues in assertions
Tests That Can't Fail (3)
Inappropriate Assertions (1)
Line hitter (1)
Redundant Assertion (1)
The Turing Test (1)

Issues in setup
General Fixture (2)
Inappropriately Shared Fixture (2)
Excessive Setup (1)
Fragility (1)
Logic in Setups (1)
Setup/Teardown Bloat (1)
The Cuckoo (1)
The Mother Hen (1)

Issues in teardown
Wet Floor (1)

Other test steps
The Secret Catcher (2)
Boiler Plate (1)
Brittle Unit Tests (1)
Fragile Unit Tests (1)

Test execution / behavior

Other test execution / behavior
Verify behaviour (6)
Non-deterministic tests (1)
Sleep Test (1)
The Flickering Test (1)

Performance
Slow Runtime (1)
The Slow Poke (1)

Test semantic / logic

Other test logic related
The Sleeper (2)
Lazy Test (1)
Conditional logic (1)
Happy Path (1)
The Inspector (1)
Wait and See (1)

Testing many things
Assertion Roulette (3)
Multiple Responsibilities (1)
Tests of Different Behaviors (1)
The Test It All (1)

problems with roots on, ii) *Patterns*, and iii) *Refactorings for problems associated with*. The colors represent the categories of test code problems from Fig. 6. The patterns, good practices, and test code refactorings aim to solve them.

The *Patterns* category refers to patterns to structure the test code, avoiding issues in the test steps. Most solutions suggested organizing the test methods according to the AAA pattern [21], [43]. This pattern organizes the test methods into three steps: (1) setup inputs and targets, (2) act on the target behavior, and (3) assert expected outcomes. It is also called the *Given-When-Then* pattern in the *Behavior Driven Development (BDD)* process [44], [45]. The *Four-Phase Test* pattern adds one step into that pattern: (4) reset to its pre-setup state [46].

The *Good practices for preventing problems with roots on* category refers to good practices covering the top test problems presented in Fig. 6. The *Test Steps* category presents good practices for organizing the test code. The *Code-related* category suggests practices to provide useful information while naming variables and test methods and classes to facilitate the identification of failures caught by the tests. The *Dependencies* category suggests practices for handling dependencies among tests and with external resources. The *Mock and stub related* category brings insights into the correct usage of the mocking frameworks. The *In association with production code* category suggests separating the responsibilities to avoid coupling between test and production codes. The *Test execution/behavior* category aids in developing tests for async tasks of the production code. The *Test semantic/logic* category presents good practices to avoid developing naive test cases and cover as many paths as possible.

Similarly, the *Refactoring for problems associated with* category refers to test code refactorings to solve the seven top problems presented in Fig. 6. The *Test steps* category presents only one refactoring to extract common arrange code in test methods into setup/teardown methods. The *Code-related* category presents refactorings to deal with code duplication

Good practices for preventing problems with roots on

Test steps
Use setup/teardown (3)
Do not mix different types of test codes (1)
Do not put assertions in the setup method (1)
Organize the code following good practices (1)
Separate setup and assert methods (1)

Code-related
Follow naming conventions (7)
Give meaningfull names (6)
Asserting strings instead of discrete values (1)
Fast, Focused Feedback (1)
Propositional Style Naming (1)
Write meaningfull assert messages (1)

Dependencies
Use mocks (3)
Keep all code related entries together (1)
Tests should run independently of each other (1)
Use fakes instead of mock (1)

Mock and stub related
Do not mock entire classes (1)
Do not use unrealistic test data (1)

In association with production code
Assert the behavior of your class's features (4)
Do not test private methods (2)
Avoid coupling your code to unit tests (1)
Minimize Test-Code Distance (1)

Test execution / behavior
Avoid use sleep statements (1)
Idempotent between test sessions (1)
Results do not vary based on the environment (1)
Static state sharing between threads (1)

Test semantic / logic
One scenario per test (7)
Asserting elements in a UI (1)
Do as little as possible (1)
Do not create tests only for the happy path (1)

Patterns

Test steps
Arrange, Act and Assert (AAA) (4)
Don't Repeat Yourself (DRY) (1)
Four-Phase Test (1)
Given-When-Then (1)

Refactorings for problems associated with

Test steps
Extract setup/teardown (1)

