Identifying and Addressing Test Smells in JavaScript: A Developer-Centric Study

Jhonatan Oliveira
UEFS
Bahia, Brazil
natanjesuss20@gmail.com

Luigi Mateus
UNEB
Bahia, Brazil
luigimee15@gmail.com

Gabriel Amaral
UEFS
Bahia, Brazil
gabrielamaralsousa@gmail.com

Tássio Virgínio
UFBA
Bahia, Brazil
tassio.virginio@ufba.br

Carla Bezerra
UFC
Quixadá, Brazil
carlailane@ufc.br

Ivan Machado
UFBA
Bahia, Brazil
ivan.machado@ufba.br

Larissa Rocha
UNEB/PGCC-UEFS
Bahia, Brazil
larissabastos@uneb.br

ABSTRACT

Test smells are poor practices in test code that can compromise maintainability, reliability, and clarity. While the concept has been widely studied in languages such as Java and Python, research on test smells in JavaScript remains limited—despite its prominence in modern development. To address this gap, we conducted a focus group study with JavaScript developers of varying experience levels to explore their perceptions of seven test smells. These smells—*Anonymous Test*, *Comments Only Test*, *Overcommented Test*, *General Fixture*, *Test Without Description*, *Transcripting Test*, and *Sensitive Equality*—are particularly relevant to the JavaScript ecosystem and had not been systematically examined in this context prior to our study. We applied thematic analysis to transcribed discussions, uncovering developers' concerns, recognition patterns, and proposed mitigation strategies. Our results show that experience level strongly influences the ability to detect and refactor test smells, with junior developers often struggling to identify more subtle patterns. To the best of our knowledge, this is the first study to investigate JavaScript developers' perceptions of test smells using a qualitative approach. Our findings reveal key challenges, offer practical insights for test improvement, and support the development of better training and tooling for JavaScript test quality.

KEYWORDS

Test Smells, Focus Group, JavaScript

1 Introduction

Software testing plays a fundamental role in ensuring the delivery of reliable and high-quality software. Among the various testing strategies, unit testing stands out as an essential practice due to its focus on verifying the correctness of individual software components in isolation [18]. By automating these tests, developers can quickly identify issues and ensure that specific units of code behave as expected. However, for unit tests to fulfill their purpose effectively, they must be well-designed, not only to achieve functional correctness but also to serve as clear and maintainable documentation of the software's expected behavior. Poorly written tests, even when automated, can lead to misinterpretation, reduced trust in the test suite, and increased maintenance effort, ultimately undermining the benefits of testing [27].

To address these challenges, Deursen et al. [31] introduced the concept of test smells, anti-patterns in test code that hinder comprehension, maintainability, and reliability. Over the years, test smells have been studied extensively in languages like Java [22, 31–33], Python [11, 35], and Scala [9]. In these studies, various tools have been developed for detecting test smells, ranging from command-line utilities [25, 35] to graphical interfaces like [34] and RAIDE [28]. However, in the JavaScript ecosystem—one of the most widely used programming languages worldwide[1] —there remains a significant gap in research and tooling for detecting test smells.

While test smells have been extensively studied in languages like Java and Python, the JavaScript ecosystem presents unique challenges due to its dynamic typing, asynchronous nature, and the widespread adoption of testing frameworks such as Jest and Vitest. Likewise, JavaScript tools that focus on code quality unfortunately lack standards to ensure test code quality. As a result, tools like linters, including the widely used ESLint, still do not provide default configurations that promote better test quality. Unlike statically typed languages, JavaScript's lack of compile-time checks increases reliance on clear test descriptions and assertions to document expected behavior. Our study focuses on these JavaScript-specific nuances, bridging the gap between generic test smell research and the practical realities of modern web development.

To bridge this gap, our study employs *SNUTS.js* [23], a tool capable of detecting seven distinct test smells: *Anonymous Test, Comments Only Test, Overcommented Test, General Fixture, Test without Description, Transcripting Test, and Sensitive Equality* [23]. These test smells, if present, can lead to increased maintenance effort and confusion, potentially compromising the reliability of the test suite. It is important to note that the test smells detected by the SNUTS.js tool had never been implemented by other detection tools

[1]https://www.tiobe.com/tiobe-index/

before. This makes understanding and analyzing these test smells particularly relevant to the developer community, as it provides new insights into testing practices in JavaScript and highlights the need for more specialized tools to improve code quality.

This paper evaluates the relevance and impact of these test smells in JavaScript projects, aiming to understand their significance within the JavaScript developer community. To achieve this, we conducted a focus group study with JavaScript developers of varying experience levels, selected based on their professional roles and familiarity with popular unit testing libraries such as Jest, Jasmine, Vitest, and Node Test Runner. Through this qualitative analysis, we aim to validate the importance of detecting test smells and highlight how tools like SNUTS.js can support better testing practices in JavaScript development.

Our results indicate that JavaScript developers recognize test smells as a significant concern within the community, with their primary impact being on code readability and maintainability. Developers also suggested refactoring strategies to mitigate the issues caused by these smells. Furthermore, we observe that more experienced developers have a stronger understanding of best practices in test code and are less susceptible to introducing test smells into the codebase. One of the key challenges for JavaScript developers is identifying specific issues, such as the improper use of methods and unnecessary mock data. This study highlights the potential to enhance best practices in writing test code, specifically within the JavaScript community.

2 Background

Software quality is essential in engineering reliable and maintainable systems. Key attributes such as readability, maintainability, and reliability are especially important when evaluating unit tests and test code [10]. Readable code eases understanding and reduces the likelihood of errors [5], while maintainable code supports efficient updates and adaptations [17]. Reliability reflects the consistent performance of software over time [15]. These quality aspects are not limited to production code—test code must also uphold them. Inadequate testing practices may lead to test smells, which compromise the clarity, upkeep, and trustworthiness of tests.

2.1 Test Smells

Unit tests involve conducting automated tests to ensure the quality of what is being developed and to validate the expected results. In this sense, performing unit tests is common in the context of software engineering. Test smells are a set of bad practices in unit test code that can lead to problems, such as difficulty in maintaining the test, difficulty in understanding the unit test, and ultimately, the loss of test effectiveness, which may allow bugs to persist [32] [28] [16]. Identifying test smells is essential to ensure the readability, coherence, and functionality of test code.

In the following, we present a summary of the test smells used in this study, which were originally adapted to the JavaScript ecosystem in our prior work [23]. We were pioneers in adapting these test smells to JavaScript, as, to the best of our knowledge, no other work discusses them in the context of JavaScript.

2.1.1 Test Without Description (TWD). This type of test smell occurs when a test block lacks a description. In the test example Listing 1, the test case has an empty description, which can affect the readability and future maintenance of the mentioned example. Test cases must have clear and coherent descriptions relevant to their defined context, ensuring they fulfill their objectives.

Listing 1: Example: Test Without Description

```
it("", () => {
    const result = getPermission();
    expect(result).toBe(true);
});
```

2.1.2 Anonymous Test (AT). This smell occurs when a name lacks a clear, descriptive meaning in its context. In Listing 2, the expression *should work* is vague and open to interpretation, reducing the clarity of the test and complicating future maintenance. Descriptive phrases are essential for readability. This smell is especially relevant in JavaScript, where testing libraries like Jest require developers to provide meaningful test descriptions.

