AgNose: A Framework-Agnostic Test Smell Detection Tool

André Cardoso[1], Denilson Vieira[1], Elvys Soares[1]

[1]Federal Institute of Alagoas (IFAL)
Maceió – AL – Brazil

{alcs4,dvs10}@aluno.ifal.edu.br, elvys.soares@ifal.edu.br

Abstract. *Test smells indicate problems in automated tests that hinder the understanding, effectiveness, and maintenance of the test code. Due to their relevance, many tools have been created to detect them in xUnit frameworks, which has led to the emergence of test framework-agnostic strategies for detecting smells. However, these strategies are still little explored in the literature, and the available tools depend on obsolete components with considerable limitations. This study proposes the AgNose tool, which extends the strategy of agnostic test smell detection. The strategy is based on creating, from the test code in different frameworks, a simplified structure aimed solely at test smell detection. To implement the proposal, we (i) identify test smells analyzed simultaneously by different detection tools, as well as their identification steps; (ii) establish the minimum XML structure that represents the important information for detecting smells; (iii) implement the AgNose tool; (iv) validate the tool using real open-source projects from public repositories. As results, we map 8 test smells and their minimal XML structure. Also, the tool validation achieved a precision of 96.9%, a recall of 87.5%, and an f-measure of 92%. The presented strategy and tool extend current strategies for agnostic test smell detection, eliminating the perceived limitations of available tools.*

1. Introduction

Automated testing plays a critical role in ensuring software quality. However, the quality of the tests themselves must also be assessed, as design or implementation issues can undermine their effectiveness. These issues, known as test smells, are symptoms in test code that indicate potential challenges related to maintenance, readability, or reliability [Soares et al. 2024]. For example, Listing 1 presents a snippet of test code extracted from the TestContainers project[1] on GitHub. One can observe that the assertion method, representing the test itself, is embedded within a decision structure on line 6. According to [Soares et al. 2023b], this is the Conditional Test Logic smell, where branching the execution flow may cause the test to terminate without performing the intended assertion, potentially leading to a false test outcome, negatively impacting its reliability.

```
1 @Test
2 void first_test() {
3   if (lastContainerId == null) {
4     lastContainerId = genericContainer.getContainerId();
5   } else {
6     assertNotEquals(lastContainerId, genericContainer.getContainerId());
7   }
8 }
```

Listing 1. The Conditional Test Logic smell.

[1]https://git.io/JtRpX

Several tools have been developed to identify test smells in various unit test automation frameworks, known as xUnit frameworks [Lopes et al. 2024]. Tools such as JNose [Virgínio et al. 2021], PyNose [Wang et al. 2021], TsDetect [Peruma et al. 2019], DARTS [Lambiase et al. 2020], and many others [Aljedaani et al. 2021] have emerged as solutions for identifying test smells across different frameworks and programming languages.

Due to their restriction to the specific frameworks for which they were designed and their execution of the same steps for detecting commonly existing test smells across various xUnit frameworks, test smell detection tools notably represent redundant development efforts [Lopes et al. 2024]. This scenario has motivated the proposal of approaches [Silva et al. 2024] and a tool called SniffML [Lopes et al. 2024] for test smell detection that are agnostic to test frameworks. However, the latest propositions in the literature rely on software components, *i.e.,* srcML [Collard et al. 2013], that are no longer maintained and face a lack of support for new features of programming languages and xUnit frameworks [Lopes et al. 2024].

This study proposes the AgNose tool, aimed at enhancing the current strategy for agnostic detection of test smells and overcoming existing limitations in available tools. By creating a simplified structure from the test code, the tool enables agnostic detection of test smells without relying on external components for code conversion.

To develop the tool, we selected a list of 8 test smells most commonly identified in detection tools [Lopes et al. 2024, Aljedaani et al. 2021]. Additionally, we mapped the steps utilized by these tools for detecting the identified smells. By comparing test code with the required steps for identifying these smells, we established the minimum necessary information for their detection — capturing only the truly relevant elements and eliminating extraneous details. Subsequently, we implemented the AgNose tool along with its strategies for code conversion and test smell detection. Finally, we validated the tool by selecting automated tests from real open-source projects on GitHub and executing the tool on them. A total of 64 tests from four distinct projects were selected, and the tool's results were compared against manual analysis conducted by two authors, achieving precision metrics of 96.9%, recall of 87.5%, and an F-measure of 92%.

The main contributions of this study are:

- A simplified representation of the code required for the effective detection of each analyzed smell in test code (Section 3);
- A tool capable of identifying test smells across multiple testing frameworks (Section 4);
- Validation of the proposed tool using real open-source projects (Section 5).

