On the prevalence of test smells in mobile development

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

Márcio Ribeiro
Federal University of Alagoas (UFAL)
Maceió, Brazil
marcio@ic.ufal.br

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

ABSTRACT

Objective: This study explores the quality of tests in Dart, the main language for mobile application development with the Flutter framework. Methods: The study begins by using the DNose tool, used to detect 14 types of test smells in code written in the Dart language. Next, we evaluate the tool's precision, accuracy, recall, and F1-score. Using this tool, we conduct a detailed analysis of tests in open-source projects extracted from the language's central repository. Results: The study starts with a dataset of 5,410 Dart-language projects, from which we were able to clone 4,154 repositories after processing. Based on the cloned projects, we generated a dataset containing 907,566 occurrences of test smells. Through our analysis, we characterized the specific types of test smells most frequently encountered and identified their causes. We observed the presence of test smells in 74% of test files. Another noticeable characteristic among the analyzed projects was the scarcity of tests, with 1,873 projects having one or no tests, which led us to expand the number of analyzed projects to a broader base. Conclusion: This research makes a significant contribution by providing insights into the quality of tests in projects from Dart's official repository, as well as by offering an open-source tool for detecting 14 types of test smells.

KEYWORDS

Tests, Test Smells, Dart, Flutter

1 Introduction

Smartphones are the primary means of accessing information, used for reading news, managing bank accounts, performing transactions, among other activities [7]. All these services are accessed through mobile applications. Currently, the leading cross-platform mobile application development framework is Flutter[1] [37].

In this study, we focus on the quality of systems developed using the Dart programming language, specifically mobile systems. As mentioned, Dart is nowadays of utmost importance in mobile application cross-platform development, being the most widely used. This study examined the quality of tests conducted in Dart projects. To assess test quality, we utilized the concept of test smells, based on previous studies that employed test smells to evaluate the quality of test code [29, 31, 32, 40].

Test Smells are anti-patterns introduced in test code that affect test quality. There are studies on test smells in the Java programming language [24, 29, 31, 32, 40], in Python [12, 43], in JavaScript [10, 15], among others. To assist in identifying test smells, several tools have been developed for some languages [3, 12, 15, 24, 40, 43].

Using the DNose tool[2], we initially conducted a study on the tool's accuracy, and subsequently performed an analysis of open-source projects written in Dart, examining the distribution of test smells, which ones occur most frequently, their causes, and their co-occurrences. This study provides an overview of the current quality of projects developed in the Dart language, crucial information for mobile application development.

As a result, we generated an extensive dataset with over 907,566 occurrences of test smells, identifying that 74% of the analyzed tests exhibit at least one of these issues. This analysis provides a comprehensive view of the current quality of projects written in Dart, offering essential insights to improve the language's ecosystem and provide the Dart community with a clear perspective on the quality level of projects developed with it.

2 Background

In this section, we present all the concepts used in the development of this paper and necessary for its understanding.

2.1 Test Smells

Test smells were defined in 2001 by Deursen et al. [38] and are now well-established as one of the main approaches to assess the quality of software tests [4, 29, 32, 33, 36]. Today, there is a comprehensive list of test smells identified across different programming languages [13]. In addition to the smells already discussed in the literature, we identified a new type specific to the Dart and JavaScript languages: "Test Without Description". In the following section, we present the test smells detected by the DNose tool.

2.1.1 ***Assertion Roulette (AR).*** Happens when a test contains more than one undocumented "expect" statement. Multiple assertion statements in a test without a descriptive message impact the maintainability, readability, and comprehensibility.

Detection: It is detected when more than one "expect" does not have a "reason". As we can see in Listing 1, in this example, we have a test smell "Assertion Roulette" on line 4, as the "expect" on line 2 includes the "reason", while the one on line 3 refers to the test's own description.

```
1 test("AssertionRoulet", () {
2         expect(1 + 2, 3, reason: "Explanation of why
              respect");
3         expect(1 + 2, 3);
4         expect(1 + 2, 3);
5 });
```

Listing 1: Example of an Assertion Roulet in Dart

[1] https://flutter.dev/

[2] https://dnose-ts.github.io/

2.1.2 ***Conditional Test Logic (CTL).*** When conditions are introduced within the test, they can change its behavior and expected outcomes.

Detection: A code block with conditional statements. We have an example in Listing 2.

```
test("Checking method", () {
  List<int> list = [1,2,3];
        list.forEach((number){
          expect(productionMethod(number),number*2);
  }
  );
});
```

Listing 2: Example of an Conditional Test Logic in Dart

2.1.3 ***Duplicate Assert (DA).*** Test smells occur when a test evaluates the same function multiple times.

Detection: A line with "expect" whose parameters equal the other "expect" inside of same test. In Listing 3 we have an example of DA.

```
test("Duplicate Assert", () {
  expect(productionMethod(1), 3, reason: "Checking the
      value");
  expect(productionMethod(2), 3, reason: "Checking the
      value");
  expect(productionMethod(3), 3, reason: "Checking the
      value");
});
```

Listing 3: Example of an Duplicate Assert in Dart

2.1.4 ***Empty Test (ET).*** A test without any executable statements is considered empty.

Detection: A test lacking any executable statements. In Listing 4 we have an example of ET.

```
test("EmptyFixture", () => {});
test("EmptyFixture", () => {     });
test("EmptyFixture", () {});
test("EmptyFixture", () {
  //comments
});
```

Listing 4: Example of an Empty Test in Dart

2.1.5 ***Exception Handling (EH).*** A test smell occurs when the pass or fail outcome of a test depends on the production method throwing an exception.