Listing 2: Example: Anonymous Test

```
it("should work", () => {
   const result = getAge();
   expect(result).toBeEqual(10);
});
```

2.1.3 Transcripting Test (TT). The specified test smell occurs when print commands are used within the test case. In the example Listing 3, it is evident that different "console" commands are used within the test block to display messages in the terminal, which can impact performance, and security, and cause visual clutter in the test.

Listing 3: Example: Transcripting Test

```
test("Test 1", () => {
   console.log("Loggin to the console");
   expect(someFinction()).toBe(true);
});
test("Test 2", () => {
   console.warn("Loggin to the console");
   expect(someFunction()).toBe(true);
});
test("Test 3", () => {
   console.error("Loggin to the console");
   expect(someFunction()).toBe(true);
});
```

2.1.4 Comments Only Test (COT). This test smell occurs when a test is fully commented out, making it inactive and potentially forgotten over time. Although commonly done during debugging or refactoring, leaving such tests in the codebase harms readability and creates uncertainty about their status, whether they were failing, obsolete, or unintentionally left behind. Listing 4 shows a fully commented-out test case.

Listing 4: Example: Comments Only Test

```
 // it("should work", () => {
 //   const result = getAge();
 //   expect(result).toBeEqual(10);
// });
```

2.1.5 *Overcommented Test (OT).*

This type of test smell refers to a test block that has too many comments. In the example Listing 5, it can be observed that the comments are described redundantly, and such comments affect the readability of the test code by adding noise without providing any additional value.

Listing 5: Example: Overcommented Test

```
test("returns user information",() => {
  // Get user information
  const userInfo = getUserInfo();
  // Expected user structure
  const expectedUser = {
    id: 1,
    username: "jhon_doe",
    email: "jhon@example.com"
  };

  // Compare with expected result
  expect(userInfo).toEqual(expectedUser);
  // Verify required fields exist
  expect(userInfo.id).toBeDefined();
  // Check data types
  expect(typeof userInfo.id).toBe('number');
});
```

2.1.6 *Sensitive Equality (SE).*

This smell occurs when an assertion relies on the *toString* method for comparison, as in Listing 6. Tests of this kind are fragile, as they depend on the specific implementation of *toString()*, making them prone to failure with internal changes. In JavaScript, this smell is particularly relevant due to dynamic typing and implicit type coercion, which enable flexible but unreliable comparisons, especially when involving string conversions.

Listing 6: Example: Sensitive Equality

```
test("it should check user age to be equal to 13", () => {
 const user = getUser();
 expect(user.age.toString()).toBeEqual('13');
});
```

2.1.7 *General Fixture (GF).*

This type of test smell occurs when the test setup defines multiple data or objects, but only a subset is used. The example Listing 7 demonstrates that the test has a *beforeEach* block, which by definition is executed before each test, ensuring that each instance is initialized with defined values. However, no test case uses the *guest* variable.

Listing 7: Example: General Fixture

```
let user;
let admin;
let guest;

beforeEach(() => {
    user = new User("Alice", 30);
    admin = new Admin("Bob", 20);
    guest = new Guest("Charlie", 25);
});

test("user should have a name", () => {
    expect(user.name).toBe("Alice");
});

test("admin should have an age", () => {
    expect(admin.age).toBe(40);
});
```

2.2 SNUTS.js

Previously, we introduced SNUTS.js [23], a user-friendly tool for detecting test smells in JavaScript. It leverages Abstract Syntax Trees (AST) to identify subtle testing issues often missed by traditional static analysis, enabling the development of higher-quality test suites. To enhance accessibility, it includes a web interface that allows repository analysis directly from the browser, with no installation required.

At launch, SNUTS.js detected seven previously unsupported test smells: Test Without Description, Anonymous Test, Transcripting Test, Comments Only Test, Overcommented Test, Sensitive Equality, and General Fixture. It now supports eight additional smells recognized by other tools[2]: Complex Snapshot, Conditional Test Logic, Identical Test Description, Non-functional Statement, Only Test, Suboptimal Assert, Verbose Test, and Verify In Setup.

By bridging detection gaps, SNUTS.js offers a unified, efficient solution for test smell analysis and has gained traction among developers and researchers aiming to improve JavaScript test quality.

3 Study Design

The focus group is a method in which researchers expand the open-ended interview format into group discussions [30]. It typically consists of 3 to 12 participants and includes a moderator to guide the discussion, following a predefined script to ensure the group remains focused. The goal is to create an environment that encourages the development of ideas and responses, leveraging the interactions and exchanges among participants to enrich the content generated on the subject [3, 13, 19, 30, 36].

This study employs the focus group method based on the guidelines proposed by Kontio et al. [20], which include the following stages: Defining the research problem, Selecting the participants, and Planning and Conducting the focus group session. A detailed description of each step is provided in the following subsections.

3.1 Defining the research problem

This study aims to better understand developers' challenges and perceptions regarding test smells in JavaScript. Thus, we aim to address the following research questions:

- **RQ1: How do JavaScript developers perceive the impact of different test smells on code quality, particularly in terms of readability, maintainability, and reliability?** This question seeks to understand developers' perspectives on how test smells influence code quality (e.g., readability, maintainability, and reliability). We also analyse whether developers perceive some test smells as more harmful than others.
- **RQ2: Which test smells are most frequently recognized by JavaScript developers, and which ones pose the biggest challenges to identify?** This question aims to investigate the overall level of awareness among developers regarding different test smells, seeking to identify which ones are more intuitive to recognize and which tend to go unnoticed or be misinterpreted by the JavaScript developer community.

[2]The source code and documentation for SNUTS.js are available at https://github.com/jhonatanmizu/SNUTS.js.

- **RQ3: What strategies do developers suggest for mitigating or refactoring test smells in JavaScript unit tests?** This question focuses on how developers propose addressing test smells when they identify them. We explore whether participants suggest concrete refactoring strategies, if they rely on existing best practices, or if they struggle to propose effective solutions.
- **RQ4: How does the level of experience (junior, intermediate, senior) influence JavaScript developers' ability to recognize test smells, assess their severity, and propose strategies to address them?** This question examines the role of experience in the detection and management of test smells, comparing how junior, mid-level, and senior developers perform in identifying these issues, proposing solutions, and assessing how seriously they perceive the impact of test smells on test quality.

3.2 Selecting the participants

To select participants, we used a background form (available on Zenodo) to gather data on their experience with JavaScript and unit testing in JavaScript, as well as their availability for the focus group. We distributed the form by contacting professors from various universities, asking them to share it with students and alumni experienced in the topic. We also shared the invitation on social media (e.g., LinkedIn and Instagram) and with companies in the field. Invitations were sent and resent over a three-month period. Still, we faced challenges in recruiting participants with experience in JavaScript testing.