2. Motivating Example

Listing 2 provides an example of converting the code from Listing 1 to XML using srcML [Collard et al. 2013]. In the example, the goal of the conversion to represent all possible information needed to identify potential faults, extend, and accurately rewrite the code. Although a crucial component for the functionality of the SniffML tool, the conversion to XML performed by srcML exhibits shortcomings when handling components from newer versions of programming languages (*e.g.,* Java lambdas and callbacks) and testing frameworks (*e.g.,* JUnit tags) [Lopes et al. 2024].

```
<function><type><name>Test</name>
<name>void</name></type> <name>first_test</name><parameter_list>()</parameter_list> <block>{<block_content>
  <if_stmt><if>if <condition>(<expr><name>lastContainerId</name> <operator>==</operator> <name>null</name></expr>)</
      condition> <block>{<block_content>
    <expr_stmt><expr><name>lastContainerId</name> <operator>=</operator> <call><name><name>genericContainer</name><
        operator>.</operator><name>getContainerId</name></name><argument_list>()</argument_list></call></expr>;</
        expr_stmt>
  </block_content>}</block></if> <else>else <block>{<block_content>
    <expr_stmt><expr><call><name>assertNotEquals</name><argument_list>(<argument><expr><name>lastContainerId</name></
        expr></argument>, <argument><expr><call><name><name>genericContainer</name><operator>.</operator><name>
        getContainerId</name></name><argument_list>()</argument_list></call></expr></argument>)</argument_list></
        call></expr>;</expr_stmt>
  </block_content>}</block></else></if_stmt>
</block_content>}</block></function>
</unit>
```

Listing 2. Example of a test case converted with srcML.

However, for the purpose of identifying the specific test smell presented by the code in the example, the representation provided in Listing 3 is more efficient. To detect the Conditional Test Logic, it is sufficient to identify a decision structure within the test — we will elaborate on the remaining rules in the next section — with no need to analyze which condition is evaluated or any statement the structure contains [Soares et al. 2023a]. This approach greatly simplifies the conversion of code across different test automation frameworks, as syntax details can be disregarded. Consequently, the conversion of test code into a simplified structure, as presented in Listing 3, to be used to search for test smells, motivates the development of this study.

```
<testcase>
  <if>
    <stmt></stmt>
    <else>
      <assert></assert>
    </else>
  </if>
</testcase>
```

Listing 3. Minimal XML of a testcase with the Conditional Test Logic smell.

3. Test Smells and Detection Steps

In this section, we present the test smells cataloged as common to more than one xUnit framework by the literature [Aljedaani et al. 2021, Lopes et al. 2024]. Each test smell is described in terms of its definition, the steps for its detection, and a minimal example of test code converted to XML that exhibits it.

3.1. Magic Number

This occurs when a test method contains undocumented or unexplained numeric literals used as parameters or identifier values [Peruma et al. 2019]. Magic Numbers are detected on `assert` statements including a numeric literal as an argument [Virgínio et al. 2021]. Figure 1 shows an example of a test method having magic numbers converted to XML.

```
@Test
void magicNumberExample() {
  int result = 10;
  assertEquals(10, result);
}
```

```
<testcase>
  <stmt></stmt>
  <assert expected="literal:10" actual:"result">
      </assert>
</testcase>
```

Figure 1. A simplified XML of the Magic Number smell.

3.2. Conditional Test Logic

Conditional Test Logic refers to conditions within test methods that can alter the expected behavior, resulting in failures to detect defects due to unmet conditions [Soares et al. 2020]. To detect it, one only needs to identify conditional and iterative statements [Virgínio et al. 2021]. For an example of code, refer to Listings 1 and 3.

3.3. Unknown Test

Test methods without assertions hinder a clear understanding of the test's purpose and may cause JUnit to report them as passed [Peruma et al. 2019]. An Unknown Test smell is detected by identifying a method that uses the @Test annotation but does not contain any assertion statements [Virgínio et al. 2021]. Example in Figure 2.

```
@Test
void UnknownTest() {
  POICategories cat = getPOICategories();
  System.out.println(category[1].name());
}
```

```
<testcase>
  <stmt></stmt>
  <stmt></stmt>
</testcase>
```

Figure 2. A simplified XML of the Unknown Test smell.

3.4. Exception Handling

Defined by the manual handling of exceptions to determine the success or failure of a test, rather than using the exception-handling functionalities provided by the xUnit framework [Soares et al. 2020]. This test smell is detected by identifying a block that contains either a throw statement or a catch clause [Virgínio et al. 2021]. An example in Figure 3.

```
@Test
void exceptionHandling() {
  try{
    a.compute();
  }catch(CalculationException e) {
    Assert.fail(e.getMessage());
  }
}
```

```
<testcase>
  <try>
    <stmt></stmt>
    <catch>
      <stmt></stmt>
    </catch
  </try>
<testcase>
```

Figure 3. A simplified XML of the Exception Handling test smell.

