FEDERAL UNIVERSITY OF CEARÁ

CAMPUS QUIXADÁ

POSTGRADUATE PROGRAM IN COMPUTING

ACADEMIC MASTER'S DEGREE IN COMPUTING

PUBLIO BLENILIO TAVARES SILVA

AROMALIA: A LANGUAGE-INDEPENDENT APPROACH TO DETECT TEST SMELLS

QUIXADÁ

2025

PUBLIO BLENILIO TAVARES SILVA

AROMALIA: A LANGUAGE-INDEPENDENT APPROACH TO DETECT TEST SMELLS

Dissertation presented to the Academic Master's Degree in Computing program of the Postgraduate Program in Computing at the Campus Quixadá of the Federal University of Ceará, as a partial requirement for obtaining the title of Master in Computing. Area of Concentration: Software Engineering.

Advisor: Profa. Dra. Carla Ilane Moreira Bezerra.
Co-advisor: Prof. Dr. Ivan do Carmo Machado.

QUIXADÁ

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S582a Silva, Publio Blenilio Tavares.
AromaLIA: A Language-Independent Approach to Detect Test Smells / Publio Blenilio Tavares Silva. – 2025.
144 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Campus de Quixadá, Programa de Pós-Graduação em Computação, Quixadá, 2025.
Orientação: Profa. Dra. Carla Ilane Moreira Bezerra.
Coorientação: Prof. Dr. Ivan do Carmo Machado.

1. Test Smell. 2. Language-Independent Approach. 3. Test Smell Detection. I. Título.

CDD 005

PUBLIO BLENILIO TAVARES SILVA

AROMALIA: A LANGUAGE-INDEPENDENT APPROACH TO DETECT TEST SMELLS

Dissertation presented to the Academic Master's Degree in Computing program of the Postgraduate Program in Computing at the Campus Quixadá of the Federal University of Ceará, as a partial requirement for obtaining the title of Master in Computing. Area of Concentration: Software Engineering.

Approved on: 18/12/2025.

EXAMINING COMMITTEE

Profa. Dra. Carla Ilane Moreira Bezerra (Advisor)
Federal University of Ceará (UFC)

Prof. Dr. Ivan do Carmo Machado (Co-advisor)
Federal University of Bahia (UFBA)

Prof. Dr. Márcio de Medeiros Ribeiro
Federal University of Alagoas (UFAL)

Prof. Dr. Breno Miranda
Federal University of Pernambuco (UFPE)

Prof. Dr. Michel Sales Bonfim
Federal University of Ceará (UFC)

To God, from whom, through whom, and for whom all things exist, without whose presence nothing would be possible. To my family (my father, my mother, my brother, and my sister) for their unwavering presence, dedication, and support, which made this achievement possible.

ACKNOWLEDGEMENTS

I thank God, who has guided me throughout this journey and to whom all achievements truly belong.

To my parents, Benedito and Consuelo, for their unwavering support throughout my academic path and for rejoicing with me in every accomplishment. I also extend my gratitude to my brother Ruben and my sister Sueli for their encouragement and support.

To Prof. Dr. Carla Ilane Moreira Bezerra, my advisor, I am deeply grateful for her guidance and invaluable support throughout this journey, from my graduation to the completion of this master's degree.

To Prof. Dr. Ivan do Carmo Machado, my co-advisor, I extend my sincere thanks for his insightful contributions, which were essential to the development of this study.

To the distinguished examining committee, composed of Prof. Dr. Márcio de Medeiros Ribeiro, Prof. Dr. Breno Miranda, and Prof. Dr. Michel Sales Bonfim, I am grateful for accepting to evaluate this work and for sharing your knowledge. Your comments and insights have undoubtedly elevated the quality of this study.

To the Graduate Program in Computer Science at the Federal University of Ceará, Campus Quixadá, I am thankful for the excellent faculty, the support, and the resources that greatly contributed to the success of this research.

Finally, to all those who contributed, directly or indirectly, to this work, I extend my sincere gratitude.

"Simple things should be simple, complex things should be possible"   (Alan Kay)

RESUMO

Maus cheiros de teste (*test smells*) são indícios de más práticas ou decisões inadequadas tomadas durante o desenvolvimento de código de teste, que podem comprometer sua manutenibilidade e capacidade de evolução. Esse tema tem ganhado crescente relevância nos últimos anos, com diversos trabalhos propondo soluções para detectar *test smells*. No entanto, a maior parte dessas soluções é projetada para funcionar com um número extremamente limitado de linguagens de programação, frequentemente apenas uma. Isso significa que aplicar técnicas de detecção de *test smells* em uma nova linguagem geralmente exige desenvolver uma solução do zero, mesmo para problemas já abordados em outras linguagens. Nesse contexto, o objetivo deste trabalho é propor uma abordagem independente de linguagem para detectar *test smells*. Para compreender como esse tópico tem sido tratado na literatura, conduzimos um mapeamento sistemático (MS) que analisou 117 estudos publicados até agosto de 2025. A partir dessa análise, identificamos, entre outros aspectos, os *test smells* mais recorrentes em múltiplas linguagens de programação, bem como aqueles considerados mais críticos, ou seja, os que demandam maior atenção da comunidade. Com base nesses achados, propusemos nossa abordagem, avaliamos sua eficácia nas linguagens C#, Java, JavaScript, TypeScript e Python em 10 diferentes *test smells* e a comparamos com abordagens específicas de linguagem já existentes. Os resultados foram promissores: nossa solução apresentou desempenho superior em termos de efetividade, alcançando precisão de 97%, *recall* de 96% e *F1-score* de 97%. Para facilitar a aplicação prática, desenvolvemos e disponibilizamos uma ferramenta de detecção de *test smells* baseada em nossa abordagem independente de linguagem. Os resultados deste trabalho têm o potencial de simplificar a inclusão de novas linguagens no contexto da detecção de maus cheiros de teste, permitindo o reaproveitamento de esforços anteriores em diferentes ecossistemas. Isso beneficia tanto profissionais da indústria que desejam utilizar ferramentas de apoio à qualidade de testes quanto pesquisadores interessados em investigar *test smells* em novas linguagens de programação.

Palavras-chave: maus cheiros de teste; abordagem independente de linguagem; detecção de maus cheiros de teste

ABSTRACT

Test smells are indicators of poor practices or suboptimal decisions made during test-code development, which can undermine its maintainability and evolution. This topic has gained increasing relevance in recent years, and numerous studies have proposed techniques to detect test smells. However, most existing solutions support only a very limited set of programming languages, often just one. As a result, applying test smell detection to a new language typically requires developing an entirely new solution, even when similar problems have already been addressed elsewhere. In this context, the objective of this work is to propose a language-independent approach for detecting test smells. To understand how this topic has been explored in the literature, we conducted a Systematic Mapping Study (SMS) analyzing 117 studies published up to August 2025. From this review, we identified, among other aspects, the most frequently discussed test smells across multiple programming languages and those considered most critical, that is, the ones that require the greatest attention from the community. Based on these findings, we developed our approach and evaluated its effectiveness in C#, Java, JavaScript, TypeScript, and Python across 10 different test smells, comparing it with existing language-specific techniques. The results were promising: our solution demonstrated superior effectiveness, achieving 97% precision, 96% recall, and a 97% F1-score. To support practical adoption, we also implemented and released a test smell detection tool built upon our language-independent methodology. The outcomes of this work have the potential to greatly simplify the incorporation of new programming languages into test smell detection workflows, enabling the reuse of prior efforts across diverse ecosystems. This benefits both industry practitioners seeking high-quality testing tools and researchers interested in studying test smells in additional languages.

Keywords: test smell; language-independent approach; test smell detection

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ABBREVIATIONS AND ACRONYMS

AST	Abstract Syntax Tree
LAAST	Language-Agnostic Abstract Syntax Tree
LLM	Large Language Model
SLR	Systematic Literature Review
SMS	Systematic Mapping Study

CONTENTS

1 INTRODUCTION

In this chapter, we present the context that motivates this Master's dissertation, along with its goals, research questions, and proposed methodology. This chapter is organized as follows: Section 1.1 presents the context that motivates our work. Section 1.2 describes the goal of this research and outlines the research questions designed to achieve it. Section 1.3 details the methodology adopted to answer the proposed research questions. In Section 1.4, we provide the replication packages for each study conducted as part of this dissertation. Finally, Section 1.5 presents the structure of the remaining chapters of this dissertation.

1.1 Context

Ensuring the quality of a software product is essential, as it directly impacts customer satisfaction, profitability, and even developer efficiency (Barney; Wohlin, 2009). This is especially evident in modern agile development, where practices such as continuous delivery accelerate the transition from feature conception to customer deployment (Tran *et al.*, 2021).

One of the primary mechanisms for ensuring software quality is testing, which helps identify errors and verify the alignment between the expected and actual behavior of the software (Liutenko *et al.*, 2019). Tests can be conducted manually (by a human) or automated through scripts (Garousi; Küçük, 2018). A significant advantage of automated testing is its repeatability and reduced effort compared to manual testing (Garousi; Küçük, 2018).

However, like production code, test code is also susceptible to quality issues. One such issue is the presence of *test smells* (Garousi; Küçük, 2018), which are poor design or implementation choices made during the creation of test cases (Aljedaani *et al.*, 2021). These suboptimal practices may arise due to system complexity or limited developer expertise (Santana *et al.*, 2024).

Developers spend approximately one-quarter of their time writing test code (Beller *et al.*, 2015). Therefore, issues affecting test code, such as test smells, can be particularly harmful, as they may result in undesirable costs and increased development effort (Garousi; Küçük, 2018). A common consequence of test code affected by test smells is reduced readability and maintainability (Panichella *et al.*, 2022), which can hinder its evolution and long-term usefulness (Tran *et al.*, 2021).

Deursen *et al.* (2001) first introduced the concept of test smells by presenting a catalog containing 11 test smells and 6 corresponding refactorings. Since then, numerous studies have proposed additional test smells, refactoring strategies, and tools to detect and resolve them (Aljedaani *et al.*, 2021). Although the majority of existing research has focused on the Java programming language (Aljedaani *et al.*, 2021), many test smells described in the literature are language-agnostic (Paul *et al.*, 2024), that is, they are not tied to language-specific constructs and can appear across a wide range of programming languages.

One example of a language-agnostic test smell is *Assertion Roulette*, which arises when a test case contains multiple undocumented assertions. This makes it difficult to understand the specific reason for a test failure (Paul *et al.*, 2024). The definition of this smell does not rely on any language-specific elements, meaning it can potentially occur in various programming environments. Other examples of language-agnostic test smells include *Conditional Test Logic*, *Duplicate Assert*, *Empty Test*, among others (Paul *et al.*, 2024).

Despite syntactic differences across programming languages, it is possible to detect many test smells using the same high-level detection rules. For example, to identify the *Duplicate Assert* smell, one must search for assertions within a test case that use the same parameters (Peruma *et al.*, 2020). While the syntax for writing assertions may vary between languages, the underlying detection logic remains consistent. This principle applies to several other test smells as well.

Aljedaani *et al.* (2021) conducted a Systematic Literature Review (SLR) to identify existing tools for detecting test smells. The study identified a total of 22 tools, with the majority (19 tools) specifically designed for Java. Furthermore, most of these tools support only a single programming language, with just one tool supporting two. Since that study, new tools have emerged to address test smells in other languages, such as *xNose* for C# (Paul *et al.*, 2024) and *PyNose* for Python (Wang *et al.*, 2021), among others. However, although many of these tools support similar sets of test smells, each one implements language-specific detection mechanisms. This means that, in recent years, adding support for a new programming language typically required building a new tool from scratch, an effort-intensive and time-consuming process. These limitations underscore the need for more general, language-agnostic solutions that facilitate the extension of test smell detection capabilities to additional programming languages with reduced development overhead.