Detection: A block that includes a throw statement or a catch clause. We can check an example with the call of a function that returns an exception in Listing 5.

```
void testFunction() {
    throw Exception("Error");
}
test("Exception Handling", () {
    try {
        testFunction();
    } catch (e) {
      expect(e.toString(), Exception("Error").toString
          ());
    }
});
```

Listing 5: Example of an Exception Handling in Dart

2.1.6 ***Ignored Test (IT).*** The Dart testing library provides developers with the ability to suppress the execution of tests. However, these ignored tests result in overhead, as they add unnecessary compilation time overhead and increase code complexity and comprehension difficulty.

Detection: A test that contains the parameter "skip:true", as we can see an example in the list below.

```
test("Some Other Test", () async {
  //Test Logic
  expect(1 + 2, 3);
}, skip: true);
```

Listing 6: Example of an Ignored Test in Dart

2.1.7 ***Magic Number (MN).*** A test smell arises from the use of undocumented and unexplained numeric literals as parameters or assigned to identifiers within a test. These "magic numbers" fail to convey the meaning or purpose of the value, making the code harder to understand.

Detection: A line with an "expect" that uses a numeric literal as its argument, as we can see an example in listing 7.

```
test("Magic Number", () => {expect(1 + 2, 3)});
test("Magic Number", () => {expect("3", "3")});
test("Magic Number", () =>{
    for (int i = 0; i < 10; i++)
        {expect((1 + 1), 2, reason: "Checking the value
            ")}
    });
```

Listing 7: Example of an Magic Number in Dart

2.1.8 ***Print Statement Fixture (PSF).*** Print statements in unit tests are unnecessary because unit tests are typically executed as part of an automated process with minimal or no human involvement. Developers might include print statements for debugging or traceability purposes, but these are often left behind unintentionally.

Detection: A line that invokes either the "print()" or "stdout.write()", as we can see an example in listing 8.

```
test("PrintStatmentFixture", () {
        //instructions and checks
        print("printing values found")
});
test("PrintStatmentFixture", () {
        //instructions and checks
        stdout.write("printing values found")
});
```

Listing 8: Example of an Print Statement Fixture in Dart

2.1.9 ***Resource Optimism (RO).*** A test smell occurs when a test optimistically assumes that an external resource (e.g., a database or an image) required by the test is already available.

Detection: A test that uses an external resource without checking the state of the object, as we can see an example in listing 9.

```
test("DetectorResourceOptimism", () {
  var file = File('file.txt');
});
```

Listing 9: Example of an Resource Optimism in Dart

2.1.10 ***Sensitive Equality (SE)***. It happens when "toString" is utilized within a test. In such cases, the tests assess objects by invoking the object's default "toString()" and matching the resulting output with a predefined string. Modifications to the "toString()" implementation can cause the test to fail.

Detection: A line that invokes the "toString()" and you "expect" function, as we can see an example in listing 10.

```
test("Sensitive Equality", () {
    String value = "value";
    expect("value", value.toString());
});
```

Listing 10: Example of an Sensitive Equality in Dart

2.1.11 ***Sleepy Fixture (SF)***. Explicitly putting a thread on hold can result in unpredictable outcomes, as the time it takes to process a task may vary between different devices. This test smell is introduced by developers when they need to pause the execution of a test for a set period (e.g., simulating an external event) and then continue with the execution.

Detection: A line that invokes the "sleep()" function, as we can see an example in listing 11.

```
test("SleepyFixture", () => {sleep(Duration(seconds:
     oneSecond))});
test("SleepyFixture",
    () async => {await Future.delayed(Duration(seconds:
          1))});
```

Listing 11: Example of an Sleepy Fixture in Dart

2.1.12 ***Test Without Description (TWD)***. A new test smell arises when a test function lacks a description, making it harder to read and understand the test code.

Detection: A test that does not have its description, as we can see an example in listing 12.

```
test("", () => {});
test(" ``, () {print("some value");});
test("   ``, () => {if (true) {}});
```

Listing 12: Example of an Test Without Description in Dart

2.1.13 ***Unknown Test (UT)***. When a test lacks an "expect", "verify()", or "assertion()" check.

Detection: A test that does not contain an assertion statement "expect", as we can see an example in listing 13.

```
test("UnknownTest", () {
  print("some value");
});
test("UnknownTest", () {
  print("some value");
  if(true){
    print("some value");
  }
});
```

Listing 13: Example of an Unknown Test in Dart

2.1.14 ***Verbose Test (VT)***. When a test has more than 30 lines, it can make the code harder to understand and may indicate that the test has multiple responsibilities, impacting its maintainability. Detection: A test with more than 30 lines without counting non executable statements and annotations.

3 Empirical Evaluation

This empirical evaluation aims to investigate the accuracy of DNose in detecting test smells in the Dart language. We designed the empirical study in four stages, as shown in Figure 1: **(i) Dataset Selection**, in which we randomly define the tests to be analyzed from real projects; **(ii) Oracle Definition**, in which we manually detect instances of test smells; **(iii) Data Collection**, where we apply DNose to collect instances of test smells; and **(iv) Data Analysis**, in which we analyze the collected data to investigate our objectives.

To construct the dataset, we randomly selected repositories from our dataset, all from the Pub.dev[3] website, which is the official repository for the Dart language. We identified the test cases and randomly selected 140 tests from them. We then followed the definition to manually detect test smells.

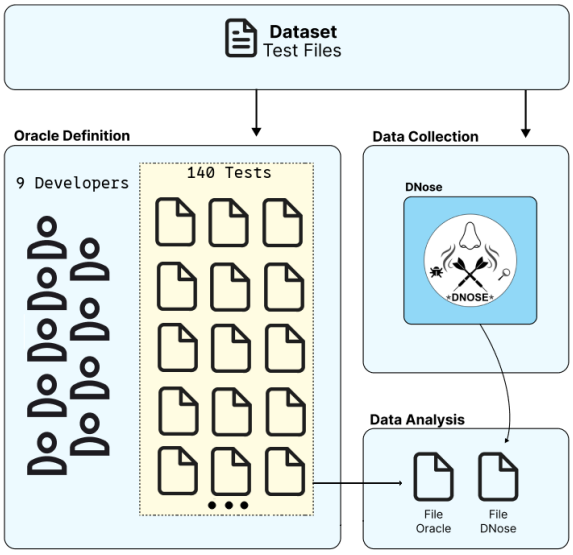

Figure 1: Empirical Accuracy Study

3.1 Oracle Definition

To manually detect instances of test smells, we followed a fully crossed design to assign coders to subjects, meaning that all subjects were analyzed by all subsets of developers [14]. The study involved 140 test cases and 9 developers with varying levels of experience, ranging from six months to four years. Additionally, their experience with Dart programming ranged from six months to two years, including unit test development.