3.5. Empty Test

Defined as test methods without executable statements which may be reported as successful tests [Peruma et al. 2019]. Detected as empty or content-commented methods [Virgínio et al. 2021]. Figure 4 provides an example.

```
@Test
void emptyTest(){
// Credentials cred = innerCredTest(FULL_SAMPLE_v1);
// assertEquals("p4ssw0rd", cred.pass);
}
```

```
<testcase>
  <comment></comment>
  <comment></comment>
</testcase>
```

Figure 4. A simplified XML of the Empty Test smell.

3.6. Duplicate Assert

When the same condition is tested multiple times within the same test method [Soares et al. 2020]. It is detected by identifying an assertion whose parameters are identical to those of another assertion within the same test method [Virgínio et al. 2021]. Figure 5 provides an example.

```
@Test
void duplicatedAssert() {
  Order o = new Order();
  o.add(new Product("mouse", 150));
  assertEquals(150, o.getTotal());
  assertEquals(150, o.getTotal());
}
```

```
<testcase>
  <stmt></stmt>
  <stmt></stmt>
  <assert expected="150" actual="o.getTotal()"/>
  <assert expected="150" actual="o.getTotal()"/>
</testcase>
```

Figure 5. A simplified XML of the Duplicate Assert test smell.

3.7. Assertion Roulette

Assertion Roulette occurs when a test method contains multiple assertions without contextualization, making it difficult to understand the test and debug failures [Soares et al. 2020]. It is detected by identifying a method with multiple assertions lacking individual error messages [Virgínio et al. 2021]. Example in Figure 6.

```
@Test
public void assertionRoulette() {
  File f = new File("file");
  assertTrue(f.exists());
  assertTrue(f.ofLength(12));
}
```

```
<testcase>
  <stmt></stmt>
  <assert expected="f.exists()"></assert>
  <assert expected="f.ofLength(12)"></assert>
</testcase>
```

Figure 6. A simplified XML of the Assertion Roulette smell.

3.8. Redundant Print

Unnecessary print statements within tests, which may consume resources or increase execution time [Peruma et al. 2019]. This test smell is detected by identifying console print methods (*e.g.,* `print()`) [Virgínio et al. 2021]. Figure 7 presents an example.

```
@Test
void redundantPrint() {
  Coord result = transf(Coord.ORIGIN, northAndUp10M);
  System.out.println("result = " + result);
  assertEquals(Coord.ORIGIN, result);
}
```

```
<testcase>
  <stmt></stmt>
  <print></print>
  <assert></assert>
<testcase>
```

Figure 7. A simplified XML of the Redundant Print test smell.

4. The AgNose tool

The tool operates in two main stages: first, it converts tests into an XML representation, and subsequently analyzes these files to detect problematic patterns. The process begins by scanning a directory for tests annotated with @Test — specifically for JUnit

— extracting their methods and converting each file into a Compilation Unit, or Abstract Syntax Tree (AST). This structured analysis ensures that essential information, such as method names, declarations, method calls, and control structures, is preserved. Each test method is subsequently converted into an individual XML file to be analyzed in the next stage.

In the subsequent phase, the tool examines these XML files to identify the test smells listed in Section 3, also adhering to the detection steps described. At the conclusion, the results are compiled into a CSV file, providing a clear overview of the issues identified in the analyzed tests.

As this initial version of the tool supports only the Java language, Java was utilized as the basis for selecting standard libraries to transform the source code into AST, i.e., Javaparser, a robust and well-established library, was employed to generate the XML as proposed in Section 3. In the following phase, Python was chosen as the primary language for implementing search mechanisms due to its ease of use and the wide availability of libraries for handling both input files (*i.e.,* XML) and output files (*i.e.,* CSV). Python excels at managing data and rapidly processing information, making it an ideal choice for organizing analysis results into CSV files.

Figure 8 illustrates a simplified UML class diagram of the AgNose tool, where parsers — responsible for converting a test file into multiple test objects — and test smell matchers are depicted. Ultimately, the tool generates a CSV file containing the test file name, test method, and identified test smells.

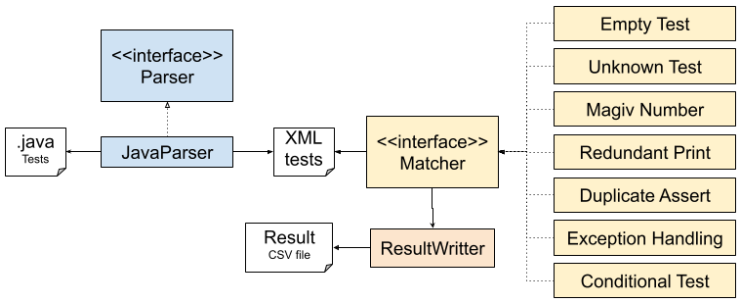

Figure 8. Simplified UML class diagram of the AgNose tool

The online repository at `https://github.com/andreoluiz/AgNose` contains the AgNose tool.

