Journal of Software Engineering Research and Development, 2026, 14:18, doi: 10.5753/jserd.2026.7537

More than one million test smells: how are Dart projects and their sentiments?

Tássio Virgínio [**Federal Institute of Tocantins** | ***tassio.virginio@ifto.edu.br***]
Márcio Ribeiro [**Federal University of Alagoas** | ***marcio@ic.ufal.br***]
Ivan Machado [**Federal University of Bahia** | ***ivan.machado@ufba.br***]

Abstract

Objective: This study investigates the quality of tests and the associated sentiments in Dart, the primary language for mobile application development with the Flutter framework. *Methods:* The study begins by using the DNose tool to detect 14 types of test smells in code written in the Dart language, along with their associated sentiments. Next, we evaluate the tool in terms of precision, recall, accuracy, 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 projects, from which 4,154 repositories were successfully cloned after processing. Based on the cloned projects, we generated a dataset containing 1,115,938 occurrences of test smells and their associated sentiments. The analysis allowed us to characterize the most frequently encountered test smells and identify their causes. We observed the presence of test smells in 77% of test files. Another key characteristic observed in the analyzed projects was the scarcity of tests: 1,873 projects had one or no tests, which led us to expand the analysis to a broader base. In addition, we analyzed the relationship between developers' sentiments and the identified test smells, enabling us to classify test smells according to sentiment. *Conclusion:* This research makes a significant contribution by providing in-depth insights into the quality of tests and associated sentiments in projects from Dart's official repository, as well as by offering an open-source tool for detecting 14 types of test smells and their associated sentiments.

Keywords: *Software testing, Test smells, Dart, Flutter, Sentiment analysis.*

1 Introduction

Smartphones are the primary means of accessing information, being widely used for activities such as reading news, managing bank accounts, and performing financial transactions (Clark, 2013). These services are predominantly accessed through mobile applications, which has intensified the demand for efficient and scalable mobile development frameworks. In this context, Flutter[1] has emerged as the leading cross-platform mobile application development framework, due to its growing adoption and industrial relevance (Statista, 2024).

Given this scenario, ensuring the quality of mobile applications developed with Flutter becomes a critical concern. In this study, we focus on the quality of mobile systems developed using the Dart programming language, which underpins Flutter and has become one of the most widely used languages for cross-platform mobile development. More specifically, we examine the quality of test code in Dart-based projects. To assess test quality, we adopt the concept of test smells, following prior studies that successfully employed test smells as indicators of deficiencies in test code quality (Santana et al., 2020a; Virgínio et al., 2020; Santana et al., 2022, 2024).

Assessing test quality in large-scale software systems requires concrete and well-established quality indicators that go beyond subjective judgment. In this context, test smells have been proposed as a practical and systematic way of identifying deficiencies in test code that may hinder its readability, maintainability, and effectiveness.

Test smells are anti-patterns introduced in test code that negatively affect test quality. A substantial body of research has investigated test smells in different programming languages, including Java (Peruma et al., 2020a; Santana et al., 2020a; Virgínio et al., 2020; Santana et al., 2022, 2024), Python (Wang et al., 2022; Fernandes et al., 2022), and JavaScript (Fard and Mesbah, 2013; Jorge et al., 2021a), among others. To support the identification of test smells in practice, several automated tools have been proposed for specific languages and ecosystems (Peruma et al., 2020a; Aljedaani et al., 2021; Virgínio et al., 2020; Jorge et al., 2021a; Wang et al., 2022; Fernandes et al., 2022).

Building upon prior work on test smells and the availability of automated detection tools, this study adopts a tool-supported approach to empirically investigate test quality in the Dart ecosystem. Using the DNose tool[2] (Virgínio et al., 2025a), we first conducted an evaluation of the tool's accuracy and then performed a large-scale analysis of open-source projects written in Dart.

In this analysis, we examined the distribution of test smells, identifying the most frequent smells, their potential causes, and their co-occurrence patterns. The results provide an overview of the current state of test quality in Dart-based projects, offering insights that are particularly relevant for the development and maintenance of mobile applications built with modern cross-platform frameworks.

As a result of this empirical analysis, we generated an extensive dataset comprising over 1,115,938 occurrences of test smells and observed that 77% of the analyzed test suites exhibit at least one such issue. These findings offer a compre-

[1] `https://flutter.dev/`

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

hensive view of the current state of test quality in Dart-based projects.

Beyond the structural and quantitative analysis of test smells, we further investigated their relationship with developers' perceptions through sentiment analysis. Specifically, we extracted sentiment scores from commit messages to examine how the presence of test smells correlates with developers' expressed emotions during software evolution.

Through this analysis, we identified which test smells are most strongly associated with negative sentiment and explored recurring patterns across different authors and project contexts. This complementary perspective enriches our findings by shedding light on the human factors underlying test quality issues, thus broadening the understanding of their impact beyond purely technical aspects.

This paper extends our previous work on test smells in Dart projects (Virgínio et al., 2025b) by broadening both the empirical scope and the analytical depth of the original study. While the initial study focused on identifying and characterizing the distribution of test smells in Dart-based projects using the DNose tool, this extended version expands the dataset, enabling analysis of over one million test smell occurrences across thousands of open-source projects. In addition to refining the empirical evaluation of DNose, this work introduces a sentiment analysis based on commit messages, enabling a systematic investigation of how test smells relate to developers' expressed emotions. By integrating quantitative test smell detection with sentiment-based insights, this extended study offers a more holistic perspective on test quality in the Dart ecosystem, encompassing both technical and human aspects of software development.

The remainder of this paper is organized as follows. Section 2 presents the background concepts necessary to understand the study, including test smells and sentiment analysis. Section 3 describes the empirical evaluation of the DNose tool, detailing the dataset construction, oracle definition, and accuracy assessment. Section 4 outlines the experimental design and research questions guiding the study. Section 5 reports the results of the large-scale analysis, including the distribution, frequency, and co-occurrence of test smells, as well as their relationship with developer sentiment. Section 6 discusses the findings, highlighting implications for researchers and practitioners and opportunities for improving test quality in the Dart ecosystem. Section 7 discusses threats to validity. Section 8 reviews related work on test smells and test quality across programming languages and ecosystems. Finally, Section 9 concludes the paper and outlines directions for future work.

2 Background

This section presents the fundamental concepts required to understand the context, motivation, and methodological choices of this study.

2.1 Test Smells

Test smells were first introduced by van Deursen et al. (van Deursen et al., 2001) as poor design choices in test code that may hinder its readability, maintainability, and effectiveness. Since then, they have become one of the most widely adopted approaches for assessing the quality of software tests (Spadini et al., 2018; Bavota et al., 2015; Santana et al., 2020a, 2022; Soares et al., 2023b). Over the years, comprehensive catalogs of test smells have been proposed and studied across different programming languages and testing ecosystems (Garousi and Küçük, 2018; Soares et al., 2023a).

In addition to the test smells already discussed in the literature, we identified a new test smell that is particularly relevant to Dart and JavaScript ecosystems, namely *Test Without Description*. In the following section, we describe the set of test smells detected by the DNose tool and discuss their characteristics in the context of Dart-based projects.

