Toward a Language-Agnostic Approach to Detect Test Smells

Publio Blenilio Tavares Silva
publioufc@alu.ufc.br
Federal University of Ceará
Quixadá, Brazil

Carla Bezerra
carlailane@ufc.br
Federal University of Ceará
Quixadá, Brazil

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

ABSTRACT

Tests play a crucial role in software development by ensuring code quality. However, test code can suffer from "smells" — poor implementation choices that hinder maintainability and evolution. Numerous studies have addressed test smells in various programming languages, proposing tools for detecting them in Java, C++, Scala, and others. These tools employ techniques such as information retrieval, metrics analysis, and abstract syntax tree (AST) parsing. However, their focus on specific languages limits their generalizability and applicability to other languages and test frameworks. This challenge is similar to issues found in code smell detection and static code analysis. Therefore, this work proposes a language-agnostic approach to detect test smells. Our approach leverages AST parsing to extract relevant information from the test code, followed by test smell detection based on this extracted data. This method aims to facilitate the detection of test smells across various programming languages and test frameworks, enhancing the tool's generalizability and usability. To check the viability of our approach, we created a proof of concept using two test smells and two different languages.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **General programming languages**; • **General and reference** → **Empirical studies**.

KEYWORDS

Test smell detection, language-agnostic

1 INTRODUCTION

One of the key practices to ensure the quality of code in software development is testing [23]. It can be either manual tests that have to be conducted by a human or automated tests consisting of test scripts that provide repeatability and require less test effort [10]. In organizations that adopt continuous delivery, it is necessary to create automated tests to shorten the time between the ideation of a feature and its delivery to the final customers while ensuring its quality [23].

However, as desirable as evolving the production code is, it is equally important to evolve the test code to remain useful over time [23]. Poorly implemented test code can cause undesirable extra costs and effort [10]. Neglecting the importance of developing good tests can be harmful since developers spend around one-quarter of their time on engineering test code [5]. One of the factors that can impact the capability to maintain and evolve test code is test smells. Test smells are developers' poor choices in implementing test code [1]. These choices, among other things, can lead to a test code that is hard to maintain and evolve [17].

Table 1: Test smell detection tools and their supported programming languages [1]

Tool	Supported Language
DARTS	Java
DrTest	Pharo
DTDetector	Java
EletricTest	Java
JNose Test	Java
OraclePolish	Java
PolDet	Java
PraDeT	Java
RAIDE	Java
RTj	Java
SoCRATES	Scala
Taste	Java
TeCReVis	Java
TEDD	Java
TeReDetect	Java
TestEvoHound	Java
TestHound	Java
TestLint	Smalltalk
TestQ	C++, Java
TRex	Java
TSDETECT	Java

The initial catalog of test smells proposed by Deursen et al. [8] consists of 11 test smells and their suggested refactorings. Since then, numerous studies have addressed test smells by proposing strategies to improve detection and refactoring in a range of programming languages and platforms, with Java being the most common one [1]. However, many of these studies have focused on only a few languages, which restricts the generalization of results to other languages or frameworks. Table 1, adapted from Aljedaani et al. [1], displays a list of test smell detection tools and their supported languages. As shown in the table, most tools support only Java. Additionally, all tools support a single language except for TestQ, which supports two languages.

Researchers have tackled similar problems in related fields like code smell detection and static code analysis by proposing language-agnostic approaches. For instance, Ducasse et al. [9] introduced a language-independent and visual method for detecting code duplication. Although code smells and test smells are related concepts, language-agnostic approaches to detecting code smells cannot be effectively scaled for detecting test smells. This is because test smells are specific to test code and involve unique practices that general detectors may not accurately identify. Schiewe et al. [21] focused on static code analysis and presented a language-agnostic approach to identify components. Their approach involves converting the source code into an intermediary format called a language-agnostic abstract-syntax tree and using parsers to extract high-level information.