1.2 Goal and Research Questions

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

To develop this approach, we built upon successful results from prior work on test smell detection, as well as research addressing language-agnostic solutions in related areas such as code smells and static code analysis. From this point on, we refer to our language-independent approach for detecting test smells as AromaLIA.

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

RQ_1 How are test smells characterized and addressed in the literature across different programming languages and frameworks?

- **$RQ_{1.1}$** Which programming languages and frameworks are most frequently addressed in discussions of test smells, and what are the most prevalent test smells associated with each?
- **$RQ_{1.2}$** What are the universal test smells across all programming languages and frameworks, and which test smells are specific to particular languages?
- **$RQ_{1.3}$** What refactorings are commonly recommended for addressing test smells in each programming language and framework?
- **$RQ_{1.4}$** What datasets of test smells are available for each programming language and framework?
- **$RQ_{1.5}$** Which test smells are most critical across different programming languages and frameworks?
- **$RQ_{1.6}$** What test smell detection or refactoring tools have been proposed for each programming language and framework?

RQ_2 How does AromaLIA compare to existing language-specific approaches for detecting the test smells?

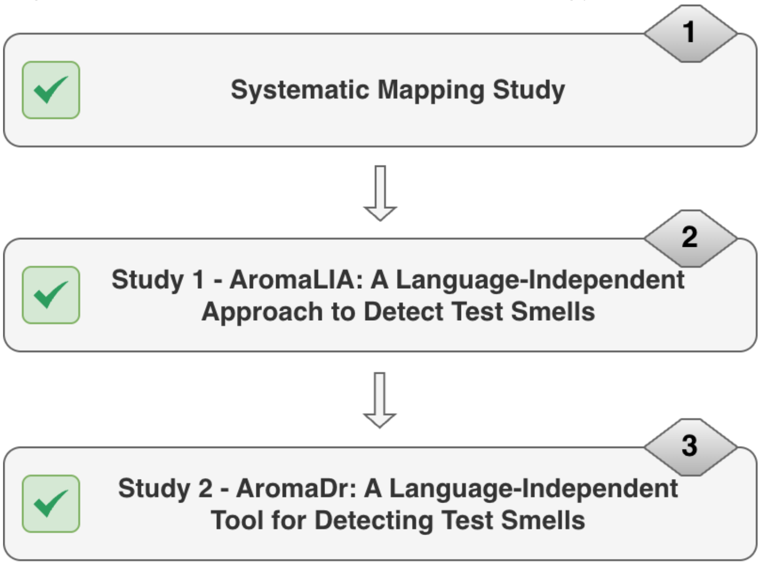
- **$RQ_{2.1}$** How effective is AromaLIA in detecting test smells compared to existing language-specific test smell detection tools?
- **$RQ_{2.2}$** Which programming languages are most challenging for AromaLIA in detecting test smells?

RQ$_{2.3}$ Which test smells are most challenging for AromaLIA to detect, both overall and within specific programming languages?

1.3 Research Methodology

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

Figure 1 – Overview of research methodology



Source: prepared by the author.

1. **SMS**: To address RQ1, we conducted a SMS titled *"Exploring Test Smells Across Programming Languages: A Systematic Mapping Study"*. This study aimed to understand how test smells are handled in the literature from a programming language perspective. Specifically, we investigated: (i) the programming languages and testing frameworks most commonly addressed in the context of test smells; (ii) the most prevalent test smells in each language; (iii) which test smells are language-specific versus language-agnostic; (iv) which test smells are considered most critical; and (v) the refactorings, datasets, and tools used for test smells across different languages. This review provided critical insights and foundational guidance for the subsequent phases of this work.
2. **Study 1 - AromaLIA: A Language-Independent Approach to Detect Test Smells**: To answer RQ2, this study proposed AromaLIA, a language-independent approach for detecting test smells. We evaluated AromaLIA by comparing it with existing language-

specific detection methods across ten test smells in five different programming languages. The results demonstrate that AromaLIA has a strong detection capability compared to the other approaches.

3. **Study 2 - AromaDr: A Language-Independent Tool for Detecting Test Smells**: This study aimed to provide a more user-friendly implementation of our approach by developing a dedicated tool, AromaDr, to support test smell detection. The tool supports a total of 10 test smells across five programming languages (C#, JavaScript, TypeScript, Java, and Python). To the best of our knowledge, this makes AromaDr the test smell detection tool with the broadest language support currently available.

1.4 Study Replicability

In scientific research, ensuring the reproducibility of studies is essential. To support open science practices Mendez *et al.* (2020), we have made the replication packages for each study conducted in this Master's dissertation publicly available on platforms such as Zenodo and GitHub. Each replication package includes, among other resources: scripts, datasets, data analysis artifacts, and the tools produced throughout the research.

Table 1 – Replication packages

Replication Package	Links
Exploring Test Smells Across Programming Languages: A Systematic Mapping Study	https://zenodo.org/records/17093675
AromaLIA: A Language-Independent Approach to Detect Test Smells	https://github.com/publiosilva/aroma-lia-evaluation-rp
AromaDr: A Language-Independent Tool for Detecting Test Smells	https://github.com/publiosilva/aromadr

Source: prepared by the author.

1.5 Organization

In this chapter, we introduced the context of this Master's dissertation, along with its objectives, research questions, and methodology. The remainder of this work is organized as follows: Chapter 2 presents the key concepts required to understand this research. Chapter 3 reviews the relevant literature. Chapter 4 details Step 1 of our methodology, which involves conducting a SMS on test smells across multiple programming languages. In Chapter 5, corresponding to Step 2, we propose our approach and evaluate it by comparing it with existing language-specific techniques. Chapter 6 introduces AromaDr, the tool that implements our language-independent test smell detection approach. Finally, Chapter 7 presents our concluding remarks and outlines directions for future research.

2 BACKGROUND

In this section, we introduce key concepts essential for understanding the rest of this work. Section 2.1 outlines the fundamental ideas related to test smells and provides illustrative examples to demonstrate how they manifest across different programming languages. Section 2.2 presents the main techniques employed for detecting test smells.

2.1 Test Smells

This section introduces the concept of test smells, presenting definitions, examples across different programming languages, detection techniques, recommended refactorings, and tools designed to detect and refactor them.

2.1.1 Concept

Code smells and test smells are related concepts that denote suboptimal practices that may compromise software maintainability (Rio *et al.*, 2024; Oliveira *et al.*, 2024). While code smells typically refer to poor practices in production code, test smells highlight issues in test code (Aljedaani *et al.*, 2021). Although neither type of smell necessarily causes functional failures, both signal potential quality problems that can hinder the maintainability and evolution of a software product (Panichella *et al.*, 2022).

The concept of test smells was first formalized by Deursen *et al.* (2001), who introduced a catalog of 11 test smells along with recommended refactorings. Since then, the catalog has expanded significantly, incorporating both general smells (applicable across multiple programming languages) and language-specific smells (Aljedaani *et al.*, 2021). Furthermore, various tools have also been developed to detect these smells in different programming environments, although the majority of studies focus on Java (Aljedaani *et al.*, 2021).

Test smells are a significant concern for software maintainability, particularly in fast-paced development environments. As research continues to evolve, identifying commonalities across languages and frameworks becomes increasingly valuable for developing more efficient and reusable detection and refactoring solutions.

2.1.2 *Definitions of Test Smells*

The catalog of test smells described in the literature is extensive. In this work, we focus on a subset of ten well-known test smells. Their definitions, along with illustrative examples in different programming languages, are presented below.

Assertion Roulette: This smell occurs when a test case contains multiple assertions without accompanying explanation messages Deursen *et al.* (2001). The main issue is that, when the test fails, it becomes difficult to identify which assertion caused the failure (Peruma *et al.*, 2020). As shown in Listings 1 (Java) and 2 (Python), the smell manifests similarly across languages despite syntactic differences.

```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 1 – Assertion Roulette Java + JUnit example (Peruma *et al.*, 2020)

Conditional Test Logic: This smell arises when a test case includes control flow statements such as conditionals (e.g., `if`, `switch`) or loops (e.g., `for`, `while`) Deursen *et al.* (2001). Although the syntax varies by language, the detection strategy remains the same: identifying control structures within test methods. Examples in Java and Python are provided in Listings 3 and 4.

Duplicate Assert: This smell occurs when the same assertion, with identical parameters, is repeated multiple times within a single test case (Peruma *et al.*, 2020). Listings 5, 6, and 7 show this smell in C#, Java, and Python, respectively. The consistency of its manifestation suggests that cross-language detection techniques can be effectively reused.

```python
def test_clone_non_bare_repo_from_local_test_server():
    repo_uri = integration_git_server_uri_for("small-repo.early.git")
    temp_folder = new_folder()
    repo = clone_repo(repo_uri, temp_folder)

    assert "ba1f63e4430bff267d112b1e8afc1d6294db0ccc" in [obj.hexsha for obj
        in repo.git.rev_list("HEAD")]

    readme_file = temp_folder / "README"
    assert readme_file.exists()
    assert readme_file.stat().st_size == 12
```

Listing 2 – Assertion Roulette Python + PyTest example

```java
@Test
void testIsAdult() {
    List<Integer> ages = List.of(17, 18, 21, 15);
    for (int age : ages) {
        boolean expected = age >= 18;
        assertEquals(expected, PersonUtil.isAdult(age));
    }
}
```

Listing 3 – Condititonal Test Logic Java + JUnit example

```python
def test_is_adult():
    ages = [17, 18, 21, 15]
    for age in ages:
        expected = age >= 18
        assert is_adult(age) == expected
```

Listing 4 – Condititonal Test Logic Python + PyTest example

```csharp
[Fact]
public void TestNoChangeWhenAddingZero()
{
    int sum = 1;
    Assert.Equal(1, sum);
    sum += 0;
    Assert.Equal(1, sum);
}
```

Listing 5 – Duplicate Assert example in C#

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

Listing 6 – Duplicate Assert example in Java

```python
def test_no_change_when_adding_zero():
    sum = 1
    assert sum == 1
    sum += 0
    assert sum == 1
```

Listing 7 – Duplicate Assert example in Python

Empty Test: As the name implies, this smell occurs when a test case lacks executable statements (Peruma *et al.*, 2020). This includes methods that are entirely empty or contain only commented-out code. Listings 8 and 9 illustrate this smell in Java and Python. Notably, in Python, empty test functions often include a `pass` statement due to syntax requirements.

```java
@Test
void testIsAdult() {
    // Not implemented yet
}
```

Listing 8 – Empty Test Java + JUnit example

```python
def test_is_adult():
    # Not implemented yet
    pass
```

Listing 9 – Empty Test Python + PyTest example

Exception Handling: This smell occurs when test cases explicitly raise exceptions or contain exception-handling constructs such as `try/catch` blocks (Peruma *et al.*, 2020). While sometimes justified, their presence can complicate test readability and maintainability. Examples are provided in Listings 10 and 11.

```java
@Test
public void test() {
    // ...
    try {
        // ...
    } catch (Exception e) {
        // ...
    }
    // ...
    throw new Exception();
}
```

Listing 10 – Exception Handling Java + JUnit example

```python
def test_exception_handling():
    # ...
    try:
        # ...
    except Exception as e:
        # ...
    # ...
    raise Exception()
```

Listing 11 – Exception Handling Python + PyTest example

Ignored Test: This smell appears when a test case is deliberately disabled, for instance using the `@Ignore` annotation in JUnit (Peruma *et al.*, 2019). The implementation of this varies depending on the programming language and test framework. Listings 12 and 13 show Python examples using the `unittest` and `pytest` frameworks, respectively. Regardless of syntax, detection involves identifying commands that disable tests.