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

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.

```
1 test("Checking method", () {
2   List<int> list = [1,2,3];
3         list.forEach((number){
4           expect(productionMethod(number),number
                *2);
5   }
6   );
7 });
```

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.

```
1 test("Duplicate Assert", () {
2   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 test smell arises when a test function lacks a proper description, reducing the readability and understandability of the test code. In Dart testing frameworks, test intentions are typically expressed through string-based descriptions passed as function arguments, unlike Java-based frameworks where method names often convey the tested behavior. The absence of such descriptions hinders the identification of executed scenarios and the interpretation of test outcomes, frequently requiring developers to inspect the test and production code to infer the test objective.

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.

2.2 Sentiment Analysis

Sentiment analysis is an area of study within Natural Language Processing (NLP) that seeks to identify and extract opinions and emotions expressed in text (Calefato et al., 2018). Recently, several studies have explored the impact of sentiments in software engineering (Islam and Zibran, 2018; Ahmed et al., 2017; Calefato et al., 2018), analyzing how developer perception can influence the quality of code and tests. In this study, we use the **AFINN-165** wordlist, developed by Nielsen (2011). It is a list of words rated for valence with an integer between minus five (negative) and plus five (positive). It enables the capture of sentiment intensity expressed in commit messages. The motivation behind this analysis lies in the observation that development contexts favorable to the accumulation of technical debts, such as *test smells*, may co-occur with negative sentiment expressions in commit messages associated with the introduction or maintenance of such code.

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.

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 (Hallgren, 2012). 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

[3] https://pub.dev/

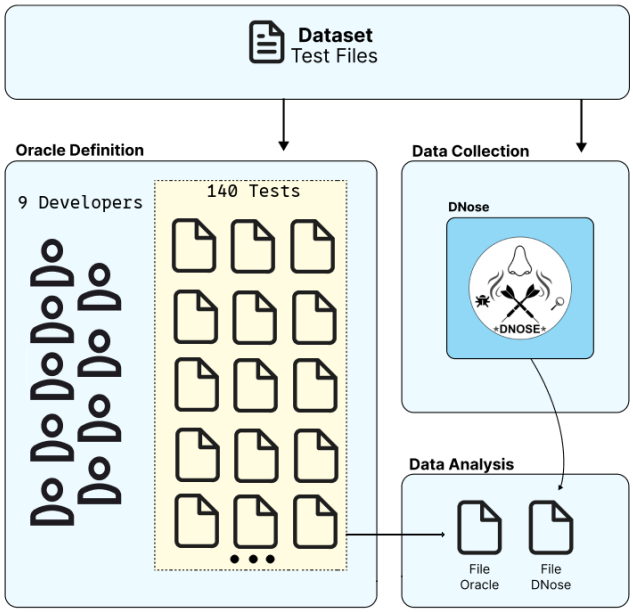

Figure 1. Empirical Accuracy Study

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.

3.2 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 (Virgínio et al., 2025a). The data for replication purposes are available online (Virgínio et al., 2026). 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 describes the approach we used to structure our study, including the research questions and study design.

4.1 Research 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.

RQ$_5$ *What is the exploratory relationship between the occurrence of different test smells and the overall sentiment score in the analyzed projects?* This question aims to explore the co-occurrence between test smells and developer sentiment. By analyzing the cumulative sentiment scores associated with each type of test smell, we can identify which smells are most frequently accompanied

by negative sentiment expressions in commit messages, serving as an initial baseline for future studies.

RQ$_6$ *What is the average intensity of negative sentiment associated with each test smell individually?* While RQ5 focuses on the overall co-occurrence, this question examines the average sentiment intensity per occurrence. This helps distinguish between smells that are frequent but co-occur with mildly negative sentiments versus those that are less common but strongly associated with negative sentiment expressions.

4.2 Design

Figure 2 presents 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.

It is important to highlight the evolution of our extraction tool, DNose, between the preceding version of this study (Virgínio et al., 2025b) and this extension. Although both versions mined the same 4,154 repositories, the extended dataset grew by approximately 23% (from 907,566 to 1,115,938 occurrences). This increase is a direct result of a major architectural improvement in DNose. The core of all detectors was completely refactored to natively inherit and implement the `RecursiveAstVisitor` from the official Dart Analyzer, ensuring an exhaustive traversal of the Abstract Syntax Tree (AST), which previously failed on deeply nested structures. Furthermore, we implemented a highly robust, fault-tolerant concurrent processing pipeline using semaphores to prevent parser crashes on complex files. Finally, the parser was upgraded to be fully compatible with Dart SDK 3.11.0, enabling it to successfully process modern syntax features. These improvements yielded a much more accurate and complete picture of the test smells present in the dataset.

5 Results

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

Figure 3 describes 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 ones. 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 problems. The third step was to run the DNose[6] on all the projects, which resulted in 2,852 projects with at least one test smell detected, meaning approximately 68% of the cloned projects had at least one occurrence of a test smell.

We found 18,094 test files "infected" with test smells, out of a total of 23,316 test files from the cloned projects, meaning that 77% of the test files in the projects have at least one test smell.

5.1 Key Findings

This section provides a synthesis of the findings for each research question, focusing on the most relevant observations.

RQ$_1$ **What is the distribution of Test Smells in Dart language projects?** We found a distribution of 68% 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 77% of test smells in the analyzed tests;

RQ$_3$ **Which Test Smells are the most frequently found?** The most frequently found Test Smells were Magic Number, Duplicate Assert and Assertion Roulette, with the following values: 427,935, 378,527 and 246,247, 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" (0.80) 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.

RQ$_5$ **What is the exploratory relationship between the occurrence of different test smells and the overall sentiment score in the analyzed projects?** Our analysis shows that Magic Number (-22,424), Duplicate Assert (-21,420), and Assertion Roulette (-13,453) are the smells that most frequently co-occur with negative sentiments in terms of total score. Conversely, Empty Test (-23) and Test Without Description (-2) have a negligible total score.

