Test Smells through the Lens of the Dart Community: A Mixed-Methods Study on Grey Literature and Developer Perceptions

Eronildo Junior
Federal University of Bahia
Salvador, Brazil
eronildosilva@ufba.br

Rafael Rocha
Federal University of Bahia
Salvador, Brazil
rafaelvieira@ufba.br

Paloma Passos
Federal University of Bahia
Salvador, Brazil
palomabcp@ufba.br

Paulo Cerqueira
Federal University of Bahia
Salvador, Brazil
cerqueira.paulo@ufba.br

Tássio Virgínio
Federal University of Bahia
Salvador, Brazil
tassio.virginio@ufba.br

Antonio Rito Silva
Instituto Superior Técnico, University of Lisbon
Lisbon, Portugal
Rito.Silva@tecnico.ulisboa.pt

Ivan Machado
Federal University of Bahia
Salvador, Brazil
ivan.machado@ufba.br

Abstract

Automated testing is a cornerstone of modern software quality, especially in rapidly growing ecosystems like Dart and Flutter. However, poor design choices in tests, known as test smells, can compromise maintainability and reliability. While test smells are widely studied in Java and C#, their prevalence and developer perceptions in the Dart ecosystem remain largely unexplored. This work aims to investigate how Dart developers perceive bad testing practices and identify which test smell patterns emerge from real-world community discussions. We conducted a mixed-methods study on grey literature (specifically Stack Overflow Q&A posts), analyzing 307 selected discussions that yielded 432 individual test smell occurrences. Applying Pareto Analysis, we find that a small subset of seven smells (the "Vital Few") accounts for approximately 80% of all occurrences, with Magic Number Test, Unknown Test, and Assertion Roulette being the most prevalent. Furthermore, we identify and formalize the Expected Resolution Omission (ERO) smell, a Dart-specific asynchronous testing anti-pattern that causes silent false positives or flaky tests. Our findings highlight critical pain points in the Dart testing community and provide a formally defined catalog to support developers and tool builders in improving test quality.

Keywords

Automated Testing, Test Smells, Dart, Stack Overflow, Pareto Analysis

1 Introduction

Software engineering faces increasing challenges in developing systems that meet strict requirements for functionality, performance, and, above all, quality [3, 15]. Automated testing has emerged as the primary mechanism for maintaining software quality over time, enabling teams to detect regressions, reduce defect rates, and verify correctness across software evolution cycles [13, 18, 19]. However, test code itself is prone to poor design choices that lead to anti-patterns known as *test smells* [14, 21, 27].

Test smells are flawed structural or design choices in test suites that degrade their maintainability, readability, and diagnostic effectiveness, even without preventing test execution [5, 6, 12]. Empirical research confirms that tests containing smells are 47% more likely to be frequently modified, 81% more associated with higher defect occurrence, and 71% more prone to failures in the production code they exercise [18, 19]. Despite these consequences, developers often perceive test smells as purely aesthetic problems, not recognizing them as technical debt that directly impacts reliability [3, 4, 10].

The vast majority of test smell research and tooling focus on mature, class-based ecosystems such as Java (JUnit) and C# (NUnit) [2, 14, 17, 22]. Meanwhile, the Dart language has experienced steady growth worldwide, accumulating over 6,000 repositories on GitHub and showing a 532% increase in adoption since 2019 [20]. This growth translates into a large and expanding base of Dart test code.

Despite this scale, the Dart ecosystem still lacks dedicated systematic studies on test code quality. Developers have no language-specific taxonomies, validated catalogs, or automated tooling to help them identify and fix poor testing practices. The most notable prior effort is *DNose* [23, 24], which introduced automated code-level detection of test smells across 4,154 open-source Dart repositories. Although DNose analyzed **static code structures** in software repositories, it does not provide insight into *how and why* developers introduce test smells, nor does it capture community perceptions and the recurring patterns developers themselves struggle with.

Automated static analysis, however, answers a fundamentally different question from developer experience. Tools such as DNose reveal *what smells exist in code*, but they cannot reveal *which smells cause the most friction in practice*, *how developers introduce them*,

or *why they struggle to recognize and fix them*. Raw occurrence frequency in repositories is a proxy for prevalence, not for developer difficulty: a smell that appears thousands of times in committed code may go unnoticed in daily work, while a less frequent but cognitively costly smell may dominate developer Q&A discussions. Bridging this gap requires a complementary, human-centered perspective on test quality.

This paper adopts a complementary perspective: we investigate test smells **through the lens of the Dart developer community**. We systematically mined Stack Overflow, the largest Q&A platform for software developers, analyzing 307 selected discussions from an initial corpus of 495 posts. Our manual, qualitative analysis, conducted independently by three researchers, identified 432 individual test smell occurrences, revealing the patterns that cause developers the most difficulty in practice.

This study is guided by the following research questions:

RQ1: *What recurring patterns of poor testing practices can be characterized as test smells in the Dart ecosystem, based on developer community discussions?*
RQ2: *What is the relative prevalence of the identified test smells, which ones constitute the "Vital Few" causes of the majority of developer pain points?*
RQ3: *Do Dart-specific language features give rise to novel test smell patterns that are absent from existing catalogs?*

This study provides four main contributions:

(1) **Empirical Characterization**: The quantitative and qualitative mapping of 432 individual test smell occurrences across 307 selected Stack Overflow discussions, documenting how classical anti-patterns manifest in the Dart testing ecosystem.
(2) **Pareto Analysis**: An empirical application of the Pareto Principle showing that only 7 smell types account for approximately 80% of all identified occurrences, the "Vital Few" (answering RQ2). Practically, targeting the top 3 smells alone would address over **53%** of all identified bad practices, providing a concrete, evidence-based priority for tool development and team policies.
(3) **Formalization of ERO**: The conceptualization, definition, and formalization of the *Expected Resolution Omission* (ERO) smell, a Dart-specific asynchronous testing anti-pattern that does not appear in existing Java-based catalogs (answering RQ3).
(4) **Open Dataset**: A publicly available replication package of 307 curated Stack Overflow discussions manually labeled by three independent researchers, enabling replication and future extensions.