Code-related
Reestructure test folder (3)
Remove unnsued method (2)
Rename test methods and classes (2)
Test the 'helper object' in isolation (2)
Create helper methods (1)
Replace print by assert (1)

Dependencies
Use dependency injection (5)
Create mocks (2)
Mock database (1)

Mock and stub related
Create a shared sttubing (1)
Match widely and verify precisely (1)

In association with production code
Create an interface (1)
Fixing production class (1)
Use abstract factory (1)
Use reflection (1)

Test execution / behavior
Use test annotation (2)
Mock the elapsed time (1)
Use explicit wait (1)

Test semantic / logic
Decompose test logic (1)
Use guard asserts to multiple assertions (1)

Fig. 7: Suggestions of patterns for code organization, good practices, and test code refactorings for preventing and fixing test smells.

and bad naming. The *Dependencies* category suggests using mocks or dependency injection to handle external dependencies. The *Mock and stub related* category presents refactorings to remove duplication related to mocks and ensure well-designed asserts. The *Association with the production code* category presents refactorings in the production class that can require adaptations in the test code. The *Test execution/behavior* category presents refactorings to handle async tasks by using specific features of the testing or mocking frameworks. Lastly, the *Test semantic/logic* category presents refactorings to decompose the logic in test methods, reducing their complexity.

To guide the adoption of testing patterns, good practices, and test code refactorings, the developers pointed out 32 resources in their answers. Most resources are blogs and books (5; 21.7% each) that present test anti-patterns and refactoring strategies to fix them. In addition, the developers suggested mocking frameworks (7; 30.4%) and pointed out the documentation for constructs of testing and mocking frameworks (4; 17.4%). Some developers also suggested online courses and videos (1; 4.4% each).

> **Finding 3:** Developers more often discuss good practices and test patterns than code-based refactoring recommendations.

E. Test behavior (RQ4)

In RQ_4, we analyzed whether developers are concerned with keeping the test code behavior after performing refactorings to fix test smells. From 101 discussions, only eight (7.9%) addressed test code behavior. In most discussions, the developers are interested in evolving legacy codes, understanding how to perform the refactoring phase of TDD, or improving the test code quality.

After reading Martin Fowler's book [2], the developer decided to refactor the test code to organize and remove redundant code. The developer linked refactoring production and test codes, asking how to test the test code refactorings [47]. As a strategy to keep test code behavior, the answer suggests:

> [...] The trick with complex refactoring of test code is to be able to run the tests against the system under test and get the same results. [...] [47]

In another discussion, the developer brings up definitions of refactoring and how to ensure that refactorings do not break the code behavior. Following those definitions, the developer's interest lies in understanding how to refactor smelly test codes and whether creating meta-tests helps keep the test code behavior [48]. The answer suggests:

> When modifying tests, keep the SUT (System Under Test) unchanged. Tests and production code keep each other in check, so varying one while keeping the other locked is safest. [...] [48]

> **Finding 4:** Discussions on how to keep the test code behavior suggest 1) locking the production code while modifying the tests and 2) checking the results of the tests before and after performing the refactoring. There is no suggestion of tools or strategies to make this process more trustworthy.

IV. DISCUSSION

This section discusses and interprets the results of each RQ we addressed in this study. Besides, this section points out some existing gaps.

In RQ_1, we raised the challenges developers face regarding test code problems and the solutions to cope with them. Developers have to overcome the barrier of (1) convincing management and development teams of the implications of test smells for the test code quality, (2) co-evolving the production and test codes, and (3) keeping themselves up-to-date about the new constructs of programming languages and testing frameworks, as they emerge. Although all the discussions occurred after the first catalogs of test smells [22], [49], we found that developers commonly ask for others' perceptions and practical experience on test code problems and proven solutions. The discussions on the Stack Exchange network highlight a gap between industry and academia, indicating the importance of effectively disclosing literature findings.

In RQ_2, we classified 54 test smells into eight high-level categories of test code problems, based on [15]. In addition, we publicized the test smells definitions through an online catalog [38] to help developers understand, prevent, detect, and refactor test smells. Practitioners could use the catalog to educate themselves on test smells concepts. To evolve the catalog, we invited researchers and developers to submit pull requests to update the website with test smells and their definitions.