RQ$_6$ **What is the average intensity of negative sentiment associated with each test smell individually?** When evaluating the average intensity, Conditional Test Logic shows the highest negative intensity (-2.03), followed by Sleepy Fixture (-1.83) and Duplicate Assert (-1.83). Notably, although Magic Number has the highest total negative impact, its average intensity (-1.65) is lower than smells like Verbose Test (-1.81) and Exception Handling (-1.75).

[4] https://huggingface.co/datasets/deepklarity/top-flutter-packages
[5] https://pub.dev/
[6] https://dnose-ts.github.io/

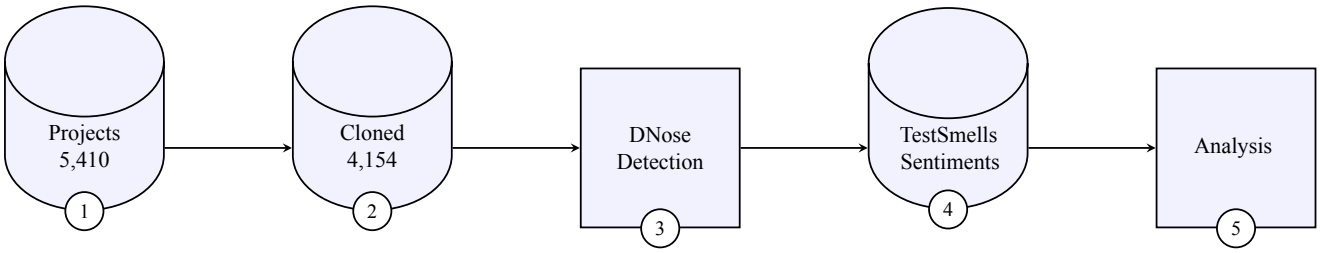

Figure 2. DNose Design: 1 - List of selected projects, 2 - Number of cloned projects, 3 - Use of DNose for test smells detection, 4 - Dataset with the list of selected test smells and Sentiments, 5 - Data analysis.

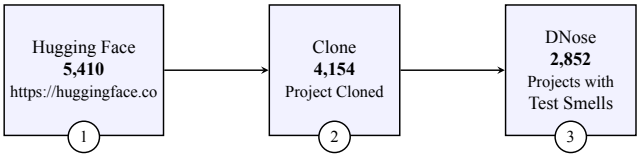

Figure 3. Project Filtering Workflow.

6 Analysis and Discussion

We conducted statistical analysis on a dataset of 4,154 Dart projects, extracted from the official Pub.dev repository. This yielded over 23,000 test files, comprising 1,115,938 occurrences of test smells.

Table 2 presents 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 dataset's substantial volume and diversity, we observed significant variability in the distribution of certain test smells. The test smells 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 detected more than one million test smells. As shown in Figure 4, these data indicate that the three most common test smells in Dart projects were magic number, duplicate assert, and assertion roulette. The magic number was found, on average, 18 times, while the duplicate assert was found an average of 16 times, and the assertion roulette appeared 11 times, with medians of 0, 2, and 1, respectively.

The Magic Number test smell stands out as the most prevalent, with 427,935 occurrences and a peak of 6,413 in a single project, suggesting its widespread and recurring nature in Dart projects. Our analysis showed an average of 18.35 occurrences per file, but with a median of zero and a high standard deviation of 101.60. This highly asymmetrical distribution indicates that while most files have few or no magic numbers, a select few are heavily burdened with thousands. This extreme distribution points to a problematic behavior: the systematic disregard for documenting numeric literals in certain contexts. It is 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.

Duplicate Assert is the second most frequent test smell, with 378,527 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 246,247 occurrences, indicating a tendency towards poorly structured tests that hinder 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 Figure 4 shows, the occurrence values of the test smells has a large discrepancy, especially for "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, as we can observe in the graph in Figure 5. The base-10 logarithm is particularly useful because it reduces the large differ-

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	10.5613	44.1038	1	2056.6848	2611	0	246247	1	47953663
Conditional Test Logic	0.4977	3.6207	0	13.3573	356	0	11605	0	311439
Duplicate Assert	16.2346	66.0735	2	4629.2686	3253	0	378527	2	107936027
Empty Test	0.0202	0.1646	0	0.0275	8	0	471	0	641
Exception Handling	0.1721	1.6884	0	2.8803	109	0	4013	0	67157
Ignored Test	0.0122	0.3032	0	0.0921	30	0	285	0	2147
Magic Number	18.3537	101.6048	0	10660.3870	6413	0	427935	0	248557583
Print Statement Fixture	0.1628	3.3847	0	11.4824	405	0	3795	0	267723
Resource Optimism	0.1561	1.4200	0	2.0408	100	0	3640	0	47584
Sensitive Equality	0.3228	3.5023	0	12.3704	273	0	7526	0	288428
Sleepy Fixture	0.0992	1.3332	0	1.7871	102	0	2313	0	41669
Test Without Description	0.0103	0.3771	0	0.1423	31	0	240	0	3318
Unknown Test	0.7260	4.2706	0	18.7653	228	0	16927	0	437531
Verbose Test	0.5324	4.4185	0	19.8065	557	0	12414	0	461808

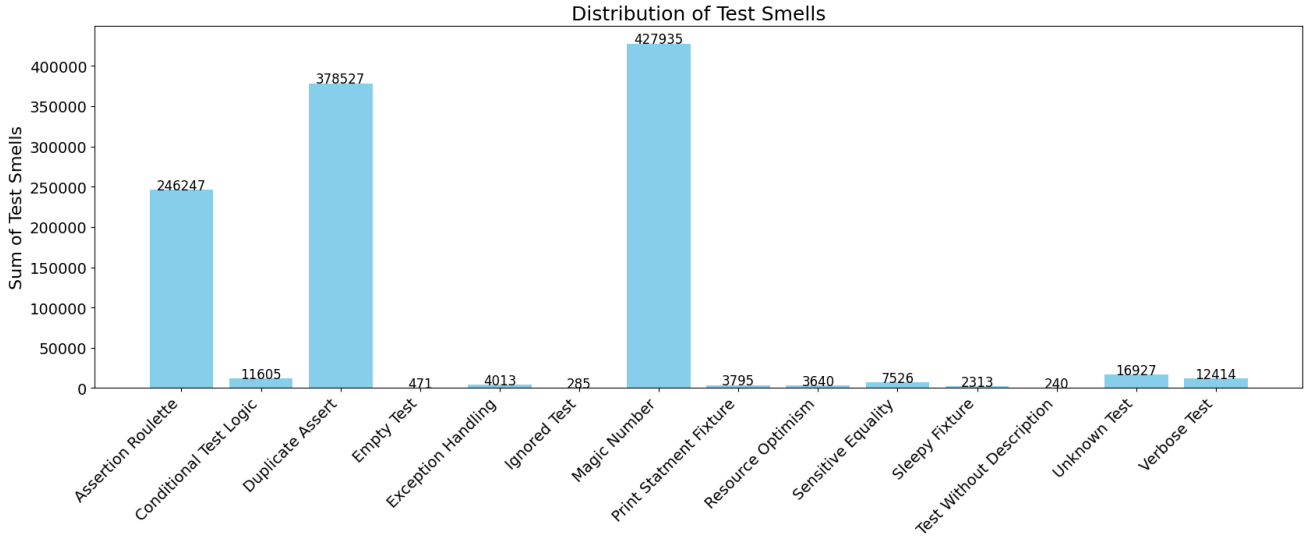

Figure 4. Distribution of Test Smells

ences 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.

From Table 2, we observe that Magic Number, Duplicate Assert, and Assertion Roulette have high standard deviations (102, 66, 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.

We identified test files exceeding 10,000 lines of code, which in itself hinders the maintainability of the tests located there. In terms of test smells, 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.

6.1 Co-occurrences

In Figure 6, we present the co-occurrence relationships among the identified test smells. The following list provides a detailed discussion of each co-occurrence pattern, including possible motivations behind these relationships. Such information may support the definition of prioritization strategies for test refactoring activities.

- **AR with DA - Correlation: 0.83 and DA with AR - Correlation: 0.86** The correlation of 0.83 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 (Aljedaani et al., 2023; Santana et al., 2020b, 2024).
- **AR with EH - Correlation: 0.84** When "Assertion Roulette" occurs, there is also a strong correlation with "Exception Handling" issues (the use of exception handling in tests). The high correlation (0.84) 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 (Martins et al., 2024; Aljedaani et al., 2023). In some languages, the solution is to use specific annotations for exception handling (Paula and Bonifácio, 2022).
- **AR with IT - Correlation: 0.79** 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 (Kim et al., 2021; Virgínio et al., 2019).
- **AR with SF - Correlation: 0.86** 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 (Camara et al., 2021; Santana et al., 2021).
- **AR with VT - Correlation: 0.82** 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 (Ding et al., 2022; Afonso and Campos, 2023).
- **DA with CTL - Correlation: 0.89** 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 (Junior et al., 2020; Peruma and Newman, 2021).
- **DA with EH - Correlation: 0.87** 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 (Santana et al., 2020b; Soares et al., 2023c).
- **DA with IT - Correlation: 0.90** The correlation of 0.90 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 (De Stefano et al., 2022; Jorge et al., 2021b).