The remainder of this paper is structured as follows. Section 2 presents the theoretical background on test smells and related work. Section 3 details the study design, search strategy, and coding process. Section 4 presents the empirical results and Pareto analysis. Section 5 formalizes the ERO smell. Section 6 discusses findings, implications, and threats to validity. Section 7 concludes the paper.

2 Theoretical Background and Related Work

This section establishes the conceptual and empirical foundation for this study. Rather than revisiting generic software testing definitions, we focus directly on the state of the art in test smells, the particularities of the Dart testing infrastructure, and how this study is positioned relative to prior work.

2.1 Test Smells and Their Impact

Test smells are defined as poor implementation choices or structural patterns in test code that indicate design flaws [5, 6, 21]. While these patterns do not prevent test execution or compile-time success, they represent technical debt that degrades the long-term maintainability, readability, and diagnostic value of test suites [3]. First formalized by Van Deursen et al. [21], the study of test smells has evolved into a key topic within software maintenance. Developers frequently introduce smells during test implementation or evolution, often due to tight deadlines, lack of training, or underuse of testing framework primitives, viewing them as cosmetic issues rather than functional risks [4, 10].

The presence of test smells in software projects has severe empirical consequences. Extensive studies reveal that tests containing smells are significantly more prone to frequent changes, harbor a higher density of defects, and are less effective at detecting faults in the production code they exercise [16, 18, 19]. Moreover, certain smells are primary drivers of test flakiness, where tests fail nondeterministically without production code changes, which erodes team confidence in automated CI pipelines [3, 5, 25]. Finally, maintaining test suites with low internal quality is highly expensive; prior reports highlight that major organizations spend millions of dollars annually solely on debugging and adjusting poorly structured test suites [2, 6].

2.2 The Dart Testing Ecosystem: A Comparison with Java

To understand how test smells manifest in the Dart ecosystem, it is useful to contrast its testing infrastructure with mature ecosystems like Java/JUnit, where classical smells were originally defined. Java testing frameworks rely heavily on class-based architectures and metadata annotations (e.g., `@Test`, `@BeforeEach`) to structure and execute tests. In contrast, Dart's testing infrastructure, via `package:test`, implements a functional and block-based paradigm using nested functions like `test()`, `group()`, and `setUp()` [8, 9]. This functional style resembles modern JavaScript frameworks and alters how tests are organized.

Furthermore, Dart's execution model is intrinsically asynchronous, featuring native language support for primitives like `Future`, `Stream`, and the `async`/`await` syntax [7]. Testing asynchronous logic requires explicit synchronization primitives. If developers omit proper synchronization, tests may exit early without running assertions or may produce silent false positives. This execution profile creates conditions for asynchronous test smells such as *Expected Resolution Omission (ERO)*, which do not arise in synchronous, class-based JUnit structures.

2.3 Related Work: Q&A Studies and Code-Level Analysis

The mining of Q&A repositories to understand testing challenges is a recognized empirical software engineering method. In Java and web ecosystems, researchers have studied Stack Overflow discussions to identify common developer misconceptions, API misuse and the prevalence of testing challenges [1, 2, 5, 6].

The pioneering work on test smells in the Dart ecosystem was conducted by Virgínio et al. [23, 24]. They introduced *DNose*, the first automated test smell detector for Dart, and evaluated the prevalence of 14 smell types across 4,154 open-source repositories, identifying over 900,000 instances of smells.

Our study differs from DNose in both the unit of analysis and the research question it addresses. DNose answers: *"which smells are most frequent in Dart code?"* Our study answers: *"which smells cause the most difficulty to developers in practice?"* These are distinct empirical questions with distinct datasets, and their answers need not coincide. More concretely, our work extends the picture drawn by Virgínio et al. in three ways. First, while DNose operates on static code structures, we capture *developer intent and confusion* as expressed in natural-language forum discussions—a dimension that automated AST analysis cannot access. Second, our Pareto analysis reveals a *priority ordering* of smells from the developer's point of view, which may differ substantially from raw occurrence counts in repositories; a smell that appears frequently in committed code is not necessarily the one that causes the most trouble when writing tests. Third, the open-coding phase of our qualitative analysis uncovered the ERO smell, a pattern tied to Dart's asynchronous execution model that does not appear in the 14-smell catalog used by DNose, suggesting the catalog itself can be extended. Table 1 contrasts the two perspectives explicitly.

Table 1: Comparison of two complementary perspectives on Dart test smells: repository-based analysis using DNose [24] and developer-oriented analysis conducted in this work.

Dimension	DNose	This Work
Data source	4,154 GitHub repositories	307 Stack Overflow Q&A posts
Unit of analysis	Source code files	Developer discussions
Research focus	Identifying test smells present in source code	Identifying test smells that concern developers
Method	Automated AST-based analysis	Manual qualitative coding
Main finding	Test smells are widespread across Dart projects	Seven smells account for approximately 80% of reported pain points
Novel contribution	Dart test-smell detection through DNose	Identification of the ERO smell and a developer-prioritized smell set

The key insight from this contrast is that a smell can be statically frequent in repositories yet rarely surface as a developer strugglem and conversely, a smell that causes acute pain in practice may not rank highest by raw occurrence count. Understanding both dimensions is essential: DNose tells tool builders *where* to look; this study tells them *what to prioritize*.

3 Methodology

To address **RQ1**–**RQ3**, we adopt an exploratory grey literature mining approach, treating community-generated Q&A discussions as primary evidence of developer perception and practical difficulty. Unlike repository-based studies that analyze static code structures to identify *what smells exist*, this study captures *which smells developers actively report as problems*—a complementary lens that automated static analysis alone cannot provide.

This section describes the study design following the empirical software engineering guidelines of Wohlin et al. [26]. The study is an **exploratory grey literature mining study**, focused on developer Q&A discussions about testing practices in the Dart ecosystem.