In RQ_3, we listed the solutions for test code problems and classified them into good practices, test patterns, and test code refactorings. Most solutions explain how to deal with the test code problem but do not demonstrate them. In addition, one good practice or test code refactoring could prevent or refactor more than one test smell. As we could not establish a link between the good practices and test code refactorings to the specific test smells, we linked the good practices and test code refactorings to the top test code problems they aim to solve. Hence, exploring good practices and test code refactorings to fix test smells is essential. Most solutions pointed to blogs, books, documentation, and testing frameworks. Despite the research community's efforts in developing tools for handling test smells and other problems in the test code, the solutions of the Stack Exchange network did not point to any of them. Besides proposing new tools, academic studies should evaluate their usefulness in real-world contexts.

In RQ_4, we observed developers are concerned with keeping the test code behavior after the refactoring. However, no well-defined strategies or tools to aid developers in refactoring test

code. It can lead developers to skepticism, missing an opportunity to improve test code quality. Therefore, researchers could focus on providing simple oracle mechanisms for test refactorings (e.g., mutation testing), or automating the catalog of refactorings in a static analyzer.

V. THREATS TO VALIDITY

Search process: We filtered the discussions related to the test code refactorings and test smells by applying a search string to the tags of each post. As developers can use different tags other than the ones we considered in our search string, we may have missed some discussions in exchange for reducing the number of non-relevant discussions returned with the search string. In addition, we only analyzed the accepted and top-voted responses for each discussion to analyze only the relevant answers that provided a refactoring action or explanation for the problems raised on the Stack Exchange network.

Generalization of findings: There are several technology-based question-and-answer websites on the Stack Exchange network. For this study, our scope focused on the top 3 websites (*Stack Overflow*, *Software Engineering*, and *Code Review*), encompassing various programming-related topics. In addition, we applied a search string to the discussions' tags to limit our analysis to test code refactorings performed with Java programming language and the JUnit testing framework.

Manual analysis bias: We performed a peer review process to mitigate bias during the selection and classification of the discussions. First, we selected a set of potential discussions, and three researchers classified them independently. We achieved an agreement level of 0.61, following Kappa statistics. Also, the selection and classification processes involved discussions aiming to solve any potential conflicts.

Popularity metric: To measure the popularity of a question and an answer, we considered counts of the number of scores on the posts. In addition, we did not consider the period of the questions (e.g., when the developers posted the questions). In our analysis, we did not distinguish between questions based on time, so there are no new questions with low counts of the number of scores and views.

VI. RELATED WORK

A. Developers' perception of test smells

Bavota et al. [50] performed the first study investigating the empirical evidence of test smells in 18 software systems and the software developers' perception of the test smells effects on the code quality. Later, the authors extended that study [51] by investigating 27 software systems and surveying 61 developers. The authors observed a high diffusion of test smells in such a study, which may lead to issues concerning the comprehensibility and maintainability of test suites and production code.

Similarly, Tufano et al. [52] surveyed 19 developers to investigate whether they could recognize occurrences of test smells in software projects. The results indicated developers do not recognize test smells and rarely remove them from the test code. Similarly, Junior et al. [53], [54] conducted empirical studies to unveil how consciously software developers insert test smells. The results indicated that experienced professionals introduce test smells during their daily programming tasks, even when using standard practices from their companies.

Spadini et al. [13] argued developers only sometimes perceive test smells as problematic, given the lack of thresholds to interpret them. The authors defined thresholds for nine test smells and empirically evaluated the perception of 31 developers on the proposed thresholds. As a result, the authors indicate that participants' perceptions agree with previously predefined severity thresholds and that test smells impact maintenance in test suites. Bai et al. [55] investigated the impact of test smells on test learning with 42 computer science students. Results indicated some test smells become less severe or do not occur with the evolution of the testing frameworks.

Instead of analyzing developers' opinions through surveys and interviews, our analysis of Stack Exchange discussions addresses the main challenges developers face in handling test smells in practice. This work differs from state-of-the-art approaches by (i) addressing a larger set of developers through the Stack Exchange network discussions among many developers and (ii) studying the definitions of test smells in practice, providing additional context to earlier studies.