From the selected test cases, we developed forms with one question per test, where the occurrence of each type of test smell was verified. The developer reviewed the code and answered whether the test smell was present or not. We used the test smell descriptions provided in Section 2.1.

[3]https://pub.dev/

3.2 Data Collection

We used DNose to detect test smells. After running the tool, the output file contained the detected test smells for each test. The automated detection with DNose was instantaneous for the selected dataset. A user-friendly graphical interface makes this process easier.

3.3 Data Analysis

We used the oracle to calculate the accuracy of DNose in relation to manual analysis. DNose detects all instances of a test smell with its exact location (line, module, method, or class). We compared the accuracy of DNose and manual analysis considering a test-level granularity. For example, when evaluating the test-level granularity, we can detect the AR test smell; thus, we collected data at the test level to analyze them both manually and automatically. Our goal is to show DNose's accuracy in pinpointing the location of test smells at the test level. Therefore, we provide the accuracy value at the test level, in terms of precision and recall.

3.4 DNose and Manual Analysis Comparison

This section presents the results of our empirical study on the accuracy of DNose. The data for replication purposes are available online [42]. Table 1 reports the data obtained with accuracy, precision, recall, and F1-score values for detecting test smells with DNose and manual analysis. This comparison considered the test-level granularity for detecting test smells.

Table 1: Confusion matrix with precision, accuracy, recall and f1-core data from the DNose tool.

TS	Accuracy(%)	Precision(%)	Recall(%)	F1-Score(%)
AR	100	100	100	100
CTL	80	80	80	80
DA	100	100	100	100
ET	100	100	100	100
EH	90	100	80	89
IT	90	100	80	89
MN	100	100	100	100
PSF	100	100	100	100
RO	100	100	100	100
SE	100	100	100	100
SF	100	100	100	100
TWD	100	100	100	100
UT	100	100	100	100
VT	100	100	100	100

4 Experiment Planning

This section outlines the approach we used to structure our study, including the research questions and study design.

4.1 Research Questions

This investigation is guided by a main research question (RQ): *What is the state of quality in projects developed in the Dart language?* To further explore this analysis, the main question was broken down into the following sub-questions:

- RQ$_1$ *What is the distribution of test smells in Dart language projects?* This question aims to understand how test smells are distributed across Dart projects, which is essential to determine whether they are concentrated in certain types of projects. This initial analysis allows us to identify general quality patterns in projects developed in this language, providing a broad perspective on the adoption of best practices within the Dart ecosystem.
- RQ$_2$ *What is the distribution of test smells in the tests of Dart language projects?* Testing is a crucial part of software development, and analyzing the distribution of test smells specifically in tests helps assess the quality of the test code. This question provides detailed information about areas that need improvement and points to practices that may be compromising the effectiveness of tests.
- RQ$_3$ *Which test smells are the most frequently found?* The purpose of this question is to identify the most common test smells and prioritize mitigation and refactoring efforts. It also helps the development community understand which problematic practices are most prevalent and which specific aspects of testing should receive the most attention during code writing or review.
- RQ$_4$ *What are the most relevant co-occurrences and their motivations?* This question determines the most relevant co-occurrences between test smells and helps understand how they interact and potentially amplify test quality issues. This analysis can reveal underlying causes for combined smell patterns and provide valuable insights into how coding practices can be improved in an integrated manner.

4.2 Design

In Figure 2, we present an overview of the processes involved in the experimental study. First, 5,410 open-source projects were selected from the Hugging Face dataset[4], all sourced from Pub.dev[5] and stored on GitHub. The dataset includes the following information: title, title link, likes, pub points, popularity, description, latest version, latest version release date, provided license, beta version status, SDK, platform, documentation link, verified publisher, dependencies, and GitHub link. The selection of the dataset was based on the number of stars the projects had.

5 Results

We used the dataset from Huggingface with a list of over 5,000 (5,410) projects in the Dart language, which were obtained from the Pub.dev repository, the official site for centralizing libraries written in the Dart language and the Flutter framework.

In Figure 3, we describe the process of filtering the projects until the final count. Initially, we had access to the list of projects provided by Hugging Face. The second step of our experiment was to clone each repository locally. Out of the total 5,410 projects, we managed to clone 4,154. During this stage, we encountered access issues with some repositories, non-existent repositories, name changes, as well as 2 or more projects located in the same repository, among other

[4]https://huggingface.co/datasets/deepklarity/top-flutter-packages
[5]https://pub.dev/

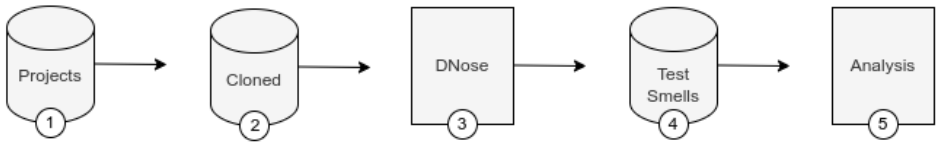

Figure 2: DNose Design: 1 - List of selected projects, 2 - Number of cloned projects, 3 - Use of DNose for test smells detection, 4 - Dataset with the list of selected test smells, 5 - Data analysis.

problems. The third step was to run the DNose[6] on all the projects, which resulted in 2,532 projects with at least one test smell detected, meaning approximately 60% of the cloned projects had at least one occurrence of a test smell.