3.1 Search Strategy

We selected Stack Overflow as the primary data source because it is the largest technical Q&A platform for software developers, with over 60 million questions and a well-structured tagging system. We used the search string:

```
[DART] unit test
```

The tag `[DART]` acts as a subject-level filter restricting results to Dart-tagged discussions. The term `unit test` captures discussions involving automated test code.

3.2 Selection Criteria and Filtering Funnel

The post selection followed a structured funnel approach to ensure only relevant, high-quality discussions were analyzed. Figure 1 illustrates the three-stage process: (i) querying Stack Overflow with the `[DART] unit test` search string, (ii) filtering posts by text content to retain only those containing concrete test code, and (iii) applying the codebook to analyze each retained discussion and extract test smell occurrences.

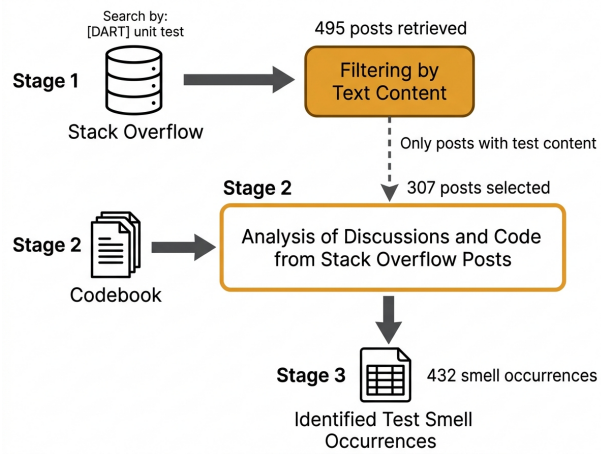

Figure 1: Overview of the study methodology.

- **Initial Collection:** The search retrieved **495 posts** matching the query.
- **Inclusion Criteria:** Posts must (a) contain or reference test code in Dart, and (b) describe a concrete testing problem, error, or discussion about test quality.
- **Exclusion Criteria:** Posts were excluded if they (a) referenced only non-test code, (b) were about test *execution infrastructure* only, or (c) lacked sufficient context to classify a smell.
- **Final Selection:** After applying these criteria, **307 posts** were retained for analysis.

3.3 Qualitative Coding Process

Each of the 307 selected posts was analyzed independently by **three researchers**. The analysis procedure for each post consisted of:

(1) Reading the full discussion (question text, code snippets, answers, and comments).
(2) Inspecting each code snippet against the test smell definitions from the preliminary catalog (21 classical smells adapted from established literature [5, 6, 21]).
(3) Labeling each identified occurrence with the corresponding smell type.
(4) Flagging recurring patterns that did not match existing categories for open-coding of novel smells.

Inter-Rater Agreement: To ensure scientific rigor, we computed **Fleiss' Kappa** (κ) as the inter-rater agreement metric. Prior to coding, a *codebook* defining each smell's operational criteria was distributed to all three raters. Each post was coded independently; disagreements were resolved through a consensus discussion, with a majority vote rule applied to unresolved cases. We achieved a $\kappa = 0.57$, which falls in the *moderate* agreement range on the Landis & Koch scale [11], providing statistical evidence that the classification was not purely subjective. The moderate level (rather than perfect) reflects the inherent ambiguity in classifying multi-label, real-world forum snippets where a single post may contain more than one smell type simultaneously.

3.4 Smell Identification Codebook

To ensure high reliability and consistency during the manual qualitative coding phase, the three researchers utilized a standardized codebook. This codebook establishes the operational definitions and concrete classification criteria for all 21 classical test smells adapted to Dart, translating conceptual definitions into actionable rater guidelines. Table 2 presents the full codebook.

4 Results

This section answers the three research questions by presenting the quantitative and qualitative findings of the grey literature analysis. A total of **432 individual test smell occurrences** were identified across the 307 selected Stack Overflow discussions.

4.1 Frequency and Prevalence (RQ1)

Figure 2 presents the absolute frequency of all identified test smell types. The distribution is markedly uneven, with a small number of smell types dominating the dataset.

Although our codebook mapped 21 classical test smells, the *Constructor Initialization* smell was not identified in any of the analyzed discussions (0 occurrences).

The *Magic Number Test* was the most prevalent issue, accounting for **100 occurrences (23.15%)** of the total. This high incidence reflects the common practice of asserting computed values, counts, or configuration thresholds using raw numeric literals instead of named constants. The second and third most frequent smells were *Unknown Test* (73 occurrences; 16.9%) and *Assertion Roulette* (59 occurrences; 13.66%), indicating that the primary challenges in the Dart developer community relate to test **readability** and **diagnosability**.

Table 3 summarizes the complete frequency ranking of all identified smells.

4.2 Pareto Analysis: The "Vital Few" (RQ2)

To prioritize the identified problems, a Pareto analysis was conducted. Figure 3 presents the cumulative distribution of smell occurrences across all types.

The results confirm the "80/20 rule": only **7 smell types** are responsible for approximately **80% of all identified occurrences**, the *"Vital Few"*:

(1) Magic Number Test (23.15%)
(2) Unknown Test (16.90%)
(3) Assertion Roulette (13.66%)
(4) Redundant Print (8.80%)
(5) Exception Handling (6.71%)
(6) Resource Optimism (5.32%)
(7) Mystery Guest (3.94%)

The remaining 14 smell types (the *"Trivial Many"*) collectively represent only 20% of occurrences. This finding has direct implications for tooling prioritization: detection and refactoring support targeting the top 7 smells would address the overwhelming majority of real-world developer pain points in the Dart community (answering **RQ2**).

Alignment with the DNose Catalog. To contextualize this ranking, it is instructive to compare the Vital Few against the smell catalog supported by DNose [24], the only automated Dart test smell detector to date. Several of the Vital Few smells are amenable to static detection: *Unknown Test* (absence of assertions), *Assertion Roulette* (multiple un-annotated `expect` calls), *Redundant Print* (presence of `print()` statements), *Resource Optimism* (direct access to external resources), and *Mystery Guest* (hidden external dependencies) can, in principle, be identified through AST traversal. By contrast, *Magic Number Test*—the single most prevalent smell in our dataset (23.15%)—requires semantic reasoning about the *meaning* of literal values, a step beyond purely structural analysis. The novel ERO smell, rooted in Dart's asynchronous execution model, is absent from DNose's catalog altogether, highlighting an extension opportunity for future detector work.