- **DA with PSF - Correlation: 0.82** The correlation of 0.82 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

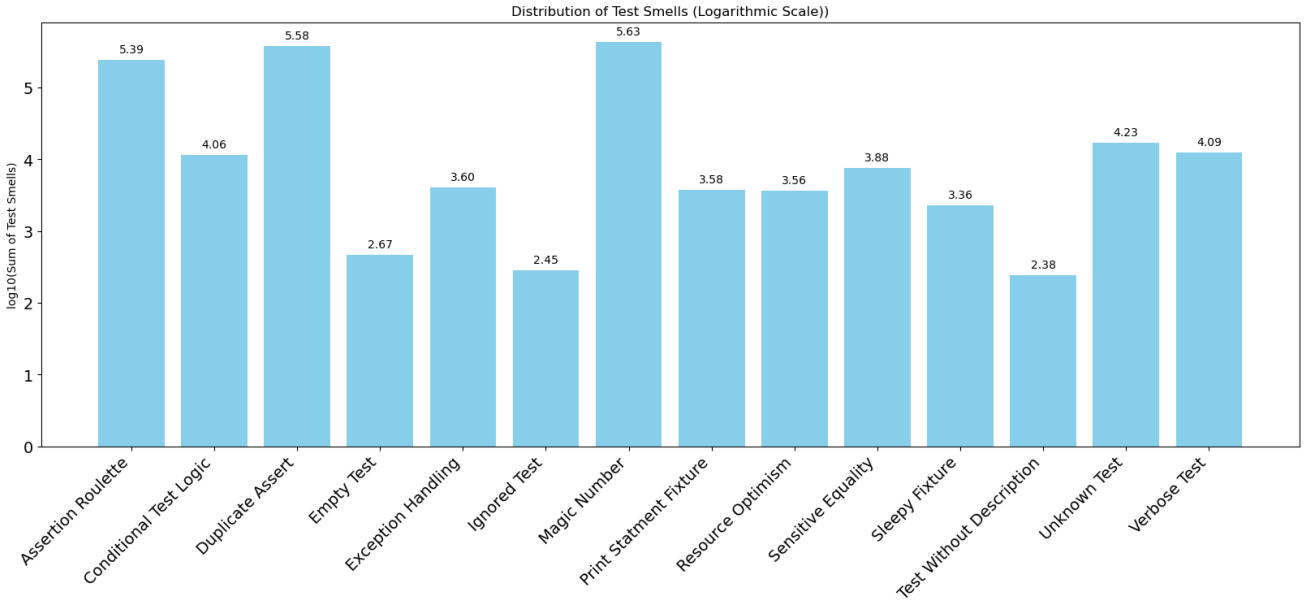

Figure 5. Distribution of Test Smells (Log)

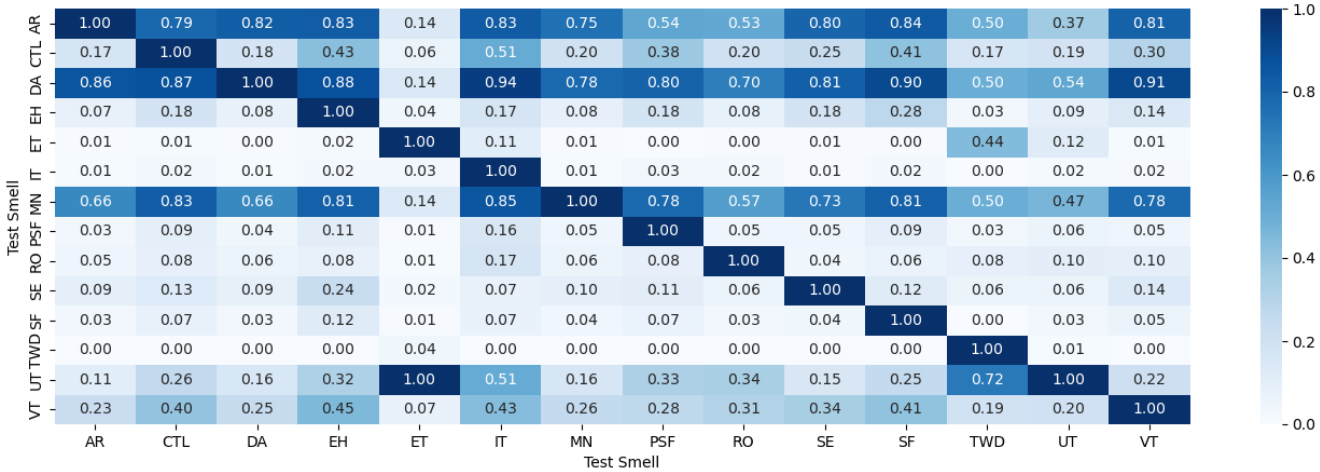

Figure 6. Co-occurrence of test smells

assertions made without clear organization, which can be confusing and difficult to maintain (Ding et al., 2022; Martins et al., 2024).

- **DA with SF - Correlation: 0.92** 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 (Veloso and Hora, 2022; Fernandes et al., 2021).
- **DA with VT - Correlation: 0.90** The 0.90 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 (Martins et al., 2021, 2024).
- **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 (Junior et al., 2021; Soares et al., 2020).
- **MN with EH - Correlation: 0.80** 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 (Peruma et al., 2019a; Soares et al., 2020).
- **MN with IT - Correlation: 0.80** 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 (Wang et al., 2022; Peruma et al., 2020b).
- **MN with SF - Correlation: 0.83** The correlation between "Magic Number" and "Sleepy Fixture" is due to the use of magic numbers within sleep calls (Santana et al., 2021; Peruma et al., 2020b).
- **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(Campos et al., 2021; Pontillo et al., 2024).

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, Duplicate 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.

The preceding analysis (RQ1–RQ4) established the technical landscape of test smells in Dart: their distribution, frequency, and co-occurrence patterns. However, test smells are not merely static code artifacts — they are introduced, tolerated, and accumulated by developers working under real-world conditions. Understanding the human dimension of this technical debt is essential because prior research has shown that developer affect and code quality are intertwined (Guzman et al., 2014; Lin et al., 2018). A developer who is frustrated, under pressure, or fatigued is more likely to produce lower-quality code, and conversely, working with degraded codebases can amplify negative sentiment.

The following section extends our investigation from the purely structural analysis to an exploratory examination of the emotional context surrounding the introduction of test smells. By analyzing the sentiment expressed in the commit messages associated with test smell occurrences (RQ5–RQ6), we aim to provide an initial, complementary perspective on whether the degradation of test code quality co-occurs with negative developer sentiment, adding a human-centered lens to the technical findings presented so far.

6.2 Sentiment Analysis

In this section, we perform the sentiment analysis associated with the detected test smells. The objective is to understand how the presence of these smells relates to developer perception, measured through sentiment scores extracted from commit messages. For this purpose, we processed the dataset with the latest version of DNose. Figure 7 describes the experiment design.

We acknowledge that commit messages are primarily technical artifacts, often brief, neutral, or automatically generated. However, the Empirical Software Engineering literature has established that they can serve as a valid proxy for extracting developer sentiment and correlating it with software artifacts (Guzman et al., 2014). Recent studies (Lin et al., 2018; Kaur et al., 2022) discuss the limitations of this approach and the multifactorial nature of developers' emotions. Furthermore, modern guidelines have been proposed for extracting emotions from commit messages (Dey et al., 2025). Building upon these foundational works, we frame our investigation not as a strict cause-and-effect relationship, but as an observational study of co-occurrence, aiming to understand whether the degradation of test code quality co-occurs with negative emotional polarity during the commit period.

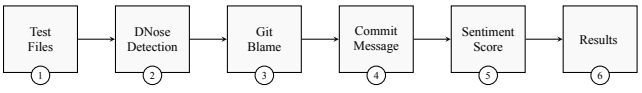

Figure 7. Sentiment Analysis Process: (1) Test files are analyzed by (2) DNose to detect test smells with their line numbers. (3) Git blame identifies the commit and author for each line. (4) The commit message is extracted. (5) A sentiment dictionary scores the message. (6) Results are aggregated per test smell type.

The experiment started with 1,115,938 test smell occurrences, from which it was possible to analyze sentiments in 2,852 projects. From each detected test smell, we identify the specific line in the source code. Using `git blame`, we retrieve the commit hash and author responsible for introducing that line. The commit message associated with this change is then extracted and analyzed using a sentiment dictionary that assigns positive and negative scores based on the words present. This process allows us to link developer sentiment directly to the introduction of specific test smells.

6.2.1 Sentiment Distribution

The analysis revealed a distribution where the majority of sentiments are neutral or positive. Of the 1,115,938 records analyzed, 168,554 (15.1%) had positive scores and 911,246 (81.7%) had neutral scores, while only 36,138 (3.2%) had negative scores. This proportion indicates that, in most cases, commits associated with test smells do not carry explicit negative connotations. However, negative cases deserve special attention as they reveal frustration or difficulty on the part of developers.

6.2.2 Total Negative Sentiment Score by Test Smell

Figure 8 presents the sum of negative sentiment scores by test smell type, showing which smells accumulate the highest negative load.

The three test smells with the highest cumulative negative impact are: *Magic Number* $(-22,424)$, *Duplicate Assert* $(-21,420)$, and *Assertion Roulette* $(-13,453)$. Together, these three smells account for the vast majority of the negativity found. This suggests that practices such as using undocumented literal values, duplicate assertions, and tests with multiple assertions without descriptions co-occur most frequently with negative sentiment expressions in commit messages.

At the other end, smells such as *Empty Test* (-23), *Ignored Test* (-10), and *Test Without Description* (-2) have almost negligible total impact. This low expressiveness can be attributed to their lower frequency of occurrence or the lower severity perceived by developers.