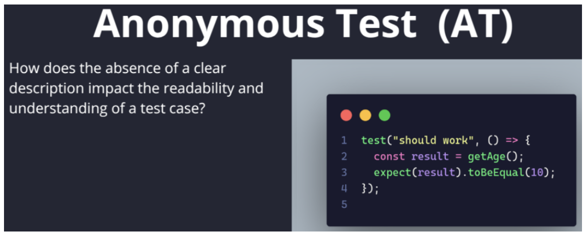
As a result, we received a total of 36 responses, from which we selected the participants, prioritizing those with experience in JavaScript and unit testing, regardless of the popular libraries they use. Thus, after analyzing the experience and availability form, only 15 people were eligible to participate in the study.

3.3 Planning and conducting the focus group session

3.3.1 Pilot Study. We conducted a pilot study with two randomly selected participants, each session lasting 30–40 minutes. The goal was to assess the clarity, flow, and effectiveness of the focus group structure and materials before the main study. The sessions revealed opportunities for improvement, particularly in the order of test smell presentation and discussion pacing. Notably, the *General Fixture* smell was moved to the end, as it imposed a higher cognitive load. Introducing simpler examples first helped create a smoother learning curve and improved participant engagement. Participants also noted that the guiding questions were clear and relevant. The contextualized scenarios fostered a comfortable, open environment, encouraging more candid responses.

3.3.2 Planning. To conduct this research, we adopted the focus group method to gather feedback and experiences from software engineers. This approach has proven effective and versatile in previous studies Kontio et al. [19].

This study aimed to understand the relevance of identifying bad practices in JavaScript test code and to gather perspectives and feedback from developers. Due to participants' varying availability, we conducted the study across three separate focus groups, each



Anonymous Test (AT)

How does the absence of a clear description impact the readability and understanding of a test case?

```
test("should work", () => {
  const result = getAge();
  expect(result).toBeEqual(10);
});
```

Figure 1: Example of question and test code used in the presentation

consisting of four to five participants. Since two people participated in the pilots, the study had thirteen participants. The sessions were held remotely via Google Meet, as scheduling a common time for all participants proved challenging. Two experienced researchers facilitated the discussions. For the study, we prepared a structured presentation featuring the seven test smells under study: *Anonymous Test, Comments Only Test, Overcommented, General Fixture, Test without Description, Transcripting Test, and Sensitive Equality.* The presentation included examples of unit test code, each accompanied by a guiding question designed to foster discussion among participants. The material used in this study is available on Zenodo.

At the beginning of each session, we introduced the concepts involved in the study and the study's objectives. To encourage meaningful discussions, we provided contextualized scenarios for each test smell. Hence, to each smell, we showed one small code snippet with a question associated with it. Figure 1 shows an example of the *Anonymous Test* smell (AT), which does not contain an adequate description of the test purpose, and its corresponding question. In this case, instead of "*should work*", an example of a more appropriate test description would be "*should return the correct age when calling getAge*". Each participant had approximately 3 to 5 minutes to review the test smell and its corresponding question. They were then asked to analyze the example code and share their perspectives on whether they identified issues or found it acceptable. Each focus group session lasted between 30 to 45 minutes. In the first session, we had 5 participants, and it lasted 43 minutes; in the second session, 4 participants, lasting 39 minutes; and in the third session, 4 participants, with a duration of 41 minutes. The following questions guided the discussion on test smells.

- **TWD**: Consider that your test case does not include a descriptive text. How does this issue impact the readability and understanding of the test case?
- **AT**: How does the absence of a clear description impact the readability and understanding of a test case?
- **TT**: How can the inclusion of logs, such as console.log, console.warn, or console.info, in a test case affect the readability, clarity, performance, and effectiveness of the tests?
- **COT**: What is the impact on the maintenance and clarity of a test case that is completely commented out, without any executable test assertions?
- **OT**: What is the impact on the maintenance and clarity of a test case that has too many comments?
- **SE**: What are the implications and challenges of using the *toString()* method to compare primitive types and data structures in tests, and how can this affect the accuracy and clarity of test results?

- **GF**: What are the risks associated with using general fixtures (mocked data) in test cases, and how can this impact the reliability and independence of the tests?

During the discussion, participants could ask questions related to the examples. After that, if a participant shared an opinion, others were encouraged to agree or disagree, explaining their point of view. They were also invited to reflect on whether they had encountered those examples of test code in their daily routine and how they believed the existing test code could be improved. In addition, we encouraged them to share valid suggestions for refactoring or best practices that could help mitigate the identified test smells.

3.3.3 Data Analysis. To analyze the extracted data and address the research questions, all focus group sessions were recorded and manually transcribed from audio. These transcriptions, available on Zenodo, served as the basis for analysis. The transcriptions (available on Zenodo) were manually done by listening to the audio recorded of the three focus groups. We adopted a *thematic analysis* approach [14, 24]. Initially, one researcher systematically organized all transcribed material, grouping participant statements according to each test smells discussion. Subsequently, three researchers engaged in a collaborative thematic coding process. This procedure consisted of three distinct phases: (1) open coding, where emerging concepts and ideas were identified in participant statements; (2) axial coding, which enabled grouping initial codes into broader thematic categories; and (3) selective coding, where we refined and consolidated the main categories that emerged from the analysis.

To ensure analytical reliability, we held multiple consensus sessions to discuss and validate the proposed categories. This process included standardizing terminology and merging similar codes into coherent themes. As a concrete example of this process, in the specific case of the "*Anonymous Test*" smell, our initial analysis identified fourteen distinct codes, which were progressively consolidated into three main categories: : (1) Impaired Readability; (2) Useless or Misleading Description; and (3) Maintenance Impact. This rigorous approach allowed us to organize raw qualitative data into structured, actionable insights, while preserving the participants' original perspectives and highlighting patterns aligned with the research goals.

We also considered the participants' roles in the software industry to gain insights and understand the general perspective of specific positions related to test smells. Thus, the distribution of experience levels was analyzed to capture varying perspectives. For instance, we investigated whether entry-level developers are capable of identifying test smells and suggesting effective refactoring strategies to reduce their prevalence in the codebase.

4 Results

This section presents the focus group findings, organized by RQs.

4.1 Participants' Profile and experience with test smells

4.1.1 Demographic Overview. The majority of participants were full-stack developers, highlighting the widespread use of JavaScript for both back-end and front-end tasks. Most of them were men, and one participant, P10, was identified as a woman. The roles and experience levels of the participants are summarized in Table 1.

