AromaDr: A Language-Independent Tool for Detecting Test Smells

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

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

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

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

ABSTRACT

Ensuring high-quality test code is critical for the success and maintainability of software projects. Poorly designed tests, known as *test smells*, can undermine this goal by making test code harder to understand, maintain, and extend. Although various tools exist to detect and refactor test smells, most are tailored to specific programming languages, limiting their effectiveness in polyglot development environments. To overcome this limitation, we introduce AromaDr, a language-independent tool capable of detecting ten common test smells across multiple programming languages, including C#, Java, JavaScript, TypeScript, and Python. The smells detected by AromaDr include: *Assertion Roulette*, *Conditional Test Logic*, *Duplicate Assert*, *Empty Test*, *Exception Handling*, *Ignored Test*, *Magic Number*, *Redundant Print*, *Sleepy Test*, and *Unknown Test*. Beyond its language-independent detection capabilities, AromaDr offers several practical features: a graphical user interface for intuitive interaction, precise localization of test smells within the code, and an API that facilitates seamless integration with other development tools. To our knowledge, AromaDr currently supports the broadest range of programming languages among available test smell detection tools. In addition, AromaDr provides several features that distinguish it from other test-smell detection tools: a graphical interface (many alternatives are command-line only), the ability to pinpoint the exact line where each test smell occurs and a REST API that enables seamless integration with other tools.

Video: https://zenodo.org/records/15467769

KEYWORDS

Test Smell, Detection, Tool, Language-Independent

1 Introduction

Ensuring the quality of developed software is essential for the success of any software project [10]. One of the primary approaches to achieving this is software testing [16]. Testing can be performed manually or by automation [5]. Automated tests offer significant advantages, including ease of execution and the ability to reproduce results consistently [5].

However, when using automated tests, it is equally important to assess the quality of the test code to ensure its continued usefulness over time [16]. The significance of maintaining high-quality test code is underscored by the fact that developers spend approximately one-quarter of their time writing tests [3]. Therefore, ensuring test quality is crucial to avoid unnecessary costs and additional development effort [5].

One of the key factors that can hinder the quality of test code is the presence of test smells [8]. Test smells are poor design choices made during the development of test code [1]. These suboptimal practices result in test code that is more difficult to read, maintain, and evolve [15]. Test smells may arise due to the inherent complexity of the system or the limited experience of the developers [12].

This topic has gained increasing relevance in recent years, with numerous studies exploring it and proposing tools to detect and refactor test smells [4, 9, 10, 12, 17, 18]. Such tools are essential, as manual detection and refactoring of test smells are often impractical in large-scale software systems [11].

However, most existing tools are language-specific, which makes it challenging to support additional programming languages in the context of test smell detection [1]. To address this limitation, we proposed a language-independent approach for detecting test smells in a previous study [14]. In that work, we did not develop a fully functional tool to implement the approach but rather presented a proof of concept demonstrating the detection of two test smells (*Assertion Roulette* and *Duplicate Assert*) in a language-independent manner.

In this context, we present AromaDr, a tool designed to detect test smells in a language-independent manner. The tool supports the detection of ten test smells: *Assertion Roulette*, *Conditional Test Logic*, *Duplicate Assert*, *Empty Test*, *Exception Handling*, *Ignored Test*, *Magic Number*, *Redundant Print*, *Sleepy Test*, and *Unknown Test*. AromaDr currently enables test smell detection across five programming languages: C#, Java, JavaScript, TypeScript, and Python. Since it is based on a language-independent approach, the detection logic was implemented once and applied uniformly across all supported languages. To the best of our knowledge, AromaDr is the test smell detection tool that supports the largest number of programming languages to date.

Beyond being language-independent (which makes it easier to add support for new programming languages) *AromaDr* offers several additional advantages over existing tools: (i) it provides a graphical user interface, whereas most existing tools are console-based; (ii) it identifies the exact line in the source code where the smell occurs; and (iii) it can be easily integrated with other tools, as it exposes an API specifically designed for this purpose.

