AromaLIA: A Language-Independent Approach to Detect Test Smells

Publio Silva
Universidade Federal do Ceará (UFC) - Campus Quixadá
Quixadá, CE, Brazil
publioufc@alu.ufc.br

Carla Bezerra
Universidade Federal do Ceará (UFC) - Campus Quixadá
Quixadá, CE, Brazil
carlailane@ufc.br

Ivan Machado
Universidade Federal da Bahia (UFBA)
Salvador, BA, Brazil
ivan.machado@ufba.br

Abstract

Test smells are indicators of poor practices or suboptimal decisions made during test code development, potentially undermining maintainability and evolution. Existing solutions for detecting test smells typically support only a limited set of programming languages (often just one), thereby requiring the development of new tools for each additional language. To address this limitation, this work proposes a language-independent approach for test smell detection. We conducted a Systematic Mapping Study (SMS) comprising 117 studies published up to August 2025, through which we identified the most frequently discussed and critical test smells across multiple programming languages. Building on these findings, we designed and evaluated our approach in C#, Java, JavaScript, TypeScript, and Python, covering 10 distinct test smells. Our evaluation achieved 97% precision, 96% recall, and a 97% F1-score. Furthermore, we implemented and released a test smell detection tool based on the proposed approach.

Keywords

Test Smell, Language-Independent Approach, Test Smell Detection

1 Introduction

Implementing tests for software products is essential to ensure software quality, particularly in agile development environments where practices such as continuous delivery accelerate the transition from feature ideation to customer deployment [27]. However, similar to production code, test code is also susceptible to quality issues that can lead to additional maintenance effort and costs [8]. Test engineers may make suboptimal design decisions when writing test code due to system complexity or limited experience [20], resulting in code that is difficult to read, understand, and maintain [24]. These poor design choices are referred to as *test smells* [1].

The concept of test smells was first introduced by Deursen et al. [6], who proposed an initial catalog of 11 test smells along with corresponding refactorings. Since then, researchers have identified many additional test smells and refactorings [14], as well as developed tools to automatically detect and refactor them across various programming languages [1]. However, most of these studies and tools focus on a small number of languages, which significantly limits their applicability [1].

To address this gap, we propose a novel language-independent approach for detecting test smells, we called AromaLIA. Our approach is grounded in the observation that test code generally follows similar structural patterns across programming languages, and that test smells tend to manifest in comparable ways regardless of the language (see examples in Listings 1 and 2).

To validate our approach, we developed a tool that implements AromaLIA and evaluated its effectiveness against existing language-specific test smell detection tools. The evaluation was conducted using a pre-classified and manually validated dataset comprising 830 test smell instances spanning ten test smells widely discussed in the literature [4, 5, 7, 9, 10, 13, 17–19, 22, 26]. In addition, we assessed the effectiveness of our approach on a multilingual project. The results show that AromaLIA outperforms language-specific methods, achieving an overall precision of 97%, a recall of 96%, and an F1-score of 97%. Furthermore, in the multi-language project, our approach identified more test smell instances than the two other tools combined.

```java
public class MyTestClass {
  @Test
  public void testNoChangeWhenAddingZero() {
      int mysum = 1;
      assertEquals(1, mysum);
      mysum += 0;
      assertEquals(1, mysum);
  }
}
```

Listing 1: Duplicate Assert example in Java

```python
class TestMyClass:
  def test_no_change_when_adding_zero():
      mysum = 1
      assert mysum == 1
      mysum += 0
      assert mysum == 1
```

Listing 2: Duplicate Assert example in Python

2 Related Work

2.1 Conventional approaches

Bodea [3] presented PyTest-Smell, a tool designed to identify test smells in Python code written using the PyTest framework. The tool detects ten different types of test smells and was validated with a detection rate exceeding 90% for smelly test suites. When applied to our dataset, PyTest-Smell produced results notably inferior to those originally reported by its authors, whereas the AromaLIA-based tool outperformed PyTest-Smell in terms of precision, recall, and F1-score.

Similarly, Paul et al. [15] proposed xNose, a tool for detecting test smells in C#, which detects sixteen types of test smells and achieved an average precision of 96.97% and an average recall of 96.03%. Although xNose supports a broader range of test smells, for the nine test smells supported by both tools, the AromaLIA-based tool achieved superior precision, recall, and F1-score metrics.

Another notable contribution is TSDETECT, developed by Peruma et al. [16], which achieved an average precision of 96% and an average recall of 97% on 65 unit test files covering 19 test smells. When considering only the ten shared smells, the AromaLIA-based tool achieved higher precision, recall, and F1-scores.

2.2 Language-independent approaches

Lopes et al. [11] proposed SniffML, a pioneering language-independent tool targeting C, C++, C#, and Java. SniffML converts test code into an XML representation using srcML and applies pattern matching to detect seven test smells: *Assertion Roulette*, *Conditional Test*, *Duplicate Assert*, *Empty Test*, *Exception Handling*, *Magic Number*, and *Unknown Test*. In its evaluation, SniffML reported a precision of 97.99%, recall of 96.90%, and an F-measure of 97.44%.

Compared with SniffML, AromaLIA supports five programming languages (C#, Java, JavaScript, Python, and TypeScript), detects all seven smells covered by SniffML along with three additional ones, and relies on a LAAST extractor [2] that can be extended beyond the four languages supported by srcML.

The detection pipelines also differ substantially. SniffML searches the generated XML for textual patterns (e.g., the keyword "assert") to identify smells such as *Duplicate Assert* and *Assertion Roulette*, which may introduce false positives when identifiers contain the substring "assert" but do not represent actual assertions. In contrast, AromaLIA constructs a typed LAAST, maps tests to a language-independent model, and applies detection rules over node types, thereby reducing such false positives.