Hence, the current study aims to tackle the challenge of language-agnostic detection of test smells. To achieve this goal, we introduce an approach that leverages the language-agnostic abstract-syntax tree outlined by Schiewe et al. [21] to extract pertinent details about tests, such as their names, test asserts, and assert messages, using parsers. Additionally, we establish language-agnostic test smell detectors that utilize the extracted information from these parsers. To check the viability of this approach, we developed a proof of concept that detects the test smells Assertion Roulette and Duplicate Assert in test code written in Java and JavaScript.

The structure of the work is as follows: Section 2 introduces the concepts and related work relevant to this study. Section 3 outlines the research objectives and the methodologies employed to conduct the study. Section 4 presents the findings of the study. In Section 5, the implications of the study findings are discussed. Section 6 explores potential threats to validity of the study. Finally, Section 7 summarizes the findings and outlines avenues for future research.

2 BACKGROUND

2.1 Test smells

The concept of code smells, akin to test smells, denotes sub-optimal coding practices leading to potential issues in maintainability. While not indicative of functional failure, they signal potential quality problems [17]. Test smells, like code smells, highlight sub-optimal engineering decisions in test code, impacting maintainability and evolution, crucial in environments like continuous delivery [23].

Studies, including the seminal work by Deursen et al. [8], introduced a catalog of 11 test smells with associated refactorings. This catalog has expanded over time, incorporating new test smells, including language-specific ones [1]. Additionally, research has yielded various tools for detecting test smells in different programming languages like Java, C++, Smalltalk, and Scala, though mostly focused on Java [1].

2.2 Test smell detection techniques

The literature presents several techniques for identifying test smells. Notable among these are Metrics, Rules/Heuristics, Information Retrieval, and Dynamic Tainting methodologies.

Metrics: Measure symptom impact using structural and semantic metrics, defining thresholds to identify smells. Tools like those by Van Rompaey et al. [24] detect General Fixture and Eager Test using metrics like Number of Objects Used in setup.

Rules/Heuristic: Supplement metrics with predefined patterns in source code. For instance, Assertion Roulette is detected by examining assertion statements. TSDETECT [18] employs this technique applied to the AST, excelling in detection and popularity.

Information Retrieval: Extract and normalize test code information, applying preprocessing steps and machine learning to discern textual features. Tools such as those proposed by Lambiase et al. [15] leverage this technique effectively.

Dynamic Tainting: Monitor code during execution, analyzing runtime data with predefined taint values. This technique is used in the work of Zhang et al. [28].

Rules/Heuristic-based detection is common due to well-defined rules, while Metrics-based detection is less frequent due to metric limitations. Dynamic-based detection is used for test dependency and rotten green tests. Information Retrieval techniques are utilized but may suffer from feature absence.

2.3 Related Work

Taniguchi et al. [22] introduces JTDog[1], a Gradle plugin designed to detect dynamic smells in test code, including rotten green tests, flaky tests, and dependent tests. Unlike static smell detection, dynamic smell detection requires dynamic analysis and additional information such as test execution and coverage data. JTDog aims to address the reduced portability of dynamic smell detection tools by integrating them into the Gradle build tool, making it more accessible and user-friendly. The effectiveness of JTDog is demonstrated by its successful application to 150 projects on GitHub, where it accurately detected dynamic smells in a significant portion of the projects.

Wang et al. [26] aimed to address the existing gap in understanding test smells in dynamically typed languages like Python. They curated a list of 17 diverse test smells and identified 18 additional Python-specific test smells through empirical analysis. They developed PYNose, a PyCharm plugin, to detect these smells in Python projects utilizing the standard Unittest framework. An empirical study on 248 Python projects revealed that test smells are prevalent, with most projects and test suites exhibiting at least one smell. The paper provides valuable insights into the prevalence and detection of test smells in Python code, offering researchers and practitioners a tool to improve the quality of test code.

Peruma et al. [18] proposed the TSDETECT tool that uses rules as a technique for detecting test smells. These rules are applied not directly to the source code but to an AST generated from the source code. By applying these rules, it is identified whether there are test smells in the code and which test smells exist. The tool was validated with a benchmark containing 65 unit test files for 19 different types of test smells. The results demonstrated that the tool has high accuracy, achieving average precision and recall above 95%. Furthermore, in the work of Aljedaani et al. [1], the tool is mentioned as one of the most popular ones, having a high number of forks in its GitHub project.