2 AromaDr Tool

The AromaDr tool implements a language-independent approach for detecting test smells, as proposed in our previous work [14]. To ensure broad usability and avoid dependency on specific development environments, we developed AromaDr as a stand-alone application rather than integrating it into existing IDEs (Integrated Development Environments). This design choice was made considering the impracticality of implementing the tool across multiple platforms, each with its own constraints and architectures.

The tool was built using a combination of modern technologies. The front-end was developed using the React web framework. For the back-end, we employed Node.js, TypeScript, and the Express framework.

To maximize portability, AromaDr is distributed as a containerized application. As a result, the only prerequisite for using the tool is having Docker installed. The setup process is straightforward and requires only two terminal commands to launch the container. Once running, the application serves a web interface that allows users to interact with the tool. Additionally, a REST API is made available, enabling integration with third-party applications via HTTP requests.

AromaDr offers two primary modes of use. In the first mode, users can paste the source code of an individual test file into the interface to analyze and identify any present test smells. In the second mode, users can input the URL of a public GitHub repository. The tool will then scan the entire repository to locate test files and analyze each one for potential test smells.

Currently, AromaDr supports five programming languages, each paired with a commonly used test framework: C# with xUnit, Java with JUnit, JavaScript or TypeScript with Jest, and Python with PyTest. The tool is capable of detecting ten types of test smells, namely: *Assertion Roulette*, *Conditional Test Logic*, *Duplicate Assert*, *Empty Test*, *Exception Handling*, *Ignored Test*, *Magic Number*, *Redundant Print*, *Sleepy Test*, and *Unknown Test*.

Figure 1 illustrates the architecture of AromaDr. The system consists of three main components: the AromaDr Web App, the AromaDr API, and the rust-code-analysis API, a third-party tool used to extract a Language-Independent Abstract Syntax Tree (LAAST) from the test code. AromaDr supports two modes of operation: file mode and project mode. In file mode, the tool can begin detecting test smells immediately, as the test code is already provided. In project mode, however, the process involves additional steps. First, the source code must be downloaded from the repository, and the test files must be identified. These steps are handled by the Repository Downloader and Test File Detector components (as shown in Figure 1). Once these steps are complete, the test smell detection process can begin.

The first step in the test smell detection process is to extract a LAAST from the test code (LAAST Generator in Figure 1). To achieve this, we utilize Mozilla's rust-code-analysis crate [2]. Since this tool requires the programming language of the source code to be specified in advance, we prompt the user to indicate both the language and the testing framework used in the test code.

As AromaDr depends on rust-code-analysis to extract the LAAST, its support for programming languages is currently limited to those supported by the tool. These include C++, C#, CSS, Go, HTML,

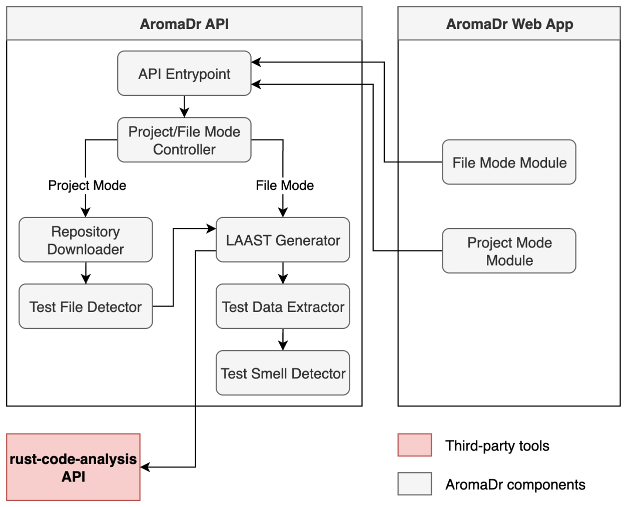


Figure 1: AromaDr Arquitecture

Java, JavaScript, JavaScript (Firefox internals), Python, Rust, and TypeScript. However, since rust-code analysis is extensible, it is possible to incorporate support for additional languages, which, in turn, would expand the scope of languages that AromaDr can analyze.