2.3 AI-based approaches

Recent studies have explored the use of Large Language Models (LLMs) for test smell detection. Lucas et al. [12] evaluated ChatGPT-4, Mistral Large, and Gemini Advanced on 30 test smells across seven programming languages, with ChatGPT-4 achieving the highest overall accuracy of 70%, although the results were limited by the small number of test cases. Similarly, Santana Jr et al. [21] investigated LLMs for both detection and refactoring tasks using GPT-4-Turbo, LLaMA 3 70B, and Gemini 1.5 Pro; Gemini achieved the highest detection accuracy, reaching 74.35% for Python and 80.32% for Java.

In contrast, our approach achieved an overall accuracy of 98%, outperforming the LLM-based approaches reported in these studies. This result suggests that a traditional, non-LLM-based approach, such as AromaLIA, can provide higher effectiveness for test smell detection.

Table 1 summarizes the conventional, language-independent, and LLM-based approaches discussed in this section, considering their supported languages, covered test smells, degree of language independence, underlying techniques, and reported accuracy. Among non-LLM tools, SniffML and AromaLIA are the only language-independent, rule-based solutions. Compared with SniffML, AromaLIA provides broader coverage in terms of both supported languages and detected smells, while achieving overall accuracy comparable to SniffML's reported results.

Table 1: Comparison of test smell detection approaches

Approach	Lang.	Smells	Lang.-indep.	Technique	Acc.
AromaLIA	5	10	Yes	LAAST + rules	98%
SniffML [11]	4	7	Yes	srcML + rules	97.4%
TSDETECT [16]	1	19	No	AST + rules	—
xNose [15]	1	16	No	AST + rules	—
PyTest-Smell [3]	1	10	No	Rules	—
LLM-based [12]	7	30	Yes	LLM	70%
LLM-based [21]	2	15	Yes	LLM	80%

3 Research Methodology

3.1 Goal and Research Questions

The primary objective of this Master's dissertation is to propose a language-independent approach for detecting test smells. The specific goals of this work are: (i) to develop a language-independent approach for test smell detection (AromaLIA), and (ii) to compare the proposed approach with existing language-specific techniques.

To guide our research and support the achievement of these goals, we formulated the following research questions:

- **RQ_1 How are test smells characterized and addressed in the literature across different programming languages and frameworks?** This question aims to understand how test smells are described and handled in various languages, providing the foundation for designing a language-independent detection approach.
- **RQ_2 How does AromaLIA compare to existing language-specific approaches for detecting test smells?** This question focuses on evaluating the effectiveness of AromaLIA and comparing it with existing language-specific approaches to determine whether a language-independent method can match or exceed their performance.

3.2 Study Steps

To answer the research questions outlined in the previous section, we followed the methodology illustrated in Figure 1. The methodology is structured into five main stages, as described below:

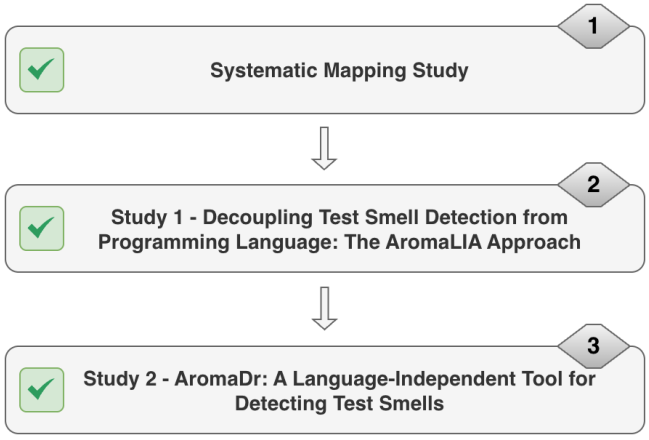

Figure 1: Overview of research methodology

To address RQ_1, we conducted a SMS titled *"Exploring Test Smells Across Programming Languages: A Systematic Mapping Study"*. This

study aimed to investigate how test smells are addressed in the literature from a programming language perspective. Specifically, we examined: (i) the programming languages and testing frameworks most commonly associated with test smells; (ii) the most prevalent test smells in each language; (iii) which test smells are language-specific versus language-agnostic; (iv) which test smells are considered most critical; and (v) the refactorings, datasets, and tools used for test smells across different languages. This review provided critical insights and laid the foundation for the subsequent phases of this work.

To answer RQ_2, we conducted a study titled *"Decoupling Test Smell Detection from Programming Language: The AromaLIA Approach"*. In this study, we introduced the *AromaLIA* approach and evaluated its performance by comparing it with existing language-specific detection methods across ten test smells in five programming languages. The results demonstrate that *AromaLIA* achieves strong detection performance, outperforming the compared approaches.

Finally, we conducted a study titled *"AromaDr: A Language-Independent Tool for Detecting Test Smells"* focused on delivering a user-friendly implementation of our approach by developing the dedicated tool AromaDr. The tool supports detection of all ten test smells across five programming languages (C#, JavaScript, TypeScript, Java, and Python). To the best of our knowledge, AromaDr currently offers the broadest programming language support among available test smell detection tools.

4 AromaLIA

Figure 2 illustrates the complete AromaLIA test smell detection process, which consists of three main steps described below.

The first step involves obtaining the test code and generating its LAAST. In our implementation, we use a tool which generates a Language-Agnostic Abstract Syntax Tree (LAAST) via Mozilla's rust-code-analysis crate [2, 23]. This tool supports multiple programming languages, including Java, JavaScript, TypeScript, Python, Go, and Rust, and can be extended to accommodate additional languages as needed.