Ducasse et al. [9] discusses the prevalent issue of duplicated code in software systems, attributing its occurrence to factors such as time constraints, performance evaluation criteria, and efficiency considerations. Despite its commonality, code duplication is widely regarded as a bad practice due to its detrimental effects on maintenance, code size, and design quality. While techniques and tools for detecting duplicated code exist, the need for language-specific parsers hinders their practical application in industrial contexts. The authors propose a language-independent approach for analyzing duplication based on string matching, textual reports, and scatter plot visualizations. The effectiveness of this approach is demonstrated through case studies conducted in different programming languages.

Rakić et al. [19] introduces the Set of Software Quality Static Analyzers (SSQSA), a framework comprising various static analysis

[1]https://plugins.gradle.org/plugin/com.github.m-tanigt.jtdog

tools to improve software quality. A key feature of these tools is their language independence, allowing them to be applied uniformly to software systems written in different programming languages. This characteristic enhances the generality and applicability of the tools, enabling consistent analysis across diverse software products. The framework utilizes an enriched Concrete Syntax Tree (eCST) as an intermediate representation of source code to achieve language independence. Currently, SSQSA supports six programming languages, including Java, C#, and COBOL, with the capability to easily add support for new languages. The paper details the framework's architecture, focusing on the eCST Generator component for generating language-independent representations. It also describes three fully functional tools developed within the framework: SMIILE for software metrics, SNEIPL for software network extraction, and SSCA for software structure change analysis. Emphasis is placed on SNEIPL, which is in intensive development.

Schiewe et al. [21] employed advancing static code analysis to meet the demands of modern software development, emphasizing its key role in automating tasks like code reviews, security assessments, and error detection. The authors highlight the inadequacies of traditional methods in recognizing high-level components and architectural issues due to their focus on low-level constructs and lack of platform independence. To tackle these challenges, the study proposes a two-step approach: first, parsing code into a Language-Agnostic Abstract-Syntax Tree (LAAST), and then employing Relative Static Structure Analyzers (ReSSA) to identify specific component types using generalized parsers. Based on the platform and system-specific requirements, these parsers can be tailored for better precision. Evaluating microservice testbeds, DeathStarBench, and TrainTicket demonstrates promising results in identifying components across diverse structures.

In our research, we draw insights from various related works. While Taniguchi et al. [22], Wang et al. [26] and Peruma et al. [18] focus on Java and Python code, we aimed to adopt a language-agnostic approach. Similarly, Ducasse et al. [9], Rakić et al. [19], and Schiewe et al. [21] have developed language-agnostic methodologies, albeit primarily for code smell detection and static code analysis. In our study, we introduce a language-agnostic approach tailored specifically for detecting test smells, leveraging the same language-agnostic AST utilized in the work of Schiewe et al. [21].

3 PROPOSED APPROACH (AROMALIA)

3.1 Goal

Our primary objective is to introduce a language-agnostic approach for detecting test smells, henceforth known as AromaLIA. This approach has three key requirements: (i) it should be adaptable to accommodate various languages or frameworks as needed, (ii) the process for detecting test smells must remain consistent across different languages and test frameworks. This ensures that the implementation of the test smell detection process remains unchanged when incorporating a new language, and (iii) any components within the final approach that are specific to a particular language should be designed for extensibility, allowing for integration with as many languages as possible.

3.2 Study Steps

This section describes the steps of the study to support research.

Step 1: Analysis of similar studies to select the techniques to be used. We searched related works in the literature with two main focuses:

- **Works dealing with test-smell detection.** From these works, we could identify which techniques can be used for test smell detection.
- **Works proposing language-agnostic code analysis solutions.** From these works, we could identify which techniques can be used to create language-agnostic code analysis solutions. Additionally, we could identify intersections with what was identified in the previous topic.