The second step involves extracting relevant information from the LAAST and converting it into a standardized and language-independent format (Test Data Extractor in Figure 1). This structured data serves as the foundation for subsequent test smell detection. The extracted data includes, but is not limited to: the names of individual test cases, the assertions within each test, the expected and actual values in assertions, explanatory messages, and other syntactic or semantic elements that can inform the presence of test smells. The complete data model derived from the LAAST is presented in Listing 1.

In the third step, the tool utilizes the test data extracted in the previous step to identify the presence of test smells (Test Smell Detector in Figure 1). The detection logic for each of the ten supported test smells is entirely language-independent, as it operates solely on the standardized data model produced during the previous stage. Since this model abstracts away language-specific syntax, the test smell detection algorithms can be implemented in any programming language without requiring adaptation for individual source languages.

Assertion Roulette. Algorithm 1 outlines the procedure for detecting the *Assertion Roulette* test smell. This smell is identified when the following two conditions are met: (1) a test contains more than one assertion, and (2) at least one of these assertions lacks an explanatory message [10]. The primary challenge is that when a test fails, it can be hard to determine which specific assertion triggered the failure [10].

Using the test data model presented in Listing 1, this detection can be performed by evaluating whether the `asserts` list associated with a test contains more than one element, and whether at least one of these elements has an empty `explanationMessage` field.

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

Listing 1: Test data model extract in the second step of the AromaDr test smell detection process

Algorithm 1 Assertion Roulette detection algorithm

```
1: function DetectAssertionRoulette(test)
2:     hasMoreThanOneAssert ← length of test.asserts > 1
3:     hasSomeAssertWithoutMessage ← False
4:     for each assert in test.asserts do
5:         if assert.message is empty then
6:             hasSomeAssertWithoutMessage ← True
7:             break
8:         end if
9:     end for
10:    return hasMoreThanOneAssert and hasSomeAssertWithoutMessage
11: end function
```

Conditional Test Logic. Algorithm 2 illustrates the process for detecting the *Conditional Test Logic* test smell. This smell is present when a test contains one or more control flow statements, such as conditionals (e.g., `if`, `switch`) or loops (e.g., `for`, `while`) [10].

In the test data model shown in Listing 1, a parameter named `statements` captures the different types of code statements found within a test. These include assignments, function or method calls, and control structures such as conditionals and loops. To detect the *Conditional Test Logic* smell, the algorithm inspects the `statements` list and checks whether it contains at least one element of type `condition` or `loop`.

Algorithm 2 Conditional Test Logic detection algorithm

```
1: function DetectConditionalTestLogic(test)
2:     for each statement in test.statements do
3:         if statement.type is condition or loop then
4:             return True
5:         end if
6:     end for
7:     return False
8: end function
```

Duplicate Assert. Algorithm 3 describes the process for detecting the *Duplicate Assert* test smell. This smell occurs when the same assertion (defined by identical parameters) is repeated multiple times within a single test case [10].

Using the test data model introduced in Listing 1, detection can be performed by generating a unique string representation for each assertion. This string is formed by concatenating the assertion's key attributes: the `actual` value, the `expected` value, and the assertion `matcher` (e.g., `equals`, `greaterThan`, `lessThan`). Once these strings are created for all assertions in a test, the algorithm checks for duplicates. The presence of any repeated strings indicates that one or more assertions are duplicated within the test.

Algorithm 3 Duplicate Assert detection algorithm

```
1: function DetectDuplicateAssert(test)
2:     seen ← new Set()
3:     for each assert in test.asserts do
4:         uniqueKey ← assert.literalActual + " | " + assert.matcher + " | " + assert.literalExpected
5:         if seen contains uniqueKey then
6:             return true
7:         end if
8:         seen.add(uniqueKey)
9:     end for
10:    return false
11: end function
```

Empty Test. Algorithm 4 presents the procedure for detecting the *Empty Test* smell. This smell is characterized by the complete absence of executable statements within a test case [10].