The second step involves generating a high-level test data model from the LAAST, capturing all essential information about the tests required for test smell detection. We introduced this intermediate step instead of applying detection rules directly to the LAAST because one of our primary goals is to minimize rework when extending the detector to new programming languages. Although the LAAST provides a consistent node representation across languages, the overall tree structure can still vary depending on the language and testing framework. Applying detection rules directly to the LAAST would therefore require reimplementing the detection logic for each new language.

To address this challenge, our approach first constructs a high-level test data model from the LAAST. This model is substantially simpler than the full LAAST while preserving all metadata required for test smell detection. It is explicitly designed to be language-independent, containing no elements tied to a specific programming language. Consequently, adding support for a new language does not require reimplementing detection logic for existing test smells, thereby reducing both development and maintenance effort.

Table 2: Test smells and their detection rules (adapted from [16])

Test Smell	Detection Rule
Assertion Roulette	A test method contains more than one assertion statement without an explanation or message parameter.
Conditional Test Logic	A test method that contains one or more control statements (e.g., conditional expressions, loops).
Duplicate Assert	A test method that contains more than one assertion statement with the same parameters.
Empty Test	A test method that does not contain a single executable statement.
Exception Handling	A test method that contains either a throw statement or a catch clause.
Ignored Test	A test method or class that is marked to be ignored or disabled during test execution.
Magic Number Test	An assertion statement that contains a numeric literal as an argument.
Redundant Print	A test method that invokes output or logging statements that are not required for test verification.
Sleepy Test	A test method that invokes explicit sleep or delay operations.
Unknown Test	A test method that does not contain a single assertion statement or any explicit expected outcome specification.

In the third step, detection rules are applied to the high-level test data model to identify test smells. Table 2 summarizes the rules for the ten smells evaluated in this work. Because these rules operate on the unified model rather than on language-specific AST nodes, the same detection logic can be applied across all supported languages once the model has been extracted. Algorithm 1 illustrates one such rule. The remaining implementations are available in our replication package.

Listing 3 defines the language-independent schema of the high-level test data model. Listing 4 presents an extracted instance corresponding to the *Duplicate Assert* examples shown in Listings 1 and 2. The fields of this model were derived from the detection rules in Table 2. Specifically, the `statements` field supports *Conditional Test Logic*, *Empty Test*, and *Exception Handling*; the `events` field supports *Redundant Print* and *Sleepy Test*; the `asserts` field supports *Assertion Roulette*, *Duplicate Assert*, *Magic Number Test*, and *Unknown Test*; and the `isIgnored` field supports *Ignored Test*. For example, detecting *Assertion Roulette* requires only counting the assertions in the `asserts` field and verifying whether any lack a `message`. This intermediary representation clearly distinguishes our approach from existing test smell detection methods. A key advantage of this design is that the resulting high-level test data model is simple enough to support manual inspection, allowing researchers and developers to understand why a particular test smell was detected. Additionally, it substantially simplifies the detection algorithms compared to approaches that operate directly on the AST, thereby improving the clarity and interpretability of the detection logic for each test smell.

4.1 Worked Example: Duplicate Assert

We illustrate the three AromaLIA steps using the *Duplicate Assert* examples in Listings 1 and 2. In Step 1, `rust-code-analysis` generates a LAAST from the test source code. In Step 2, a language-specific extractor populates the high-level test data model. This

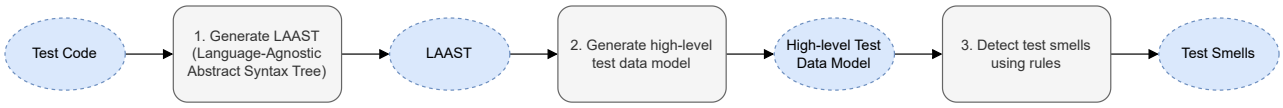

Figure 2: AromaLIA Test Smell Detection Process

is the only step that varies across languages. In Step 3, language-independent detection rules operate on this model (Algorithm 1). Because LAAST nodes are largely consistent across languages, helper functions for locating test methods, invocations, and assertions can be reused, thereby reducing the effort required to support additional languages.

```
interface TestSuiteFull {
  filePath?: string;
  name: string;
  isExclusive: boolean;
  isIgnored: boolean;
  tests: {
    name: string;
    isExclusive: boolean;
    isIgnored: boolean;
    startLine: number;
    endLine: number;
    statements: {
      type:
        | "assignment"
        | "call"
        | "condition"
        | "loop"
        | "exceptionHandling"
        | "exceptionThrowing"
        | "other";
    }[];
    events: {
      name: string;
      type: "assert" | "print" | "sleep" | "unknown";
    }[];
    asserts: {
      matcher: string;
      literalExpected?: string;
      literalActual?: string;
      message?: string;
    }[];
  }[];
}
```

Listing 3: High-level test data model (schema)

```
{
  "asserts": [
    {
      "literalActual": "mysum",
      "matcher": "==",
      "literalExpected": "1", // ...
    },
    {
      "literalActual": "mysum",
      "matcher": "==",
      "literalExpected": "1", // ...
    }
  ], // ...
}
```

Listing 4: Extracted model instance for Duplicate Assert (Java and Python)

Listing 5 presents an excerpt of the LAAST for the Java example. Listing 6 illustrates how a Java-specific extractor identifies test methods (via the `@Test` annotation) and assertions (via the `assert` method prefix), producing a model instance such as the one shown in Listing 4.