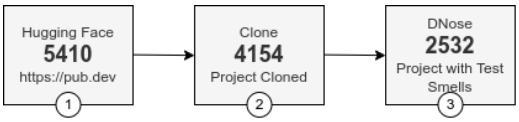

Figure 3: Project Filtering Workflow: The initial dataset consisted of 5,410 projects, but only 4,154 projects could be cloned, and 2,532 contained test smells.

We found 14,917 test files "infected" with test smells, out of a total of 20,006 test files from the cloned projects, meaning that 74% of the test files in the projects have at least one test smell.

5.1 Answering the Research Questions

We summarized the answers for each RQ as follows:

- RQ_1 What is the distribution of Test Smells in Dart language projects? **We found a distribution of 60% of test smells in the analyzed projects;**
- RQ_2 What is the distribution of Test Smells in the tests of Dart language projects? **We found a distribution of 74% of test smells in the analyzed tests;**
- RQ_3 Which Test Smells are the most frequently found? **The most frequently found Test Smells were Assertion Roulette, Duplicate Assert and Magic Number, with the following values: 200,403, 305,384 and 353,684, respectively.**
- RQ_4 What are the most relevant co-occurrences and their motivations? **The most relevant co-occurrences of "test smells" are discussed in Section 6.1, focusing on correlations above 80%, revealing significant relationships between them. For example, the strong correlation between "Assertion Roulette" and "Duplicate Assert" (0.86) indicates that tests with multiple undocumented assertions often contain redundant checks. Similarly, "Magic Number" and "Ignored Test" (0.85) suggest that the lack of clarity in numeric literals may lead to test neglect. These patterns highlight the need for targeted refactoring to mitigate negative impacts on code quality and maintainability.**

[6]https://anonymous/

6 ANALYSIS AND DISCUSSION

Statistical analysis was conducted on a robust dataset of 4,154 Dart projects, extracted from the official Pub.dev repository. This yielded over 20,000 test files and a remarkable 907,566 occurrences of test smells.

In Table 2, we present the statistics obtained from our dataset. This table includes the Test Smell Name, Average of test smells found per file, Standard Deviation, Median, Square Mean, Max, Min, Sum, Center, and Squares Sum.

Despite the substantial volume and diversity of the dataset, we observed significant variability in the distribution of certain test smells. Notably, Magic Number, Duplicate Assert, and Assertion Roulette showed standard deviations considerably higher than their respective averages. This indicates that these smells aren't uniformly distributed across projects but are instead concentrated in a few highly problematic files.

This concentration can be attributed to differences in project maturity levels and a lack of consolidated testing standards within the Dart community. Therefore, our data isn't flawed; rather, it accurately reflects the current state of testing practices in this ecosystem. This asymmetry, far from being statistical noise, is a crucial finding. It highlights areas of low quality and presents clear opportunities for improvement through refactoring, linters, or educational initiatives aimed at enhancing test maintainability and readability.

We found over 907,566 test smells. As shown in Figure 4, **from these data we can see that the three most common test smells found in Dart projects** were assertion roulette, duplicate assert, and magic number. The assertion roulette was found, on average, 10 times, while the duplicate assert was found an average of 15 times, and the magic number appeared 17 times, with medians of 1, 2, and 0, respectively.

Our statistical analysis, drawing from a robust dataset of 4,154 Dart projects on Pub.dev, unearthed over 900,000 test smell occurrences across more than 20,000 test files. While this sheer volume and diversity are noteworthy, a closer look reveals significant variability in the distribution of certain test smells. As presented in Table 2, our dataset provides statistics including the Test Smell Name, Average occurrences per file, Standard Deviation, Median, Square Mean, Max, Min, Sum, Center, and Squares Sum.

The Magic Number test smell stands out as the most prevalent, with a staggering 353,684 occurrences and a peak of 6,413 instances in a single project, suggesting its widespread and recurring nature in Dart projects. Our analysis showed an average of 17.67 occurrences per file, but with a median of zero and a high standard deviation of 105.37. This highly asymmetrical distribution indicates that while most files have few or no magic numbers, a select few

Table 2: Test Smells Statistics: Generated by the DNose tool.

Test Smell	Mean	Standard Deviation	Median	Mean Square	Max	Min	Sum	Center	Sum of Squares
Assertion Roulette	10.0171	44.6755	1	2096.2395	2611	0	200403	1	41937367
Conditional Test Logic	0.4615	3.5592	0	12.8810	362	0	9232	0	257698
Duplicate Assert	15.2646	62.5343	2	4143.5474	3253	0	305384	2	82895810
Empty Test	0.0209	0.1622	0	0.0267	6	0	419	0	535
Exception Handling	0.1704	1.7444	0	3.0720	107	0	3410	0	61458
Ignored Test	0.0118	0.3151	0	0.0994	30	0	237	0	1989
Magic Number	17.6789	105.3744	0	11416.3177	6413	0	353684	0	228394852
Print Statement Fixture	0.1393	3.4987	0	12.2604	405	0	2786	0	245282
Resource Optimism	0.1409	1.0499	0	1.1221	44	0	2819	0	22449
Sensitive Equality	0.2706	2.2917	0	5.3249	113	0	5414	0	106530
Sleepy Fixture	0.0928	1.3192	0	1.7490	102	0	1857	0	34991
Test Without Description	0.0044	0.2466	0	0.0608	31	0	89	0	1217
Unknown Test	0.6227	3.9897	0	16.3051	228	0	12458	0	326200
Verbose Test	0.4686	3.1454	0	10.1130	309	0	9374	0	202320

are heavily burdened with thousands of them. This extreme distribution points to a problematic behavior: the systematic disregard for documenting numeric literals in certain contexts. It's highly likely that developers are copying and pasting tests or code snippets without proper parameterization, reusing "magic" numbers without explaining their purpose. This significantly hinders test readability and maintainability, especially when these numbers represent isolated IDs, validation limits, or status codes.