B. Test code refactoring

van Deursen et al. [22] introduced the concept of test smells and proposed a catalog describing test smells and refactorings to fix them. Complementary, Meszaros et al. [49] and Bowes et al. [56] broadened the definition of test smells and listed relevant principles for test code. Later, Guerra et al. [20] explored those test smells definitions and proposed a catalog of 15 test code refactorings to fix them. In addition, the authors proposed a representation that can ease the analysis of whether the refactoring did not change the test code behavior.

Turning the attention to the empirical studies, Kummer [57] studied whether 20 developers recognize and refactor test smells. Results pointed out developers refactor test smells by chance. Similarly, Soares et al. [58] investigated how developers refactor test code to eliminate test smells. The authors surveyed 73 open-source developers to assess their preference and motivation to choose between smelly and refactored test code samples. In another work, Soares et al. [59] investigated whether the JUnit 5 features help refactor test code to remove test smells. They conducted a mixed-method study to analyze the usage of the testing framework features in 485 Java open-source projects, identifying the features helpful for fixing test smells and proposing test code refactorings.

To understand how often and which strategies developers use to refactor the test code for fixing test smells, Santana et al. [3] surveyed 87 developers and interviewed eight other developers. Results indicated most participants consider relevant to refactor test smells but only sometimes do it. Similarly, Campos et al. [60] asked software developers to refactor the test code of their projects and remove existing test smells.

The results indicated developers must learn how to refactor test code to remove the test smells.

The studies mentioned above asked for the developers' opinions, while other studies mined the projects' commits history to understand test code refactorings. Peruma et al. [61] investigated the relationship between refactorings and their effect on test smells in 250 open-source Android Apps. Results showed that refactoring operations in test and non-test files differ, and the refactorings co-occur with test smells. Kim et al. [62] conducted an empirical study on the test smells evolution and maintenance in 12 open-source projects. The authors analyzed the commits that removed test smells and concluded the test smells removal was due to maintenance activities. Kim et al. [63] studied the maintenance activities of developers on test annotations. They created a taxonomy by manually inspecting and classifying a sample of test annotation changes and documenting the motivations driving these changes.

Differently, Alomar et al. [64], [65] discussed the importance of considering the developer's experience as part of solutions for code refactoring. The authors conducted an empirical study on 800 open-source projects to investigate the relationship between the developers' experience and the number of refactoring activities. As a result, the authors found several developers apply refactorings, but only a few are responsible for the production and test code. Furthermore, the authors reported no correlation between experience and motivation due to refactoring.

Instead of analyzing developers' opinions or refactoring activities through the projects' histories, our study analyzes discussions from the Stack Exchange network. Those discussions can anticipate the problems that developers face with their respective solutions in practice.

C. Developers' perceptions of Stack Exchange topics

Several studies have analyzed Stack Exchange network discussions on particular topics in the Software Engineering field [28], [29], [32], [66], [67]. Openja et al. [29] analyzed 260,023 release engineering questions using topic modeling. The authors examined the developers' topics of interest and their difficulties reported on Stack Overflow. The results indicated developers discussed 38 release engineering topics, among which software testing is the most challenging topic and the most important contributor to software quality assurance.

Other topics include discussions on smells that occur in different software artifacts. Choi et al. [32] performed a preliminary study on 925 discussions on Stack Overflow about code clones. Results showed most discussions are related to refactoring with the need for more support for clone refactoring tools. Tian et al. [66] analyzed the developers' perception of architectural smells through 207 discussions on Stack Overflow. The results indicated developers use general terms to describe architectural smells caused by violating architectural patterns and design principles.

Tahir et al. [67] mined 17,126 Stack Overflow posts and manually analyzed the top 100. The results showed developers use Stack Overflow to ask for general code smell assessments rather than particular refactoring solutions. Tahir et al. [28] also analyzed how developers discuss code smells and anti-patterns across three technical Stack Exchange sites. Results showed developers often discuss the downsides of implementing specific design patterns and flag them as potential anti-patterns that developers should avoid.

Regarding code refactoring to improve its overall design, Pinto and Kamei [68] mined Stack Overflow posts to study discussions around refactoring tools. The results indicated developers prefer multi-language refactoring tools. However, they do not use them due to usability issues and a lack of trust in the refactoring process. Peruma et al. [11] analyzed 9489 refactoring discussions on Stack Overflow to investigate the trends and challenges that the developers face in refactoring software artifacts in practice. The authors automatically assigned the discussions to one of the topics: (i) code optimization, (ii) architecture and design patterns, (iii) unit testing, (iv) tools and IDEs, and (v) database. As for unit testing, developers face challenges with writing test cases, mainly to accommodate refactored production code. As a result, the authors summarized the key challenges and conclusions for relevant stakeholders considering each topic.