We found several works addressing the detection of test smells [3, 4, 6, 7, 11–16, 18, 20, 25, 28]. These works use some of the test smell detection techniques described in Section 2.2. We also found some studies dealing with solutions for language-independent code analysis [9, 21]. These studies addressed issues such as language-independent code smell detection and language-independent static code analysis. From the analysis of these studies, we identified an intersection between the works of Peruma et al. [18] and Schiewe et al. [21] regarding the two focuses mentioned earlier, which is their use of abstract syntax trees (ASTs) for detecting test smells and performing static code analysis, respectively. Schiewe et al. [21] extends this by employing a language-agnostic AST to achieve broader applicability across different programming languages.

Based on the studies, we decided to use the same strategy as the work of Peruma et al. [18] with the use of rules to identify test smells from an AST generated from the source code. However, we opted to use the same type of language-agnostic AST used in Schiewe et al. [21] so that the final solution for test smell detection would also be language-independent. These choices satisfy the first requirement described in Section 3.1, as the chosen AST, in addition to being language-agnostic, was based on Mozilla's rust-code-analysis crate [2], which already supports various languages such as Java, JavaScript, TypeScript, Python, Go, and Rust, and can be extended to other languages.

Step 2: Code architecture design to meet requirements. Next, we designed a solution combining features from both studies of Peruma et al. [18] and Schiewe et al. [21]. Figure 1 shows a high-level architecture of the proposed solution.

As Figure 1 shows, the first step of the solution involves using the LAAST parser to convert the source code into a language-agnostic AST. Next, the AST is used by an AST Test Extractor, a component responsible for extracting relevant information about the tests from the AST and providing them in a standardized format.

An important point to highlight is that each language's tests follow a specific structure. Moreover, sometimes, within the same language, the test structure can vary depending on the testing framework used. Therefore, there will be an implementation of the AST Test Extractor component for each language or framework for which test smell detection is desired. However, the extracted test data format will always follow the same interface.

After extracting relevant test data from the AST into a standardized format, the data is passed to Test Smell Detectors. These components detect whether a given test exhibits a certain test smell.

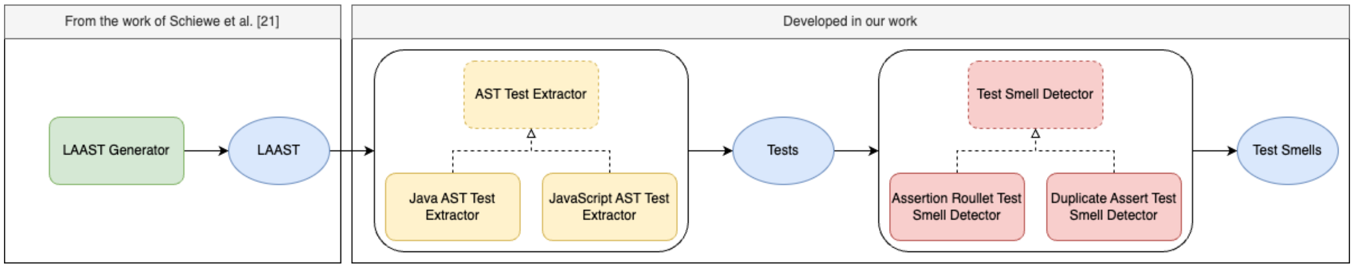
From the work of Schiewe et al. [21]
Developed in our work
LAAST Generator
LAAST
AST Test Extractor
Java AST Test Extractor
JavaScript AST Test Extractor
Tests
Test Smell Detector
Assertion Roullet Test Smell Detector
Duplicate Assert Test Smell Detector
Test Smells

Figure 1: High-level architecture of the proposed solution

Hence, the Test Smell Detector component will be implemented for each test smell desired to be detected. An important point is that implementing a Test Smell Detector will be language-agnostic because the previous step ensures that the test data received by the Test Smell Detector will always be in a standardized format.

With this architecture, we can meet requirements 2 and 3 described in Section 3.1, as the implementation of test smell detection will be consistent regardless of the language in which the tests were written, and step two, which requires specific implementation for each language/framework, was designed to allow extension to as many languages as possible.