Using the test data model described in Listing 1, detection is straightforward: the algorithm simply checks whether the list of `statements` (representing all statements present in the test) is empty. If the list contains no elements, the test is considered empty and flagged accordingly.

Exception Handling. Algorithm 5 outlines the procedure for detecting the *Exception Handling* test smell. This smell occurs when a test method includes explicit exception management, such as `throw` or `catch` clauses [10].

Algorithm 4 Empty Test detection algorithm

```
1: function DetectEmptyTest(test)
2:     if test.statements is empty then
3:         return True
4:     end if
5:     return False
6: end function
```

In the test data model introduced in Listing 1, each test includes a `statements` list representing all types of statements present in its body. This list may contain elements of type `exceptionHandling` (representing `catch` blocks) and `exceptionThrowing` (representing `throw` statements). To detect this smell, the algorithm checks whether the `statements` list contains at least one element of either type. If so, the test is flagged as containing exception handling logic.

Algorithm 5 Exception Handling detection algorithm

```
1: function DetectExceptionHandling(test)
2:     for each statement in test.statements do
3:         if statement.type is exceptionHandling or exceptionThrowing then
4:             return True
5:         end if
6:     end for
7:     return False
8: end function
```

Ignored Test. Algorithm 6 outlines the process for detecting the *Ignored Test* test smell. This smell occurs when a test is deliberately skipped during execution, often due to the presence of a command that prevents the test from running [10]. The detection process is based on identifying such a command in the test code.

Since the mechanism for skipping tests can vary across different programming languages, in the previous step we analyze the LAAST of the test code to check for the presence of any skip command. If found, the `isIgnored` parameter in the test data model (Listing 1) is set to `true`. Therefore, the algorithm to detect this smell is simple: it checks whether the `isIgnored` parameter is set to `true`.

Algorithm 6 Ignored Test detection algorithm

```
1: function DetectIgnoredTest(test)
2:     return test.isIgnored
3: end function
```

Magic Number Test. Algorithm 7 outlines the procedure for detecting the *Magic Number Test* test smell. This smell occurs when one or more assertions in the test contain numeric literals [10], also known as "magic numbers."

Using the test data model derived in the previous step (Listing 1), the algorithm iterates over the list of assertions and checks whether either the `literalActual` or `literalExpected` parameters contain numeric literals. If either of these parameters is identified as a numeric literal, the test is flagged as having a *Magic Number Test* smell.

Algorithm 7 Magic Number Test detection algorithm

```
1: function DetectMagicNumberTest(test)
2:     if length of test.asserts > 1 then
3:         for each assert in test.asserts do
4:             if IsNumeric(assert.literalActual) or IsNumeric(assert.literalExpected) then
5:                 return True
6:             end if
7:         end for
8:     end if
9:     return False
10: end function
```

Redundant Print. Algorithm 8 outlines the process for detecting the *Redundant Print* test smell. This smell occurs when a test contains one or more unnecessary `print` statements [10].

The detection process varies depending on the programming language used in the test. To address this, the test data model generated in the previous step (Listing 1) includes a parameter called `events`, which represents a list of all events occurring within the test code. These events may include function or method calls, with common event types (such as assertion events) explicitly categorized. Any unrecognized events are classified as `unknown`. One of the event types explicitly identified is the `print` event. Therefore, to detect the *Redundant Print* test smell, the algorithm simply checks whether the list of events contains at least one event of type `print`.

Algorithm 8 Redundant Print detection algorithm

```
1: function DetectRedundantPrint(test)
2:     for each event in test.events do
3:         if event = "print" then
4:             return true
5:         end if
6:     end for
7:     return false
8: end function
```

Sleepy Test. Algorithm 9 outlines the process for detecting the *Sleepy Test* test smell. This smell is identified when a test includes a `wait` or `sleep` command, which introduces unnecessary delays in the test execution [10].