Crucially, even where the two studies share smell categories, their questions differ fundamentally. DNose's prevalence rankings are derived from occurrence counts across 4,154 repositories and reveal *what smells exist in code* [24]. Our Pareto ranking reveals *which smells cause the most difficulty in practice*. These two dimensions need not coincide: a smell that is statically widespread in

Table 2: Codebook classical test smells: identification criteria used in the qualitative analysis.

Test Smell	Core Concept	Identification Criteria (Rater Guidelines)
Magic Number Test	Numeric literals without explicit meaning in assertions.	Direct use of numeric literals inside `expect` or test setup; absence of named constants for values whose origin or semantics are non-obvious; repeated literals without explanation.
Unknown Test	A test that validates no observable behavior.	No `expect` or `expectLater` statements; production code is invoked, but its return value or side effects are never asserted; `verify` calls alone do **not** count as assertions.
Assertion Roulette	Multiple undocumented assertions in one test.	Two or more `expect` statements without a `reason` parameter; failure reports cannot pinpoint which specific expectation was violated; `verify` calls are excluded from the assertion count.
Redundant Print	`print()` calls inside test bodies.	One or more `print()` statements in the test body; output is used instead of or alongside assertions to inspect values; console output may obscure CI test reports.
Exception Handling	Improper or absent exception verification.	`try-catch` blocks used without formal assertions on exception type or message; exception-producing calls made without an `expect(..., throwsA(...))` wrapper.
Resource Optimism	Naive reliance on external resources.	Test accesses external files, databases, or network endpoints without mocks; no guard, stub, or `setUp`/`tearDown` ensures resource availability; failures may be intermittent or environment-dependent.
Mystery Guest	Hidden external state or configuration.	Test depends on global variables, environment variables, or external configuration not visible inside the test; dependency is not initialized, passed in, or documented within the test scope.
Dependent Test	Tests coupled by shared mutable state.	Mutable global state is modified in one test and read or modified in another; no `setUp()` or `tearDown()` resets shared state; shared `List` or `Map` objects are mutated across tests without isolation.
Ignored Test	Tests permanently excluded from execution.	Test marked with `skip: true` or `@Skip` without a documented resolution plan; test body is present but never executed in CI; comments such as "TODO" or "temporarily disabled" lack a linked task.
Empty Test	A test with no statements or assertions.	`test()` body is completely empty or contains only comments; no production code is invoked and no `expect` is present.
General Fixture	Over-broad shared test fixtures.	A `setUp` or fixture object initializes data or dependencies not used by all tests in the group; shared objects are not reset or contextualized per test, creating implicit dependencies.
Eager Test	A single test exercising too many behaviors.	Test calls multiple production methods not strictly required for the action under test; many `expect` statements validate unrelated aspects of the SUT; primary behavioral concern is unclear.
Lazy Test	Multiple tests sharing the same production method.	Several test functions invoke the same production method under different conditions; tests cover only the "happy path" and omit edge cases or error scenarios; assertions provide limited behavioral coverage.
Conditional Test Logic	Control-flow constructs inside test bodies.	`if`, `else`, or `switch` statements inside the test body; `for`, `while`, or `forEach` loops containing `expect` calls; assertions depend on runtime-evaluated conditions.
Duplicate Assert	Multiple identical assertions in one test.	The same variable or property is asserted with the same matcher more than once, without an intermediate state change.
Redundant Assertion	Assertions whose result is always predetermined.	Assertion is always true or always false regardless of the SUT (e.g., `expect(true, false)`); final assertion guarantees success or failure independently of execution.
Sensitive Equality	Overly rigid value comparisons.	String comparison via `toString()` without accounting for whitespace or casing; exact numeric equality for floating-point values without tolerance; test fails on trivial formatting changes unrelated to functionality.
Sleepy Test	Fixed-time delays instead of synchronization.	Explicit `sleep` or `Future.delayed` call is used to wait for asynchronous operations; assertion follows a fixed-duration pause rather than directly awaiting completion.
Constructor Initialization	Complex object construction inside tests.	Constructor performs validation, calculations, or method calls beyond field assignment; test setup depends on objects whose constructors produce side effects.
Default Test	Template-generated tests not adapted to real logic.	Test is auto-generated by a framework scaffold; validates example or placeholder behavior rather than domain-specific requirements; test structure closely matches the default framework template.
Verbose Test	Excessively long or complex test bodies.	Test contains redundant variables, assertions, or setup steps; unnecessary manual cleanup such as setting variables to `null`; simpler implementation could achieve the same verification goal.