```python
@unittest.skip("Disabled for now as this test is too flaky")
def test_skipped_example(self):
    self.peerGroup.addConnectedEventListener(self.connectedListener)
    # ...
```

Listing 12 – Ignored Test Python + Unittest example

Magic Number Test: This smell arises when numeric literals, commonly referred to as "magic numbers", are used directly in assertions (Peruma *et al.*, 2020). Since these values lack contextual meaning, they should be replaced by well-named constants or variables. Listings 14 and 15 present examples in Java and Python.

```
@pytest.mark.skip(reason="Disabled for now as this test is too flaky")
def test_skipped_example():
    self.peerGroup.addConnectedEventListener(self.connectedListener)
    # ...
```

Listing 13 – Ignored Test Python + PyTest example

```
@Test
public void test() {
    // ...
    assertEquals(result, 342);
}
```

Listing 14 – Magic Number Java + JUnit example

```
def test():
    # ...
    assert result == 342
```

Listing 15 – Magic Number Python + PyTest example

Redundant Print: This smell is present when print statements are used within test cases (Peruma *et al.*, 2020). These are typically remnants of debugging activity and often go unnoticed in the final code. Listings 16 and 17 demonstrate this smell in Java and Python.

```
@Test
public void test() {
    // ...
    System.out.println("Test");
    // ...
}
```

Listing 16 – Redundant Print Java + JUnit example

```
def test():
    # ...
    print("Test")
    # ...
```

Listing 17 – Redundant Print Python + PyTest example

Sleepy Test: This smell occurs when a test case includes commands that intentionally delay execution, such as `Thread.sleep()` in Java. Such delays can make test suites slower and less reliable. Listings 18 and 19 offer examples in Java and Python.

```java
@Test
public void test() {
    // ...
    Thread.sleep(1000);
    // ...
}
```

Listing 18 – Sleepy Test Java + JUnit example

```python
def test():
    # ...
    time.sleep(1)
    # ...
```

Listing 19 – Sleepy Test Python + PyTest example

Unknown Test: This smell arises when a test method lacks any assertion statements (Peruma *et al.*, 2020). Such tests may give a false impression of test coverage without actually validating behavior. Examples can be found in Listings 20 and 21.

```java
@Test
public void test() {
    SystemManager systemManager = new SystemManager();
    // ...
    systemManager.doSomething();
    // ...
}
```

Listing 20 – Unknown Test Java + JUnit example