This situation can be attributed to several factors. First, the absence of coding standards for tests within the relatively young Dart/Flutter ecosystem leads to less formal enforcement of good testing practices, particularly concerning named constants or reusable fixtures. Second, the low maturity of many analyzed projects in our sample, which included a broad range of open-source initiatives, often exhibit simple, undertested, or poorly structured architectures, directly impacting test quality. Finally, the limited tool and IDE support for detecting test smells or suggesting best practices in Dart/Flutter environments, compared to more mature languages like Java, also contributes to the problem.

Following Magic Number, Duplicate Assert is the second most frequent test smell, totaling 305,384 occurrences. This highlights opportunities for refactoring, such as splitting tests into multiple, more focused ones, which would greatly improve test readability. Assertion Roulette also appears frequently, with 200,403 occurrences, indicating a tendency towards poorly structured tests that complicate failure identification.

Despite these challenges, the high frequency of these test smells, particularly Magic Number, is a valuable finding, not mere statistical noise. This volume reflects genuine and recurring flaws in Dart's test-writing style. These elevated metrics, combined with extreme values, reveal patterns of technical debt concentrated in specific areas of the ecosystem. This insight offers clear targets for intervention through refactoring, linter implementation, and educational initiatives aimed at improving test maintainability and readability.

As we can observe in the graph in Figure 4, the occurrence values of the test smells show a large discrepancy, especially due to the test smells "Assertion Roulette," "Duplicate Assert", and "Magic Number." This large variation in values makes it difficult to analyze the other test smells, which, although detected in smaller quantities, are equally relevant to our study. When the data spans a wide range of values, as in our case, the visualization of test smells with lower occurrences becomes obscured by the higher values, which compromises the comparative analysis between the different types of test smells.

To solve this problem and allow for a fairer and more intuitive comparison between all the test smells, we applied a base-10 logarithmic transformation to the occurrence values. The base-10 logarithm is particularly useful because it reduces the large differences in magnitude, compressing the values proportionally while maintaining their relative relationships. This makes the graphs more balanced, allowing smaller occurrences to be clearly visualized without being "overpowered" by the presence of larger values. The logarithmic transformation helps to reveal patterns and trends that would be difficult to identify on a linear scale, in addition to making it easier to compare test smells of different orders of magnitude.

Magic Number, Duplicate Assert, and Assertion Roulette have high standard deviations (105, 62, and 44, respectively), indicating that their distribution varies greatly across projects. In contrast, Empty Test has the lowest standard deviation (0.16), suggesting a more uniform distribution. Tests such as Empty Test and Ignored Test have low median values, indicating that, in most cases, they are infrequent but appear in specific projects.

Some test smells reach extreme values per test file: Assertion Roulette with 2,611, Duplicate Assert with 3,253, and Magic Number with 6,413. This suggests that certain projects may be concentrating problematic practices and not prioritizing the quality of their tests. Some test smells that occurred in a very specific manner, such as Print Statement Fixture and Sleepy Fixture, indicate issues in the maintenance and debugging process, potentially impacting test efficiency.

Conditional test logic and exception handling have lower average values, which suggests that these smells are less common but should still be monitored.