6.2.3 Mean Negative Sentiment Intensity

In addition to the sum, we analyzed the mean of negative sentiment scores to understand the individual intensity of each test smell type. Figure 9 illustrates these results.

The results show that *Conditional Test Logic* has the most negative mean (-2.03), indicating that its presence is strongly associated with intense negative sentiments. *Sleepy Fixture* and *Duplicate Assert* follow with means of approximately -1.83.

An interesting finding is that, although *Magic Number* has the highest total impact, its mean (-1.65) is lower than smells such as *Verbose Test* (-1.81) and *Exception Handling* (-1.75). This indicates that the high total impact of *Magic Number* is due to its high frequency of occurrence, and not necessarily to a higher negative intensity per individual occurrence.

Test Without Description had the least negative mean (-1.00), suggesting that when this smell causes frustration, it tends to be moderate.

6.2.4 Author Perspective

The sentiment analysis by author revealed that negativity is concentrated among a few contributors. The top three authors with the highest accumulated negative scores presented sums of $-2,221$, $-2,081$, and $-1,753$ respectively. To investigate whether these authors are maintainers of popular projects, we cross-referenced the commit data with the Hugging Face dataset containing project popularity metrics.

Our analysis revealed an association between high negative sentiment and the maintenance of widely-used libraries. The author ranked third in negativity maintains 18 distinct projects with a combined popularity exceeding 10,000 likes, including some of the most popular Flutter packages such as *provider*, *freezed*, and *riverpod*. The top-ranked negative author contributes to 12 projects, including *flutter_redux* and *i18n_extension*. Even the second-ranked author maintains 4 projects, including *conduit*. This pattern strongly supports prior research showing that maintainers of large-scale and widely-used projects tend to experience higher levels of stress and frustration (Raman et al., 2020; Miller et al., 2019). Core developers often face overwhelming workloads, constant demands for features and fixes, and a perceived lack of appreciation for their efforts, which can manifest as negative sentiment in their commit messages (Guzman et al., 2014).

This concentration suggests that the responsibility of maintaining mature or high-profile projects can intensify the expression of negative sentiments. On the other hand, 778 distinct authors contributed at least one commit with negative sentiment, indicating that frustration associated with test smells is a phenomenon distributed across the Dart community, consistent with findings from Sinha et al. Sinha et al. (2016) on sentiment distribution in open-source projects.

6.2.5 Experience and Test Smell Introduction Rate

To investigate whether developer experience influences the introduction of test smells, we calculated the experience of each author based on the time span between their first and last commit in our dataset. We then computed the rate of test smell introduction per commit for each experience group, using a **normalized rate** that divides the total test smells by the total commits within each group, providing a more robust measure that accounts for varying activity levels. Table 3 presents the results of this analysis, considering only authors with at least one year of experience and a minimum of 10 commits in the dataset.

The columns in Table 3 are defined as follows: **Experience** represents the time span between the first and last com-

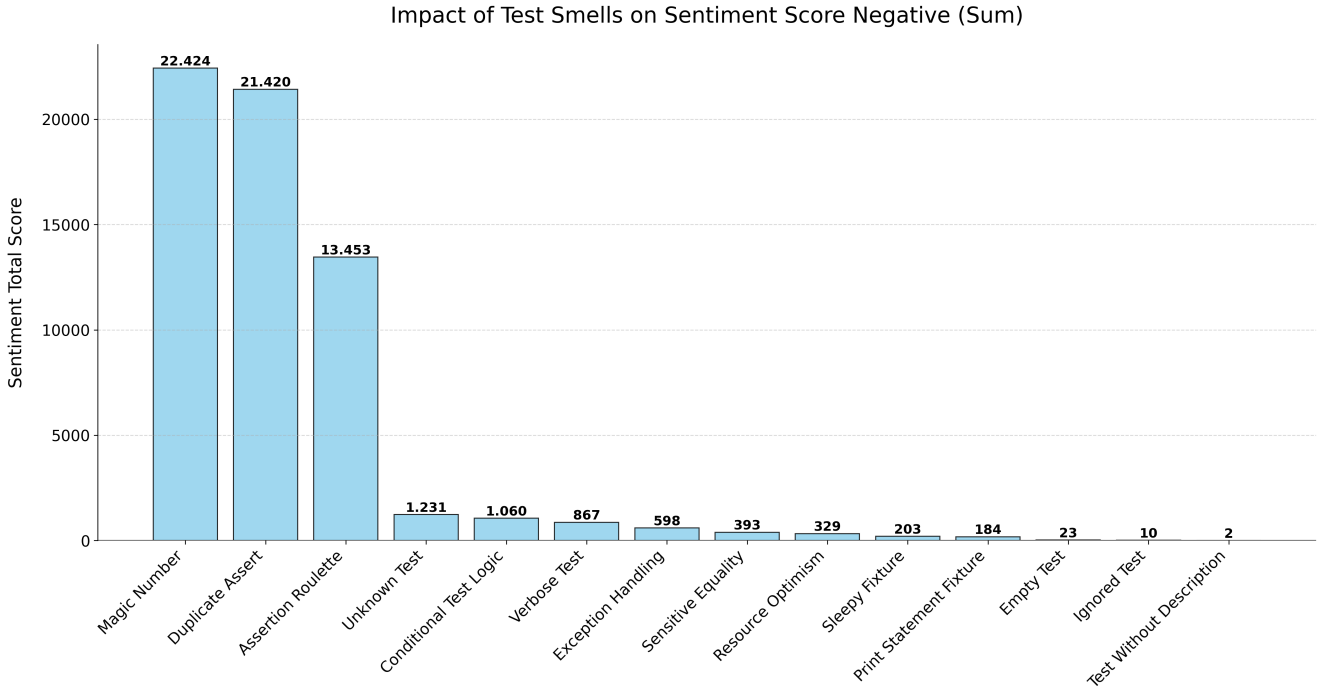

Figure 8. Total Negative Sentiment Score by Test Smell Type (Sum)

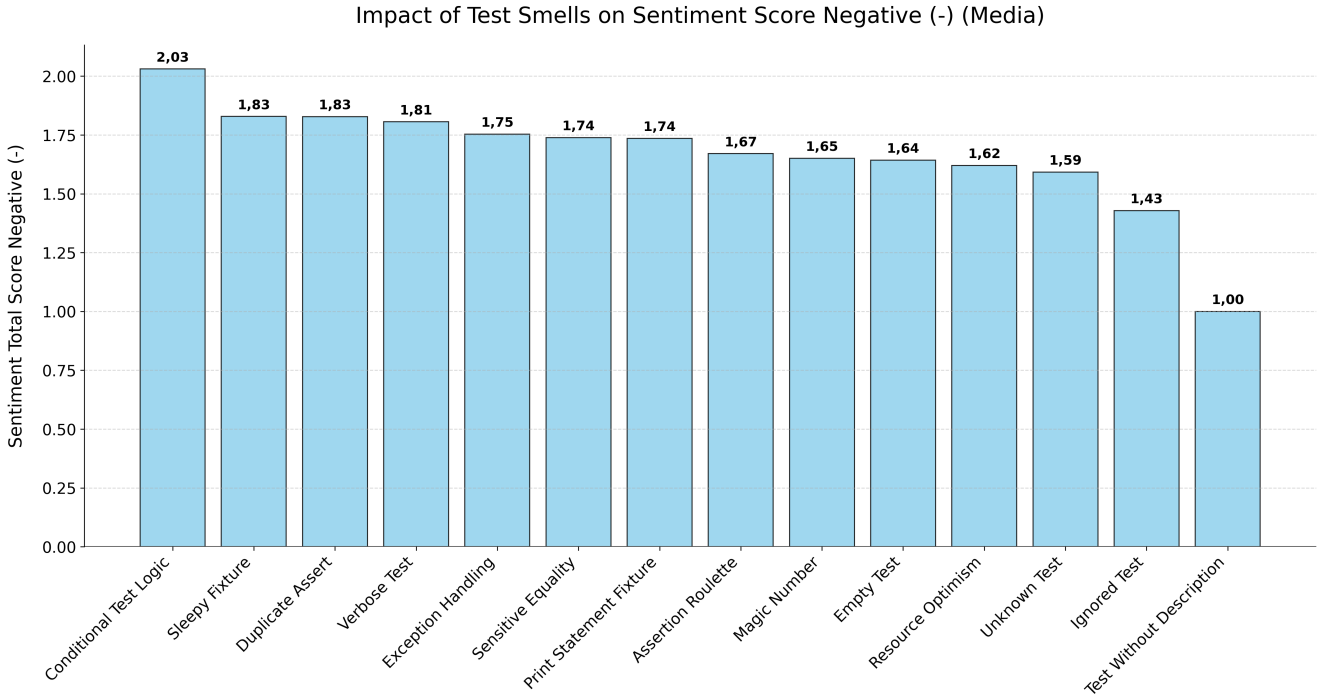

Figure 9. Mean Sentiment Score per Test Smell

Table 3. Relationship between developer experience and test smell introduction rate.

Experience	Authors	Avg. Commits	Total TS	Total Commits	Norm. Rate	Reduction
1–2 years	501	79	110,246	39,871	2.77	–
2–4 years	810	113	230,824	91,600	2.52	−8.9%
4–6 years	597	236	309,845	141,200	2.19	−20.6%
6+ years	269	549	345,031	147,699	2.34	−15.5%

mit; **Authors** is the number of developers in each group; **Avg. Commits** is the average number of commits per author; **Total TS** is the sum of test smells introduced by the group; **Total Commits** is the sum of commits by the group; **Norm. Rate** is the normalized rate calculated as Total TS divided by Total Commits; and **Reduction** represents the percentage reduction compared to the reference group (1–2 years).

The analysis reveals a clear pattern: developers with **1–2 years of experience** introduce an average of **2.77 test smells per commit**, while developers with **4–6 years of experience** introduce only **2.19 test smells per commit**, representing a **reduction of 20.6%**. This normalized rate provides a fairer comparison as it accounts for the significant difference in commit volume between groups, experienced developers (6+ years) contribute an average of 549 commits compared to 79 commits for beginners.

Interestingly, developers with 6+ years of experience show a slightly higher normalized rate (2.34) than the 4–6 years group (2.19), though still significantly lower than beginners (2.77). This may be attributed to the fact that highly experienced developers often work on more complex, legacy systems where test smells may accumulate over time due to technical debt. The grouped analysis reveals that the relationship is not linear but follows a pattern where intermediate-to-senior developers (4–6 years) achieve the best results in terms of test code quality.

6.2.6 Summary of Findings

The sentiment analysis reveals important nuances about how developers perceive test smells:

- The majority of test smell occurrences are associated with neutral (911,246) or positive (168,554) sentiments, totaling 96.8%;
- The test smells with the highest total negative impact (*Magic Number*, *Duplicate Assert*, *Assertion Roulette*) are also the most frequent;
- The mean intensity of negativity is not directly correlated with frequency, *Conditional Test Logic* is the most intense, but not the most frequent;
- The negativity is concentrated within specific contexts, probably linked to the maintainers of high-profile projects;