Name	Position	Experience
P1	Fullstack Developer	1-2 years with JavaScript 1-2 years with Testing
P2	Fullstack Developer	1-2 years with JavaScript 1-2 years with Testing
P3	QA Developer	1-2 years with JavaScript 1-2 years with Testing
P4	Tech Lead	5-10 years with JavaScript 5-10 years with Testing
P5	Fullstack Developer	5-10 years with JavaScript 3-4 years with Testing
P6	Fullstack Developer	3-4 years with JavaScript 1-2 years with Testing
P7	QA Developer	1-2 years with JavaScript 1-2 years with Testing
P8	Fullstack Developer	1-2 years with JavaScript 1-2 years with Testing
P9	Fullstack Developer	3-4 years with JavaScript 3-4 years with Testing
P10	Backend Developer	5-10 years with JavaScript 1-2 years with Testing
P11	Fullstack Developer	1-2 years with JavaScript 1-2 years with Testing
P12	Fullstack Developer	1-2 years with JavaScript 1-2 years with Testing
P13	Frontend Developer	5-10 years with JavaScript 3-4 years with Testing

Table 1: Participants' profile

4.1.2 Experience Distribution. The participants were categorized into three distinct experience levels based on their years of practice with JavaScript and testing:

Entry-Level: The entry-level developers were mainly junior professionals with 1-2 years of experience in JavaScript and testing. These participants had limited exposure to identifying and addressing test smells, which was reflected in their lower engagement with refactoring suggestions.

Mid-Level: The mid-level developers, with 3-4 years of experience in JavaScript and unit testing, showed a stronger grasp of identifying test smells. They contributed more effectively by suggesting potential refactoring solutions.

Senior-Level: Senior developers, with 5-10 years of experience, were able to identify advanced test smells such as General Fixture and Anonymous Test, demonstrating a deeper understanding of testing principles. Their suggestions for mitigating test smells were more insightful, reflecting their extensive experience with best practices.

4.1.3 Role-Specific Observations. The participants represented four different professional roles:

QA Developers: Two QA developers participated, and while their experience with testing tools and automation was apparent, they showed more difficulty in recognizing subtle test smells. This may be attributed to their primary focus on test automation rather than code quality.

Fullstack Developers: The majority of participants were fullstack developers, with a solid background in both frontend and backend JavaScript development. Most had 1-2 years of experience with unit testing. These developers were proactive in emphasizing best practices such as version control and proper commenting, which helps in mitigating test smells.

Tech Lead: The tech lead, with 5-10 years of experience, exhibited a strong focus on the quality of test code and suggested advanced strategies for avoiding test smells, likely due to their leadership role and deeper familiarity with design patterns.

Frontend Developers: Only one frontend developer participated, making it difficult to generalize the role's involvement with test smells. This participant, with 5-10 years of experience in JavaScript, contributed insightful observations but did not represent the broader group of frontend developers.

4.2 RQ1: Impact of different test smells on code quality

Participants consistently identified test smells as harming code quality, particularly in terms of readability and maintainability. The most commonly discussed test smells were Overcommented Test, Transcripting Test, General Fixture, and Anonymous Test. Developers emphasized that these smells make test code harder to understand and maintain, leading to increased effort in debugging and refactoring.

4.2.1 Test Without Description (TWD). The *TWD Test Smell* was a starting point in our focus group discussions. Participants identified three main issues: (1) Difficulty in debugging; (2) Loss of test code's documentation value; and (3) Misinterpretation.

Regarding the **difficulty in debugging the code**, participants expressed concerns about the impact of this smell on test code readability. If a test fails or requires maintenance, one would need to read the entire body of the code to fully understand what is being tested. P12 emphasized how this could affect productivity in the workflow "*[...]imagine if there were multiple lines of code—you'd have to interrupt your workflow to check why the test is failing or passing.*".

Additionally, some participants warned that writing tests without descriptions **compromises the documentary value** of the test code. This was noted by P2 "*A test should also serve as documentation, and the moment it lacks a description, it loses that value [...]*" P13 added "*I agree that a test serves as a form of documentation. And if you don't describe what you're testing, you're losing one of its functionalities.*" One participant pointed out a potential **interpretation issue**, as reading the code might lead to different understandings of what is being tested. As P10 observed, "*Besides requiring more time to understand what the code is doing, there is also the risk of misinterpreting its intent.*"

Our focus group data reveal a strong consensus among participants regarding this smell. Out of 13 participants, 11 (84.62%) agreed that the *TWD Test Smell* is problematic to the code, while the remaining participants did not express their opinions.

4.2.2 Anonymous Test (AT). The *AT Test Smell* was identified without much difficulty by the participants. Three central problems were highlighted: (1) Impaired Readability, (2) Useless and Misleading Description, and (3) Maintenance Impact. It was pointed out that this smell could be even more harmful than the previous one *TWD Test Smell*, as P2 commented "*This would be even worse than the first case (TWD), because, first, it's not describing anything, so you force the need to read the code body.*"

Participants particularly highlighted the challenges related to the lack of clarity in the test, **how it impacts readability**, potentially confusing future consultations. P3 emphasized that "*When a test isn't clear in its description, it becomes completely out of context. We can't define the context without having to read the entire code*". This lack of clarity can harm how the test is interpreted leading to a **useless and misleading description**, as emphasized by P6 "*[...]the description doesn't help at all. And even if someone else reads it, because it's so vague, they might think it's about one thing when, in reality, it's about something else.*" P9 also commented on the impact on **maintenance** of lacking a clear description "*What I see is that the lack of a clear description causes the developer [...] might end up testing something more than once due to the lack of clarity.*" Additionally, P12 reinforced this issue "*[...]If you don't understand what a test is doing, someone might come later and remove it or change it without realizing its purpose.*" Some participants also stated that they do not see any positive aspects in providing a poor description lacking details (P8) and highlighted how this can leave the test completely out of context (P3).

Our focus group dataset demonstrates significant consensus among participants about the AT smell (76,92%). Besides, ten of the thirteen participants actively commented on the smell, while the others did not express their opinions about it.

4.2.3 Transcripting Test (TT). The Transcripting Test (TT) smell sparked diverse reactions during the focus groups. While many participants criticized the use of 'console.log' in test files, others expressed neutral or even positive perspectives in specific contexts. From the discussions, three distinct themes emerged: Violation of Principles, Test Structure Issues, and Code Pollution.

Violation of Principles was one of the most strongly voiced concerns. Participants argued that using logs in tests goes against the core principles of unit testing. P9 noted, "*If we are using logs in unit tests, we are violating one of the fundamental principles of unit testing, which is interacting with the external environment by redirecting strings to the user's console output [...]*" Similarly, P1 pointed out that keeping logs after the debugging phase is not advisable: "*During development it's necessary, but [...] I don't think it's good practice to keep logs in your code.*"

Another concern was the **Test Structure** implications of using logs, especially in complex or lengthy tests. According to P4, "*If the log is trying to describe something in the test, [...] maybe the test is too long and should be split.*" Similarly, P8 suggested that reliance on logs might indicate poorly structured test cases: "*If you need a console.log to understand an error message [...] that's a problem.*" *Code pollution* was also highlighted. Participants mentioned that excessive or leftover logs could lead to cluttered and unreadable test code. P2 stated, "*If each test has two or three logs [...] it becomes a huge code bloat.*" P3 added, "*This can become dead code, which clutters [...] and ends up being ignored.*"

Despite these criticisms, some participants viewed the use of logs as acceptable under certain conditions. This perspective gave rise to two emergent themes: Acceptable Temporary Use and Acceptable Use for Backend Development. Regarding *temporary use*, P5 shared, "*I think it's fine to use console.log when you're testing and building something [...].*" P6 echoed this view, noting they personally don't mind logs during development: "*I'm not a fan, but I don't mind [...]*".