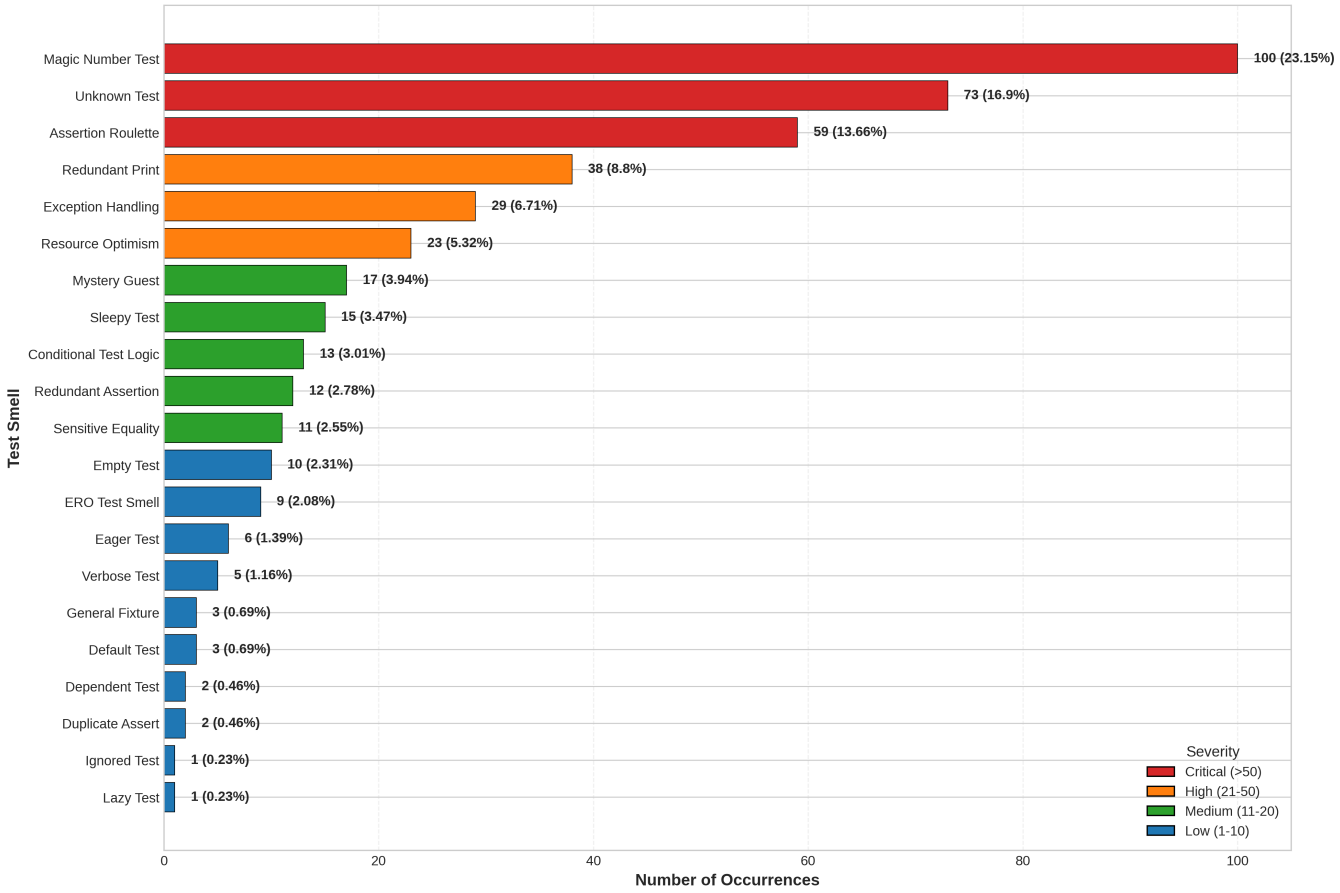

Figure 2: Absolute frequency of test smells identified in 307 Stack Overflow discussions.

Table 3: Test smell occurrences identified in Stack Overflow discussions ($N = 432$).

Test Smell	Count	%
Magic Number Test	100	23.15%
Unknown Test	73	16.90%
Assertion Roulette	59	13.66%
Redundant Print	38	8.80%
Exception Handling	29	6.71%
Resource Optimism	23	5.32%
Mystery Guest	17	3.94%
ERO (novel)	9	2.08%
Other (14 types)	84	19.44%
Total	**432**	**100%**

committed code is not necessarily the one developers struggle with most when writing or debugging tests, and vice versa. Our results therefore complement, rather than replicate, DNose's findings by providing tool builders with an evidence-based *priority ordering* grounded in observed developer pain rather than raw occurrence frequency.

4.3 Code Examples of the Most Prevalent Smells

This subsection illustrates the three most prevalent test smells identified in the Pareto analysis, *Magic Number Test*, *Unknown Test*, and *Assertion Roulette*, and their remediation through standard refactoring practices.

- ***Magic Number Test.*** Numeric literals are used directly in assertions without descriptive constants. Figure 4 shows the anti-pattern and its refactored version using named constants.
- ***Unknown Test.*** Production code is executed without assertions, potentially creating a false sense of security when no exception is thrown. Figure 5 shows the anti-pattern and its refactored version.
- ***Assertion Roulette.*** Multiple assertions lack descriptive messages, making failures difficult to diagnose. Figure 6 shows the problem and its remediation using the `reason` parameter.