- These findings suggest that improvement initiatives should prioritize smells with high frequency and high intensity, such as *Duplicate Assert* and *Sleepy Fixture*.

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 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.
  Regarding the sentiment analysis, the latent construct of interest (developer frustration associated with test smells) is only indirectly reflected in our chosen proxy (sentiment polarity in commit messages). We explicitly acknowledge the "inference gap" pointed out by the literature: commit messages are not originally designed to capture emotional states and can be highly technical (Guzman et al., 2014; Lin et al., 2018; Sinha et al., 2016). To mitigate this threat, we restrict our claims to measuring *co-occurrence* rather than causality. We do not assert that developers were necessarily aware of the test smells or that the smells directly caused the expressed frustration. Instead, our findings indicate that the degradation of test code quality (test smells) co-occurs with negative sentiment at the moment of the commit, possibly driven by shared underlying factors such as high complexity or schedule pressure.
  Furthermore, our sentiment extraction relies on a general exploratory methodology. While recent studies propose using domain-specific pre-trained Language Models (LLMs) to accurately identify emotions in software artifacts (Dey et al., 2025), our approach serves as an initial exploratory step to establish baseline data regarding the co-occurrence of test smells and sentiments in Dart. Future work may employ these advanced models for a more comprehensive emotional analysis.

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

- **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.
  The sentiment analysis relies on a single general-purpose sentiment dictionary to classify commit messages. This dictionary may not fully capture the nuances of software engineering terminology, where words like "bug", "fix", or "kill" may have different connotations than in general language. Additionally, commit messages are often terse and lack the context needed for accurate sentiment classification. To mitigate this threat, we focused on relative comparisons between test smell types rather than absolute sentiment scores.

8 Related Work

The study by Peruma et al. Peruma et al. (2019b). is the one that most closely resembles ours. In their work, Peruma et al. conducted a large-scale study of 656 open-source Android projects that used the JUnit framework. The goal of the study was to understand the existence and distribution of test smells in this context. Their findings showed that almost all Android applications containing unit tests exhibited test smells introduced in the early stages of development, and that these smells tend to persist throughout the application's lifetime. They also observed that test smells increase as the number of unit test files grows, showing a strong positive correlation (0.90) between the volume of test smells and the volume of test files.

Compared to our study, both investigations found a high prevalence of test smells in their respective mobile ecosystems. In the Dart study, 77% of the test files were "infected" with at least one smell, while in the Android study, only 3% (21 out of 656) of the applications with unit tests did not exhibit test smells. The discrepancy in the number of analyzed projects between the two studies may account for the difference in these values.

There is a similarity in the types of the most frequent smells. In our study, the three most common test smells were Magic Number (MN) with 427,935 occurrences, Duplicate Assert (DA) with 378,527, and Assertion Roulette (AR) with 246,247. Peruma et al.'s study also found that Assertion Roulette (AR) occurred more frequently than other smells and was prevalent in more than 50% of the analyzed applications. Other highly common smells in Android included Magic Number Test, Exception Handling, Lazy Test, and Eager Test. One possible reason for the differences in these occurrences is that DNose does not detect the Lazy Test and Eager Test smells (Peruma et al., 2019b).

Regarding sentiment analysis in software engineering, several studies have explored how developer emotions and perceptions relate to software artifacts. Guzman et al. (2014) analyzed 60,425 commit comments from 90 top-rated GitHub projects to understand developer sentiment and its relationship with project characteristics, finding that Java projects tend to have more negative commit comments and that projects with more geographically distributed teams tend to have higher positive polarity. Sinha et al. (2016) conducted a large-scale analysis of developer sentiment in 2,251,585 commit logs from 28,466 GitHub projects, finding that negative sentiment was approximately 10% more prevalent than positive sentiment and observing a strong correlation between the number of files changed and the sentiment expressed in commit messages. More recently, Islam and Zibran (2018) developed SentiStrength-SE, a sentiment analysis tool specifically adapted for software engineering contexts, addressing the limitation of general-purpose sentiment dictionaries that may misclassify technical terms.

Calefato et al. (2018) presented Senti4SD, a classifier trained on Stack Overflow data to detect polarity in developer communications. Ahmed et al. (2017) proposed SentiCR, specifically designed for code review comments, achieving higher accuracy than general-purpose tools. These specialized approaches highlight the importance of domain-specific sentiment analysis in software engineering research. Our study contributes to this body of work by linking sentiment analysis directly to test code quality through test smells, an approach that, to the best of our knowledge, has not been explored in previous literature.

9 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 (Virgínio et al., 2025a), revealed an alarming presence of test smells in 77% of the test files analyzed, totaling 1,115,938 occurrences. From a project perspective, it is important to note that 68% of the analyzed projects contained at least one test smell.

The results highlight that Magic Number (427,935 occurrences), Duplicate Assert (378,527 occurrences), and Assertion Roulette (246,247 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.