The theme of **acceptable use in backend contexts** was introduced by P13, who described scenarios where logs were essential for debugging backend systems: "*Logs are a tool—if you don't have another way to inspect the code [...] logs sometimes make it easier to identify the problem.*" P10 also made a distinction: while logs are common and helpful in backend services, they are rare in tests and can harm test readability.

Overall, while some participants expressed neutrality or contextual acceptance, the majority (61.45%) acknowledged that logs in tests can negatively affect test quality—whether by violating principles, complicating structure, or polluting the code. These findings suggest that the Transcripting Test smell deserves attention in practice, particularly to ensure cleaner, principle-aligned, and maintainable test code.

4.2.4 *Comments Only Test (COT).*

The discussions around the *Comments Only Test* smell revealed three main categories: Dead Code and Visual Clutter, Acceptable Cases, and Versioning Misuse. Most participants perceived this smell as problematic, especially when entire test blocks are commented out without justification.

Several participants emphasized that commented-out tests make the code harder to read and maintain (**Dead code and visual clutter**). P5 noted: "*[. . .] it's basically dead code [. . .] just causes more problems.*" P6 added: "*[. . .] commented code is rarely uncommented [. . .] just delete it.*" P4 mentioned: "*These comments are useless [. . .]*", highlighting that such code adds noise with no real benefit. Some participants acknowledged that commenting out tests **might be acceptable in very specific scenarios**. P10 stated: "*[. . .] only while still developing the feature [. . .]*", and P8 said: "*[. . .] if it was a feature that didn't work and might need to be reverted.*" However, they stressed the importance of including a comment explaining the reason, otherwise it becomes meaningless.

According to P6, this smell reflects **poor use of version control tools**: "*[. . .] if you keep good commit practices per PR, you can trace and revert anything.*" Instead of relying on commented code to preserve past versions, participants argued that developers should trust the version history.

In this context, the focus group data reveal that the participants view the *COT Test Smell* as a relevant issue. In fact, 92.31% of them agreed that commenting out entire test blocks is problematic, highlighting that such practices lead to visual clutter and make the test suite harder to navigate.

4.2.5 *Overcommented Test (OT).*

The *Overcommented Test* (OT) smell was widely recognized in our focus group discussions. Participants highlighted four main concerns: (1) redundant explanations that simply repeat what the code already expresses, (2) comments compensating for poorly written test code, (3) maintenance issues due to outdated comments, and (4) a few acceptable cases for comments, such as educational contexts or very complex logic.

Redundant Explanations were the most frequently mentioned issue. Many participants found comments that mirror the test code to be unnecessary and even harmful to readability. As P3 put it, "*People use comments to document poorly written code [...] but there are better ways to do that.*". P8 emphasized that such comments are avoidable with good naming: "*The correct approach would be to have explanatory code with well-named variables and functions.*"

The use of comments to compensate for **Poorly Written Code** was another recurring theme. Several participants saw heavy commenting as a sign of unclear or low-quality code. P1 stated, "*If you add too many comments, it means your code isn't good.*" P12 linked this to clean code principles: "*Too many comments make the test visually cluttered [...] it feels lazy.*" Participants also discussed the **Impact on Maintenance**, especially when comments become outdated. P4 warned, *"When the test is changed [...] the comment loses its meaning.*" This can create confusion and lead to incorrect assumptions during future maintenance.

Despite the overall criticism, some participants acknowledged **Acceptable Cases** for using comments. P5 mentioned, "*Sometimes you find a line you don't understand [...] that's when a comment makes sense.*" P8 noted that comments might be justified in learning environments. Others, like P10 and P13, emphasized using comments only when the code is particularly complex or unclear at first glance.

Overall, the OT smell was actively discussed by 10 out of 13 participants. Notably, ten participants (76,92% of those who commented) identified *OT Test smell* as problematic. The general consensus points to overcommenting as a clear code smell that should be addressed through better code structure, naming, and readability rather than through excessive documentation.

4.2.6 *Sensitive Equality (SE).*

Regarding the *Sensitive Equality (SE)* test smell, we identified four thematic categories in participants' reflections: Unjustified Complexity, Impact on Reliability, Performance and Readability Issues, and Valid Use Cases for `toString`.

Several participants emphasized the issue of **unjustified complexity**, pointing out that using `toString` in test assertions adds unnecessary layers to the code. As P3 stated, *"There's no need to use it [...] I only use* `toString` *when I really have to convert some type"*. P2 expressed a similar stance: *"Is there any reason to use* `toString` *in this case?"*, while P4 argued that *"it makes the test more verbose"*. According to P12, *"You're testing something that wouldn't happen in the real world [...] which feels strange"*.

Others pointed out the **impact on reliability**, noting that converting values into strings may prevent tests from catching meaningful errors. For instance, P6 remarked, *"If* `getUser` *returns a number and for some reason it returns a string instead, the test won't catch that"*. In a similar vein, P5 explained, *"The error happens, but not for the right reasons [...] and that may cause confusion"*. A few participants mentioned **performance and readability issues**, especially when `toString` is used excessively or unnecessarily. P1 commented, *"It impacts performance by using resources unnecessarily"*, and P6 added, *"It's just an extra method to read [...] it doesn't help with readability or the effectiveness of the test"*.

Finally, some participants acknowledged **valid use cases for `toString`**, though with caution. P10 described a case where backend values are numbers, but the frontend expects strings, noting: *"It could cause a bug [...] it might behave differently on different servers"*. P12 added that while *"you could stringify objects to compare them [...] there are better tools for that"*.

These findings suggest that the SE smell is widely recognized as problematic due to its potential to obscure test logic and introduce fragility. Most participants (9 out of 13 - 69.23%) recognized this smell and expressed concerns about the use of the `toString` method in assertions, especially when it masks the real behavior of the system under test. The focus group participants largely agreed that relying on `toString` in assertions should be avoided, with a preference for more transparent and type-safe testing practices.

4.2.7 *General Fixture (GF).*

Regarding the *General Fixture (GF)* smell, participants demonstrated a clear understanding of its drawbacks. Their reflections revealed four thematic categories: Lack of independence between tests, Unnecessary resource allocation, Tight coupling and risk of false positives, and Justified use of mocks.