```python
def test():
    systemManager = SystemManager()
    # ...
    systemManager.doSomething()
    # ...
```

Listing 21 – Unknown Test Python + PyTest example

2.2 Test Smell Detection

Researchers have proposed various techniques for detecting test smells across multiple programming languages (Aljedaani *et al.*, 2021). Four prominent approaches for test smell detection include Metrics, Rules/Heuristics, Information Retrieval, and Dynamic Tainting (Aljedaani *et al.*, 2021). Each technique applies different principles and methodologies to identify suboptimal patterns in test code.

Metrics-based detection measures the impact of test smell symptoms using structural and semantic metrics, typically relying on predefined thresholds (Aljedaani *et al.*, 2021). For instance, tools developed by Rompaey *et al.* (2007) can detect smells such as General Fixture and Eager Test by analyzing metrics like the Number of Objects Used in setup methods. Although this approach is valuable, its adoption is less widespread due to the limitations and variability of metric-based thresholds (Aljedaani *et al.*, 2021).

Rules/Heuristics are widely used in test smell detection because they rely on well-defined patterns observable in the source code (Aljedaani *et al.*, 2021). These techniques often supplement metric-based detection to improve accuracy (Aljedaani *et al.*, 2021). A common example is the identification of the Assertion Roulette smell by analyzing assertion statements (Aljedaani *et al.*, 2021). TSDetect (Peruma *et al.*, 2020), a popular tool for Java, applies this technique by parsing the Abstract Syntax Tree (AST) to identify smells efficiently.

Information Retrieval techniques extract and normalize information from test code, applying natural language processing and machine learning to identify smells based on textual features (Aljedaani *et al.*, 2021). Tools like those proposed by Lambiase *et al.* (2020) exemplify this method. However, these techniques can struggle when relevant features are missing or inadequately represented in the code (Aljedaani *et al.*, 2021).

Dynamic Tainting involves monitoring test code during execution, using runtime data annotated with predefined taint values to detect undesirable behaviors (Aljedaani *et al.*, 2021). This approach is particularly effective for identifying test dependencies and rotten green tests (tests that pass but hide underlying issues) (Aljedaani *et al.*, 2021). An example of this technique is found in the work of Zhang *et al.* (2014).

Rule and heuristic-based techniques are the most commonly employed, given their reliance on deterministic patterns (Aljedaani *et al.*, 2021). Dynamic analysis and information retrieval are used in more specialized contexts, while metrics-based techniques are less frequently adopted due to their dependency on carefully calibrated thresholds (Aljedaani *et al.*, 2021).

3 RELATED WORK

In this section, we discuss relevant studies that are related to our work in various aspects. This chapter is organized as follows: in Section 3.1, we present a review of the literature on test smells; in Section 3.2, we examine existing approaches for test smell detection and compare them with the approach proposed in this study. Finally, in Section 3.3, we summarize our conclusions.

3.1 Literature reviews on test smells

Garousi e Küçük (2018) conducted a multivocal literature review of test smells, examining 166 sources from both the scientific and grey literature. Their study provides practitioners with a detailed classification of test smells, along with recommendations for test automation, guidelines, techniques, and tools for their prevention, detection, and correction. An interesting finding of this work is that the authors identified some test smells as *generic* (i.e., independent of any specific framework), while others were *framework-specific*. However, they also reported that some test smells initially observed in particular contexts (for example, within the JUnit framework) are likely to appear in other contexts as well.

Rwemalika *et al.* (2022) aimed to address gaps in the research on test smells in System User Interactive Tests (SUITs). The authors conducted an exploratory analysis and a multivocal literature review, resulting in a catalog of 35 SUIT-specific test smells. They also analyzed the prevalence of these smells in both industrial and open-source projects. Although this work focused on a specific context (SUITs), the authors discovered several test smells that are well-known in other contexts, such as the *Eager Test* smell. This finding highlights that some test smells are general and can manifest across a wide variety of scenarios.

Tran *et al.* (2021) aimed to develop a comprehensive model for assessing the quality of test cases and test suites. To achieve this, they conducted a literature review of 49 secondary studies. As a result, the paper presents a model that supports the assessment and improvement of test artifact quality. The authors examined both test smells and other quality attributes, highlighting the relationship between test smells and the overall quality of the test code.

Aljedaani *et al.* (2021) sought to provide practitioners and researchers with a catalog of tools for detecting test smells. To achieve this, the authors conducted a literature review and identified 22 such tools. They analyzed the test smells detected by each tool, the programming languages and frameworks supported, the detection strategies used, and the extent to which each tool is adopted. The supported languages include Java, Scala, Smalltalk, and C++, with Java being the most prominent. Most tools were designed specifically for Java. The results also revealed that most of the existing tools for detecting test smells are language-specific; among the 22 tools reported, only one supported more than one programming language.

Based on these studies, several research gaps emerge. None of the existing literature reviews specifically focuses on test smells from a language-oriented perspective. Therefore, in our work, we conducted a SMS to explore test smells from a programming language standpoint and to investigate topics such as the most common programming languages studied in the context of test smells, the test smells that are most prevalent in each language, those that are language-specific and those that occur across multiple languages, the tools available for each language, among other aspects.

Previous literature reviews have also highlighted that, while the same test smells can occur across multiple programming languages, most existing detection approaches remain language-specific. Consequently, in this master's dissertation, we address this issue by proposing a language-independent approach for detecting test smells.

3.2 Test Smell Detection Approaches

3.2.1 Traditional Test Smell Detection Approaches

Recent research has introduced several tools aiming to detect test smells in software projects to improve test code quality and maintainability. Bodea (2022) presented PyTest-Smell, a tool designed to identify test smells in Python code written using the PyTest framework. Their goal was to provide an easy-to-install and user-friendly solution for Python developers. The tool detects ten different types of test smells and was validated with a detection rate exceeding 90% for smelly test suites.

Similarly, Paul *et al.* (2024) proposed xNose, a tool for detecting test smells in C#, thereby expanding research into a language that had previously been underexplored in this context. The xNose tool detects sixteen types of test smells and was validated using three distinct projects, achieving an average precision of 96.97% and an average recall of 96.03%.

Another notable contribution is that of Peruma *et al.* (2020), who developed TSDETECT, a tool that employs rule-based techniques over AST to detect test smells. TSDETECT achieved an average precision of 96% and an average recall of 97% in benchmarks performed on 65 unit test files covering 19 distinct test smells. Owing to its substantial number of GitHub forks, it is widely regarded as one of the most adopted tools in this domain, as noted by Aljedaani *et al.* (2021).

In the literature, SniffML (Lopes *et al.*, 2024) is a pioneering tool for language-independent test smell detection, 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*. In its evaluation, the tool achieved outstanding results, with a precision of 97.99%, a recall of 96.90%, and an F-measure of 97.44%.

In our work, we propose a language-independent approach to test smell detection and evaluate a tool based on this approach by comparing its results with those obtained from the aforementioned tools. In the AromaLIA-based tool, we implemented exactly the same set of test smells supported by PyTest-Smell. However, when applied to our dataset, PyTest-Smell produced results notably inferior to those originally reported by its authors. In contrast, the AromaLIA-based tool outperformed PyTest-Smell in terms of precision, recall, and F1-score.

Regarding the xNose tool, it supports a broader range of test smells than the AromaLIA-based tool used in our study. As a result, we could not compare the detection results for smells that we did not implement. Moreover, xNose does not support one of the test smells included in our work. Nevertheless, for the nine test smells supported by both tools, the AromaLIA-based tool achieved superior precision, recall, and F1-score metrics.

With respect to TSDETECT, its use of an AST makes it conceptually related to our approach, as both rely on AST-based, rule-driven detection. However, a key distinction lies in our use of a language-agnostic AST rather than a conventional language-specific one. Furthermore, instead of applying rules directly to the AST, we introduce an intermediate processing layer that enables more language-independent detection algorithms, thereby reducing the effort required to support new programming languages. TSDETECT supports all ten test smells included in the scope of our study, as well as nine additional smells. For these nine extra smells, a direct comparison was not possible. Nonetheless, when considering only the ten shared smells, the AromaLIA-based tool achieved higher precision, recall, and F1-scores.

When comparing SniffML with the AromaLIA-based tool, our tool supports a wider range of programming languages and detects all the test smells identified by SniffML (plus three additional ones). Another key advantage 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 Language-Agnostic Abstract Syntax Tree (LAAST), which supports a significantly larger number of programming languages and can be further extended to include even more in the future (Ardito *et al.*, 2020).

Additionally, the test smell detection approach differs between the tools. SniffML searches for patterns in the code via the XML generated by SRCML. For example, to locate assertions, it looks for the keyword "assert," which allows it to detect test smells such as *Duplicate Assert* and *Assertion Roulette*. However, this strategy can produce false positives. If a developer creates a method whose name contains the word "assert" but is not actually an assertion, SniffML may incorrectly identify it as such, potentially compromising test smell detection.

In contrast, the AromaLIA-based tool uses more precise rules that account for differences between programming languages and rely on the node types in the AST generated from the test code. This approach reduces the likelihood of such false positives and improves the accuracy of test smell detection.

3.2.2 LLM-based Test Smell Detection Approaches

Recent studies in the literature investigated the use of Large Language Model (LLM) for test smell detection. If successful, this approach holds great potential as a language-independent solution, enabling the detection of test smells across multiple programming languages with minimal development effort.

In the work of Lucas *et al.* (2024), the authors evaluated three large language models (ChatGPT-4, Mistral Large, and Gemini Advanced) for their ability to detect test smells. The study assessed the models on 30 types of test smells across seven different programming languages. A key limitation of this work, however, was the small number of test cases for certain languages. In some cases, only a single test case was available. Among the evaluated models, ChatGPT-4 achieved the highest accuracy, reaching an overall rate of 70%.

Similarly, SANTANA JUNIOR *et al.* (2025) examined the use of LLMs for both detection and refactoring tasks. The evaluated models were GPT-4-Turbo, LLaMA 3 70B, and Gemini 1.5 Pro. The authors tested these models on Java and Python samples. Among these, Gemini achieved the highest detection accuracy, with 74.35% for Python and 80.32% for Java.

In our work, we did not explicitly report the accuracy of the AromaLIA-based tool, as precision, recall, and F1-score already provide a comprehensive view of performance.

However, the accuracy value is available in our replication package. Our approach achieved an overall accuracy of 98%, outperforming the results of the LLM-based approaches reported in the mentioned studies. Therefore, while LLM-based methods show great promise for reducing development effort in test smell detection, our results indicate that a more traditional, non-LLM-based approach, such as AromaLIA, may offer higher effectiveness in detecting test smells.

3.2.3 Comparison of works on test smell detection approaches

Table 2 presents a comparison between our work and the previously mentioned test smell detection approaches across six aspects: the number of supported languages, the number of supported test smells, language independence, the specific supported languages, the detection technique used, and the reported overall accuracy of each approach. For some studies, the authors did not report overall accuracy, therefore, these were not included in the table.

As shown in the table, our approach stands out compared to those using more traditional techniques, particularly in the number of supported languages. Our approach also supports a reasonable number of test smells relative to the other methods. Regarding LLM-based approaches, the method proposed by Lucas *et al.* (2024) supports a larger number of languages; however, our approach achieved a higher overall accuracy.

3.3 Conclusion

The studies presented in this section highlight efforts over the past few years to develop solutions for test smell detection, reflecting the growing recognition of the topic's importance. Many existing approaches are language-specific, which limits their applicability to a single programming language and requires the development of new tools to accommodate additional languages in the context of test smell detection. More recently, initiatives proposing language-independent solutions have emerged, such as SniffML (Lopes *et al.*, 2024). As noted earlier, however, this approach relies on a tool to extract a form of AST from the test code that currently supports only four languages, which restricts its generalizability.