In contrast, our study is the first to look into test smells within the discussions of the Stack Exchange sites. More specifically, our study investigates the challenges developers face in test code refactoring, the test code problems in practice, and the test code refactorings suggested to fix test smells.

VII. Conclusions and Future Work

In this study, we investigated the discussions about test smells and test code refactoring on the Stack Exchange network. We aimed to leverage knowledge about the corrective actions developers take to deal with them and the main challenges developers face to correct test smells.

To accomplish our goal, we sorted the discussion topics into 30 categories. Each category describes questions developers ask when they face any issue during test code evolution. The yielded results indicate most topics lie in the *Understanding which code structures to test (12)* and *Understanding test smells (12)* categories. We observed developers are interested in others' perceptions and hands-on experience handling issues of test code. In addition, there is an indication the developers often ask whether test smells or anti-patterns are either good or bad testing practices than code-based refactoring recommendations.

In future work, we plan to design a catalog of test smells and their definitions to help bridge the gap between tool supporters and developers to ensure that refactoring does not break tests.

REFERENCES

[1] V. Alizadeh, M. Kessentini, M. W. Mkaouer, M. Ó Cinnéide, A. Ouni, and Y. Cai, "An interactive and dynamic search-based approach to software refactoring recommendations," *IEEE Transactions on Software Engineering*, vol. 46, no. 9, pp. 932–961, 2020.

[2] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[3] R. Santana, D. Fernandes, D. Campos, L. Soares, R. Maciel, and I. Machado, "Understanding practitioners' strategies to handle test smells: A multi-method study," in *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, ser. SBES '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 49–53.

[4] C. Dibble and P. Gestwicki, "Refactoring code to increase readability and maintainability: A case study," *J. Comput. Sci. Coll.*, vol. 30, no. 1, p. 41–51, oct 2014.

[5] A. Christopoulou, E. A. Giakoumakis, V. E. Zafeiris, and V. Soukara, "Automated refactoring to the strategy design pattern," *Information and Software Technology*, vol. 54, no. 11, pp. 1202–1214, 2012.

[6] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," in *International Conference on Software Maintenance, 2002. Proceedings.*, 2002, pp. 576–585.

[7] M. Katić and K. Fertalj, "Towards an appropriate software refactoring tool support," in *WSEAS international conference on applied computer science*, 2009, pp. 140–145.

[8] R. Oliveira, R. de Mello, E. Fernandes, A. Garcia, and C. Lucena, "Collaborative or individual identification of code smells? on the effectiveness of novice and professional developers," *Information and Software Technology*, vol. 120, p. 106242, 2020.

[9] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *Journal of Software Engineering Research and Development*, vol. 5, no. 1, pp. 1–28, 2017.

[10] E. Tempero, T. Gorschek, and L. Angelis, "Barriers to refactoring," *Communications of the ACM*, vol. 60, no. 10, pp. 54–61, 2017.

[11] A. Peruma, S. Simmons, E. A. AlOmar, C. D. Newman, M. W. Mkaouer, and A. Ouni, "How do i refactor this? an empirical study on refactoring trends and topics in stack overflow," *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–43, 2022.

[12] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 101–110.

[13] D. Spadini, M. Schvarcbacher, A.-M. Oprescu, M. Bruntink, and A. Bacchelli, "Investigating severity thresholds for test smells," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 311–321.

[14] M. Hozano, A. Garcia, B. Fonseca, and E. Costa, "Are you smelling it? investigating how similar developers detect code smells," *Information and Software Technology*, vol. 93, pp. 130–146, 2018.

[15] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, 2018.

[16] V. Garousi, K. Petersen, and B. Ozkan, "Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review," *Information and Software Technology*, vol. 79, pp. 106–127, 2016.

[17] I. Karac and B. Turhan, "What do we (really) know about test-driven development?" *IEEE Software*, vol. 35, no. 4, pp. 81–85, 2018.

[18] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Information and software technology*, vol. 50, no. 9-10, pp. 833–859, 2008.