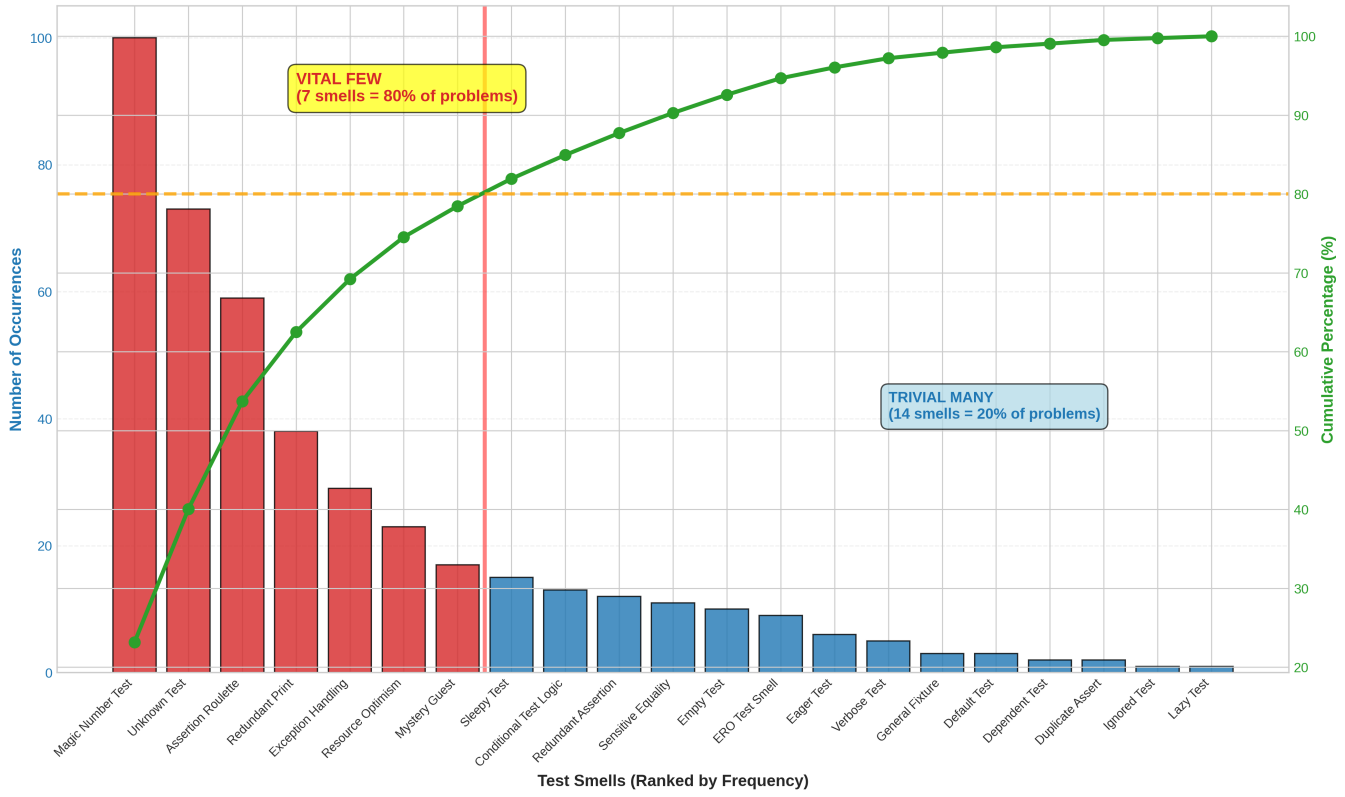

Figure 3: Pareto Analysis: "Vital Few" vs. "Trivial Many" in Dart developer discussions.

Magic Number Test (with smell)

```dart
import 'package:test/test.dart';

void main() {
  test('Discount with magic numbers', () {
    final discount = calculateDiscount(150);
    expect(discount, equals(15));
  });
}

double calculateDiscount(int basePrice) {
  return basePrice * 0.1;
}
```

Magic Number Test (without smell)

```dart
import 'package:test/test.dart';

void main() {
  test('Discount without magic numbers', () {
    const int basePrice = 150;
    const int expectedDiscount = 15;
    final discount = calculateDiscount(basePrice);
    expect(discount, equals(expectedDiscount));
  });
}
double calculateDiscount(int price) {
  const double rate = 0.1;
  return price * rate;
}
```

Figure 4: Magic Number Test: smell (left) vs. refactored (right).

5 Expected Resolution Omission (ERO): A Novel Dart-Specific Smell (RQ3)

Beyond confirming classical smells, the empirical analysis uncovered a recurring pattern specific to Dart's asynchronous execution model, which we designate the **Expected Resolution Omission (ERO)**. This smell was explicitly identified in **9 discussions (2.08%)** from the analyzed dataset.

5.1 Definition

ERO occurs when a test involves operations that return a `Future` or a `Stream`, but the developer fails to properly await their resolution before the test method exits. The smell encompasses five distinct sub-patterns:

(1) **Missing `await`/`expectLater`**: asserting a `Future` value without awaiting its resolution.

Unknown Test (with smell)

```dart
import 'package:test/test.dart';

void main() {
  test('Calculate Discount Test', () {
    var cart = ShoppingCart();
    cart.addItem(Item(price: 100));
    // No assertion made
    var discount = cart.calculateDiscount();
  });
}
```

Unknown Test (without smell)

```dart
import 'package:test/test.dart';

void main() {
  test('Calculate Discount Test', () {
    var cart = ShoppingCart();
    cart.addItem(Item(price: 100));
    var discount = cart.calculateDiscount();
    expect(discount,10,reason:'10%% of the total price');
  });
}
```

Figure 5: Unknown Test: smell (left) vs. refactored (right).

Assertion Roulette (with smell)

```dart
import 'package:test/test.dart';

void main() {
  test('Test with Assertion Roulette'
      ,(){
    final values = [10, 20, 30];
    expect(values.length, 4);
    expect(values[0], 5);
    expect(values.contains(50), true);
  });
}
```

Assertion Roulette (without smell)

```dart
import 'package:test/test.dart';

void main() {
  test('Test without Assertion Roulette', () {
    final values = [10, 20, 30];
    expect(values.length, 4, reason: 'Expected 4 elements');
    expect(values[0], 5, reason: 'First element expected 5');
    expect(values.contains(50), true, reason: 'Expected 50 in
        list');
  });
}
```

Figure 6: Assertion Roulette: smell (left) vs. refactored (right).

(2) **Unnecessary `await`**: awaiting synchronous values, adding confusion without benefit.
(3) **Missing `throwsA`**: testing `Futures` that throw exceptions without the correct exception matcher.
(4) **Incorrect use of `completes`**: using `expect(await future, completes)` instead of `expectLater(future, completes)`.
(5) **Ignoring asynchronous behavior**: testing a `Future` without any synchronization, leading to false positives.

5.2 Why ERO is Dart-Specific

In Java (JUnit), test methods execute synchronously within a managed thread pool, so omitting synchronization primitives typically causes an obvious compilation error or exception. In Dart, however, a test function not marked `async` may exit immediately after its last synchronous instruction. Assertions placed inside `.then()` callbacks or stream listeners may never execute, causing the test to pass falsely (*False Positive*), or to execute after test tear-down, causing *test leak* errors. This phenomenon was repeatedly observed in the analyzed Stack Overflow discussions, where developers reported tests that pass locally but emit errors in CI, or where async assertions silently never fire.

5.3 Code Example

Expected Resolution Omission (with smell)

```dart
test('fetches user', () {
  final result = repository.fetchUser(1);//
      without await
  expect(result,equals(mockUser));
});
```

Expected Resolution Omission (without smell)

```dart
test('fetches user', () async {
  final result = await repository.fetchUser(1);
  expect(result, equals(mockUser));
});
```

5.4 Recommended Fixes

- Always mark test functions as `async` when they involve `Future` or `Stream`.
- Use `await` on any `Future` before asserting its value.
- For exception testing, use `expectLater(future, throwsA(isA<Exception>()))`.
- For completion testing, use `expectLater(future, completes)`.