For Python (Listing 2), only Step 2 differs. The LAAST structure changes (Listing 7), and the extractor relies on `test_` function prefixes and `assert_statement` nodes, rather than JUnit annotations and `assert*` method calls (Listing 8). Steps 1 and 3 remain unchanged.

```
class_declaration (MyTestClass)
|-- modifiers: public
|-- class keyword
|-- identifier: MyTestClass
|-- class_body
    |-- method_declaration (testNoChangeWhenAddingZero)
        |-- modifiers
        |   |-- marker_annotation: @Test
        |   |-- public
        |-- return_type: void
        |-- identifier: testNoChangeWhenAddingZero
        |-- formal_parameters: ()
        |-- block
            ...
            |-- expression_statement (ASSERT #1)
            |   |-- method_invocation
            |       |-- identifier: assertEquals
            |       |-- argument_list
            |           |-- 1 (expected)
            |           |-- mysum (actual)
            ...
            |-- expression_statement (ASSERT #2)
                |-- method_invocation
                    |-- identifier: assertEquals
                    |-- argument_list
                        |-- 1 (expected)
                        |-- mysum (actual)
```

Listing 5: LAAST excerpt for Duplicate Assert (Java)

For the *Duplicate Assert* examples discussed above, Algorithm 1 implements the corresponding rule from Table 2 on the extracted model (Listings 3 and 4). The algorithm remains straightforward, as much of the underlying complexity has been abstracted in the preceding steps.

5 Results

5.1 Test Smells Across Languages (RQ_1)

In the first step of this work, we conducted a study analyzing 117 papers published between 2006 and 2025. This study identified 11 programming languages and 15 testing frameworks discussed in the literature in the context of test smells. Among these, Java was the most frequently studied language, and JUnit was the most commonly used testing framework. The use of Java has remained consistent and has steadily increased over time (see Figure 3). In recent years, however, Python has experienced significant growth, surpassing Java in 2024. When analyzing test smells by category [25], code-related test smells emerged as the most frequently studied.

5.1.1 Most Cited and Universal Test Smells. We identified 213 distinct test smells across the selected papers. Of these, 37 were observed in more than one programming language or framework (i.e., universal smells), while 176 were language-specific. The ten most frequently cited smells, based on the number of paper mentions,

are *Assertion Roulette* (63), *Eager Test* (59), *Conditional Test Logic* (48), *Duplicate Assert* (45), *Exception Handling* (42), *Magic Number* (41), *Sensitive Equality* (41), *Mystery Guest* (41), *Unknown Test* (39), and *General Fixture* (39). Among the universal smells, *Unknown Test* appears in 11 language/framework combinations, the broadest coverage, followed by *Conditional Test Logic*, *Exception Handling*, and *Magic Number*, each appearing in 10 combinations.

```
class JavaHighLevelTestDataModelExtractor {
  extract(node: LAASTNode): TestSuite[] {
    const classes = helpers.findAllClassDeclarations(node);
    return classes.map(classDeclaration => ({
      tests: this.extractTests(classDeclaration.node),
      // ...
    }));
  }

  private extractTests(node: LAASTNode): Test[] {
    const methods = helpers.findAllMethodDeclarations(node);
    return methods
      .filter(method => method.modifiers.includes('Test'))
      .map(method => ({
        asserts: this.extractAsserts(method),
        // ...
      }));
  }

  private extractAsserts(node: LAASTNode): TestAssert[] {
    const invocations = helpers.findAllMethodInvocations(node);
    return invocations
      .filter(i => i.identifier.startsWith('assert'))
      .map(i => this.extractAssertData(i));
  }

  // ...
}
```

Listing 6: High-level test data model extractor (Java)

```
class_definition (TestMyTestClass)
  |-- class keyword
  |-- identifier: TestMyTestClass
  |-- colon: :
  |-- block
      |-- function_definition (test_no_change_when_adding_zero)
          |-- def keyword
          |-- identifier: test_no_change_when_adding_zero
          |-- parameters: ()
          |-- colon: :
          |-- block
              ...
              |-- assert_statement (ASSERT #1)
              |   |-- assert keyword
              |   |-- comparison_operator
              |       |-- identifier: mysum
              |       |-- == operator
              |       |-- integer: 1
              ...
              |-- assert_statement (ASSERT #2)
                  |-- assert keyword
                  |-- comparison_operator
                      |-- identifier: mysum
                      |-- == operator
                      |-- integer: 1
```

Listing 7: LAAST excerpt for Duplicate Assert (Python)

5.1.2 The Criticality Paradox. Despite their high citation frequency, *Assertion Roulette* and *Eager Test* are generally rated as having low to moderate criticality in terms of both maintainability and developer perception. In contrast, *Sleepy Test* (including one very-high maintainability rating), *Ignored Test*, *Resource Optimism*, and *Mystery Guest* rank among the most critical for maintainability, yet have received comparatively less research attention. From a developer-perception perspective, *Magic Number* is consistently regarded as having very low criticality. This *criticality paradox* (the extensive study of less severe smells alongside the relative neglect of more impactful ones) motivates the need for language-independent detection approaches that can prioritize a broader and more practically relevant set of smells, as pursued in AromaLIA.

```
class PythonHighLevelTestDataModelExtractor {
  extract(node: LAASTNode): TestSuite[] {
    const classes = helpers.findAllClassDefinitions(node);
    return classes.map(c => ({
      tests: this.extractTests(c),
      // ...
    }));
  }

  private extractTests(node: LAASTNode): Test[] {
    const functions = helpers.findAllFunctionDefinitions(node);
    return functions
      .filter(f => f.identifier.startsWith('test_'))
      .map(f => ({
        asserts: this.extractAsserts(f),
        // ...
      }));
  }

  private extractAsserts(node: LAASTNode): TestAssert[] {
    const assertStatements =
        helpers.findAllAssertStatements(node);
    return assertStatements
      .map(assert => this.extractAssertData(assert));
  }

  // ...
}
```