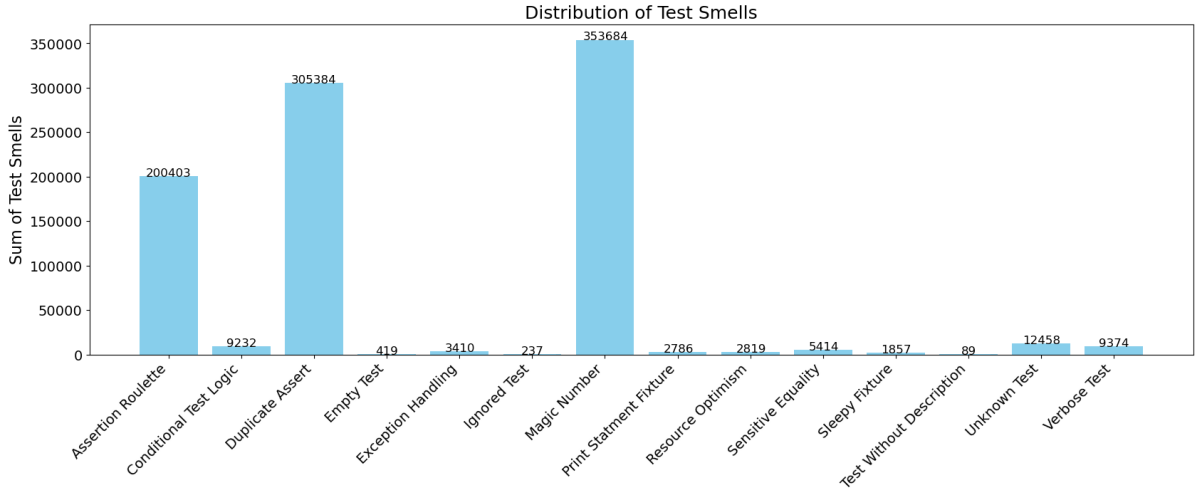

Figure 4: Distribution of Test Smells

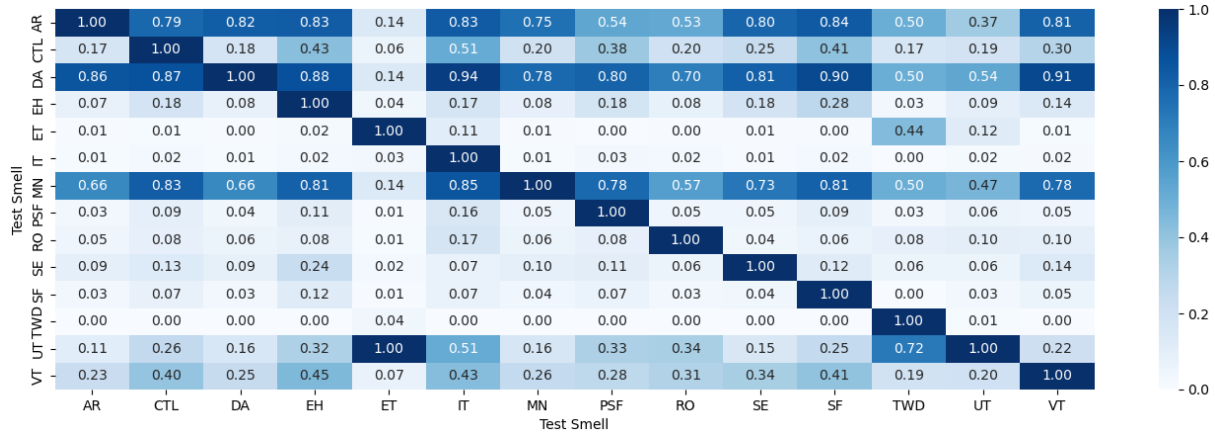

Figure 5: Co-occurrence of test smells

6.1 Co-ocurrences

In Figure 5, we can see the co-occurrences between the test smells. In the following list, we describe each co-occurrence in more detail, along with their motivations. This information can be used to generate prioritization strategies for refactorings.

- **AR with DA - Correlation: 0.82 and DA with AR - Correlation: 0.86** The correlation of 0.82 suggests that when "Assertion Roulette" occurs (a test with more than one assertion without description), it is very likely that "Duplicate Asserts" (assertions checking the same function) will also be present. This indicates that the tests are making repeated assertions about the same functionality, making the test redundant and hard to maintain. This combination suggests a lack of clarity in the test design, where assertions are not made efficiently [2, 30, 31].
- **AR with EH - Correlation: 0.83** When "Assertion Roulette" occurs, there is also a strong correlation with "Exception Handling" issues (the use of exception handling in tests). The high correlation (0.83) indicates that, in addition to using the exception system for making assertions, two anti-patterns are being used together. This suggests that the tests rely on error handling to perform their checks [2, 21]. In some languages, the solution is to use specific annotations for exception handling [22].
- **AR with IT - Correlation: 0.83** The correlation between "Assertion Roulette" and "Ignored Test" indicates that tests with many assertions are being ignored in Dart projects and not executed. This type of information is concerning, as the motivation for ignoring the test is unclear. It may suggest that the tests are poorly structured, leading to tests being disregarded or failing to run, or unnecessary at the coverage level [19, 41].
- **AR with SF - Correlation: 0.84** The correlation between "Assertion Roulette" and "Sleepy Fixture" (slow or poorly

designed fixture) shows that tests with "Assertion Roulette," in addition to difficulty in understanding the test's motivation, use wait times for specific calls, causing the test to take longer than necessary. This can become a systematic problem due to the number of tests with this issue within the same project. Another problem that sleep can cause is the non-repeatability of the test, potentially leading to inconsistent results, sometimes passing and other times failing [5, 28].

- **AR with VT - Correlation: 0.81** When "Assertion Roulette" occurs, there is a good chance the test will also be "Verbose Test" (with excessive unnecessary details). Verbose tests are those with excessive outputs or information, making the test code harder to understand and maintain. The correlation indicates that the redundancy of assertions can lead to an overload of information in the tests, making them harder to analyze and maintain [1, 9].
- **DA with CTL - Correlation: 0.87** The high correlation between "Duplicate Assert" and "Conditional Test Logic" suggests that tests with duplicate assertions often involve excessive conditional logic. This may indicate that the test is using conditions to analyze the same method with different parameters and flows, making it more complex and harder to understand, which hinders its maintainability [18, 25].
- **DA with EH - Correlation: 0.88** The correlation between "Duplicate Assert" and "Exception Handling" indicates that the tests use many assertions for validation based on exceptions. This suggests that, in addition to redundancy in assertions, the tests are not properly handling exceptions, which could lead to test failures when the code throws unexpected errors [30, 35].
- **DA with IT - Correlation: 0.94** The correlation of 0.94 between "Duplicate Assert" and "Ignored Test" is extremely strong. This suggests that when there is assertion duplication, it is very common for the test to be ignored. Redundancy in tests can hinder proper execution or result in coverage failures, causing these tests to be ignored [8, 16].
- **DA with PSF - Correlation: 0.80** The correlation of 0.80 between "Duplicate Assert" and "Print Statement Fixture" suggests that tests with duplicate assertions often use print statements for debugging. This suggests that the tests may be poorly structured, with multiple assertions made without clear organization, which can be confusing and difficult to maintain [9, 21].
- **DA with SF - Correlation: 0.90** The correlation between "Duplicate Assert" and "Sleepy Fixture" suggests that tests with multiple redundant assertions often have inefficient test setups, resulting in slower. This may indicate that the tests are not optimized and have performance issues due to improper test environment configuration [11, 39].
- **DA with VT - Correlation: 0.91** The 0.91 correlation suggests that when "Duplicate Assert" occurs, it is very likely the test will also be "Verbose Test". In other words, the tests have many assertions or unnecessary outputs, making them harder to understand and maintain [20, 21].
- **MN with CTL - Correlation: 0.83** The correlation between "Magic Number" and "Conditional Test Logic" suggests that tests with "Magic Numbers" (fixed literal values) often include conditional logic. This can make the test harder to understand and modify because the use of fixed values complicates the reading and maintenance of the test code [17, 34].
- **MN with EH - Correlation: 0.81** When "Magic Numbers" are combined with "Exception Handling," there is also a tendency to use exceptions within the test. This may suggest that tests with fixed numbers are often used within exception checks or to trigger an exception, which can lead to failures or unexpected behaviors [23, 34].
- **MN with IT - Correlation: 0.85** The correlation between "Magic Number" and "Ignored Test" suggests that tests with magic numbers are often ignored. This may be due to the lack of clarity about what these values mean, making the tests harder to understand and likely to be neglected or fail during execution [26, 43].
- **MN with SF - Correlation: 0.81** The correlation between "Magic Number" and "Sleepy Fixture" is due to the use of magic numbers within sleep calls [26, 28].
- **UT with ET - Correlation: 1.00** The correlation of 1.00 between "Unknown Test" and "Empty Test" is perfect, indicating that all empty tests have no assertions[6, 27].