- Avoid placing assertions inside `.then()` callbacks without returning the `Future` to the test runner.

The ERO smell directly answers **RQ3**: Dart-specific language features, principally its native `Future`/`Stream` model and the absence of implicit test-thread synchronization, give rise to a category of async test smells that do not appear in Java-based test smell catalogs.

6 Discussion and Threats to Validity

6.1 Answers to Research Questions

RQ1: The analysis of 307 Stack Overflow discussions identified **432 individual test smell occurrences** distributed across 20 classical smell types and 1 novel type (ERO). The most prevalent patterns were *Magic Number Test* (100 occurrences; 23.15%), *Unknown Test* (73; 16.9%), and *Assertion Roulette* (59; 13.66%). These three smells alone account for more than half of all identified occurrences (53.81%), revealing that the Dart community's primary struggles are concentrated in test **readability**, **diagnosability**, and the management of literal values in UI-intensive test scenarios.

RQ2: The Pareto analysis confirmed the 80/20 rule: only **7 smell types** account for approximately **80% of all occurrences** (the *Vital Few*): Magic Number Test, Unknown Test, Assertion Roulette, Redundant Print, Exception Handling, Resource Optimism, and Mystery Guest. The remaining 14 types (the *Trivial Many*) collectively represent only 20% of occurrences. This concentration provides a concrete, evidence-based priority list: detection and refactoring support targeting these seven smells would address the overwhelming majority of real-world developer pain points in the Dart community. For tool builders, this ranking directly guides implementation priorities.

RQ3: Yes. The open-coding phase revealed the **Expected Resolution Omission (ERO)**, a novel smell rooted in Dart's native asynchronous execution model. ERO occurs when tests involving `Future` or `Stream` operations omit proper synchronization (`expectLater` or `await`), causing tests to exit before assertions run, producing silent false positives. This pattern was explicitly identified in 9 discussions (2.08%) and does not appear in any Java-based catalog, as JUnit tests execute synchronously on managed threads. ERO is a direct consequence of Dart's cooperative, event-loop scheduling model and represents a structurally new category of test smell that requires Dart-specific detection rules.

6.2 Implications for Developers

The Pareto analysis provides an actionable priority list for the Dart/Flutter community. Eliminating the top three smells (*Magic Number*, *Unknown Test*, and *Assertion Roulette*) would address more than **53%** of identified bad practices. Development teams should prioritize: (i) named constants for assertion values; (ii) descriptive test names; and (iii) `reason` arguments in `expect` calls to improve failure diagnosis.

The prevalence of *Magic Number Test* (23.15%) reflects the use of raw numeric literals in assertions, which can make tests brittle and increase maintenance costs when production thresholds change.

ERO has a particularly high safety impact because it can produce *perpetually green* tests that mask failures in asynchronous logic. False confidence in the test suite may allow critical defects to reach production, making ERO a high-priority target for developer education and linting.

ERO is rooted in Dart's single-threaded, event-loop execution model. When asynchronous operations returning a `Future` or `Stream` are not properly synchronized with `await`, the test may reach its final brace before the operation completes. The test runner can therefore mark it as passed while assertions in subsequent callbacks, such as `.then()`, execute later or fail to execute as intended. This behavior can allow functional errors to pass through CI/CD undetected, posing a threat to software quality.

6.3 Threats to Validity

Construct Validity. The primary threat is the subjectivity of the manual coding process: two researchers may classify the same code snippet under different smell categories. We mitigated this by: (1) distributing a structured *codebook* with operational definitions for each smell before coding began; (2) requiring three independent coders; and (3) measuring inter-rater agreement with **Fleiss' Kappa ($\kappa = 0.57$)**, which indicates *moderate agreement* on the Landis & Koch scale [11]. The moderate value (as opposed to substantial or near-perfect) is expected given the multi-label nature of the task: a single forum post frequently manifests more than one smell simultaneously, increasing classification ambiguity at the boundaries between categories. Remaining disagreements were resolved by consensus, with a majority-vote rule for unresolved cases.

Internal Validity. Researchers may exhibit *confirmation bias* when searching for pre-defined smell categories, potentially overlooking novel patterns or misclassifying ambiguous snippets. This risk was mitigated by including an open-coding stage in the protocol, in which patterns not matching the predefined codebook were explicitly flagged and resolved by consensus. The discovery of ERO itself is an outcome of this open-coding mechanism.

External Validity. The results may not generalize beyond Stack Overflow or the specific search string used. First, because Stack Overflow posts are primarily in English, challenges faced by non-English-speaking Dart communities may be under-represented. Second, the search string `[DART] unit test` is oriented toward logical unit tests and may under-sample Flutter widget tests (`testWidgets`), potentially underestimating UI-specific smells such as Widget Setup and Residual State. Future work should include `widget test` and `flutter test` terms. Third, Stack Overflow users may not represent the broader Dart developer population, as developers differ in experience, incentives, and problem visibility. Developers who do not post publicly or work in private codebases remain outside the sample; thus, the reported rankings reflect community-visible pain points rather than overall Dart testing activity.

Reliability. To ensure replicability, the complete dataset of 307 labeled Stack Overflow discussions is made publicly available. Other researchers can independently re-apply the codebook and verify the classification decisions.

7 Conclusion and Future Work

This paper presented an empirical grey literature mining study on test smells in the Dart developer community. Analyzing 307 Stack

Overflow discussions from an initial pool of 495 posts, we identified **432 test smell occurrences** and addressed three research questions.

For **RQ1**, the most prevalent smells were *Magic Number Test* (23.15%), *Unknown Test* (16.9%), and *Assertion Roulette* (13.66%), indicating that developers primarily struggle with test clarity, diagnosability, and literal values in UI-intensive scenarios.

For **RQ2**, Pareto analysis showed that **7 smell types** account for **approximately 80% of occurrences**, providing an evidence-based priority list for tool development, education, detection, and refactoring efforts.