As part of the data model returned in the previous step (Listing 1), events are classified into various types. One of these explicitly classified event types is `sleep`. Therefore, to detect the *Sleepy Test* test smell, the algorithm simply checks whether there is at least one event of type `sleep` in the list of events.

Unknown Test. Algorithm 10 outlines the procedure for detecting the *Unknown Test* test smell. This smell occurs when a test contains no assertions [10].

Detecting this test smell using the test data model derived in the previous step (Listing 1) is straightforward. The algorithm simply checks whether the list of assertions is empty. If it is, the test is flagged as an *Unknown Test*.

Algorithm 9 Sleepy Test detection algorithm

```
1: function DetectSleepyTest(test)
2:     for each event in test.events do
3:         if event = "sleep" then
4:             return true
5:         end if
6:     end for
7:     return false
8: end function
```

Algorithm 10 Unknown Test detection algorithm

```
1: function DetectUnknownTest(test)
2:     if length of test.asserts = 0 then
3:         return True
4:     end if
5:     return False
6: end function
```

3 Example of Use

This section presents a usage example of the AromaDr tool. To begin, the user must first download the source code from the public GitHub repository [13]. Next, the user must execute two Docker commands (Docker must be installed and running), as described in the `README.md` file within the repository. Once the container is running, the user can access the tool by navigating to http://localhost:8000 in a web browser.

Figure 2 shows the initial view of the AromaDr web interface. The default mode, shown in the figure, is the file mode. In this mode, the user selects the language and testing framework used in the test, pastes the test code into the provided field, and clicks the *Detect Test Smells* button to initiate the analysis. In project mode, the user must select the programming language and testing framework used in the project, provide the URL of the public GitHub repository, and then click the *Detect Test Smells* button. The tool will automatically retrieve the test files from the specified repository and perform the smell detection across the entire project.

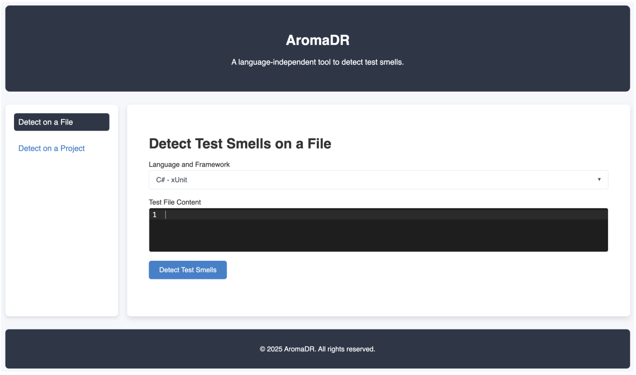
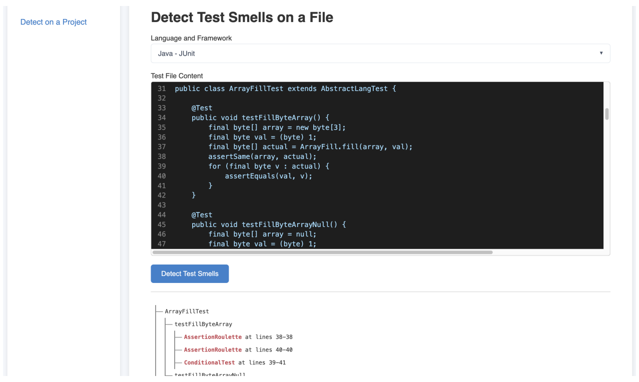

AromaDR
A language-independent tool to detect test smells.
Detect on a File
Detect on a Project
Detect Test Smells on a File
Language and Framework
C# - xUnit
Test File Content
Detect Test Smells
© 2025 AromaDR. All rights reserved.