[19] E. A. AlOmar, M. W. Mkaouer, C. Newman, and A. Ouni, "On preserving the behavior in software refactoring: A systematic mapping study," *Information and Software Technology*, vol. 140, p. 106675, 2021.

[20] E. M. Guerra and C. T. Fernandes, "Refactoring test code safely," in *International Conference on Software Engineering Advances (ICSEA 2007)*. IEEE, 2007, pp. 44–44.

[21] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[22] A. Deursen, L. M. Moonen, A. Bergh, and G. Kok, "Refactoring test code," Centre for Mathematics and Computer Science, NLD, Tech. Rep., 2001.

[23] M. Greiler, A. Van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 322–331.

[24] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory study," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '19. USA: IBM Corp., 2019, p. 193–202.

[25] P. S. Kochhar, X. Xia, and D. Lo, "Practitioners' views on good software testing practices," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 61–70.

[26] D. J. Kim, "An empirical study on the evolution of test smell," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2020, pp. 149–151.

[27] D. Posnett, E. Warburg, P. Devanbu, and V. Filkov, "Mining stack exchange: Expertise is evident from initial contributions," in *2012 International Conference on Social Informatics*, 2012, pp. 199–204.

[28] A. Tahir, J. Dietrich, S. Counsell, S. Licorish, and A. Yamashita, "A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites," *Information and Software Technology*, vol. 125, p. 106333, 2020.

[29] M. Openja, B. Adams, and F. Khomh, "Analysis of modern release engineering topics : – a large-scale study using stackoverflow –," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 104–114.

[30] T. Bhasin, A. Murray, and M.-A. Storey, "Student experiences with github and stack overflow: An exploratory study," in *2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2021, pp. 81–90.

[31] "Java: code duplication in classes and their junit test cases," Stack Overflow, 2012, Accessed on 12.29.2022. [Online]. Available: https://stackoverflow.com/questions/10781050/java-code-duplication-in-classes-and-their-junit-test-cases

[32] E. Choi, N. Yoshida, R. G. Kula, and K. Inoue, "What do practitioners ask about code clone? a preliminary investigation of stack overflow," in *2015 IEEE 9th International Workshop on Software Clones (IWSC)*, 2015, pp. 49–50.

[33] F. Gomes, E. P. d. Santos, S. Freire, M. Mendonça, T. S. Mendes, and R. Spínola, "Investigating the point of view of project management practitioners on technical debt: A preliminary study on stack exchange," in *Proceedings of the International Conference on Technical Debt*, ser. TechDebt '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 31–40.

[34] E. P. Santos, F. Gomes, S. Freire, M. Mendonça, T. S. Mendes, and R. Spínola, "Technical debt on agile projects: Managers' point of view at stack exchange," in *Proceedings of the XXI Brazilian Symposium on Software Quality*, ser. SBQS '22. New York, NY, USA: Association for Computing Machinery, 2023.

[35] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[36] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[37] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: A critical review and guidelines," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 120–131. [Online]. Available: https://doi.org/10.1145/2884781.2884833

[38] "Data collection and analysis," 2022, Accessed on 12.23.2022. [Online]. Available: https://github.com/arieslab/testsmells

[39] "If your unit test code "smells" does it really matter?" Software Engineering, 2011, Accessed on 12.23.2022. [Online]. Available: https://softwareengineering.stackexchange.com/questions/77313/

[40] "Do i need to test helper/setup methods?" Software Engineering, 2009, Accessed on 12.23.2022. [Online]. Available: https://stackoverflow.com/questions/1004866/do-i-need-to-test-helper-setup-methods

[41] "Unit test coding standards," Software Engineering, 2010, Accessed on 12.23.2022. [Online]. Available: https://softwareengineering.stackexchange.com/questions/15925/

[42] "Unit testing anti-patterns catalogue," Software Engineering, 2008, Accessed on 12.23.2022. [Online]. Available: https://stackoverflow.com/questions/333682/unit-testing-anti-patterns-catalogue

[43] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.

[44] J. Smart, *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon and Schuster, 2014.

[45] J. Korhonen, "Automated model generation using graphwalker based on given-when-then specifications," 2020.