For **RQ3**, we uncovered the **Expected Resolution Omission (ERO)**, a novel smell rooted in Dart's asynchronous execution model. ERO occurs when tests involving `Future` or `Stream` operations fail to synchronize properly with `await` or `expectLater`, potentially causing false positives or silent assertion non-execution. This smell is not represented in existing Java-based catalogs and reflects Dart's single-threaded, event-loop model and lack of implicit test-level asynchronous synchronization.

Future Work. The findings motivate three extensions: (i) a Dart-specific static analysis linter targeting the "Vital Few" smells; (ii) formal ERO detection rules for tools such as *DNose* [23]; and (iii) broader search strategies incorporating Dart/Flutter terms such as `widget test` and `testWidgets`.

Artifact Availability

The dataset and scripts (`kappa.py`, `pareto.py`) are available at *https://figshare.com/s/f719a336f1be519edaf8*.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001; Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB), grants PIE0002/2022 and BOL0997/2025; and CNPq, grants 315840/2023-4 and 403361/2023-0.

References

[1] Mumtahina Ahmed, Md Nahidul Islam Opu, Chanchal Roy, Sujana Islam Suhi, and Shaiful Chowdhury. 2026. Exploring challenges in test mocking: Developer questions and insights from StackOverflow. *Journal of Systems and Software* 235 (2026), 112748. doi:10.1016/j.jss.2025.112748

[2] 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*. 170–180. doi:10.1145/3463274.3463335

[3] 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. doi:10.1007/s10664-014-9313-0

[4] Humberto Damasceno, Carla Bezerra, Denivan Campos, Ivan Machado, and Emanuel Coutinho. 2023. Test smell refactoring revisited: What can internal quality attributes and developers' experience tell us? *Journal of Software Engineering Research and Development* 11, 1 (2023), 13–1. doi:10.5753/jserd.2023.3195

[5] 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. doi:10.1016/j.jss.2017.12.013

[6] Vahid Garousi, Baris Kucuk, and Michael Felderer. 2018. What we know about smells in software test code. *IEEE Software* 36, 3 (2018), 61–73. doi:10.1109/MS.2018.2875843

[7] Google. 2025. Dart: Language Tour. https://dart.dev/language. Acesso em 13 jul. 2025.

[8] Google. 2025. flutter_test API Reference. https://api.flutter.dev/flutter/flutter_test/. Acesso em 13 jul. 2025.

[9] Google. 2025. package:test. https://pub.dev/packages/test. Acesso em 13 jul. 2025.

[10] Nildo Silva Junior, Luana Martins, Larissa Rocha, Heitor Costa, and Ivan Machado. 2021. How are test smells treated in the wild? A tale of two empirical studies. *Journal of Software Engineering Research and Development* 9 (2021), 9–1. doi:10.5753/jserd.2021.1802

[11] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174. doi:10.2307/2529310

[12] Luana Martins, Heitor Costa, and Ivan Machado. 2024. On the diffusion of test smells and their relationship with test code quality of Java projects. *Journal of Software: Evolution and Process* 36, 4 (2024), e2532. doi:10.1002/smr.2532

[13] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. 2022. Test smells 20 years later: detectability, validity, and reliability. *Empirical Softw. Engg.* 27, 7 (Dec. 2022), 40 pages. doi:10.1007/s10664-022-10207-5

[14] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1650–1654. doi:10.1145/3368089.3417921

[15] Valeria Pontillo, Luana Martins, Ivan Machado, Fabio Palomba, and Filomena Ferrucci. 2025. An empirical investigation into the capabilities of anomaly detection approaches for test smell detection. *Journal of Systems and Software* 222 (2025), 112320. doi:10.1016/j.jss.2024.112320

[16] Railana Santana, Luana Martins, Tássio Virgínio, Larissa Rocha, Heitor Costa, and Ivan Machado. 2024. An empirical evaluation of RAIDE: A semi-automated approach for test smells detection and refactoring. *Science of Computer Programming* 231 (2024), 103013. doi:10.1016/j.scico.2023.103013

[17] Tushar Sharma, Pratibha Mishra, and Rohit Tiwari. 2016. Designite: a software design quality assessment tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities* (Austin, Texas) *(BRIDGE '16)*. Association for Computing Machinery, New York, NY, USA, 1–4. doi:10.1145/2896935.2896938

[18] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the relation of test smells to software code quality. In *2018 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 1–12. doi:10.1109/ICSME.2018.00010

[19] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 4–15. doi:10.1145/2970276.2970340

[20] Lionel Sujay Vailshery. 2025. Cross-platform mobile frameworks used by developers globally 2019-2023. https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/ Accessed: 2025-07-14.

[21] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, Vol. 92. Citeseer, 95.

[22] Tássio Virgínio, Luana Martins, Railana Santana, Adriana Cruz, Larissa Rocha, Heitor Costa, and Ivan Machado. 2021. On the test smells detection: an empirical study on the jnose test accuracy. *Journal of Software Engineering Research and Development* 9 (2021), 8–1. doi:10.5753/jserd.2021.1893

[23] Tássio Virgínio, Marcio Ribeiro, and Ivan Machado. 2025. DNose: Dart Test Smell Detector. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 976–982. doi:10.5753/sbes.2025.11564

[24] Tássio Virgínio, Márcio Ribeiro, and Ivan Machado. 2025. On the prevalence of test smells in mobile development. In *Anais do X Simpósio Brasileiro de Testes de Software Sistemático e Automatizado* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 84–93. doi:10.5753/sast.2025.14327

[25] Jing Wang, Weixi Zhang, Weiwei Wang, Ruilian Zhao, and Ying Shang. 2025. Predicting the Root Cause of Flaky Tests Based on Test Smells. In *2025 IEEE/ACM 22nd International Conference on Software and Systems Reuse (ICSR)* (Ottawa, ON, Canada). IEEE Press, 84–94. doi:10.1109/ICSR66718.2025.00015

[26] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer. doi:10.1007/978-3-642-29044-2

[27] Min Zhang, Tracy Hall, and Nathan Baddoo. 2011. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: research and practice* 23, 3 (2011), 179–202. doi:10.1002/smr.521