The co-occurrence analysis in this section reveals that test smells are not isolated. They are typically surrounded with other badly-patterned areas of poor quality that make problems with test maintenance and understanding even worse. The correlations, particularly with Assertion Roulette, Duplicated Assert, and Magic Number, are very strong. These test smells tend to come along with one another making the tests have redundant and unclear assertions, too much conditional logic, improper exception handling, and reliance on undocumented literal values-all of which drastically reduce readability and maintainability.

Together, these patterns result in tests that are very hard to understand and debug, much slower to execute, and even more likely to be ignored, they accrue enormous amounts of technical debt into the Dart ecosystem.This deeper view of how test smells interact will be critical for the Dart developer community because it provides actionable insights into where prioritizing and guiding refactoring efforts should be directed.It highlights the urgent need for better test-writing practices through style enforcement tools (linters) and educational initiatives that raise overall quality in projects created with Flutter.

7 Threats to Validity

- **Conclusion Validity**
  In the study, the validity of the conclusion may be affected by how the data on test smells were collected and analyzed. The DNose[7] tool was used to automatically detect test smells, and the accuracy of this detection is crucial for the validity of the conclusions. To mitigate this threat, an accuracy study of the tool was conducted. Another threat is the manual detection of test smells to generate the oracle used to define the tool's accuracy, which may introduce errors in manual detection. To mitigate this threat, a fully crossed design was used. The sample size and the diversity of the analyzed projects may

[7] https://dnose-ts.github.io/

also impact the validity of the conclusion. To mitigate this threat, a large number of projects from the main repository of the Dart language were used.

- **Construct Validity**
  The definitions of test smells used in the study are based on previous works in other languages. Even though they are well-established, there should be a presentation of these test smells in the Dart language. To mitigate this threat, examples of test smells in the Dart language were included.
- **External Validity**
  The study selected open-source projects from Dart's official repository, Pub.dev. These projects may not be representative of all existing Dart projects, especially those that are private and corporate.

8 Conclusion and Future Work

This study investigated the quality of tests in Dart projects, with a particular focus on the libraries available in its central repository, which are used for developing web, desktop, and mobile applications. The analysis, conducted with the DNose tool, revealed an alarming presence of test smells in 74% of the test files analyzed, totaling 907,566 occurrences. From a project perspective, it is important to note that 60% of the analyzed projects contained at least one test smell.

The results highlight that Magic Number (353,684 occurrences), Duplicate Assert (305,384 occurrences), and Assertion Roulette (200,403 occurrences) are the most frequent test smells, with a significant difference compared to the others. This indicates a concerning trend of poorly structured and hard-to-read test code in Dart projects, particularly regarding duplicated assertions, lack of descriptions for assertions, and the undefined usage of specific numbers within the code.

Additionally, the high variability in the distribution of certain test smells, as indicated by the standard deviations, suggests that some projects concentrate problematic testing practices, neglecting code quality. The presence of test smells such as Print Statement Fixture and Sleepy Fixture raises questions about developers' knowledge of debugging tools, which impacts the maintainability and efficiency of tests.

The co-occurrence analysis reinforces the influence of Magic Number, Duplicate Assert, and Assertion Roulette on overall test quality, demonstrating their frequent association with other test smells. The findings of this study provide a crucial perspective on the current state of Dart projects and, by extension, the entire Flutter framework ecosystem, revealing the need for greater attention to test code quality within the community.

As future work, we intend to conduct a practical study to assess the improvements that can be achieved by using linters during coding, aiming to identify test smells before they are committed to the production code. Another proposal is to investigate the persistence of these test smells, analyzing how long it takes for them to be removed after their introduction. Finally, we plan to conduct a detailed study on repositories to analyze, both quantitatively and qualitatively, whether developer experience and sentiments influence the introduction of test smells.

ARTIFACT AVAILABILITY

The material generated in the example and the video are available on Zenodo: https://zenodo.org/records/14869744. The DNose site is available at https://dnose-ts.github.io/, and the DNose source code is available on GitHub at https://github.com/tassiovirginio/dnose and a demo is running at https://dnose.onrender.com/ .

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001; CNPq grants 140587/2024-1, 315840/2023-4 and 403361/2023-0; and FAPESB grant PIE0002/2022.

REFERENCES

[1] João Afonso and José Campos. 2023. Automatic Generation of Smell-free Unit Tests. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE, 9–16.

[2] Wajdi Aljedaani, Mohamed Wiem Mkaouer, Anthony Peruma, and Stephanie Ludi. 2023. Do the Test Smells Assertion Roulette and Eager Test Impact Students' Troubleshooting and Debugging Capabilities?. In *Proceedings of the 45th International Conference on Software Engineering: Software Engineering Education and Training* (Melbourne, Australia) *(ICSE-SEET '23)*. IEEE Press, 29–39. https://doi.org/10.1109/ICSE-SEET58685.2023.00009

[3] 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

[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] Bruno Camara, Marco Silva, Andre Endo, and Silvia Vergilio. 2021. On the use of test smells for prediction of flaky tests. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing*. 46–54.

[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] B. Clark. 2013. Cellular phones as a primary communications device: What are the implications for a global community? *Global Media Journal* 12 (01 2013).