4 PROOF OF CONCEPT OF AROMALIA

We created a POC implementing the architecture mentioned earlier using the TypeScript language. The first step in creating the POC was to define the types and interfaces described by the architecture mentioned earlier. Listing 1 shows the interface of ASTTestExtractor. The interface contains only the extract method, which takes the source code's AST as input and returns a list of tests obtained through the AST.

```
interface ASTTestExtractor {
  extract(ast: ASTProgram): Test[];
}
```

Listing 1: ASTTestExtractor interface

Listing 2 shows the type returned by the extract method of ASTTestExtractor. The interface contains the test case name, its assertions, and its annotations. Each assertion contains the actual value, the expected value, the combinator used, and the assertion's explanation message. Depending on the type of combinator, some of these values may be undefined. It is important to emphasize that this type was specifically made for the context of this work. This interface contains the information we need to detect the two selected test smells. To detect other test smells, we will likely need to expand this interface to accommodate data necessary for the detection of other types of test smells.

Listing 3 shows the interface of TestSmellDetector. The interface contains only the isPresent method, which takes a test in the format specified in Listing 2 and returns a boolean value indicating whether the received test contains a certain type of test smell or not.

We chose to apply the POC to two widely used programming languages and testing frameworks, namely Java with JUnit and JavaScript with Jest. The Java language was choosen because how it is possible to see on Table 1 is the language supported by the majority of the existing test smell detection tools and it can be useful in the future to compare our tool with existing ones. On the other hand, the JavaScript language was choosen because it is not a language commonly approached in the test smell detection context which brings some novelty. We also selected two test smells whose detection process is relatively straightforward, namely Duplicate Assert and Assertion Roulette. Detecting the Duplicate Assert test smell involves searching for test cases with more than one assertion statement with the same parameters. Detecting the Assertion Roulette test smell involves searching for test cases with more than one assertion statement, where at least one does not have a message or explanation.

```
interface Test {
  name: string;
  asserts: TestAssert[];
  annotations?: TestAnnotation[];
}

interface TestAssert {
  literalActual?: string;
  matcher?: string;
  literalExpected?: string;
  message?: string;
}

interface TestAnnotation {
  name: string;
  value?: string;
}
```

Listing 2: Test interface

```
interface TestSmellDetector {
  isPresent(test: Test): boolean
}
```

Listing 3: TestSmellDetector interface

To test the POC, we selected a simple example of the smells Duplicate Assert and Assertion Roulette for Java and JavaScript. The example of Duplicate Assert for Java and JavaScript can be seen in Listing 4 and Listing 5, respectively. The example of Assertion Roulette for Java and JavaScript can be seen in Listing 6 and Listing 7, respectively.

Something that becomes clear with these examples is what we mentioned in Section 3: even though the test cases are the same in both languages, the way the tests are structured varies significantly. For example, in Java with JUnit, the name of a test case is defined by the method's name in the test class (Listing 4). In contrast, in Java with Jest, the name of the test case is defined by the first parameter of the test function (Listing 5). Another example is with the assertions. In Java with JUnit, the expected value and the actual value in an assert are generally defined as the first and second parameters of an assert function (Listing 6). In contrast, in JavaScript with Jest, the actual value is defined in the expect function, and the expected value is defined in the toBe function or an equivalent function that is chained to the expect function call (Listing 7).

```java
@Test
public void testXmlSanitizer() {
  boolean valid = XmlSanitizer.isValid("Fritzbox");
  assertEquals("Fritzbox is valid", true, valid);
  System.out.println("Pure ASCII test - passed");

  valid = XmlSanitizer.isValid("Fritz Box");
  assertEquals("Spaces are valid", true, valid);
  System.out.println("Spaces test - passed");

  valid = XmlSanitizer.isValid("Frutzbux");
  assertEquals("Frutzbux is invalid", false, valid);
  System.out.println("No ASCII test - passed");

  valid = XmlSanitizer.isValid("Fritz!box");
  assertEquals("Exclamation mark is valid", true, valid);
  System.out.println("Exclamation mark test - passed");

  valid = XmlSanitizer.isValid("Fritz.box");
  assertEquals("Exclamation mark is valid", true, valid);
  System.out.println("Dot test - passed");

  valid = XmlSanitizer.isValid("Fritz-box");
  assertEquals("Minus is valid", true, valid);
  System.out.println("Minus test - passed");

  valid = XmlSanitizer.isValid("Fritz-box");
  assertEquals("Minus is valid", true, valid);
  System.out.println("Minus test - passed");
}
```