Listing 8: High-level test data model extractor (Python)

Algorithm 1 Duplicate Assert detection algorithm

```
function DetectDuplicateAssert(test)
    seen ← new Set()
    for each assert in test.asserts do
        uniqueKey ← assert.literalActual + " | " + assert.matcher + " | " + assert.literalExpected
        if seen contains uniqueKey then
            return true
        end if
        seen.add(uniqueKey)
    end for
    return false
end function
```

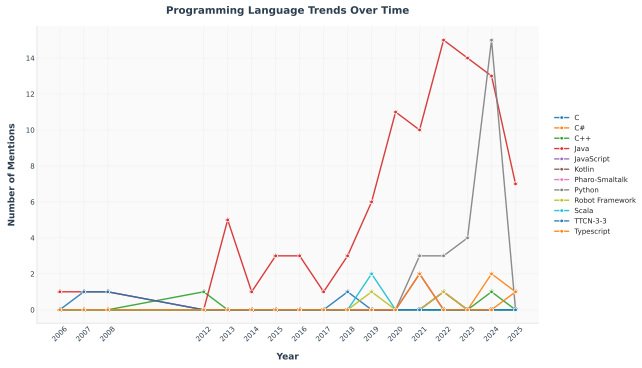

Figure 3: Language trend

5.1.3 Refactorings, Datasets, and Tools. Beyond prevalence and criticality, the SMS also cataloged supporting artifacts reported in the literature. We identified 94 distinct refactorings associated with 56 test smells, as well as 16 datasets spanning C++/GoogleTest, C#, C#/xUnit, Java, Java/JUnit, Python/Unittest, and Robot Framework, covering a total of 73 distinct test smells. In addition, we identified 36 tools for detecting or refactoring test smells across C#, Java, JavaScript, TypeScript, Python, Scala, TTCN-3, and Robot Framework; among these, 22 specifically target Java.

5.1.4 Key Takeaways. Seven findings from the SMS guide future research and motivated this work:

- **Java monoculture:** 85 of 117 papers focus on Java; Python reached eight papers in 2024 alone, surpassing Java that year.
- **Language influences smell patterns:** behavior-related smells account for 13.3% of Python occurrences, 16.7% for TypeScript, and 6.8% for Java.
- **Criticality paradox:** *Assertion Roulette* (63 mentions) and *Eager Test* (59) are the most studied, yet *Sleepy Test* ranks highest for maintainability criticality with far fewer studies.
- **Inconsistent criticality:** the same smell receives opposing ratings across papers (e.g., *Eager Test* described as both "highly harmful" and "not harmful"; *Assertion Roulette* from "irrelevant" to "medium severity").
- **Detection–refactoring imbalance:** among 36 tools, 31 support detection only and five support both detection and refactoring.
- **Limited AI adoption:** only one of the 117 papers applies LLMs to test smell refactoring.
- **Sparse multi-language support:** only one prior language-independent tool (SniffML [11]) was identified before AromaLIA.

5.2 AromaLIA Evaluation (RQ_2)

In the second step of this work, we presented our proposed approach and evaluated it against three language-specific tools: xNose for C#, TSDETECT for Java, and PyTest-Smell for Python. SniffML [11], the closest language-independent alternative, was not included in this controlled comparison because it could not be executed on our multilingual dataset. Instead, we compare it at the approach level in Table 1. The evaluation considered ten test smells. Table 3 reports the precision, recall, and F1-score for each tool. Our approach achieved an overall precision of 97.9%, recall of 96.9%, and F1-score of 97.4%. As illustrated in Figure 4, and based on 95% confidence intervals, AromaLIA's F1-score is higher than that of all compared tools.

5.2.1 Evaluation Dataset. To establish ground truth for tool comparison, we constructed a dataset of test cases sourced from GitHub. We searched for Java/JUnit and Python/PyTest test files (queries are available in our replication package), manually reviewed candidate files, and labeled occurrences of the ten test smells until obtaining at least 20 instances per smell. In total, we collected files from 105 projects (58 Java and 47 Python), resulting in 166 original test cases (84 Java and 82 Python).

To extend coverage to C#, JavaScript, and TypeScript, we used ChatGPT-4o-mini to translate the original test cases between Java and Python (each case was translated at most once). All translated code was then manually validated, and the corresponding test smell labels were re-verified. The final dataset comprises 830 manually validated test cases (166 per language). Table 4 summarizes the distribution across test smells and programming languages. The complete dataset and source files are available in the replication package.

Table 3: Precision, Recall, and F1-score for all test smell detection tools

Tool	Precision	Recall	F1-score
AromaLIA overall	97.9%	96.9%	97.4%
AromaLIA C#	97.9%	97.2%	97.6%
AromaLIA Java	97.3%	98.5%	97.9%
AromaLIA Python	98.5%	96.5%	97.5%
AromaLIA JavaScript	98.3%	97.1%	97.7%
AromaLIA TypeScript	97.8%	95.2%	96.5%
pytest-smell	82.6%	14.4%	24.5%
tsdetect	73.5%	83.9%	78.4%
xNose	83.0%	86.1%	84.5%