[8] Manuel De Stefano, Fabiano Pecorelli, Dario Di Nucci, and Andrea De Lucia. 2022. A preliminary evaluation on the relationship among architectural and test smells. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 66–70. https://doi.org/10.1109/SCAM55253.2022.00013

[9] Jianshu Ding, Guisheng Fan, Huiqun Yu, and Zijie Huang. 2022. Automatic identification of high-impact bug report by product and test code quality. *International Journal of Software Engineering and Knowledge Engineering* 32, 06 (2022), 893–916.

[10] Amin Milani Fard and Ali Mesbah. 2013. JSNOSE: Detecting JavaScript Code Smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 116–125. https://doi.org/10.1109/SCAM.2013.6648192

[11] Daniel Fernandes, Ivan Machado, and Rita Maciel. 2021. Handling test smells in python: Results from a mixed-method study. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*. 84–89.

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

[13] 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

[14] Kevin A Hallgren. 2012. Computing inter-rater reliability for observational data: an overview and tutorial. *Tutorials in quantitative methods for psychology* 8, 1 (2012), 23.

[15] Dalton Jorge, Patricia Machado, and Wilkerson Andrade. 2021. Investigating Test Smells in JavaScript Test Code. In *Anais do VI Simpósio Brasileiro de Testes de Software Sistemático e Automatizado* (Joinville). SBC, Porto Alegre, RS, Brasil, 36–45. https://sol.sbc.org.br/index.php/sast/article/view/18790

[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] 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.

[18] Nildo Silva Junior, Larissa Rocha, Luana Almeida Martins, and Ivan Machado. 2020. A survey on test practitioners' awareness of test smells. *arXiv preprint arXiv:2003.05613* (2020).

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

[20] Luana Martins, Carla Bezerra, Heitor Costa, and Ivan Machado. 2021. Smart prediction for refactorings in the software test code. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering* (Joinville, Brazil) *(SBES '21)*. Association for Computing Machinery, New York, NY, USA, 115–120. https://doi.org/10.1145/3474624.3477070

[21] 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. https://doi.org/10.1002/smr.2532 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2532

[22] Estevan Paula and Rodrigo Bonifácio. 2022. TestAXE: Automatically Refactoring Test Smells Using JUnit 5 Features. In *Anais Estendidos do XIII Congresso Brasileiro de Software: Teoria e Prática* (Uberlândia/MG). SBC, Porto Alegre, RS, Brasil, 89–98. https://doi.org/10.5753/cbsoft_estendido.2022.227655

[23] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the distribution of test smells in open source Android applications: an exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '19)*. IBM Corp., USA, 193–202.

[24] 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

[25] Anthony Peruma and Christian D Newman. 2021. On the distribution of" simple stupid bugs" in unit test files: An exploratory study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 525–529.

[26] Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. An Exploratory Study on the Refactoring of Unit Test Files in Android Applications. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 350–357. https://doi.org/10.1145/3387940.3392189

[27] Valeria Pontillo, Dario Amoroso d'Aragona, Fabiano Pecorelli, Dario Di Nucci, Filomena Ferrucci, and Fabio Palomba. 2024. Machine learning-based test smell detection. *Empirical Software Engineering* 29, 2 (2024), 55.

[28] Railana Santana, Daniel Fernandes, Denivan Campos, Larissa Soares, Rita Maciel, and Ivan Machado. 2021. Understanding practitioners' strategies to handle test smells: a multi-method study. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*. 49–53.

[29] 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* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 374–379. https://doi.org/10.1145/3422392.3422510

[30] 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*. 374–379.

[31] 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. https://doi.org/10.1016/j.scico.2023.103013

[32] Railana Santana, Luana Martins, Tássio Virgínio, Larissa Soares, Heitor Costa, and Ivan Machado. 2022. Refactoring Assertion Roulette and Duplicate Assert test smells: a controlled experiment. In *Anais do XXV Congresso Ibero-Americano em Engenharia de Software* (Córdoba). SBC, Porto Alegre, RS, Brasil, 263–277. https://doi.org/10.5753/cibse.2022.20977

[33] Elvys Soares, Manoel Aranda III, Davi Romão, and Márcio Ribeiro. 2023. The Open Catalog of Test Smells. Available at https://test-smell-catalog.readthedocs.io/en/latest/index.html.

[34] Elvys Soares, Márcio Ribeiro, Guilherme Amaral, Rohit Gheyi, Leo Fernandes, Alessandro Garcia, Baldoino Fonseca, and André Santos. 2020. Refactoring Test Smells: A Perspective from Open-Source Developers. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing* (Natal, Brazil) *(SAST '20)*. Association for Computing Machinery, New York, NY, USA, 50–59. https://doi.org/10.1145/3425174.3425212

[35] Elvys Soares, Márcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André Santos. 2023. Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date? *IEEE Transactions on Software Engineering* 49, 3 (2023), 1152–1170. https://doi.org/10.1109/TSE.2022.3172654

[36] 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)*. 1–12. https://doi.org/10.1109/ICSME.2018.00010

[37] Statista. 2024. Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2023. https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/. Accessed: 2024-07-15.

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

[39] Victor Veloso and Andre Hora. 2022. Characterizing High-Quality Test Methods: A First Empirical Study. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 265–269. https://doi.org/10.1145/3524842.3529092

[40] Tássio Virgínio, Luana Martins, Larissa Rocha, Railana Santana, Adriana Cruz, Heitor Costa, and Ivan Machado. 2020. JNose: Java Test Smell Detector. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering* (Natal, Brazil) *(SBES '20)*. Association for Computing Machinery, New York, NY, USA, 564–569. https://doi.org/10.1145/3422392.3422499

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

[42] Machado.I Virgínio.T, Ribeiro.M. 2025. On the prevalence of test smells in mobile development. https://doi.org/10.5281/zenodo.14869744

[43] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. 2022. PyNose: a test smell detector for python. In *PyNose: a test smell detector for python* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 593–605. https://doi.org/10.1109/ASE51524.2021.9678615