The issue of **lack of independence between tests** was highlighted by participants who observed that using shared mock data can lead to unintended dependencies. P2 warned, *"One of the test blocks could change the existing values [...] this would break the other tests."* P4 shared a similar concern: *"If another test modifies some mocked data, it will end up affecting other tests."* Participants also emphasized the problem of **unnecessary resource allocation**. They noted that including mock data irrelevant to certain tests can result in bloated test suites and reduced maintainability. As P4 noted, *"There's a mock for the guest, but it isn't used [...] it gets in the way. Unnecessary code always causes problems."*

Concerns about **tight coupling and the risk of false positives** were also prominent. P3 explained, *"If there's an issue with a fixture, all the test cases linked to it will fail [...] this dependency can be a risk."* He further noted the danger of false positives: *"Poorly built fixtures may cause false positives [...] I'd have to build many inputs to make the test pass."* P6 added, *"I'm really against mocked data because it's not testing anything [...] that's a bit dangerous."* P5 agreed, stating, *"There are few exceptions where mocks make sense [...] I feel like I'm not testing anything."*

Despite these concerns, some participants acknowledged the **justified use of mocks**, particularly in unit testing. P9 stated, *"In unit tests, mocks are useful [...] you're testing behavior, not external dependencies."* P10 echoed this: *"It's useful if you're testing the entity itself [...] but for database functions, mocking is safer."* P12 emphasized isolation: *"You recreate mocks before each test [...] it avoids interference."* Likewise, P13 remarked, *"If the block is well-scoped, I don't see a risk."* P11 concluded, *"If it's just for the constructor, and not scattered, it's fine."*

These insights reflect a nuanced understanding of the GF smell: while general fixtures can introduce critical issues related to test fragility and maintainability, participants recognized that mocks can be useful when used judiciously within isolated and well-structured test scenarios. In conclusion, while opinions on using *General Fixture (GF)* varied, 30.77% of participants recognized it as a source of concern.

To provide a clearer overview of the findings, Table 2 summarizes the categories identified for each test smell. These categories reflect the various dimensions through which participants interpreted and evaluated the smells.

Table 2: Summary of categories for each test smell (RQ1)

Test Smell	Identified Categories
Anonymous Test (AT)	Impaired Readability, Useless or Misleading Description, Maintenance Impact
Comments Only Test (COT)	Dead code and visual clutter, Acceptable cases, Versioning misuse
Overcommented Test (OT)	Redundant explanations, Poorly written code, Maintenance impact, Acceptable cases
General Fixture (GF)	Lack of independence, Unnecessary resource allocation, Coupling and false positives, Justified use of mocks
Test Without Description (TWD)	Difficulty in debugging, Loss of test code's documentation value, Misinterpretation
Transcripting Test (TT)	Violation of principles, Test structure issues, Code pollution, Acceptable temporary use, Acceptable in backend
Sensitive Equality (SE)	Unjustified complexity, Impact on reliability, Performance and readability issues, Valid use cases for `toString`

4.3 RQ2: The test smells most commonly recognized and the most challenging to identify

The study revealed that some test smells are more easily recognized than others. Comments Only Test (COT) and Overcommented Test (OT) were the most commonly identified smells, with 92.31% and 76.92% of participants recognizing their negative impact, respectively. These smells were seen as straightforward and easy to spot due to their visual prominence in the code.

On the other hand, Sensitive Equality (SE) and General Fixture (GF) were more challenging for participants to identify. Only 23.08% of participants recognized the issue with `toString` in SE, and 30.77% struggled to identify the problem with unused mock data in GF. This suggests that some test smells require a deeper understanding of testing principles and are less intuitive to detect.

- **Junior Developers**: Junior developers, in particular, had difficulty identifying more subtle test smells like SE and GF. Their lack of experience with testing best practices made it harder for them to recognize these issues.
- **Senior Developers**: Senior developers, on the other hand, were more adept at identifying advanced test smells. For example, P4, a senior developer, was the only participant to correctly identify the issue with unused mock data in GF, demonstrating a deeper understanding of testing patterns.

4.4 RQ3: Strategies for dealing with smells in JavaScript

Participants proposed several strategies for addressing test smells, although the suggestions were often limited and varied based on experience level.

Test Without Description (TWD). Although the participants did not suggest direct refactorings, their criticisms of the smell reveal practices that can be adopted to improve the clarity and maintainability of tests. Participants P2 and P13 emphasized that tests should serve as documentation, but this value is lost when they are written without descriptions. On the other hand, participants such as P6, P12, and P10 mentioned that the absence of descriptions forces developers to analyze the entire test code, increasing the effort required to understand it and making it more vulnerable to misinterpretation.

Comments Only Test (COT). P8 and P6 suggested using version control tools and maintaining good commit standards to avoid the need for commented-out code. P6 emphasized that "*we have code versioning tools that allow us to track changes; if good commit standards are maintained per Pull Request, you can review and revert if needed.*"

Overcommented Test (OT). The participants proposed several concrete strategies to address OT tests, which can be categorized into two main approaches. First, *Improving Code Quality* was emphasized as a primary solution, with P1 stating, "*Code should be clear* *enough to understand in seconds without comments*," and P8 recommending "*explanatory code with well-named variables and functions.*" This approach prioritizes self-documenting test structures over supplemental explanations. Second, *Aggressive Comment Removal* was widely supported, particularly for redundant or obsolete annotations. P5 argued, "*Comments that just repeat the next line's functionality should be removed—they add nothing but extra text.*" The participants' recommendations align with the broader principle that maintainable tests should communicate intent through structure rather than annotations. P12 reinforced this view: "*Clean Code principles emphasize clarity without excessive comments.*" These findings suggest that teams should treat verbose test comments as a smell, addressing them through code improvements rather than documentation. The proposed measures, derived directly from practitioner insights, offer actionable steps to enhance test readability and maintainability.

Anonymous Test (AT). P3 suggested refactoring based on the context of the test case, stating that: "*it's about finding a context where one or more test cases are, right? And declaring the test according to the context it operates in, right? According to what it does and its context.*"

Transcripting Test (TT). Although participants did not propose refactorings directly, their criticisms highlight practices to improve test clarity and maintainability. Among the main implicit suggestions, participants P1 and P4 stated that logs should not be kept in environments such as staging or production. As P1 noted *"I don't see it as a good practice to keep logs [...] in production or staging."* This suggests that logs should be removed after the development phase. It was also observed that relying on logs to understand errors may indicate clarity issues in the code, as highlighted by P8 *"If you need logs to understand errors, the code may have clarity problems."* This suggests that complex or unclear tests may need to be redesigned to eliminate the need for logs.

General Fixture (GF). No specific refactoring strategies were proposed for GF, but P4 highlighted the importance of avoiding unused mock data, suggesting that developers should ensure that all fixtures are relevant to the tests being executed.

Sensitive Equality (SE). Although no specific refactorings were suggested by the participants regarding the *SE Test Smell*, they emphasized the importance of addressing this issue in test code due to its potential impacts on reliability, performance, and readability. Another concern raised was that this behavior introduces unnecessary complexity into the test code, as illustrated by P1's comment: *"I think the problem lies in using additional methods unnecessarily, right? It ends up affecting performance and using resources unnecessarily."*

Overall, the lack of broader refactoring suggestions indicates a need for greater awareness of best practices and tools to help developers address test smells more effectively.

4.5 RQ4: Influence of experience on smells

Experience level played a significant role in how participants perceived and addressed test smells. Junior developers often struggled to recognize more subtle smells and were less likely to propose refactoring strategies. In contrast, senior developers demonstrated a stronger understanding of testing best practices and were more proactive in identifying and addressing test smells.