Even more recently, with the advent of LLM, there has been increasing interest in leveraging these solutions for test smell detection. This strategy has significant potential to reduce the effort required to detect test smells across new languages, as it does not necessitate

Table 2 – Comparison of Test Smell Detection Approaches

Approach	# Lang.	# Smells	Lang.-Indep.	Supported Languages	Detection Technique	Accuracy
AromaLIA (our work)	5	10	Yes	C#, Java, JavaScript, Python, TypeScript	Language-independent + Rule-based	98%
PyTest-Smell (Bodea, 2022)	1	10	No	Python	Rule-based	—
xNose (Paul *et al.*, 2024)	1	16	No	C#	Rule-based	—
TSDETECT (Peruma *et al.*, 2020)	1	19	No	Java	Rule-based	—
SniffML (Lopes *et al.*, 2024)	4	7	Yes	C, C++, C#, Java	Language-independent + Rule-based	—
LLM-based (Lucas *et al.*, 2024)	7	30	Yes	C#, Java, JavaScript, Python, Ruby, Smalltalk, TTCN-3	LLM	70%
LLM-based (SANTANA JUNIOR *et al.*, 2025)	2	15	Yes	Java, Python	LLM	80%

Source: prepared by the author.

building a detection tool from scratch. However, when comparing the results obtained by our approach in this work, we observe that the accuracy of LLM-based approaches remains lower, indicating that further research is needed to improve the correctness of detections using such methods.

4 EXPLORING TEST SMELLS ACROSS PROGRAMMING LANGUAGES: A SYSTEMATIC MAPPING STUDY

This chapter outlines the initial step of the methodology employed in this Master's dissertation: a SMS focusing on test smells across various programming languages. The chapter is structured as follows: Section 4.2 details our systematic research methodology, Section 4.3 presents our comprehensive findings and analysis, Section 4.4 provides an in-depth examination and interpretation of the results, Section 4.5 addresses potential threats to validity and our mitigation strategies, and Section 4.6 concludes with key insights and directions for future research.

4.1 Introduction

Testing stands as one of the fundamental practices for ensuring code quality in software development (Tran *et al.*, 2021). While manual testing requires human intervention, automated testing offers a more efficient alternative through test scripts that provide repeatability with significantly reduced effort (Garousi; Küçük, 2018). For organizations embracing continuous delivery, automated testing becomes particularly crucial, enabling them to accelerate the journey from feature conception to customer delivery while maintaining high quality standards (Tran *et al.*, 2021).

However, just as production code must evolve to remain relevant, test code requires similar attention to maintain its effectiveness over time (Tran *et al.*, 2021). The consequences of poorly implemented test code extend beyond mere inefficiency; they translate into significant extra costs and unnecessary effort (Garousi; Küçük, 2018). This concern becomes particularly pressing when we consider that developers dedicate approximately one-quarter of their time to engineering test code (Beller *et al.*, 2015). Among the various factors that can compromise the maintainability and evolution of test code, test smells emerge as a critical concern. Test smells represent poor design choices made by developers during test implementation (Aljedaani *et al.*, 2021), and these choices can ultimately result in test code that becomes difficult to maintain and evolve (Panichella *et al.*, 2022).

The foundation of test smell research was established by Deursen *et al.* (2001), who introduced an initial catalog of 11 test smells along with their corresponding refactoring strategies. This seminal work has since inspired a wealth of research, with numerous studies expanding the field by proposing new test smells, developing refactoring techniques, and creating detection and refactoring tools. The research landscape has been further enriched by comprehensive literature reviews that have compiled extensive catalogs of test smells (Garousi; Küçük, 2018; Garousi *et al.*, 2019), systematically identified and analyzed test smell detection tools (Aljedaani *et al.*, 2021), and explored test smells within specific domains (Rwemalika *et al.*, 2022). Despite this substantial body of work, a critical gap remains: none of these studies specifically examines the prevalence of test smells across different programming languages and frameworks. Consequently, fundamental questions remain unanswered, such as which test smells are language-specific and which are more prevalent in particular programming environments.

Overlooking programming language perspectives in test smell research has created a significant practical challenge. Many studies propose state-of-the-art solutions for test smell detection and refactoring that are tailored to specific programming languages, often focusing on just one language at a time. This narrow scope severely limits the reusability of research findings across different programming environments. Consequently, researchers find themselves repeatedly building solutions from scratch to address identical problems in several languages, a process that is both complex and time-consuming. The development of more general and reusable solutions could dramatically simplify this process, but achieving this goal requires first illuminating the current landscape, identifying potential research directions, and establishing a foundation for future investigations.

To address this critical gap, our work explores the intricate relationship between test smells and the programming languages and frameworks in which they manifest. Through a comprehensive SMS, we gathered and analyzed data from papers published up to August 2025, ultimately selecting 117 papers that explicitly addressed test smells while citing specific programming languages or test frameworks. Our investigation revealed a rich landscape of 213 distinct test smells distributed across 11 programming languages (including Java, Python, JavaScript, TypeScript, C#, and others) and 15 test frameworks (such as JUnit, PyTest, Jest, xUnit, and others).

Our findings reveal several noteworthy patterns. Java emerges as the most frequently referenced language, with JUnit standing out as the predominant testing framework. In contrast, Python has gained substantial traction in recent years, reflecting an increasing diversity within the research landscape. Regarding test smells, *Assertion Roulette* and *Eager Test* are the most frequently discussed in the literature, whereas *Unknown Test* demonstrates the broadest coverage across programming languages. We also identified language-specific test smells, such as those observed exclusively in TTCN-3.

Beyond prevalence, we examined test smell criticality from two complementary perspectives: developers' perceptions and their impact on maintainability. Interestingly, despite their frequent discussion, *Assertion Roulette* and *Eager Test* are not perceived as highly critical. Instead, we found the most severe test smells to be *Sleepy Test*, *Ignored Test*, *Resource Optimism*, and *Mystery Guest*. Finally, our study contributes by cataloging a broad set of refactoring strategies for different test smells, along with curated datasets and a catalog of detection and refactoring tools.

4.2 Research Methodology

This section outlines the procedures used to conduct a systematic mapping study on test smells across programming languages. This review follows the guidelines established by Ampatzoglou *et al.* (2019).

4.2.1 Goal And Research Questions

The main goal of this work is to analyze test smells in the literature from a programming language perspective, providing comparisons and insights on this topic. To achieve this goal, we defined the following research questions:

RQ1. *Which programming languages and frameworks are most frequently addressed in discussions of test smells, and what are the most prevalent test smells associated with each?* This question aims to identify the prevalence of certain programming languages and frameworks in test smell discussions and determine the most frequently addressed test smells for each.

RQ2. *What are the universal test smells across all programming languages and frameworks, and which test smells are specific to particular languages?* This question seeks to identify test smells common across all (or most) programming languages and frameworks, as well as those specific to certain languages and frameworks.

RQ3. *What refactorings are commonly recommended for addressing test smells in each programming language and framework?* This question aims to identify the techniques used to refactor test smells across different languages and frameworks.

RQ4. *What datasets of test smells are available for each programming language and framework?* This question aims to identify and catalog datasets of test smells across different languages and frameworks.

RQ5. *Which test smells are most critical across different programming languages and frameworks?* This question aims to gather evidence from the literature about the severity of various test smells in different programming languages and frameworks. We analyzed criticality from two perspectives: a general developer's perception of the test smell and the impact of the test smell on maintainability.

RQ6. *What test smell detection or refactoring tools have been proposed for each programming language and framework?* This question seeks to identify detection and refactoring tools proposed for each language and framework, providing practical and useful information for both academics and practitioners aiming to address test smells in specific languages or frameworks.

With these research questions, we aimed not only to provide a language perspective on test smells through RQ1 and RQ2, but also to offer insights that can support future research. These insights can help develop solutions to expand the set of studied languages in the context of test smells or create general approaches for detecting and refactoring test smells across multiple languages. Specifically, our contributions include common refactorings for test smells that could be implemented in new tools (RQ3), datasets of test smells in various languages that could be used to validate new studies (RQ4), identification of the most critical test smells that require greater attention (RQ5), and a list of tools that can serve as a reference or comparison basis for new studies (RQ6).

4.2.2 Search Strategy

From the research questions, we identified relevant terms and their synonyms to be used as keywords for the search. We organized the keywords into three groups, separated by AND clauses. The first group aims to retrieve works specifically addressing test smells. The second group targets works that deal with refactoring, datasets, or tools related to test smells. The third group focuses on filtering works that discuss programming languages. The search string used is as follows: *("test smell" OR "test smells") AND ("tool" OR "detect" OR "refactoring" OR "dataset") AND ("language" OR "languages" OR "code").*

To capture as many relevant works as possible, we left the start of the search period open and limited only the end to August 2025. We also selected relevant and high-quality digital libraries to conduct the automatic search, as shown in Table 3.

Table 3 – Digital libraries for automatic search

Name	URL
ACM Digital Library	http://portal.acm.org
El Compendex	http://www.engineeringvillage.com
IEEE Digital Library	http://ieeexplore.ieee.org
Science@Direct	http://www.sciencedirect.com
Scopus	http://www.scopus.com
Springer Link	http://link.springer.com

Source: prepared by the author.

4.2.3 Selection Strategy

The search described in the previous sections yielded a substantial number of studies (787 in total). The first step was to remove duplicate papers, which resulted in the elimination of 333 duplicates. Next, we applied the inclusion and exclusion criteria detailed in Table 4.

Table 4 – Inclusion and exclusion criteria

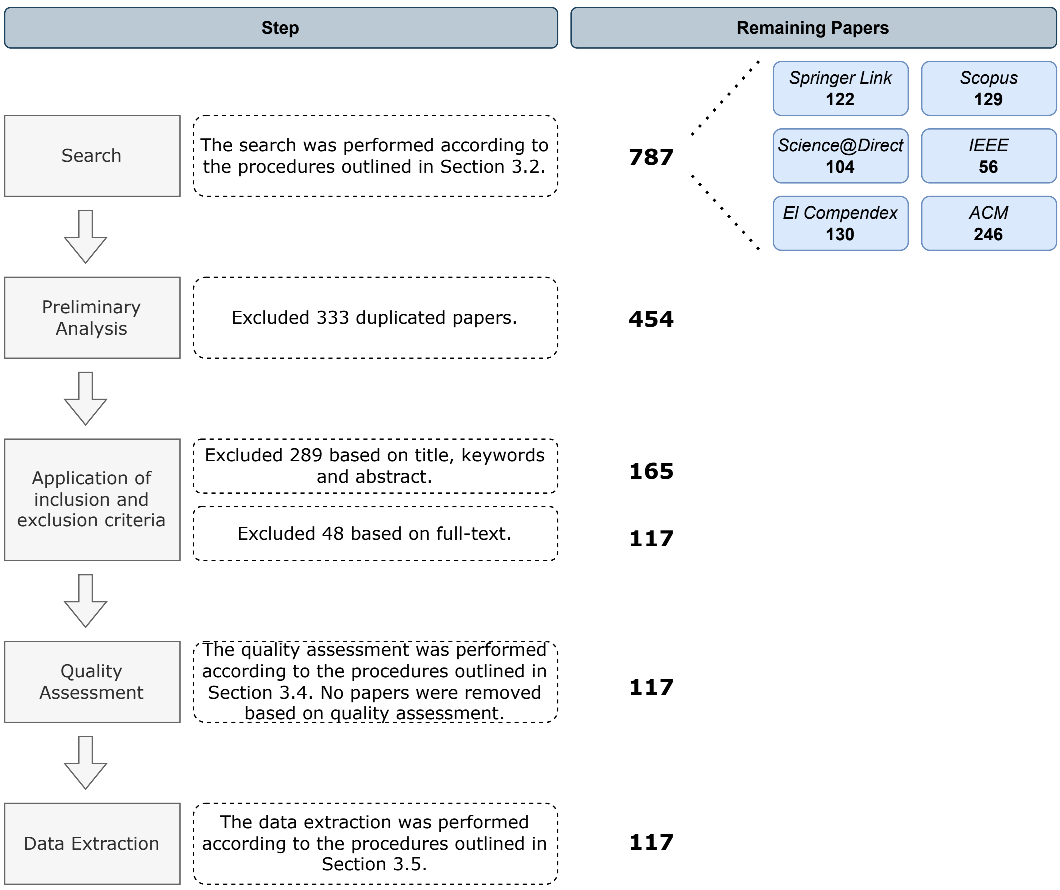
Inclusion Criteria	Exclusion Criteria
Address test smells in one or more programming languages or frameworks	Does not address test smells or does not address test smells in some programming languages or frameworks
Papers in English	Papers in any language other than English
Primary studies	Secondary and tertiary studies
	Duplicated papers
	Proceedings

Source: prepared by the author.

By applying the inclusion and exclusion criteria, 276 papers were excluded based solely on their titles, keywords, and abstracts. Following this, we reviewed the full texts of the remaining papers. During this stage, we removed 48 papers based on the inclusion and exclusion criteria applied to the full texts. This process resulted in a final set of 117 papers. Subsequently, we conducted a quality assessment of these papers, but no papers were excluded based on this assessment. The entire process is illustrated in Figure 2.

For the paper selection process, we employed a peer review approach involving three of the four co-authors. The review team consisted of two doctoral-level researchers with extensive expertise in test smells and software testing, and one master's student with moderate experience in the domain. This composition ensured both methodological rigor and diverse perspectives during the selection process. We conducted the selection process in two distinct phases: (1) initial screening based on titles, abstracts, and keywords, and (2) full-text review for papers that passed the initial screening. To maintain methodological transparency, any papers with discordant initial assessments were automatically advanced to the full-text review phase, allowing for more comprehensive evaluation. This approach ensured we did not prematurely exclude potentially relevant papers due to limited information in abstracts or titles. In cases where reviewers disagreed on paper inclusion or exclusion, the authors had a discussion where they debated their viewpoints, carefully considering how each paper aligned with our established selection criteria. The inter-rater agreement was high throughout the selection process, with minimal disagreement among reviewers.

Figure 2 – Selection process



Source: prepared by the author.

4.2.4 *Quality Assessment*

We conducted a quality assessment to evaluate the strength of the evidence in the studies included in this SMS. We derived the questions (Table 5) for the evaluation from the 11 quality criteria outlined by Machado *et al.* (2014), which cover four key aspects: reporting, rigor, credibility, and relevance. We used a binary scale for the assessment: “yes” (1) and “no” (0).

With 11 questions, the maximum achievable score is 11. The studies received scores ranging from 5 to 10, with a median of 9 and an average of 8.61. These results indicate that the papers included in this SMS exhibit high quality. We did not exclude any papers based on the quality assessment.

Table 5 – Quality assessment questions

Number	Question	Issue
1	Is the paper based on research and not merely a “lessons learned” report based on expert opinion?	Reporting
2	Is there a clear statement of the aims of the research?	Reporting
3	Is there an adequate description of the context in which the research was carried out?	Reporting
4	Was the research design appropriate to address the aim of the research?	Rigour