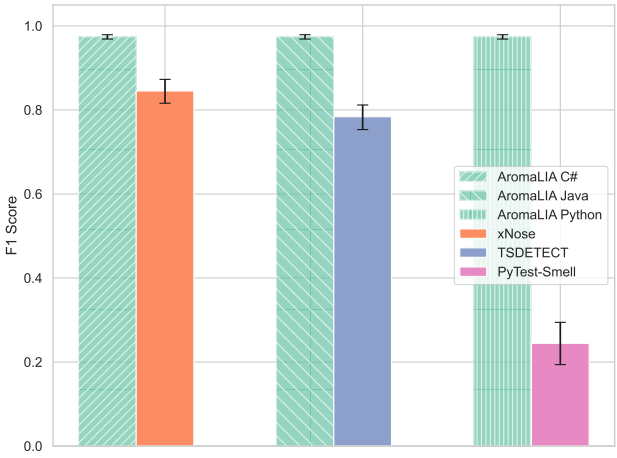

Figure 4: Comparison of F1-Score for each tool (95% CI)

Table 4: Test smell distribution and dataset size by language

Smell / Language	Python	Java	JavaScript	TypeScript	C#
Assertion Roulette	72	75	77	82	73
Conditional Test Logic	46	45	50	49	44
Duplicate Assert	28	29	28	29	30
Empty Test	23	22	22	22	22
Exception Handling	38	53	45	41	41
Ignored Test	26	23	29	29	23
Magic Number Test	46	47	49	49	47
Redundant Print	35	32	34	34	35
Sleepy Test	30	30	31	31	30
Unknown Test	53	48	53	48	48
Number of files	166	166	166	166	166

5.2.2 Performance Across Programming Languages. We examined whether AromaLIA's detection performance varied across programming languages. As shown in Table 3, the per-language F1-scores are 97.9% for Java, 97.6% for C#, 97.7% for JavaScript, 97.5% for Python, and 96.5% for TypeScript, the lowest among the five. These

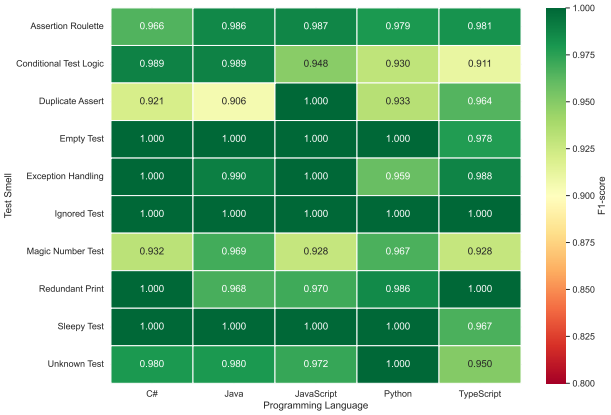

Figure 5: AromaLIA F1-score by test smell and programming language

differences are minimal, occurring only at the level of decimal precision. This stability aligns with AromaLIA's language-independent design: only Step 2 (the language-specific model extractor) varies across languages, while the detection rules remain shared.

5.2.3 Per-Smell Detection Challenges. We also investigated whether certain test smells were more difficult to detect. Figure 5 presents AromaLIA's F1-scores for each test smell across programming languages. The tool achieved a perfect F1-score (100%) for *Ignored Test* in every language. The most challenging smells were *Magic Number Test*, *Duplicate Assert*, and *Conditional Test Logic*. However, the observed differences were small and limited to minor variations in F1-scores.

Several factors help explain these localized difficulties. For *Conditional Test Logic*, detection must account for a wide range of control-flow constructs. In TypeScript, failures were often associated with `for...of` loops that the extractor did not map. In contrast, equivalent JavaScript cases used conventional `for` loops and were detected correctly, contributing to TypeScript's slightly lower overall F1-score. For *Magic Number Test*, literals may appear indirectly (e.g., in PyTest, `approx()` passes numeric values as parameters rather than as direct assertion operands). For *Duplicate Assert*, false positives occasionally arose when `rust-code-analysis` omitted string literals in the LAAST, resulting in identical empty keys for otherwise distinct assertions.

5.2.4 Multi-Language Project Evaluation. A key advantage of a language-independent approach such as AromaLIA is its applicability to multi-language projects, enabling the use of a single detection solution instead of separate tools for each language. To evaluate this advantage, we applied AromaLIA, TSDETECT (Java/JUnit), and PyTest-Smell (Python/PyTest) to the Apache Beam project [1], a public repository containing substantial test code in both languages. We measured the number of detected instances for the same ten test smells used in the evaluation. Precision and recall were not computed, as no manually validated ground truth is available for the entire project. Precision and recall were not computed, as no manually validated ground truth is available for the entire

[1]https://github.com/apache/beam

Table 5: Test smell instance counts per smell on Apache Beam (Java/JUnit and Python/PyTest)

Test Smell	AromaLIA Java	AromaLIA Python	PyTest-Smell	TSDETECT
Assertion Roulette	16,542	46	18	4,064
Conditional Test Logic	2,422	242	13	1,044
Duplicate Assert	1,488	17	1	634
Empty Test	21	0	0	9
Exception Handling	797	24	3	6,036
Ignored Test	59	0	0	193
Magic Number Test	2,585	26	4	12,924
Redundant Print	25	1	0	25
Sleepy Test	98	1	0	59
Unknown Test	6,452	913	0	4,297
Overall	30,489	1,270	39	29,285

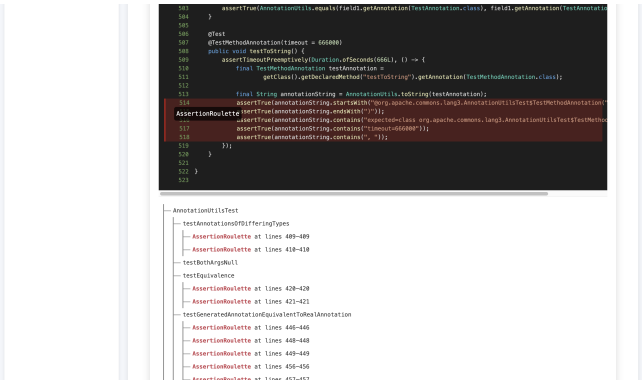

Figure 6: AromaDr Web Interface

project. Overall, AromaLIA identified 31,759 instances across both languages, compared with 29,324 detected by the language-specific tools combined. Table 5 breaks down the counts by test smell.