- **Junior Developers**: Junior developers tended to focus on implementation details rather than the broader impact of test smells. They often relied on existing examples and were less likely to evaluate the quality of the test code. In particular, their lack of experience with testing best practices made it harder for them to identify more subtle test smells like SE and GF.
- **Mid-Level Developers**: Mid-level developers showed greater confidence in identifying test smells and were more efficient than junior developers in proposing refactoring strategies. They emphasized the importance of best practices, such as code versioning and removing unnecessary comments.
- **Senior Developers**: Senior developers were able to identify more advanced test smells, such as GF and AT, and suggested effective refactoring strategies. Their experience allowed them to recognize patterns that could lead to maintenance issues and advocate for better test structure and modular design. For example, P4, a senior developer, was the only participant to correctly identify the issue with unused mock data in GF, demonstrating a deeper understanding of testing patterns.

5 Discussion

This section synthesizes cross-cutting insights and reflects on their broader implications for software testing in JavaScript.

First, the study reveals that developers' **perceptions of test smells are shaped not only by experience level**, but also by their exposure to testing principles and code review practices. For instance, while most participants identified visual smells like *Overcommented Test (OT)* and *Comments Only Test (COT)* with ease, more **subtle structural issues** like *General Fixture (GF)* and *Sensitive Equality (SE)* proved difficult to recognize—particularly for junior developers. This suggests that visibility plays a critical role in test smell awareness, which could inform how we design educational resources and tooling.

Second, despite recognizing the presence of smells, **many developers struggled to suggest concrete refactorings**—especially for smells like *SE* and *GF*. This indicates a gap not just in awareness, but also in **actionable knowledge**. These findings support the need for **training resources** and **tooling that go beyond detection**, offering contextualized suggestions for improvement. Third, the findings show a tension between **pragmatism and idealism** in testing practices. While some developers were strict about removing code smells (e.g., logs or commented code), others accepted them in specific development stages. This tension points to a nuanced view of smells—not as absolute faults, but as **context-dependent symptoms**. Future tools should consider this, potentially allowing developers to mark smells as "temporarily accepted" with expiration warnings or commit-time flags.

Fourth, tools such as Jest and Vitest enhance test expressiveness by providing structured matchers and support for snapshot testing. However, **creating truly expressive and maintainable tests still depends largely on the developer's skill** and commitment to best practices, especially in writing clear and meaningful test descriptions that accurately convey the test's intent.

Additionally, the focus group highlighted **the importance of naming and test intent**, especially in smells like *Anonymous Test (AT)* and *Test Without Description (TWD).* Participants consistently emphasized that tests are not only for verification but also serve as documentation. This dual role reinforces the idea that tools like *SNUTS.js* should not only identify structural issues but also **assist in improving test expressiveness**.

Finally, while *SNUTS.js* was not the central focus of the discussion, the study indirectly reinforces its relevance. The test smells that it detects align with those developers found most harmful. This provides preliminary validation of the tool's focus and hints at its potential to **bridge the gap between theory and practice**, particularly if paired with mentoring and educational initiatives for junior developers.

6 Threats to Validity

This study presents several threats to validity. **Conclusion validity** may be affected by the subjective nature of developers' perceptions, which can vary widely. This variability makes it difficult to draw objective conclusions about the severity and impact of test smells. Regarding **internal validity**, group dynamics during discussions could have influenced participant responses. Some participants might have conformed to majority opinions or hesitated to challenge dominant voices. Additionally, the moderation style and the framing of questions may have shaped the conversation, potentially emphasizing certain viewpoints. In terms of **construct validity**, variations in how participants interpreted test smells—especially General Fixture and Sensitive Equality—suggest that the concept may not have been consistently understood, weakening the link between theory and observation. **External validity** is also limited, as the 13 participants may not reflect the broader JavaScript developer community. Their feedback was collected in a controlled setting, which does not fully replicate the constraints of real-world development environments, such as time pressure or tooling diversity. References to specific tools like Jest or SonarQube further limit generalizability, since not all developers use the same technologies. Finally, **reliability** may be impacted by participants' uncertainty about some smells, indicating inconsistencies in understanding that could affect the stability of the results.

7 Related Work

Recent studies have examined test smells from multiple perspectives, including identification, impact, and developer perception [1, 2, 12, 26]. Garousi and Küçük [12] conducted a comprehensive review of testing research, emphasizing the value of grey literature and noting that many test smells stem from professional experience rather than empirical evidence—limiting their generalizability.

Bavota et al. [4] demonstrated that test smells hinder code comprehension and increase maintenance costs, though their work did not explore developer perceptions. This gap was later addressed by Campos et al. [6], who found that severity judgments are highly context-dependent and that many developers lack refactoring knowledge. Similarly, Souza [29] observed diverse perceptions and a strong interest in automated tools, but inconsistent mitigation strategies.

Despite these contributions, little research focuses specifically on test smells in the JavaScript ecosystem. To address this, our study collects empirical data on JavaScript developers' experiences using focus groups—a methodology from social research that structures open-ended discussions to surface practical insights [21, 30]. Widely applied in software engineering [7, 8, 19], focus groups enable us to explore the emergence of smells, developer strategies, and potential tooling needs in a real-world context.

By centering on JavaScript, we aim to deepen the understanding of test smells in this widely used environment and support the development of more effective practices for managing them.

8 Conclusions

This study highlights the influence of experience levels and professional roles on the ability to identify and address test smells in JavaScript code. The results indicate that junior developers tend to struggle with detecting test smells, while mid-level and senior developers demonstrate a stronger understanding of best practices. Tech leads, in particular, showed a proactive approach in implementing patterns to improve test code quality.

Among the most frequently identified test smells were Overcommented Test, Transcripting Test, General Fixture, and Anonymous Test. Participants noted that excessive comments, lack of test descriptions, and the presence of logging statements negatively impact test readability and maintainability. Senior developers were particularly effective in recognizing the need for refactoring strategies. Best practices for mitigating test smells include maintaining clear test descriptions, adopting structured version control practices, and using context-based refactoring. However, the study also revealed gaps in participants' ability to suggest concrete refactorings, indicating an opportunity for education to improve the test quality.

To the best of our knowledge, this is the first study to investigate JavaScript developers' perceptions of test smells, offering novel insights into how different experience levels affect their ability to recognize and address these issues.

ARTIFACT AVAILABILITY

The background form, the material used to support the focus group sessions, the example code illustrating each test smell, and the thematic coding are available in: https://zenodo.org/records/16756800.

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001; CNPq grants 315840/2023-4 and 403361/2023-0; and FAPESB grant PIE0002/2022. This work was also partially supported by UEFS-AUXPPG 2025 and CAPES-PROAP 2025 grants.

REFERENCES

[1] 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. In *Proceedings of the 25th International Conference on Evaluation*

and Assessment in Software Engineering (Trondheim, Norway) *(EASE '21)*. Association for Computing Machinery, New York, NY, USA, 170–180. https://doi.org/10.1145/3463274.3463335

[2] Manoel Aranda, Naelson Oliveira, Elvys Soares, Márcio Ribeiro, Davi Romão, Ullyanne Patriota, Rohit Gheyi, Emerson Souza, and Ivan Machado. 2024. A Catalog of Transformations to Remove Smells From Natural Language Tests. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (Salerno, Italy) *(EASE '24)*. Association for Computing Machinery, New York, NY, USA, 7–16. https://doi.org/10.1145/3661167.3661225

[3] Sharon L. Baker. 1991. Improving Business Services through the Use of Focus Groups. *RQ* 30, 3 (1991), 377–385. http://www.jstor.org/stable/25828811

[4] 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 (2015), 1052–1094.