Listing 4: Duplicate Assert Java example

```javascript
test('testXmlSanitizer', () => {
  let valid = XmlSanitizer.isValid('Fritzbox');
  expect(valid).toBe(true);
  console.log('Pure ASCII test - passed');

  valid = XmlSanitizer.isValid('Fritz Box');
  expect(valid).toBe(true);
  console.log('Spaces test - passed');

  valid = XmlSanitizer.isValid('Frutzbux');
  expect(valid).toBe(false);
  console.log('No ASCII test - passed');

  valid = XmlSanitizer.isValid('Fritz!box');
  expect(valid).toBe(true);
  console.log('Exclamation mark test - passed');

  valid = XmlSanitizer.isValid('Fritz.box');
  expect(valid).toBe(true);
  console.log('Dot test - passed');

  valid = XmlSanitizer.isValid('Fritz-box');
  expect(valid).toBe(true);
  console.log('Minus test - passed');

  valid = XmlSanitizer.isValid('Fritz-box');
  expect(valid).toBe(true);
  console.log('Minus test - passed');
});
```

Listing 5: Duplicate Assert JavaScript example

We created and made available on GitHub a project[2] containing examples of smelly tests in Java and JavaScript, the ASTs of both codes, implementations of the ASTTextExtractor interface for Java and JavaScript, implementations of the TestSmellDetector interface for the Duplicate Assert and Assertion Roulette test smells, and ready-to-use examples of using these classes. To test the project, simply clone the GitHub repository and follow the instructions in the README.md file.

```java
@Test
public void testCloneNonBareRepoFromLocalTestServer()
    throws Exception {
  Clone cloneOp = new Clone(
    false,
    integrationGitServerURIFor("small-repo.early.git"),
    helper().newFolder()
  );

  Repository repo = executeAndWaitFor(cloneOp);

  assertThat(
    repo,
    hasGitObject("
        ba1f63e4430bff267d112b1e8afc1d6294db0ccc")
  );

  File readmeFile = new File(repo.getWorkTree(), "README"
      );
  assertThat(readmeFile, exists());
  assertThat(readmeFile.length(), equalTo(12L));
}
```

Listing 6: Assertion Roulette Java example

```javascript
test('testCloneNonBareRepoFromLocalTestServer', async ()
    => {
  const cloneOp = new Clone(
    false,
    integrationGitServerURIFor('small-repo.early.git'),
    helper().newFolder()
  );
  const repo = await executeAndWaitFor(cloneOp);

  expect(repo).toHaveGitObject('
      ba1f63e4430bff267d112b1e8afc1d6294db0ccc');

  const readmeFile = new File(repo.getWorkTree(), 'README
      ');
  expect(readmeFile.exists()).toBe(true);
  expect(readmeFile.length).toBe(12);
});
```

Listing 7: Assertion Roulette JavaScript example

5 DISCUSSION

The outcomes of this study provided valuable insights into the methodologies employed in detecting test smells within similar studies. We noted a disparity between this domain and other fields like static code analysis and code smell detection, where language-agnostic approaches are more prevalent. This discrepancy underscores the need for further exploration and development within the test smell detection domain, particularly regarding language-agnostic strategies.

Additionally, our study acquainted us with potential methodologies for devising language-agnostic solutions for test smell detection. We contributed a high-level architectural framework, offering a foundational scaffold for future studies into language-agnostic

[2]https://github.com/publiosilva/aroma-lia/blob/main/README.md

test smell detection. Moreover, we operationalized this framework by drafting a POC that effectively identified two distinct test smells across test cases in disparate programming languages, employing identical detection mechanisms.