The sentiment analysis in this study was conducted as an exploratory investigation using the AFINN-165 lexical dictionary developed by Nielsen (2011). This dictionary allowed us to capture the intensity of sentiments expressed in commit messages associated with test smells. Our findings revealed that the majority of commits are neutral or positive (96.8%), while only 3.2% carry negative sentiments. However, the accumulated negative sentiment scores highlight that specific test smells, such as Magic Number, Duplicate Assert, and Assertion Roulette, co-occur most frequently with negative sentiment expressions. We emphasize that these findings reflect co-occurrence patterns rather than causal relationships, serving as an initial baseline for future research in this area.

Regarding the sentiment analysis, we plan to extend this study by incorporating multiple sentiment dictionaries specifically designed for software engineering contexts. Dictionaries such as SentiStrength-SE (Islam and Zibran, 2018), Senti4SD (Calefato et al., 2018), and SentiCR (Ahmed et al., 2017) have been developed to better capture the sentiment nuances in developer communications. Furthermore, recent advances in domain-specific pre-trained Language Models for emotion detection in software artifacts (Dey et al., 2025) offer a promising direction to move beyond lexicon-based approaches and achieve a more comprehensive emotional analysis of commit messages. Using these specialized models in combination with multiple dictionaries would significantly increase the robustness and coverage of the sentiment analysis, reducing the bias introduced by relying on a single general-purpose lexicon.

Data Availability

The complete replication package for this study is publicly available on Zenodo (Virgínio et al., 2026).

Acknowledgements

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

References

Afonso, J. and Campos, J. (2023). Automatic generation of smell-free unit tests. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 9–16. IEEE.

Ahmed, T., Bosu, A., Iqbal, A., and Rahimi, S. (2017). Senticr: A customized sentiment analysis tool for code review interactions. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 106–111. IEEE.

Aljedaani, W., Mkaouer, M. W., Peruma, A., and Ludi, S. (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*, ICSE-SEET '23, page 29–39. IEEE Press.

Aljedaani, W., Peruma, A., Aljohani, A., Alotaibi, M., Mkaouer, M. W., Ouni, A., Newman, C. D., Ghallab, A., and Ludi, S. (2021). Test smell detection tools: A systematic mapping study. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, EASE '21, page 170–180, New York, NY, USA. Association for Computing Machinery.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., and Binkley, D. (2015). Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20:1052–1094.

Calefato, F., Lanubile, F., Maiorano, F., and Novielli, N. (2018). Sentiment polarity detection for software development. In *Empirical Software Engineering*, volume 23, pages 1352–1382. Springer.

Camara, B., Silva, M., Endo, A., and Vergilio, S. (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*, pages 46–54.

Campos, D., Rocha, L., and Machado, I. (2021). Developers' perception on the severity of test smells: an empirical study. In *24th Iberoamerican Conference on Software Engineering (CIbSE 2021)*, pages 192–205. Curran Associates. Also available as arXiv preprint arXiv:2107.13902.

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

De Stefano, M., Pecorelli, F., Di Nucci, D., and De Lucia, A. (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)*, pages 66–70.

Dey, K., Singh, J., Cao, H., and Palma, F. (2025). Pushing feelings: Emotion and sentiment in software commit messages. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

Ding, J., Fan, G., Yu, H., and Huang, Z. (2022). Automatic identification of high-impact bug report by product and test code quality. *International Journal of Software Engineering and Knowledge Engineering*, 32(06):893–916.

Fard, A. M. and Mesbah, A. (2013). Jsnose: Detecting javascript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125.

Fernandes, D., Machado, I., and Maciel, R. (2021). Handling test smells in python: Results from a mixed-method study. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, pages 84–89.

Fernandes, D., Machado, I., and Maciel, R. (2022). Tempy: Test smell detector for python. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, SBES '22, page 214–219, New York, NY, USA. Association for Computing Machinery.

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

Guzman, E., Azócar, D., and Li, Y. (2014). Sentiment analysis of commit comments in github: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 352–355. ACM.

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

Islam, M. R. and Zibran, M. F. (2018). Sentistrength-se: Exploiting domain specificity for improved sentiment analysis in software engineering text. In *Journal of Systems and Software*, volume 145, pages 125–146. Elsevier.

Jorge, D., Machado, P., and Andrade, W. (2021a). Investigating test smells in javascript test code. In *Anais do VI Simpósio Brasileiro de Testes de Software Sistemático e Automatizado*, page 36–45, Porto Alegre, RS, Brasil. SBC.

Jorge, D., Machado, P., and Andrade, W. (2021b). Investigating test smells in javascript test code. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing*, SAST '21, page 36–45, New York, NY, USA. Association for Computing Machinery.

Junior, N. S., Martins, L., Rocha, L., Costa, H., and Machado, I. (2021). How are test smells treated in the wild? a tale of two empirical studies. *Journal of Software Engineering Research and Development*, 9:9–1.

Junior, N. S., Rocha, L., Martins, L. A., and Machado, I. (2020). A survey on test practitioners' awareness of test smells. In *Proceedings of the XXIII Iberoamerican Conference on Software Engineering (CIbSE 2020)*, pages 462–475. Curran Associates. Also available as arXiv preprint arXiv:2003.05613.

Kaur, R., Chahal, K., and Saini, M. (2022). Analysis of factors influencing developers' sentiments in commit logs: insights from applying sentiment analysis. *e-Informatica Software Engineering Journal*, 16(1).

Kim, D. J., Chen, T.-H., and Yang, J. (2021). The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empirical Software Engineering*, 26.

Lin, B., Zampetti, F., Bavota, G., Di Penta, M., Lanza, M., and Oliveto, R. (2018). Sentiment analysis for software engineering: How far can we go? In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 94–104. ACM.