Figure 2: AromaDr Initial View (File Mode)

After clicking the *Detect Test Smells* button in file mode, the identified test smells are displayed, as shown in Figure 3. The bottom section of the page presents a hierarchical view of the test structure, including the test suites and the individual tests within each suite. Alongside this structure, the detection results are shown, indicating the name of each detected test smell and the corresponding line numbers where each smell occurs.

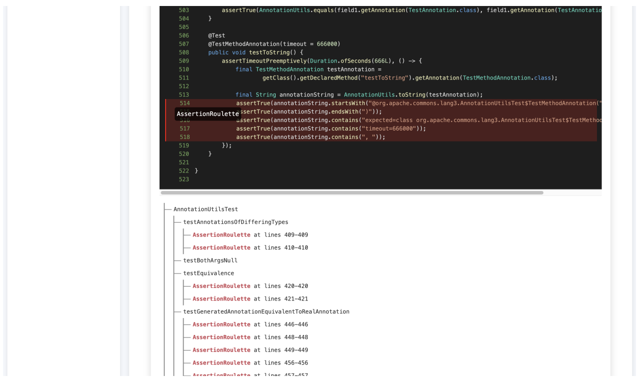


```
31  public class ArrayFillTest extends AbstractLangTest {
32
33      @Test
34      public void testFillByteArray() {
35          final byte[] array = new byte[3];
36          final byte val = (byte) 1;
37          final byte[] actual = ArrayFill.fill(array, val);
38          assertSame(array, actual);
39          for (final byte v : actual) {
40              assertEquals(val, v);
41          }
42      }
43
44      @Test
45      public void testFillByteArrayNull() {
46          final byte[] array = null;
47          final byte val = (byte) 1;
```

```
— ArrayFillTest
  — testFillByteArray
    — AssertionRoulette at lines 38-38
    — AssertionRoulette at lines 40-40
    — ConditionalTest at lines 39-41
  — testFillByteArrayNull
```

Figure 3: AromaDr File Mode Test Smell Detection

Figure 4 shows the results of test smell detection using the project mode. The output is similar to that of file mode, with some additional features. In project mode, all test files in the repository are listed, and for each file, the lines containing detected test smells are visually highlighted in red. Additionally, users are provided with the option to download a JSON file containing the detection results for all analyzed test files, enabling further inspection.



```
503         assertTrue(AnnotationUtils.equals(field1.getAnnotation(TestAnnotation.class), field1.getAnnotation(TestAnnotatio
504     }
505
506     @Test
507     @TestMethodAnnotation(timeout = 666000)
508     public void testToString() {
509         assertTimeoutPreemptively(Duration.ofSeconds(666L), () -> {
510             final TestMethodAnnotation testAnnotation =
511                     getClass().getDeclaredMethod("testToString").getAnnotation(TestMethodAnnotation.class);
512
513             final String annotationString = AnnotationUtils.toString(testAnnotation);
514             assertTrue(annotationString.startsWith("@org.apache.commons.lang3.AnnotationUtilsTest$TestMethodAnnotation("
515             assertTrue(annotationString.endsWith(")"));
516             assertTrue(annotationString.contains("expected=class org.apache.commons.lang3.AnnotationUtilsTest$TestMethod
517             assertTrue(annotationString.contains("timeout=666000"));
518             assertTrue(annotationString.contains(", "));
519         });
520     }
521
522 }
523
```

```
— AnnotationUtilsTest
  — testAnnotationsOfDifferingTypes
    — AssertionRoulette at lines 409-409
    — AssertionRoulette at lines 410-410
  — testBothArgsNull
  — testEquivalence
    — AssertionRoulette at lines 420-420
    — AssertionRoulette at lines 421-421
  — testGeneratedAnnotationEquivalentToRealAnnotation
    — AssertionRoulette at lines 446-446
    — AssertionRoulette at lines 448-448
    — AssertionRoulette at lines 449-449
    — AssertionRoulette at lines 456-456
    — AssertionRoulette at lines 457-457
```

Figure 4: AromaDr Project Mode Test Smell Detection

It is also possible to use the AromaDr tool through its REST API. The `README.md` file in the tool's GitHub repository provides detailed instructions for using the AromaDr API in both modes (file-based and project-based). This feature enables seamless integration of the tool into automated pipelines or other external applications that require programmatic access to test smell detection.