Moreover, while a part of our proposed strategy needs language-specific implementations for test case extraction from the AST, the core detection mechanism remains consistent across languages. This architectural design choice ensures the stability and universality of the detection process, irrespective of the programming language employed. Furthermore, our approach was intentionally engineered with extensibility in mind, enabling seamless adaptation to new languages without the need to reinvent the wheel. This aligns with the principle endorsed by Aljedaani et al. [1], which emphasizes the importance of critically evaluating the need for entirely new tool creation versus leveraging existing ones to foster efficiency and enable robust frameworks.

Lastly, we acknowledge that while our solution is currently limited by the language support provided by Mozilla's rust-code-analysis crate [2], our choice of a versatile tool emphasizes our commitment to scalability and adaptability. As the tool's language support expands, so will the breadth of languages accommodated by our solution, ensuring its relevance and applicability in diverse programming ecosystems.

6 THREATS TO VALIDITY

This section discusses threats to the study's validity according to the classification of Wohlin et al. [27].

Internal validity. The validity of the selected techniques to detect test smells might be questioned. To mitigate this threat, we selected techniques that have been successfully utilized in other works. This approach demonstrates that the techniques we chose have a track record of effectiveness and reliability in similar contexts, bolstering the internal validity of our study.

Construct validity. Limiting our study to only two languages (Java and JavaScript) could raise concerns about the generalizability of our approach. To address this threat, we justified our choice of languages. For instance, Java is widely used and serves as a representative language in the context of test smell detection. By selecting a widely used language like Java, we increased the likelihood that our findings will be applicable to a broad range of scenarios, thus enhancing construct validity.

External validity. The selection of valid examples of test smells is crucial for the external validity of our study. To mitigate this threat, we utilized examples from a well-known dataset of test smells [3]. This approach ensures that the test smells used in our study are recognized and accepted by the broader research community, enhancing the generalizability of our findings beyond the specific dataset or context used in our study.

Conclusion validity. Testing a limited number of languages and test smells may raise concerns about the robustness and generalizability of our conclusions. To address this threat, we justified our choice of languages and test smells as representative samples. Additionally, we acknowledged the limitations of our study and proposed avenues for future research, such as testing the approach with new languages and test smells. This approach demonstrates our commitment to expanding our research scope and enhancing the validity of our conclusions over time.

[3]https://testsmells.org/pages/testsmellexamples.html

7 FINAL REMARKS

In this work, we conducted a study to propose a language-agnostic solution for detecting test smells. We reviewed related literature to understand which techniques for detecting test smells have been used and what strategies have been employed in related areas to create language-agnostic solutions. After this preliminary study, we defined a high-level architecture for a language-agnostic approach to test smell detection using techniques and tools successfully used in related work. We also designed a POC implementing the proposed architecture, which successfully detected the Duplicate Assert and Assertion Roulette test smells in test cases written in Java and JavaScript using the same code for test smell detection.

This work can serve as a basis for future research in language-agnostic test smell detection. Additionally, the POC produced in this work provides the necessary framework for detecting new test smells and other languages not covered in this study. The results of this work can also serve as a basis for comparison for studies aiming to implement solutions for language-agnostic test smell detection using other techniques such as metrics, information retrieval, and dynamic tainting.

In future work, we intend to: (i) expand the set of test smells that can be detected by the tool produced in this work; (ii) expand the set of languages supported by the tool produced in this work by creating new implementations of the ASTTestExtractor interface for other languages and frameworks, including those that use paradigms such as functional programming; (iii) validate the tool with a larger set of test cases considering also other languages and other test smells; and (iv) create a tool for language-agnostic test smell detection using techniques different from those used in this work and compare the results.