5.3 AromaDr Tool

In the third study of this work, we enhanced the evaluation prototype into a dedicated tool, AromaDr, with a focus on user experience. AromaDr comprises the Web App, the AromaDr API, and the rust-code-analysis API for LAAST extraction. The tool supports file mode (single test file) and project mode (public GitHub repository).

AromaDr implements the AromaLIA approach and provides a graphical user interface, representing a significant advantage over many existing tools that rely solely on command-line interfaces. Figure 6 shows the web interface in project mode, where users provide a public GitHub repository URL and receive detection results across all test files, with smells and corresponding line numbers highlighted for each file.

Moreover, AromaDr supports the detection of ten test smells across five programming languages, which, to the best of our knowledge, makes it the test smell detection tool with the broadest language support currently available. AromaDr is capable of reporting the exact line of code where a test smell occurs, whereas several existing tools only indicate the file in which the test smell is located.

Beyond its web-based graphical interface, AromaDr also offers integration through a REST API, enabling seamless incorporation

into CI/CD pipelines. Another key advantage of AromaDr is its minimal setup requirements: the tool's only dependency is Docker, as it runs entirely within Docker containers, which enhances platform independence and simplifies deployment across environments.

6 Discussion and Impact

This work makes several contributions of interest to both researchers and practitioners, as outlined below.

Foundation for Future Research. A key contribution is the analysis of the prevalence of test smells across programming languages, conducted in the first step of this study. This analysis provides a solid foundation for future research, particularly for incorporating understudied programming languages into the test smell research landscape. In the same study, we compiled several comprehensive catalogs: (i) a catalog of test smells; (ii) a catalog of test smell refactorings; (iii) a catalog of test smell datasets; and (iv) a catalog of tools for detecting and refactoring test smells. Together, these catalogs constitute valuable resources for the research community.

For example, the test smell catalog distinguishes between universal and language-specific test smells, enabling future studies to investigate whether language-specific smells are genuinely exclusive to certain languages or can also occur in others. Another significant contribution is the catalog of test smell datasets. Creating such datasets from scratch is a time-consuming task. Therefore, the availability of ready-to-use datasets substantially lowers the barrier for empirical research. In total, we reported 16 test smell datasets spanning five programming languages. Similarly, the catalog of tools is beneficial for both academic and industrial contexts, supporting comparative studies on test smell detection and refactoring, and helping practitioners select tools that best fit their project environments. Finally, we introduced a multi-perspective classification of test smell criticality. This classification enables researchers to address a key limitation in existing work, namely, the *criticality paradox*, which describes the tendency of prior research to focus predominantly on less critical test smells.

Language-Independent Detection Approach. The primary contribution of this work is AromaLIA, a language-independent approach for detecting test smells. AromaLIA demonstrates the ability to identify test smells across multiple programming languages while achieving high precision, recall, and F1-score values. As part of the evaluation of AromaLIA, we constructed a multi-language test smell dataset, which we made publicly available to support future research and replication efforts.

Test Smell Detection Tooling. Finally, we introduced AromaDr, a tool that implements the AromaLIA approach and supports the detection of ten test smells across five programming languages. AromaDr can be used in future research as a baseline for comparison with other test smell detection solutions. In addition, it is suitable for industrial adoption, as it was designed with ease of use in mind and supports seamless integration into CI/CD pipelines via a REST API.

7 Conclusion

In this work, we propose AromaLIA, a language-independent approach for detecting test smells. We validated our approach by developing an AromaLIA-based tool and comparing it with existing test smell detection tools for C#, Java, and Python. The effectiveness of the tools was evaluated using a manually validated, pre-classified dataset containing 830 instances of test code encompassing 10 types of test smells. The AromaLIA-based tool achieved a precision of 97%, a recall of 96%, and an F1-score of 97%, outperforming all language-specific tools.

Our approach lays the foundation for test smell detection solutions that are more reusable and broadly applicable, thereby facilitating the integration of new languages. This approach is a significant contribution for both researchers and developers because it not only enables test smell detection on code written in different programming languages using a single unified solution, but also simplifies the process of adding support for new languages by significantly reducing the required rework.

Artifact Availability

The replication packages for the three studies in this work are publicly available:

- **SMS: Exploring Test Smells Across Programming Languages**: https://zenodo.org/records/17093675
- **AromaLIA Evaluation** (dataset, scripts, and detection rules): https://github.com/publiosilva/aroma-lia-evaluation-rp
- **AromaDr: A Language-Independent Tool for Detecting Test Smells**: https://github.com/publiosilva/aromadr

Acknowledgements

The authors gratefully acknowledge the financial support provided for this research. Ivan Machado acknowledges partial support from the CAPES – Finance Code 001; Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB), grant PIE0002/2022; and CNPq, grants 315840/2023-4 and 403361/2023-0. Carla Bezerra is supported by 403304/2025-3.