4 Comparison with other Tools

Several tools have been proposed for detecting and refactoring test smells. For our comparison, we selected (i) at least one tool that supports a language also handled by AromaDr and (ii) another language-independent tool. The rest of this section compares these tools with AromaDr in terms of their capabilities.

TsDetect is a command-line tool for detecting test smells in Java projects using the JUnit testing framework [10]. It identifies 19 distinct test smells, including *Assertion Roulette*, *Conditional Test Logic*, *Constructor Initialization*, *Default Test*, *Duplicate Assert*, *Eager Test*, *Empty Test*, *Exception Handling*, *General Fixture*, *Ignored Test*, *Lazy Test*, *Magic Number Test*, *Mystery Guest*, *Redundant Print*, *Redundant Assertion*, *Resource Optimism*, *Sensitive Equality*, *Sleepy Test*, and *Unknown Test*. Users provide a CSV file listing paths to production and test code; the tool outputs another CSV listing the detected smells. While TsDetect supports more smell types than AromaDr, our tool is language-independent (currently supporting five languages) and provides line-level precision for each smell, unlike TsDetect, which only reports the affected file.

PyNose is a tool designed to detect test smells in Python tests using the Unittest framework [18]. It currently supports 18 test smells, 17 language-agnostic and one specific to Python, including *Assertion Roulette*, *Conditional Test Logic*, *Constructor Initialization*, *Default Test*, *Duplicate Assert*, *Empty Test*, *Exception Handling*, *General Fixture*, *Ignored Test*, *Lack of Cohesion of Test Cases*, *Magic Number Test*, *Obscure In-Line Setup*, *Redundant Assertion*, *Redundant Print*, *Sleepy Test*, *Suboptimal Assert*, *Test Maverick*, and *Unknown Test*. The tool works as a plugin for PyCharm, an integrated development environment (IDE) for Python by JetBrains. In addition to in-IDE detection, PyNose allows results to be exported to a JSON file. Unlike PyNose, AromaDr is not tied to any specific IDE, as its language-independent design reduces the relevance of IDE integration. However, we have developed an API to support future platform integrations. Our tool also supports exporting detection results to JSON. A key advantage of our tool over PyNose is its language-agnostic detection mechanism, currently supporting five programming languages, while PyNose supports only Python.

XNose detects test smells in C# test code using the xUnit framework [9], supporting 16 smells: *Assertion Roulette*, *Conditional Test Smell*, *Inappropriate Assertions*, *Constructor Initialization*, *Duplicate Assert*, *Empty Test*, *Eager Test*, *Ignored Test*, *Lack of Cohesion of Test Cases*, *Magic Number Test*, *Obscure In-Line Setup*, *Redundant Assertion*, *Redundant Print*, *Sleepy Test*, *Sensitive Equality*, and *Unknown Test*. Like TsDetect, XNose is a command-line tool, whereas AromaDr offers a graphical user interface for easier use. While XNose supports more smells, AromaDr covers five programming languages compared to XNose's C#-only support.

To our knowledge, besides AromaDr, the only tool supporting JavaScript is STEEL [6]. STEEL detects 15 test smells, including *Assertion Roulette*, *Conditional Test Logic*, *Eager Test*, *Lazy Test*, *Duplicate Assert*, *Magic Number Test*, *Redundant Print*, *Empty Test*, *Exception Handling*, *Redundant Assertion*, *Unknown Test*, *Mystery Guest*, *Resource Optimism*, *Ignored Test*, and *Sleepy Test*, and also reports various test quality metrics. STEEL is a command-line tool, while AromaDr offers a graphical interface to ease detection. Our tool supports JavaScript plus four other languages, using a language-independent detection method that simplifies adding new languages without separate detection code.