5. Results and Discussion

The selection of projects was based on publicly available repositories on GitHub. Four projects containing a significant number of automated tests were chosen, totaling 64 test methods, thereby enabling a detailed analysis of the distribution and quantity of tests in each project. The selection of tests was carried out randomly, without prior control over selection criteria, ensuring an uncontrolled and diverse sample. Table 1 presents the results of the tool's analysis of the selected tests and projects. The table shows the occurrences of test smells such as Assertion Roulette (AR), Empty Test (ET), Conditional Test Logic (CTL), Magic Number (MN), Duplicate Assert (DA), Exception Handling (EH), Unknown Test (UT), and Redundant Print (RP). From this sample, Magic Number and Assertion Roulette were identified as the most frequent test smells.

Table 1. Test smells identified in the analyzed projects

Project	AR	ET	CTL	MN	DA	EH	UT	RP
Netflix/conductor	6			13		5	6	
soabase/exhibitor	3		4	2		3		
Java-Master				3				
Anuken/Mindustry	6	2	3	5	1		1	
Total	**15**	**2**	**7**	**23**	**1**	**8**	**7**	**0**

For validation purposes, two authors conducted an independent and manual review of the selected tests, analyzing the identified test smells. Discrepancies were less than 5% of the cases and were resolved in dedicated meetings involving a third author to reach a consensus. From the number of true positives (63), true negatives (373), false positives (2), and false negatives (9), the precision, recall, and F-measure metrics were calculated, achieving values of 96.9%, 87.5%, and 92%, respectively. These results demonstrate that the proposed detection strategy is highly effective, with robust indicators of accuracy in classifications.

Considering the motivating example, the simplified detection methodology proves to be a promising solution. It can be practically validated by human evaluators and is also highly flexible, allowing easy extension to other programming languages. This adaptability makes the strategy applicable to various development contexts without significant challenges.

6. Related Work

Agnostic test smell detection is a relatively unexplored research area. The first known study by Silva et al. [Silva et al. 2024] proposed an agnostic detection method using AST parsing to identify test smells. Their tool demonstrated proof-of-concept capabilities in detecting two test smells across two programming languages. Another study by Lopes et al. [Lopes et al. 2024] introduced a strategy based on transforming test code into XML, detecting seven test smells across three languages using srcML. However, their approach faces limitations with newer language versions and xUnit frameworks. The AgNose tool builds upon these foundations by offering a reliable detection mechanism with simplified translations, facilitating automated analysis and human verification without reconstructing the original test.

7. Conclusion and Future Work

This study proposes an optimized method for detecting test smells by leveraging a simplified semantic representation of test cases, thereby reducing computational overhead while allowing human verification of generated structures. Given the growing relevance of test smell detection, this approach contributes to improving software quality by facilitating the identification and analysis of problematic patterns.

As future work, the next step will involve expanding the range of supported programming languages as well as the list of detectable test smells. This extension will broaden the tool's applicability to encompass a greater variety of patterns, enabling a more comprehensive and detailed analysis of test code.

References

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

Collard, M. L., Decker, M. J., and Maletic, J. I. (2013). srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International conference on software maintenance*, ICSME '13', pages 516–519.

Lambiase, S., Cupito, A., Pecorelli, F., De Lucia, A., and Palomba, F. (2020). Just-in-time test smell detection and refactoring: The DARTS project. In *Proceedings of the 28th international conference on program comprehension*, ICPC, pages 441–445.

Lopes, G., Romão, D., Soares, E., Ribeiro, M., Amaral, G., Gheyi, R., and Machado, I. (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, pages 231–241.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2019). On the distribution of test smells in open source Android applications: An exploratory study. In *29th Annual International Conference on Computer Science and Software Engineering*, CASCON, pages 193–202.

Silva, P., Bezerra, C., and Machado, I. (2024). Toward a language-agnostic approach to detect test smells. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software*, SBES '24, pages 686–692.

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

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

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

Soares, E., Ribeiro, M., and Santos, A. (2024). A multimethod study of test smells: Cataloging removal and new types. In *Proceedings of the XXIII Brazilian Symposium on Software Quality*, SBQS '24, pages 676–686.

Virgínio, T., Martins, L., Santana, R., Cruz, A., Rocha, L., Costa, H., and Machado, I. (2021). On the test smells detection: an empirical study on the jnose test accuracy. *Journal of Software Engineering Research and Development*, 9:8–1.

Wang, T., Golubev, Y., Smirnov, O., Li, J., Bryksin, T., and Ahmed, I. (2021). Pynose: A test smell detector for python. In *2021 36th IEEE/ACM international conference on automated software engineering (ASE)*, ASE, pages 593–605.