Martins, L., Bezerra, C., Costa, H., and Machado, I. (2021). Smart prediction for refactorings in the software test code. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, SBES '21, page 115–120, New York, NY, USA. Association for Computing Machinery.

Martins, L., Costa, H., and Machado, I. (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):e2532.

Miller, C., Widder, D. G., Kästner, C., and Vasilescu, B. (2019). Why do people give up flossing? a study of contributor disengagement in open source. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Society*, ICSE-SEIS '19, pages 116–127. IEEE.

Nielsen, F. Å. (2011). A new anew: Evaluation of a word list for sentiment analysis in microblogs. *arXiv preprint arXiv:1103.2903*.

Paula, E. and Bonifácio, R. (2022). Testaxe: Automatically refactoring test smells using junit 5 features. In *Anais Estendidos do XIII Congresso Brasileiro de Software: Teoria e Prática*, pages 89–98, Porto Alegre, RS, Brasil. SBC.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2019a). 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*, CASCON '19, page 193–202, USA. IBM Corp.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2019b). 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*, CASCON '19, page 193–202, USA. IBM Corp.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2020a). 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*, ESEC/FSE 2020, page 1650–1654, New York, NY, USA. Association for Computing Machinery.

Peruma, A. and Newman, C. D. (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)*, pages 525–529. IEEE.

Peruma, A., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2020b). 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*, ICSEW'20, page 350–357, New York, NY, USA. Association for Computing Machinery.

Pontillo, V., Amoroso d'Aragona, D., Pecorelli, F., Di Nucci, D., Ferrucci, F., and Palomba, F. (2024). Machine learning-based test smell detection. *Empirical Software Engineering*, 29(2):55.

Raman, N., Cao, M., Tsvetkov, Y., Kästner, C., and Vasilescu, B. (2020). Stress and burnout in open source: Toward finding, understanding, and mitigating unhealthy interactions. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '20, pages 57–60. ACM.

Santana, R., Fernandes, D., Campos, D., Soares, L., Maciel,

R., and Machado, I. (2021). Understanding practitioners' strategies to handle test smells: a multi-method study. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, pages 49–53.

Santana, R., Martins, L., Rocha, L., Virgínio, T., Cruz, A., Costa, H., and Machado, I. (2020a). Raide: a tool for assertion roulette and duplicate assert identification and refactoring. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, SBES '20, page 374–379, New York, NY, USA. Association for Computing Machinery.

Santana, R., Martins, L., Rocha, L., Virgínio, T., Cruz, A., Costa, H., and Machado, I. (2020b). Raide: a tool for assertion roulette and duplicate assert identification and refactoring. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, pages 374–379.

Santana, R., Martins, L., Virgínio, T., Rocha, L., Costa, H., and Machado, I. (2024). An empirical evaluation of raide: A semi-automated approach for test smells detection and refactoring. *Science of Computer Programming*, 231:103013.

Santana, R., Martins, L., Virgínio, T., Soares, L., Costa, H., and Machado, I. (2022). Refactoring assertion roulette and duplicate assert test smells: a controlled experiment. In *Anais do XXV Congresso Ibero-Americano em Engenharia de Software*, pages 263–277, Porto Alegre, RS, Brasil. SBC.

Sinha, V., Lazar, A., and Sharber, B. (2016). Analyzing developer sentiment in commit logs. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 520–523. ACM.

Soares, E., Aranda, M., Oliveira, N., Ribeiro, M., Gheyi, R., Souza, E., Machado, I., Santos, A., Fonseca, B., and Bonifácio, R. (2023a). Manual tests do smell! cataloging and identifying natural language test smells. In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11.

Soares, E., III, M. A., Romão, D., and Ribeiro, M. (2023b). The open catalog of test smells. Available at `https://test-smell-catalog.readthedocs.io/en/latest/index.html`.

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

Soares, E., Ribeiro, M., Gheyi, R., Amaral, G., and Santos, A. (2023c). Refactoring test smells with junit 5: Why should developers keep up-to-date? *IEEE Transactions on Software Engineering*, 49(3):1152–1170.

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., and Bacchelli, A. (2018). On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12.

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.

van Deursen, A., Moonen, L., van den Bergh, A., and Kok, G. (2001). Refactoring test code. In Marchesi, M. and Succi, G., editors, *Refactoring Test Code*. Proceedings 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001).

Veloso, V. and Hora, A. (2022). Characterizing high-quality test methods: A first empirical study. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 265–269.

Virgínio, T., Martins, L., Rocha, L., Santana, R., Cruz, A., Costa, H., and Machado, I. (2020). Jnose: Java test smell detector. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, SBES '20, page 564–569, New York, NY, USA. Association for Computing Machinery.

Virgínio, T., Ribeiro, M., and Machado, I. (2025a). Dnose: Dart test smell detector. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software*, pages 976–982, Porto Alegre, RS, Brasil. SBC.

Virgínio, T., Ribeiro, M., and Machado, I. (2025b). On the prevalence of test smells in mobile development. In *Anais do X Simpósio Brasileiro de Testes de Software Sistemático e Automatizado*, pages 84–93, Porto Alegre, RS, Brasil. SBC.

Virgínio, T., Ribeiro, M., and Machado, I. (2026). Dataset: More than one million test smells. `https://doi.org/10.5281/zenodo.18436532`. 10.5281/zenodo.18436532.

Virgínio, T., Santana, R., Martins, L. A., Soares, L. R., Costa, H., and Machado, I. (2019). On the influence of test smells on test coverage. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pages 467–471.

Wang, T., Golubev, Y., Smirnov, O., Li, J., Bryksin, T., and Ahmed, I. (2022). Pynose: a test smell detector for python. In *PyNose: a test smell detector for python*, ASE '21, page 593–605. IEEE Press.