5	Was there a control group with which to compare the treatments?	Rigour
6	Was the data collected in a way that addressed the research issue?	Rigour
7	Was the data analysis sufficiently rigorous?	Rigour
8	Has the relationship between researcher and participants been considered to an adequate degree?	Credibility
9	Is there a clear statement of findings?	Credibility
10	Is the study valuable for research or practice?	Relevance
11	Are there any practitioner-based guidelines?	Relevance

Source: prepared by the author.

4.2.5 Data Extraction and Synthesis

For data extraction, we designed a form, shown in Table 6, to address the four research questions defined for this SMS. The questions in the form cover the following aspects: test smells, programming languages and frameworks mentioned in the studies, detection or refactoring tools for test smells, and evidence regarding the criticality of each test smell. We extracted all data in plain text format.

Some test smells appear in the literature under different names, even though they describe the same underlying issue. To avoid reporting duplicate test smells, we relied on an open catalog of test smells (Soares *et al.*, 2023), which served as a reference to standardize the terminology used in this SMS. The same catalog also provides definitions for several of the test smells considered in this study. For new test smells not included in the catalog, the original papers provide definitions, and Appendix A lists those papers.

Table 6 – Data extraction form

Question	Description	RQs
Languages and Frameworks Mentioned	Programming languages and frameworks discussed in the study	RQ1/RQ2
Test Smells Identified	Test smells associated with the mentioned languages/frameworks	RQ1/RQ2
Test Smells Refactorings	Refactorings for the identified test smells	RQ3
Test Smells Datasets	Test smells datasets used or referenced on the selected papers	RQ4
Test Smells Criticality	Assessment or discussion of the severity or impact of the identified test smells	RQ5
Detection or Refactoring Tool Proposed	Description of any detection or refactoring tools proposed in the study	RQ6

Source: prepared by the author.

4.3 Results

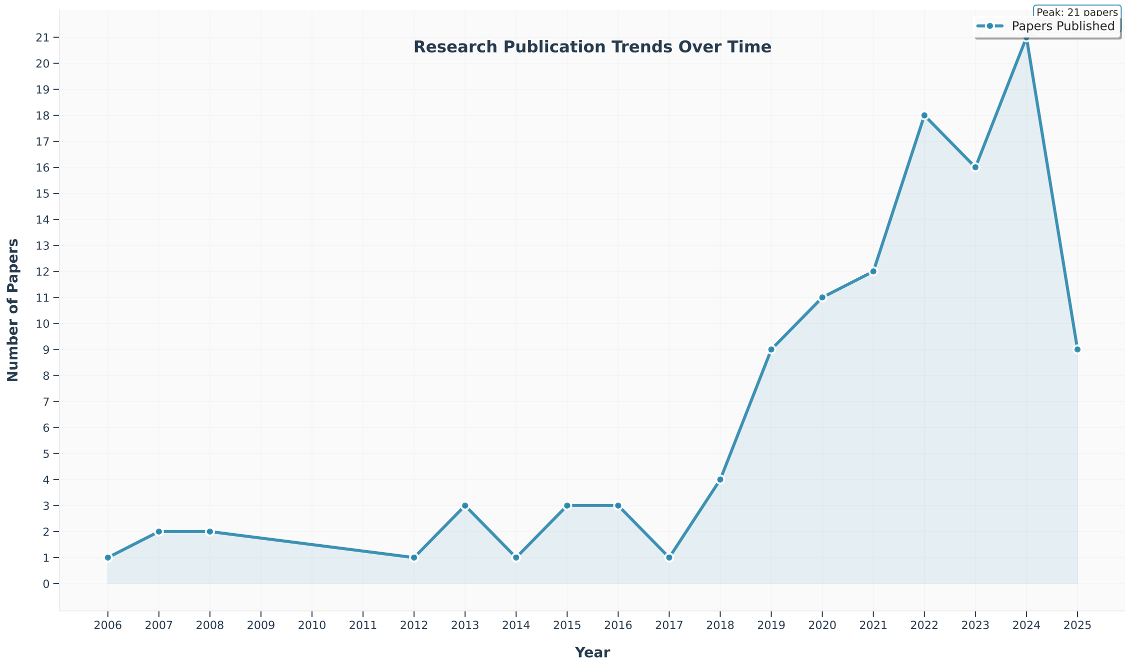
In this section we present an overview of the results we obtained and answer the research questions.

4.3.1 Overview

After applying the process shown in Figure 2, we retained 117 papers. Figure 3 indicates the number of studies (out of the 117 accepted) per year. The figure demonstrates a notable increase in studies involving test smells since 2018, indicating that researchers have shown growing interest in this topic.

Our analysis revealed several key quantitative insights: (1) The 117 papers collectively identified 213 distinct test smells across 11 programming languages and 15 testing frameworks, with Java/JUnit being the most frequently studied combination (appearing in 62 papers); (2) The most universal test smell was *Unknown Test*, appearing in 11 different language/framework combinations, followed by *Conditional Test Logic*, *Exception Handling*, and *Magic Number*, each appearing in 10 combinations; (3) A total of 94 distinct refactoring techniques were identified, with *Extract Method* being the most frequently mentioned (44 times); (4) The criticality analysis covered 33 test smells, with *Sleepy Test* identified as the most critical; and (5) The tooling landscape includes 36 tools, with 61% currently available and only 14% supporting both detection and refactoring capabilities.

Figure 3 – Final papers per year



Source: prepared by the author.

4.3.2 *RQ1: Which programming languages and frameworks are most frequently addressed in discussions of test smells, and what are the most prevalent test smells associated with each?*

To answer this research question, we analyzed all the languages and frameworks related to test smells that the selected papers mentioned. Table 7 shows these languages and frameworks along with the number of studies that mentioned each one. The analyzed studies mention a total of 11 programming languages. The papers also mention 15 different testing frameworks: 1 for C#, 3 for Java, 2 for JavaScript, 7 for Python, 1 for Scala, 2 for TypeScript, and one generic framework, which is Robot (Bisht, 2013).

The languages identified span different typing systems and programming paradigms. Static typing languages include C, C#, C++, Java, Kotlin, Scala, and TTCN-3, while dynamic typing languages include JavaScript, Python, and Smalltalk. The majority of languages (8 out of 11) use strong typing, with only C and C++ using weak typing. Most languages support multiple paradigms, with Java, C#, C++, JavaScript, Kotlin, Python, and TypeScript being multi-paradigm languages supporting object-oriented, functional, and/or procedural programming. Only Smalltalk is purely object-oriented, while TTCN-3 is declarative and specifically designed for telecommunications testing (Grabowski *et al.*, 2003).

Figure 4 shows a trend of how often each language presented in Table 7 has been

Table 7 – Languages and frameworks

Language	Framework	Number of Papers
C	-	1
C++	-	1
C++	GoogleTest	1
C#	-	1
C#	xUnit	1
Java	JUnit	62
Java	-	27
Java	TestNG	4
Java	Selenium WebDriver	1
JavaScript	Chai	1
JavaScript	Jest	1
Kotlin	-	1
Python	Unittest	10
Python	PyTest	8
Python	Nose	2
Python	-	1
Python	absl	1
Python	GoogleTest	1
Python	Numpy	1
Python	Tensorflow	1
Scala	ScalaTest	3
Smalltalk	-	1
TTCN-3	-	2
Typescript	Jest	2
Typescript	Chai	1
-	Robot Framework	2

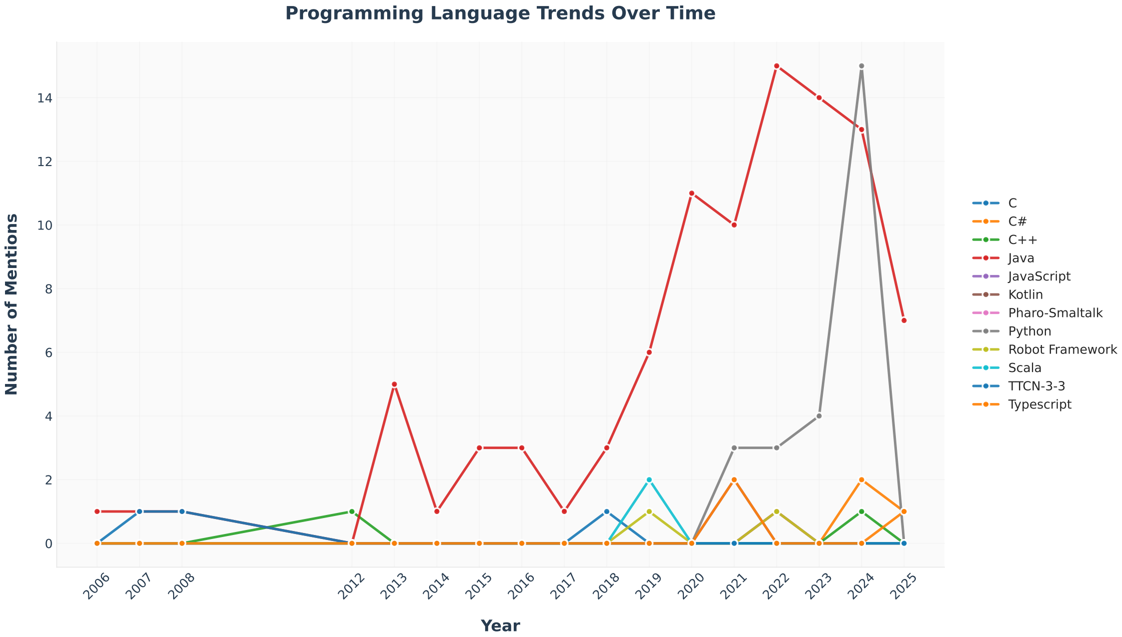
Source: prepared by the author.

mentioned over the years. As can be seen in the figure, Java appeared almost every year, and in the most recent years, the number of mentions has increased. Python is a language that has started to appear with some frequency in the last few years. Some languages, like C and C++, have only been cited in one year. This trend shows that the language most frequently mentioned in the context of test smells is Java. Python, on the other hand, has only begun to appear regularly in the last few years, but its frequency of mentions has grown rapidly, surpassing Java in 2024. Other languages lack analysis in the context of test smells.

Figure 5 shows the same trend for the frameworks associated with each language. It demonstrates that the framework most commonly associated with Java in the majority of the papers is JUnit.

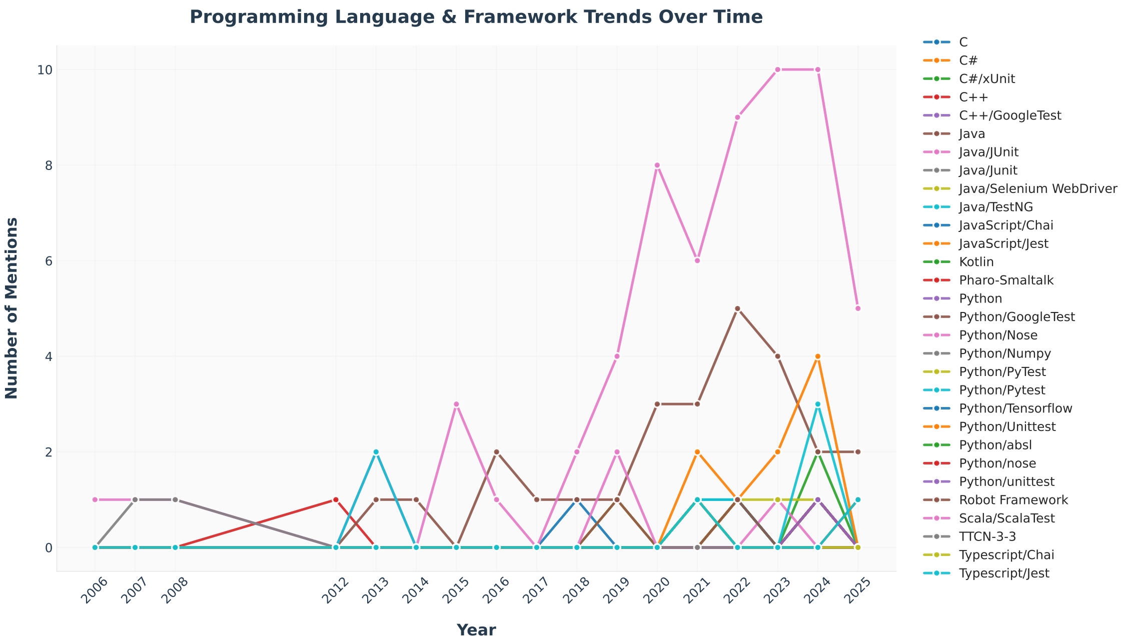
We analyzed the test smells most frequently discussed in relation to tests in each language. The test smell most commonly addressed in discussions about test smells in Java is *Eager Test*, with 51 mentions. In Python, the most widely discussed test smells are *Conditional Test Logic* and *Exception Handling*, with ten mentions each. The other languages have few

Figure 4 – Language trend



Source: prepared by the author.

Figure 5 – Language/framework trend



Source: prepared by the author.

mentions of each test smell.

We also examined the most frequently addressed test smells across all languages. Table 8 shows the top 20 most mentioned test smells across all languages. The table shows that the test smells *Assertion Roulette* and *Eager Test* stand out as the most cited. In this table, we omitted languages that do not mention any of these 20 test smells. One interesting fact is that Java is related to all 20 test smells, while Python is associated with 19 of them.

Table 8 – Top 20 most cited test smells across all languages

Test Smell	C	C#	C++	Java	Kotlin	Python	Scala	JavaScript	Typescript	Total
Assertion Roulette	0	2	1	49	0	7	2	1	1	63
Eager Test	0	1	0	51	0	2	2	1	1	59
Conditional Test Logic	0	3	1	31	0	10	0	2	1	48
Duplicate Assert	0	3	1	32	0	7	0	1	1	45
Exception Handling	0	2	1	27	0	10	0	1	1	42
Magic Number	0	2	2	28	0	7	0	1	1	41
Sensitive Equality	0	1	0	37	0	1	2	0	0	41
Mystery Guest	0	0	0	36	0	1	2	1	1	41
Unknown Test	0	3	1	26	0	7	0	1	1	39
General Fixture	0	0	0	32	1	4	2	0	0	39
Sleepy Test	0	1	0	23	0	8	0	1	1	34
Resource Optimism	0	0	0	31	0	1	0	1	1	34
Empty Test	0	2	1	22	0	7	0	0	0	32
Ignored Test	0	1	0	23	0	5	0	1	1	31
Redundant Assertion	0	1	0	21	0	6	0	1	1	30
Lazy Test	0	0	0	24	0	1	2	1	1	29
Redundant Print	0	1	0	12	0	7	0	2	1	23
Obscure Test	0	0	1	18	0	3	0	0	0	23
Constructor Initialization	0	1	0	16	0	5	0	0	0	22
Duplicated Test Code	1	0	0	11	0	0	0	0	0	13

Source: prepared by the author.

We also analyzed the distribution of test smells across programming languages by category. For this, we classified the test smells according to the types defined in the test smell catalog of Soares *et al.* (2023), which are: code-related, dependencies, design-related, issues in test steps, test execution (behavior), and test semantics (logic). Table 9 shows the distribution of test smells across the languages.

The most common types of test smells are code-related, followed by test semantic (logic) issues and issues in test steps. The remaining types occur less frequently overall. Java exhibits a relatively balanced distribution of test smell types, with no single type showing a significantly higher occurrence than the others. Python exhibits a higher proportion of code-related test smells and issues in test steps. An interesting case is TTCN-3, where code-related issues dominate, a behavior that may be related to the fact that TTCN-3 is a language specifically designed for writing test code; hence, most problems tend to be code-related rather than belonging to other categories, such as Dependencies or Design-related. Another noteworthy observation is that dynamically typed languages, such as Python, TypeScript (which supports optional typing), and JavaScript, exhibit more test execution (behavior) test smells compared to statically typed languages, such as Java.

Table 9 – Test smell categories per language

Language	Counts						Percentages					
	Code	Dep	Design	Steps	Exec	Sem	Code	Dep	Design	Steps	Exec	Sem
C	1	0	0	0	0	0	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%
C#	8	1	4	6	2	6	29.6%	3.7%	14.8%	22.2%	7.4%	22.2%
C++	5	0	1	2	0	2	50.0%	0.0%	10.0%	20.0%	0.0%	20.0%
Java	211	92	51	142	47	166	30.4%	13.3%	7.4%	20.5%	6.8%	23.9%
Kotlin	0	0	0	1	0	0	0.0%	0.0%	0.0%	100.0%	0.0%	0.0%
Python	36	9	15	38	16	20	30.0%	7.5%	12.5%	31.7%	13.3%	16.7%
Scala	0	2	0	4	0	6	0.0%	16.7%	0.0%	33.3%	0.0%	50.0%
Smaltalk	1	0	0	0	0	0	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%
TTCN-3	54	0	8	6	0	6	72.0%	0.0%	10.7%	8.0%	0.0%	8.0%
Typescript	4	3	1	2	3	5	22.2%	16.7%	5.6%	11.1%	16.7%	27.8%
Javascript	3	2	1	2	2	4	21.4%	14.3%	7.1%	14.3%	14.3%	28.6%
Robot	26	3	2	5	1	4	63.4%	7.3%	4.9%	12.2%	2.4%	9.8%

Source: Abbreviations: Code = Code related, Dep = Dependencies, Design = Design related, Steps = Issues in test steps, Exec = Test execution (behavior), Sem = Test semantic (logic)

4.3.3 RQ2: What are the universal test smells across all programming languages and frameworks, and which test smells are specific to particular languages?

To address this question, we analyzed test smells that appeared in multiple languages or frameworks, as well as those that appeared in only a single language or framework. We defined universal test smells as those present in at least two languages, indicating that there is evidence in the literature that these test smells can occur across multiple languages. Conversely, language-specific test smells are those observed in only one language, indicating that, to date, there is no evidence in the current literature that these test smells occur in multiple languages.

Table 10 shows the test smells that appeared in multiple languages or frameworks. In total, 37 test smells appeared in more than one language or framework, with mentions ranging from 2 to 11 across different languages or frameworks. As seen in the table, the test smell that appeared in the most languages or frameworks was *Unknown Test*, with 11 different language/framework associations, followed by *Conditional Test Logic*, *Exception Handling*, and *Magic Number*, which were each associated with 10 different languages/frameworks. Table 8 shows that these test smells are also among the most frequently cited.

Appendix B shows the list of test smells that appeared in a single language or framework. In total, we identified 176 test smells in the languages or frameworks: C++, Java (without specifying any framework), Java/JUnit, Java/TestNG, Python/PyTest, Python/Unittest, Robot Framework, Smaltalk, TTCN-3, and Typescript/Jest. As shown in the table, Java is the language with the most test smells that appear exclusively within a single language, which

Table 10 – Test smells that appear in more than one language or framework

Test Smell	N. of Languages/Frameworks
Unknown Test	11
Conditional Test Logic	10
Exception Handling	10
Magic Number	10
Assertion Roulette	9
Duplicate Assert	9
Eager Test	9
Redundant Print	9
Ignored Test	8
Sleepy Test	8
Obscure Test	7
Redundant Assertion	7
Empty Test	6
General Fixture	6
Lazy Test	6
Mystery Guest	6
Resource Optimism	5
Sensitive Equality	5
Constructor Initialization	4
Duplicated Test Code	4
Obscure In-Line Setup	4
Dead Field	3
Default Test	3
Dependent Test	3
Lack Of Cohesion Of Test Methods	3
Non-Functional Statement	3
Programming Paradigms Blend	3
Redudant Print	3
Test Maverick	3
Undefined Test	3
Verifying In Setup Method	3
Bad Naming	2
Inappropriate Assertion	2
Indirect Testing	2
Lack Of Cohesion	2
Private Method Test	2
Redudant Test	2

Source: prepared by the author.

aligns with the fact that Java (and its frameworks) was the most frequently mentioned in papers addressing test smells. An interesting observation is that, apart from Java, the languages or frameworks with the highest number of test smells exclusive to a single language were those specifically designed for writing tests, such as Robot (Bisht, 2013) and TTCN-3 (Grabowski *et al.*, 2003).

4.3.4 RQ3: What refactorings are commonly recommended for addressing test smells in each programming language and framework?

To answer this question, we collected the refactoring techniques for test smells reported in the reviewed studies. A total of 56 test smells had associated refactorings, resulting in 94 distinct refactorings across three programming languages (Java, Scala and Python) and seven frameworks (JUnit, TestNG, ScalaTest, Robot Framework, Unittest, PyTest and Nose). Some test smells had multiple refactorings reported. For example, for the *Assertion Roulette* smell, two possible refactorings are *Add Assertion Explanation* (demonstrated in Listing 22) and *Extract Method* (illustrated in Listing 23). Additionally, some refactorings address multiple test smells; for instance, *Extract Method* is also used to refactor the *Duplicate Assert* smell.

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

  Repository repo = executeAndWaitFor(cloneOp);

  // explanatory message added
  assertThat(
    "repo should has correct git object",
    repo,
    hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc")
  );

  File readmeFile = new File(repo.getWorkTree(), "README");

  // explanatory message added
  assertThat("readme file should exist", readmeFile, exists());

  // explanatory message added
  assertThat(
    "readme file length should be 12L",
    readmeFile.length(),
    equalTo(12L)
  );
}
```