[46] G. Meszaros, "xunit patterns - four-phase test," 2011, Accessed on 12.26.2022. [Online]. Available: http://xunitpatterns.com/Four%20Phase%20Test.html

[47] "Refactoring and test driven development," Stack Overflow, 2009, Accessed on 12.26.2022. [Online]. Available: https://stackoverflow.com/questions/657645/

[48] "How do i refactor unit tests?" Stack Overflow, 2015, Accessed on 12.26.2022. [Online]. Available: https://stackoverflow.com/questions/31434305/

[49] G. Meszaros, S. M. Smith, and J. Andrea, "The test automation manifesto," in *Extreme Programming and Agile Methods - XP/Agile Universe 2003*, F. Maurer and D. Wells, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 73–81.

[50] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "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)*, 2012, pp. 56–65.

[51] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, p. 1052–1094, 2015.

[52] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 4–15.

[53] N. S. Junior, L. R. Soares, L. A. Martins, and I. Machado, "A survey on test practitioners' awareness of test smells," in *Iberoamerican Conference on Software Engineering*, vol. abs/2003.05613, 2020.

[54] N. Silva Junior, L. Martins, L. Rocha, H. Costa, and I. Machado, "How are test smells treated in the wild? a tale of two empirical studies," *Journal of Software Engineering Research and Development*, vol. 9, no. 1, p. 9:1 – 9:16, Sep. 2021.

[55] G. R. Bai, K. Presler-Marshall, S. R. Fisk, and K. T. Stolee, "Is assertion roulette still a test smell? an experiment from the perspective of testing education," in *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2022, pp. 1–7.

[56] D. Bowes, T. Hall, J. Petric, T. Shippey, and B. Turhan, "How good are my tests?" in *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics*, ser. WETSoM '17. New York, NY, USA: IEEE Press, 2017, p. 9–14.

[57] M. Kummer, O. Nierstrasz, and M. Lungu, "Categorising test smells," *Bachelor Thesis. University of Bern*, 2015.

[58] E. Soares, M. Ribeiro, G. Amaral, R. Gheyi, L. Fernandes, A. Garcia, B. Fonseca, and A. Santos, "Refactoring test smells: A perspective from open-source developers," in *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, ser. SAST 20. New York, NY, USA: Association for Computing Machinery, 2020, p. 50–59.

[59] E. Soares, M. Ribeiro, R. Gheyi, G. Amaral, and A. M. Santos, "Refactoring test smells with junit 5: Why should developers keep up-to-date," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.

[60] D. Campos, L. Rocha, and I. Machado, "Developers perception on the severity of test smells: an empirical study," *XXIV Ibero-American Conference on Software Engineering*, 2021.

[61] A. Peruma, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "An exploratory study on the refactoring of unit test files in android applications," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 350–357.

[62] D. J. Kim, T.-H. P. Chen, and J. Yang, "The secret life of test smells-an empirical study on test smell evolution and maintenance," *Empirical Software Engineering*, vol. 26, no. 5, pp. 1–47, 2021.

[63] D. J. Kim, N. Tsantalis, T.-H. Chen, and J. Yang, "Studying test annotation maintenance in the wild," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 62–73.

[64] E. A. AlOmar, A. Peruma, C. D. Newman, M. W. Mkaouer, and A. Ouni, "On the relationship between developer experience and refactoring: An exploratory study and preliminary results," in *IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW'20. New York, NY, USA: ACM, 2020, p. 342–349.

[65] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. D. Newman, and A. Ouni, "Behind the scenes: On the relationship between developer experience and refactoring," *Journal of Software: Evolution and Process*, vol. 0, p. e2395, 2021.

[66] F. Tian, P. Liang, and M. A. Babar, "How developers discuss architecture smells? an exploratory study on stack overflow," in *2019 IEEE International Conference on Software Architecture (ICSA)*, 2019, pp. 91–100.

[67] A. Tahir, A. Yamashita, S. Licorish, J. Dietrich, and S. Counsell, "Can you tell me if it smells? a study on how developers discuss code smells and anti-patterns in stack overflow," in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, ser. EASE'18. New York, NY, USA: Association for Computing Machinery, 2018, p. 68–78.

[68] G. H. Pinto and F. Kamei, "What programmers say about refactoring tools? an empirical investigation of stack overflow," in *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, ser. WRT '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 33–36.