In the literature, SniffML [7] is another tool claiming language independence, supporting C, C++, C#, and Java. It detects seven test smells: *Assertion Roulette*, *Conditional Test*, *Duplicate Assert*, *Empty Test*, *Exception Handling*, *Magic Number*, and *Unknown Test*. SniffML requires a GitHub project URL for detection, a feature we also implemented to analyze entire projects. Our tool offers several advantages over SniffML. It supports a wider range of programming languages, detects all the code smells identified by SniffML (plus three additional ones) and provides a graphical user interface, in contrast to SniffML's command-line interface. Another key benefit is that SniffML relies on the srcML tool to extract an XML representation of the test code, which currently supports only four languages. In contrast, our tool uses a more versatile extractor to generate the LAAST (Language-Agnostic Abstract Syntax Tree), which supports a significantly larger number of languages and can be extended to accommodate even more in the future [2].

5 Conclusion

In this study, we present AromaDr, a tool for detecting test smells that follow a language-independent strategy, meaning that adding support for a new language does not require re-implementing the detection algorithms. Currently, AromaDr detects ten well-known test smells and supports five languages: C#, Java, JavaScript, TypeScript, and Python. To our knowledge, it is the test-smell detection tool with the broadest language coverage.

Our tool offers a graphical user interface, an advantage over several existing command-line tools. Moreover, it pinpoints the exact line in which each test smell occurs, whereas some competing tools merely indicate the file that contains the smell. Consequently, AromaDr is useful for practitioners (who can analyze projects written in any supported language) and researchers (who can study test smells using AromaDr or extend the tool with additional smells or languages). We provide our tool under the MIT License, which means it is open to modifications and improvements to, for example, add support for new languages and test smells.

Future work will focus on extending support to additional programming languages and broadening the set of detectable test smells. We also plan to introduce language-independent refactoring capabilities, enabling both researchers and practitioners to more effectively improve the quality of their test code. Additionally, we aim to integrate our tool with popular configuration management platforms such as GitHub and GitLab, and to evaluate its practical impact through real-world usage by developers.

ARTIFACT AVAILABILITY

We provide our data and artifacts under open licenses at: https://github.com/publiosilva/aromadr

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 (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] 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* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 179–190. doi:10.1145/2786805.2786843

[4] 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

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

[6] Dalton Jorge, Patricia Machado, and Wilkerson Andrade. 2021. Investigating Test Smells in JavaScript Test Code. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing* (Joinville, Brazil) *(SAST '21)*. Association for Computing Machinery, New York, NY, USA, 36–45. doi:10.1145/3482909.3482915

[7] 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 XXIII Brazilian Symposium on Software Quality (SBQS '24)*. Association for Computing Machinery, New York, NY, USA, 231–241. doi:10.1145/3701625.3701662

[8] 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. doi:10.1007/s10664-022-10207-5

[9] Partha P. Paul, Md Tonoy Akanda, M. Raihan Ullah, Dipto Mondal, Nazia S. Chowdhury, and Fazle M. Tawsif. 2024. xNose: A Test Smell Detector for C . In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE Computer Society, Los Alamitos, CA, USA, 370–371. doi:10.1145/3639478.3643116

[10] 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

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

[12] 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. *Sci. Comput. Program.* 231, C (Jan. 2024), 20 pages. doi:10.1016/j.scico.2023.103013

[13] Publio Silva. 2025. *AromaDr GitHub Repository*. https://github.com/publiosilva/aromadr Accessed: 2025-05-18.

[14] Publio Silva, Carla Bezerra, and Ivan Machado. 2024. Toward a Language-Agnostic Approach to Detect Test Smells. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 686–692. doi:10.5753/sbes.2024.3647

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

[16] 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

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

[18] 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)*. IEEE Computer Society, Los Alamitos, CA, USA, 593–605. doi:10.1109/ASE51524.2021.9678615