ARTIFACT AVAILABILITY

We provide our data under open licenses at: https://github.com/publiosilva/aroma-lia/blob/main/README.md

ACKNOWLEDGMENTS

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

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 (EASE '21)*. Association for Computing Machinery, New York, NY, USA, 170–180. https://doi.org/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. https://doi.org/10.1016/j.softx.2020.100635

[3] P. Baker, D. Evans, J. Grabowski, H. Neukirchen, and B. Zeiss. 2006. TRex - The Refactoring and Metrics Tool for TTCN-3 Test Specifications. In *Testing: Academic Industrial Conference - Practice And Research Techniques (TAIC PART'06)*. 90–94. https://doi.org/10.1109/TAIC-PART.2006.35

[4] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 770–781. https://doi.org/10.1145/2786805.2786823

[5] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 179–190. https://doi.org/10.1145/2786805.2786843

[6] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2019. SoCRATES: Scala radar for test smells. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala (Scala '19)*. Association for Computing Machinery, New York, NY, USA, 22–26. https://doi.org/10.1145/3337932.3338815

[7] Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew P. Black, and Anne Etien. 2019. Rotten Green Tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 500–511. https://doi.org/10.1109/ICSE.2019.00062

[8] Arie Deursen, Leon M.F. Moonen, A. Bergh, and Gerard Kok. 2001. *Refactoring test code.* Technical Report. NLD.

[9] S. Ducasse, M. Rieger, and S. Demeyer. 1999. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*. 109–118. https://doi.org/10.1109/ICSM.1999.792593

[10] Vahid Garousi and Barış Küçük. 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software* 138 (2018), 52–81. https://doi.org/10.1016/j.jss.2017.12.013

[11] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. 2013. Automated Detection of Test Fixture Strategies and Smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 322–331. https://doi.org/10.1109/ICST.2013.45

[12] Michaela Greiler, Andy Zaidman, Arie van Deursen, and Margaret-Anne Storey. 2013. Strategies for avoiding text fixture smells during software evolution. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 387–396. https://doi.org/10.1109/MSR.2013.6624053

[13] Chen Huo and James Clause. 2014. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 621–631. https://doi.org/10.1145/2635868.2635917

[14] Negar Koochakzadeh and Vahid Garousi. 2010. A tester-assisted methodology for test redundancy detection. *Adv. Soft. Eng.* 2010, Article 6 (jan 2010), 13 pages. https://doi.org/10.1155/2010/932686

[15] Stefano Lambiase, Andrea Cupito, Fabiano Pecorelli, Andrea De Lucia, and Fabio Palomba. 2020. Just-In-Time Test Smell Detection and Refactoring: The DARTS Project. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)*. Association for Computing Machinery, New York, NY, USA, 441–445. https://doi.org/10.1145/3387904.3389296

[16] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. 2018. Automatic Test Smell Detection Using Information Retrieval Techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 311–322. https://doi.org/10.1109/ICSME.2018.00040

[17] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. 2022. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering* 27, 7 (20 Sep 2022), 170. https://doi.org/10.1007/s10664-022-10207-5

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

[19] Gordana Rakić, Zoran Budimac, and Miloš Savić. 2013. Language independent framework for static code analysis. In *Proceedings of the 6th Balkan Conference in Informatics (BCI '13)*. Association for Computing Machinery, New York, NY, USA, 236–243. https://doi.org/10.1145/2490257.2490273

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

[21] 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. https://doi.org/10.1109/ACCESS.2022.3160485

[22] Masayuki Taniguchi, Shinsuke Matsumoto, and Shinji Kusumoto. 2021. JTDog: a Gradle Plugin for Dynamic Test Smell Detection. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1271–1275. https://doi.org/10.1109/ASE51524.2021.9678529

[23] 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. https://doi.org/10.1016/j.infsof.2021.106620

[24] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. 2007. On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. *IEEE Transactions on Software Engineering* 33, 12 (2007), 800–817. https://doi.org/10.1109/TSE.2007.70745

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

[26] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. 2021. PyNose: A Test Smell Detector For Python. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 593–605. https://doi.org/10.1109/ASE51524.2021.9678615

[27] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering.* Springer Science & Business Media.

[28] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 385–396. https://doi.org/10.1145/2610384.2610404