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work the author(s) used ChatGPT in order to correct any syntax and grammatical errors in the text. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

References

[1] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D. Newman, Abdullatif Ghallab, and Stephanie Ludi. 2021. Test Smell Detection Tools: A Systematic Mapping Study. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering* (Trondheim, Norway) *(EASE '21)*. Association for Computing Machinery, New York, NY, USA, 170–180. doi:10.1145/3463274.3463335

[2] Luca Ardito, Luca Barbato, Marco Castelluccio, Riccardo Coppola, Calixte Denizet, Sylvestre Ledru, and Michele Valsesia. 2020. rust-code-analysis: A Rust Library to Analyze and Extract Maintainability Information from Source Codes. *SoftwareX* 12 (2020), 100635. doi:10.1016/j.softx.2020.100635

[3] Alexandru Bodea. 2022. Pytest-Smell: A Smell Detection Tool for Python Unit Tests. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 793–796. doi:10.1145/3533767.3543290

[4] Denivan Campos, Luana Martins, and Ivan Machado. 2023. An Empirical Study on the Influence of Developers' Experience on Software Test Code Quality. In

Proceedings of the 21st Brazilian Symposium on Software Quality (Curitiba, Brazil) *(SBQS '23)*. Association for Computing Machinery, New York, NY, USA, Article 3, 10 pages. doi:10.1145/3571473.3571481

[5] Zhongyan Chen, Suzanne M. Embury, and Markel Vigo. 2023. Who Is Afraid of Test Smells? Assessing Technical Debt from Developer Actions. In *Testing Software and Systems*. Springer, Cham, Switzerland, 160–175.

[6] Arie Deursen, Leon M.F. Moonen, A. Bergh, and Gerard Kok. 2001. *Refactoring Test Code*. Technical Report. CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands.

[7] Anna Rita Fasolino and Porfirio Tramontana. 2024. Test Smells Learning by a Gamification Approach. In *Proceedings of the 3rd ACM International Workshop on Gamification in Software Development, Verification, and Validation* (Vienna, Austria) *(Gamify 2024)*. Association for Computing Machinery, New York, NY, USA, 30–33. doi:10.1145/3678869.3685687

[8] Vahid Garousi and Barış Küçük. 2018. Smells in Software Test Code: A Survey of Knowledge in Industry and Academia. *Journal of Systems and Software* 138 (2018), 52–81. doi:10.1016/j.jss.2017.12.013

[9] Ayesha Kiran, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A Comprehensive Investigation of Modern Test Suite Optimization Trends, Tools and Techniques. *IEEE Access* 7 (2019), 89093–89117. doi:10.1109/ACCESS.2019.2926384

[10] Xiangjun Liu and Ping Yu. 2022. Randoop-TSR: Random-based Test Generator with Test Suite Reduction. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware* (Hohhot, China) *(Internetware '22)*. Association for Computing Machinery, New York, NY, USA, 221–230. doi:10.1145/3545258.3545280

[11] Gustavo Lopes, Davi Romão, Elvys Soares, Márcio Ribeiro, Guilherme Amaral, Rohit Gheyi, and Ivan Machado. 2024. A Road to Find Them All: Towards an Agnostic Strategy for Test Smell Detection. In *Proceedings of the 23rd Brazilian Symposium on Software Quality* (Brasília, Brazil) *(SBQS '24)*. Association for Computing Machinery, New York, NY, USA, 231–241. doi:10.1145/3701625.3701662

[12] Keila Lucas, Rohit Gheyi, Elvys Soares, Márcio Ribeiro, and Ivan Machado. 2024. Evaluating Large Language Models in Detecting Test Smells. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software*. SBC, Porto Alegre, RS, Brasil, 672–678. doi:10.5753/sbes.2024.3642

[13] Luana Martins, Taher A. Ghaleb, Heitor Costa, and Ivan Machado. 2024. A Comprehensive Catalog of Refactoring Strategies to Handle Test Smells in Java-based Systems. *Software Quality Journal* 32, 2 (2024), 641–679. doi:10.1007/s11219-024-09663-7

[14] Gerard Meszaros. 2007. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, Boston, MA, USA.

[15] Partha P. Paul, Md Tonoy Akanda, M. Raihan Ullah, Dipto Mondal, Nazia S. Chowdhury, and Fazle M. Tawsif. [n. d.]. xNose: A Test Smell Detector for C#.

[16] 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. doi:10.1145/3368089.3417921

[17] Anthony Peruma and Christian D. Newman. 2021. On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study. In *Proceedings of the 18th International Conference on Mining Software Repositories (MSR '21)*. IEEE, 525–529. doi:10.1109/MSR52588.2021.00067

[18] 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. doi:10.1145/3387940.3392189

[19] Railana Santana, Luana Martins, Tássio Virgínio, Larissa Rocha, 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 (CIBSE)*. SBC, Porto Alegre, RS, Brasil, 263–277. doi:10.5753/cibse.2022.20977

[20] 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 (Jan. 2024). doi:10.1016/j.scico.2023.103013

[21] E. G. Santana Jr, Jander Pereira Santos Junior, Erlon P. Almeida, Iftekhar Ahmed, Paulo Anselmo da Mota Silveira Neto, and Eduardo Santana de Almeida. 2025. Evaluating LLMs Effectiveness in Detecting and Correcting Test Smells: An Empirical Study. arXiv:2506.07594 [cs.SE] https://arxiv.org/abs/2506.07594

[22] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[23] Micah Schiewe, Jacob Curtis, Vincent Bushong, and Tomas Cerny. 2022. Advancing Static Code Analysis With Language-Agnostic Component Identification. *IEEE Access* 10 (2022), 30743–30761. doi:10.1109/ACCESS.2022.3160485

[24] 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, 1 (Sept. 2021), 9:1–9:16. doi:10.5753/jserd.2021.1802

[25] 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.

[26] 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. doi:10.1109/TSE.2022.3172654

[27] Huynh Khanh Vi Tran, Michael Unterkalmsteiner, Jürgen Börstler, and Nauman bin Ali. 2021. Assessing Test Artifact Quality—A Tertiary Study. *Information and Software Technology* 139 (2021), 106620. doi:10.1016/j.infsof.2021.106620