Listing 22 – Refactored Assertion Roulette Java example

```java
@Test
public void testRepositoryContainsExpectedGitObject() throws Exception {
    Clone cloneOp = new Clone(
        false,
        integrationGitServerURIFor("small-repo.early.git"),
        helper().newFolder()
    );
    Repository repo = executeAndWaitFor(cloneOp);
    assertThat(
        repo,
        hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc")
    );
}

@Test
public void testReadmeFileExists() throws Exception {
    Clone cloneOp = new Clone(
        false,
        integrationGitServerURIFor("small-repo.early.git"),
        helper().newFolder()
    );
    Repository repo = executeAndWaitFor(cloneOp);
    File readmeFile = new File(repo.getWorkTree(), "README");
    assertThat(readmeFile, exists());
}

@Test
public void testReadmeFileHasCorrectLength() throws Exception {
    Clone cloneOp = new Clone(
        false,
        integrationGitServerURIFor("small-repo.early.git"),
        helper().newFolder()
    );
    Repository repo = executeAndWaitFor(cloneOp);
    File readmeFile = new File(repo.getWorkTree(), "README");
    assertThat(
        readmeFile.length(),
        equalTo(12L)
    );
}
```

Listing 23 – Refactored Assertion Roulette Java example

The most frequently mentioned refactoring technique was *Extract Method*, appearing 44 times across different test smells, followed by *Add Assertion Explanation* (16 mentions) and *Setup External Resource* (9 mentions). Other commonly cited refactorings include *Extract Class* (8 mentions), *Use Parameterized Test* (7 mentions), and *Introduce Equality Method* (6 mentions). The test smell with the most diverse refactoring options was *Exception Handling*, with 11 different refactoring techniques reported, followed by *Assertion Roulette* and *Conditional Test Logic*, each with seven different refactoring approaches.

Table 11 presents the most frequently cited refactorings. We omitted from this table test smells with only a single reported refactoring, as well as those for which all refactorings have the same number of mentions. As shown, the most common refactorings for the test smells *Assertion Roulette* and *Eager Test* (the two most frequently cited test smells) are *Add Assertion Explanation* and *Extract Method*, respectively. Additionally, the *Extract Method* appears most often overall, as it can address multiple test smells. A comprehensive list of refactorings, organized by language and framework, is available in Appendix C.

Table 11 – Test smells and favorite refactorings

Test Smell	Refactoring	N. of Mentions
Assertion Roulette	Add assertion explanation	16
Eager Test	Extract method	11
Duplicate Assert	Extract method	7
Sensitive Equality	Introduce equality method	6
Duplicated Test Code	Extract method	5
Mystery Guest	Setup external resource	5
Resource Optimism	Setup external resource	4
General Fixture	Extract method	4
Lack Of Cohesion Of Test Methods	Extract class	3
Indirect Testing	Extract method	3
Unknown Test	Add assertion	2
Conditional Test Logic	Extract method	2
Test Run War	Make resource unique	2

Source: prepared by the author.

4.3.5 *RQ4: What datasets of test smells are available for each programming language and framework?*

To answer this question, we collected the test smell datasets mentioned or used in the referenced works. We manually examined these datasets to verify their availability. After this process, we identified a total of 16 test smell datasets. Table 12 provides an overview of the identified datasets. These datasets cover the following languages/frameworks: C++/GoogleTest, C#, C#/xUnit, Java, Java/JUnit, Python/Unittest, and Robot Framework. In total, the datasets encompass 73 different test smells across these seven languages/frameworks. The list of test smells in each dataset is available in the replication package.

Table 12 – Test smells dataset overview

Language/ Framework	N. of Test Smells	Dataset URL
C#	4	https://figshare.com/articles/dataset/Reinforcement_Learning_from_Static_Quality_Metrics/25983166?file=46868443
C#/xUnit	7	https://figshare.com/articles/dataset/A_Road_to_Find_Them_All_Towards_an_Agnostic_Strategy_for_Test_Smell_Detection/26444968
C#/xUnit	16	https://github.com/Partha-SUST16/xNose/tree/main/results
C++/GoogleTest	7	https://figshare.com/articles/dataset/A_Road_to_Find_Them_All_Towards_an_Agnostic_Strategy_for_Test_Smell_Detection/26444968
Java	16	https://github.com/bhpachulski/SAST21-Paper
Java	4	https://github.com/darioamorosodaragona-tuni/ML-Test-Smell-Detection-Online-Appendix
Java/JUnit	1	https://github.com/unittesting-nonpublic/private-keep-out_replication-package
Java/JUnit	1	https://doi.org/10.5281/zenodo.11267987
Java/JUnit	4	https://zenodo.org/records/3337892#.XswWby-w3yU
Java/JUnit	7	https://figshare.com/articles/dataset/A_Road_to_Find_Them_All_Towards_an_Agnostic_Strategy_for_Test_Smell_Detection/26444968
Java/JUnit	7	https://figshare.com/s/b1d6b70e10837aaf3f17?file=41800866
Java/JUnit	19	https://testsmells.org/pages/testsmellexamples.html
Java/JUnit	20	https://figshare.com/s/7b8bf9a7580001929f63?file=27729300
Java/JUnit	21	https://figshare.com/s/da2b1903e7b209b1f77e?file=50726838
Python/Unittest	14	https://figshare.com/s/4789bd212185042cdad0?file=49671705
Python/Unittest	17	https://se.cite.ehime-u.ac.jp/data/Fushihara_SEAA2023/
Python/Unittest	18	https://se.cite.ehime-u.ac.jp/data/Fushihara_SEAA2024/
Robot Framework	16	https://github.com/kabinja/suit-smells-replication-package

Source: prepared by the author.

4.3.6 *RQ5: Which test smells are most critical across different programming languages and frameworks?*

To answer this question, we collected evidence directly from the papers regarding the criticality of various test smells. Specifically, when a paper indicated that a particular test smell (such as *Assertion Roulette* or *Eager Test*) was critical, we carefully examined the text to understand the criteria or rationale behind that assessment. In other words, our evaluation was grounded in the authors' own discussions, observations, or metrics rather than our subjective judgment. Once we gathered criticality evidence from multiple papers, we standardized these assessments by organizing them into a five-level scale ranging from very low to very high criticality. This approach allowed us to compare and synthesize insights across different studies systematically.

The test smells for which we collected criticality evidence included: *Ambiguous*

Display Name, *Assertion Roulette*, *Conditional Test Logic*, *Constructor Initialization*, *Duplicate Assert*, *Duplicate Display Name*, *Eager Test*, *Empty Test*, *Exception Handling*, *General Fixture*, *Ignored Test*, *Inconsistent Display Name*, *Inconsistent Style*, *Lazy Test*, *Magic Number*, *Missing Display Name*, *Multilingual Display Name*, *Mystery Guest*, *Non-Functional Statement*, *Obscure Test*, *Programming Paradigms Blend*, *Redundant Assertion*, *Redundant Display Name*, *Resource Optimism*, *Sensitive Equality*, *Sleepy Test*, *Test Run War*, *Unclear Display Name*, *Undefined Test*, *Ungrammatical Display Name*, *Unknown Test*, and *Verifying In Setup Method*.

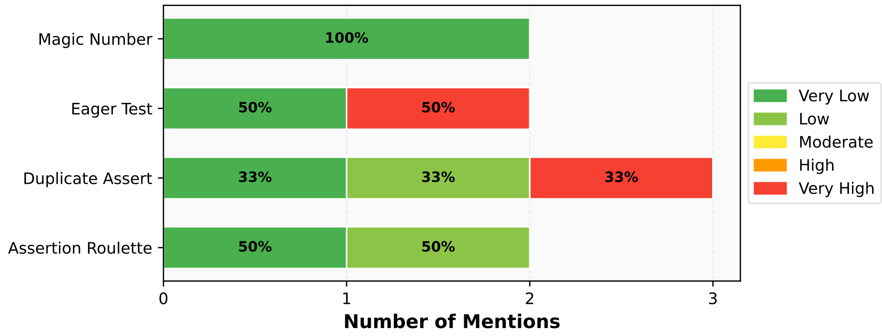
Most papers focused on criticality in terms of maintainability, while others examined it from a general developer's perspective. However, some papers did not specify the subject of criticality, and we excluded them from the analysis. We also identified one study that approached test smell criticality from the perspective of energy consumption, classifying test smells into four categories based on their energy impact: Strong (Assertion Roulette, Lazy Test, Eager Test, Magic Number Test), Moderate (Dependent Test, Unknown Test, Verbose Test), Weak (Sensitive Equality, Conditional Test Logic), and Very Weak (Redundant Assertion). While this represents a novel perspective on test smell criticality, we did not include it in our main analysis due to the limited evidence from a single study.