[5] Raymond P. L. Buse and Westley R. Weimer. 2008. A Metric for Software Readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery, 121–130. https://doi.org/10.1145/1390630.1390647

[6] 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).

[7] Maya Daneva and Niv Ahituv. 2012. What agile ERP consultants think of requirements engineering for inter-organizational ERP Systems: Insights from a Focus Group in BeNeLux. In *16th International Conference on Evaluation and Assessment in Software Engineering (EASE 2012)*. 284–288. https://doi.org/10.1049/ic.2012.0037

[8] Maya Daneva, Andrea Herrmann, Nelly Condori-Fernandez, and Chong Wang. 2019. Understanding the Most In-demand Soft Skills in Requirements Engineering Practice: Insights from Two Focus Groups. In *Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering* (Copenhagen, Denmark) *(EASE '19)*. Association for Computing Machinery, New York, NY, USA, 284–290. https://doi.org/10.1145/3319008.3319352

[9] 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* (London, United Kingdom) *(Scala '19)*. Association for Computing Machinery, New York, NY, USA, 22–26. https://doi.org/10.1145/3337932.3338815

[10] Norman E. Fenton and Shari Lawrence Pfleeger. 1998. *Software Metrics: A Rigorous and Practical Approach* (2nd ed.). PWS Publishing Co., USA.

[11] Daniel Fernandes, Ivan Machado, and Rita Maciel. 2022. TEMPY: Test Smell Detector for Python. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering* (<conf-loc>, <city>Virtual Event</city>, <country>Brazil</country>, </conf-loc>) *(SBES '22)*. Association for Computing Machinery, New York, NY, USA, 214–219. https://doi.org/10.1145/3555228.3555280

[12] 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. https://doi.org/10.1016/j.jss.2017.12.013

[13] Ronald E. Goldsmith. 2000. The Focus Group Research Handbook. *The Service Industries Journal* 20, 3 (07 2000), 214. https://www.proquest.com/scholarly-journals/focus-group-research-handbook/docview/203357973/se-2 Copyright - Copyright Frank Cass and Co. Ltd Jul 2000.

[14] Judith Green and Nicki Thorogood. 2018. Qualitative methods for health research. (2018).

[15] International Organization for Standardization. 2001. *ISO/IEC 9126: Software engineering — Product quality*. Technical Report ISO/IEC 9126. International Organization for Standardization. https://www.iso.org/standard/22749.html

[16] Dalton Jorge, Patricia Machado, and Wilkerson Andrade. 2021. Investigating Test Smells in JavaScript Test Code. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing* (Joinville, Brazil) *(SAST '21)*. Association for Computing Machinery, New York, NY, USA, 36–45. https://doi.org/10.1145/3482909.3482915

[17] Rick Kazman, Phil Bianco, James Ivers, and John Klein. 2020. *Maintainability*. Technical Report CMU/SEI-2020-TR-006. Software Engineering Institute, Carnegie Mellon University. https://doi.org/10.1184/R1/12954908

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

[19] J. Kontio, L. Lehtola, and J. Bragge. 2004. Using the focus group method in software engineering: obtaining practitioner and user experiences. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04.* 271–280. https://doi.org/10.1109/ISESE.2004.1334914

[20] J. Kontio, L. Lehtola, and J. Bragge. 2004. Using the focus group method in software engineering: obtaining practitioner and user experiences. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04.* 271–280. https://doi.org/10.1109/ISESE.2004.1334914

[21] Oliver T. Massey. 2011. A proposed model for the analysis and interpretation of focus groups in evaluation research. *Evaluation and Program Planning* 34, 1 (2011), 21–28. https://doi.org/10.1016/j.evalprogplan.2010.06.003

[22] Gerard Meszaros, Shaun M. Smith, and Jennitta Andrea. 2003. The Test Automation Manifesto. In *Extreme Programming and Agile Methods - XP/Agile Universe 2003*, Frank Maurer and Don Wells (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg.

[23] Jhonatan Oliveira, Luigi Mateus, Tássio Virgínio, and Larissa Rocha. 2024. SNUTS.js: Sniffing Nasty Unit Test Smells in Javascript. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 720–726. https://doi.org/10.5753/sbes.2024.3563

[24] Diana Papaioannou, Anthea Sutton, and Andrew Booth. 2016. Systematic approaches to a successful literature review. *Systematic approaches to a successful literature review* (2016), 1–336.

[25] 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1650–1654. https://doi.org/10.1145/3368089.3417921

[26] Valeria Pontillo. 2024. Insights Into Test Code Quality Prediction: Managing Machine Learning Techniques. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (Salerno, Italy) *(EASE '24)*. Association for Computing Machinery, New York, NY, USA, 2. https://doi.org/10.1145/3661167.3661268

[27] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th international workshop on automation of software test (AST)*. IEEE, 36–42.

[28] Railana Santana, Luana Martins, Larissa Rocha, Tássio Virgínio, Adriana Cruz, Heitor Costa, and Ivan Machado. 2020. RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering* (<conf-loc>, <city>Natal</city>, <country>Brazil</country>, </conf-loc>) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 374–379. https://doi.org/10.1145/3422392.3422510

[29] Gabriel Crespo de Souza. [n. d.]. Test Smells: An Industry Developer Perspective. https://bdm.unb.br/handle/10483/36332

[30] J.F. Templeton. 1994. *The Focus Group: A Strategic Guide to Organizing, Conducting and Analyzing the Focus Group Interview*. Probus Publishing Company. https://books.google.com.br/books?id=CRTFQgAACAAJ

[31] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. 2001. In *Refactoring Test Code*, M. Marchesi and G. Succi (Eds.). Proceedings 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001).

[32] Tássio Virgínio, Luana Almeida Martins, Larissa Rocha Soares, Railana Santana, Heitor Costa, and Ivan Machado. 2020. An empirical study of automatically-generated tests from the perspective of test smells. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering* (<conf-loc>, <city>Natal</city>, <country>Brazil</country>, </conf-loc>) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 92–96. https://doi.org/10.1145/3422392.3422412

[33] 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* (Salvador, Brazil) *(SBES '19)*. Association for Computing Machinery, New York, NY, USA, 467–471. https://doi.org/10.1145/3350768.3350775

[34] 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, 1 (Sep. 2021), 8:1 – 8:14. https://doi.org/10.5753/jserd.2021.1893

[35] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. 2022. PyNose: a test smell detector for python. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 593–605. https://doi.org/10.1109/ASE51524.2021.9678615

[36] Richard Widdows, Tia A. Hensler, and Marlaya H. Wyncott. 1991. The Focus Group Interview: A Method for Assessing Users' Evaluation of Library Service. *College and Research Libraries* 52, 4 (1991), 352–359. https://doi.org/10.5860/crl_52_04_352