Some test smells, such as *Ambiguous Display Name*, *Constructor Initialization*, *Duplicate Display Name*, *Exception Handling*, *General Fixture*, *Inconsistent Display Name*, *Inconsistent Style*, *Lazy Test*, *Missing Display Name*, *Multilingual Display Name*, *Programming Paradigms Blend*, *Redundant Assertion*, *Redundant Display Name*, *Sensitive Equality*, *Test Run War*, *Unclear Display Name*, *Undefined Test*, *Ungrammatical Display Name*, *Unknown Test*, and *Verifying In Setup Method*, had limited evidence in the literature regarding their criticality (with only one paper classifying the criticality of these smells). Due to this scarcity of evidence, we decided not to analyze these test smells further.

Figure 6 presents the criticality of the test smells *Magic Number*, *Eager Test*, *Duplicate Assert*, and *Assertion Roulette* based on the general developer's perception. Among these four test smells, the only one with 100% agreement was *Magic Number*, which received a very low criticality classification. The classification of the *Assertion Roulette* test smell ranged from very low to low criticality, indicating that, in the developer's perception, this is not a critical test smell. The other two test smells (*Eager Test* and *Duplicate Assert*) exhibited some discrepancy in classification, ranging from very low to high criticality.

Figure 7 shows the criticality of the test smells *Sleepy Test*, *Resource Optimism*,

Figure 6 – Test smells criticality concerning developer's perception in general



Source: prepared by the author.

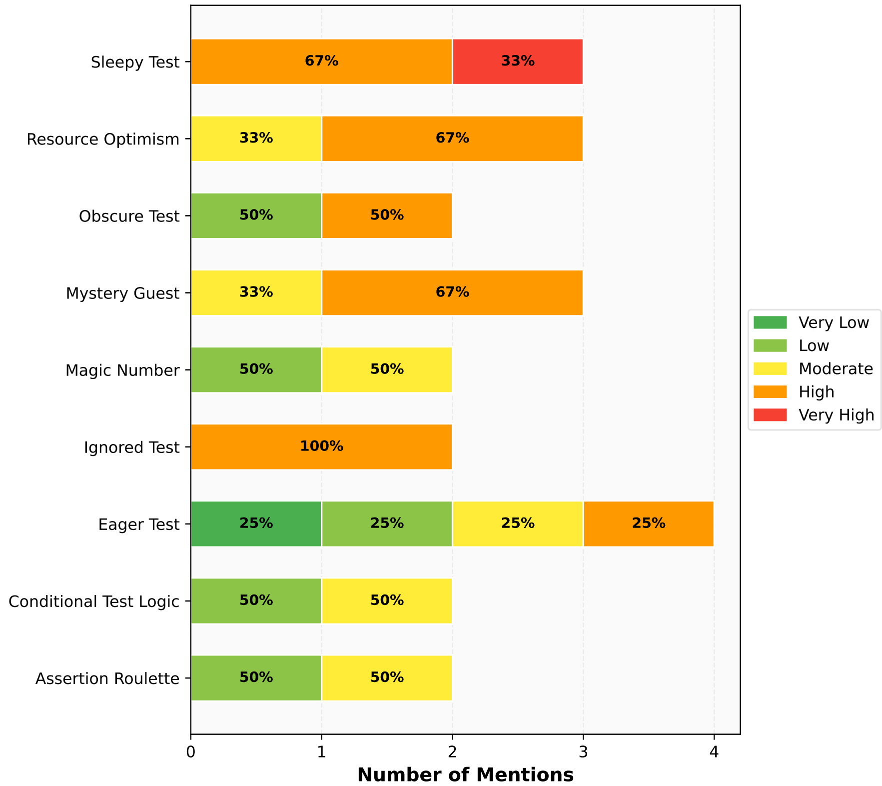
Obscure Test, *Mystery Guest*, *Magic Number*, *Ignored Test*, *Eager Test*, *Conditional Test Logic*, and *Assertion Roulette* in terms of maintainability. The only test smell with 100% agreement in its classification was the *Ignored Test*, which received a high criticality classification. The test smells *Sleepy Test*, *Resource Optimism*, and *Mystery Guest* were also classified as high criticality. Notably, the *Sleepy Test* received a very high criticality rating from one paper, indicating that it is a particularly critical test smell with respect to maintainability. The classifications for *Magic Number*, *Conditional Test Logic*, and *Assertion Roulette* varied from low to moderate criticality. The remaining two test smells, *Obscure Test* and *Eager Test*, displayed a higher level of disagreement in classification, with *Eager Test* receiving four different ratings.

4.3.7 RQ6: What test smell detection or refactoring tools have been proposed for each programming language and framework?

To answer this question, we collected tools related to test smells mentioned in the literature, including extensions to existing tools. In total, we identified 36 tools for C#, Java, JavaScript, TypeScript, Python, Scala, and TTCN-3, as shown in Table 13. Of these, 22 tools (61%) are currently available, while 14 (39%) are unavailable (e.g., Better Code Hub Extension, Evosuite Extension, TASTE, TestComet & Teslo, TSGame, TestHound, DFD, DFI, STEEL, and Pysta). Java has the most significant number of available tools (15), followed by Python (4). We also found tools designed for specialized testing languages, such as Ukwikora and sonar-ikora-plugin for Robot Framework, and TRex for TTCN-3. However, only one tool, STEEL, is available for JavaScript, highlighting a notable gap for that language.

Refactoring capabilities within the ecosystem remain limited: only five tools (RAIDE, TESTLER, DARTS, TRex, and UTRefactor) support both detection and automatic refactoring, accounting for just 14% of the available solutions. This fact highlights a broader gap in the

Figure 7 – Test smells criticality concerning maintainability



Source: prepared by the author.

field, namely the need for tools that not only identify test smells but also provide automated refactoring support. A particularly noteworthy case is UTRefactor, introduced in one of the most recent studies (2025), which leverages a LLM to refactor test smells. This approach demonstrates considerable potential, yet it remains largely underexplored.

4.4 Discussion

This systematic mapping study provides several insights into the current state of test smell research and outlines potential directions for future investigations. Based on the analysis of 117 papers published between 2006 and 2025, we identified noteworthy patterns that reveal limitations in the current body of work while also highlighting promising avenues for further exploration.

Takeaway 1: The Java Monoculture Problem. Our results clearly indicate the existence of a *Java monoculture* within test smell research, as 85 out of the 117 papers focus exclusively on Java. This fact presents both opportunities and challenges. On one hand, the emphasis on Java has deepened our understanding of how test smells manifest in this language, which may also inform insights for similar object-oriented languages. On the other hand, this

Table 13 – Detection and refactoring tools

Language	Framework	Tool	Capabilities	Availability	Papers
C#	xUnit	XNose	Detection	Available	S99
Java	JUnit	Better Code Hub (Extension)	Detection	Unavailable	S34
Java	JUnit	DesigniteJava (Extension)	Detection	Available	S85
Java	JUnit	Evosuite extension	Detection	Unavailable	S40
Java	JUnit	JNose	Detection	Available	S11
Java	JUnit	JTDog	Detection	Available	S33
Java	JUnit	RAIDE	Detection and Refactoring	Available	S23, S37
Java	JUnit	SimilarTestAnalysis	Detection	Available	S17
Java	JUnit	SniffTest	Detection	Available	S77
Java	JUnit	TASTE	Detection	Unavailable	S50
Java	JUnit	TestComet & Teslo	Detection	Unavailable	S93
Java	JUnit	TESTLER	Detection and Refactoring	Available	S19
Java	JUnit	TSDETECT	Detection	Available	S79
Java	JUnit	TSGame	Detection	Unavailable	S97
Java	JUnit, TestNG	FSE_Ignore_Test	Detection	Available	S43
Java	JUnit, TestNG	TestHound	Detection	Available	S54
Java	Selenium Web-Driver	TEDD	Detection	Available	S21
Java		DARTS	Detection and Refactoring	Available	S45
Java		DFD	Detection	Available	S72
Java		DFI	Detection	Unavailable	S80
Java		JNose	Detection	Available	S70
Java		Unamed	Detection	Available	S1
JavaScript, Typescript	Chai, Jest	STEEL	Detection	Unavailable	S4
Python	Nose, PyTest, Unittest	Pysta	Detection	Unavailable	S84
Python	PyTest	PyTest-Smell	Detection	Available	S14
Python	PyTest, Unittest	TEMPY	Detection	Available	S3
Python	Unittest	PYNOSE	Detection	Available	S31
Scala	ScalaTest	SoCRATES	Detection	Available	S13, S48
TTCN-3		TRex	Detection and Refactoring	Available	S89, S91
	Robot Framework	sonar-ikora-plugin	Detection	Available	S83
	Robot Framework	Ukwikora	Detection	Available	S78
C++, C#, Java	GoogleText, xU-nit, JUnit	SniffML	Detection	Available	S102
Java	JUnit	UTRefactor's	Detection and Refactoring	Available	S104
Java	JUnit	Sojourner under Sabotage	Detection	Available	S106
Java	JUnit	Viscount	Detection	Available	S111, S112
Python	PyTest	PyTeRor	Detection	Available	S117

Source: prepared by the author.

narrow focus may not accurately reflect real-world software development practices, where developers commonly employ a variety of languages, each with its own unique characteristics.

A temporal analysis reveals the same trend. Although the use of other programming languages in test smell research has grown over the years, Java's dominance has not diminished. In fact, its use has steadily increased. Some languages, such as Python, have recently gained more attention (e.g., eight papers published in 2024 alone). However, other widely used industrial languages (such as JavaScript, TypeScript, Go, and Rust) remain underrepresented. This gap between research focus and industry practice may limit the generalizability of current findings. Therefore, further studies should prioritize underexplored languages to provide a more comprehensive and representative understanding of test smells.

Takeaway 2: Language Characteristics Influence Test Smell Patterns. Analyzing test smell categories across languages with different characteristics reveals patterns suggesting that language design decisions may influence the types of smells that occur most frequently. For instance, in dynamically typed languages such as Python, TypeScript, and JavaScript, test smells from the *test execution–behavior* category appear more often (13.3% for Python, 16.7% for TypeScript, and 14.3% for JavaScript) compared to statically typed languages like Java, where the occurrence is lower (6.8%).

Another striking example is TTCN-3, a language specifically designed for writing tests, in which most test smell occurrences are *code-related* (72%). Similarly, Python's permissive syntax leads to higher proportions of test smells categorized as *code-related* (30%) and *issues in test steps* (31.7%).

These observations reinforce the previous takeaway: expanding research to encompass more diverse programming languages is crucial. Current tools and techniques (mainly developed for Java) may not accurately capture the nuances or address the unique challenges presented by other languages.

Takeaway 3: The Criticality Paradox. Another particularly concerning finding from this systematic mapping study is the disconnect between the test smells most frequently studied in research and those perceived as most critical in practice. *Assertion Roulette* and *Eager Test* stand out as the most extensively investigated test smells. However, they are generally considered to exhibit low to moderate criticality with respect to both their impact on maintainability and developers' perceived severity. Conversely, *Sleepy Test*, identified in our results as the most critical test smell for maintainability, has received far less research attention.

This paradox suggests that future studies should focus more on practically impactful problems rather than continuing to emphasize well-studied but less severe smells. One possible explanation for this imbalance is that research has tended to prioritize test smells that are easier to detect, such as *Assertion Roulette*, thereby overlooking more complex yet more critical ones like

Resource Optimism. Redirecting research efforts toward these underexplored, high-impact test smells would help ensure that future work aligns more closely with real-world testing challenges.

Takeaway 4: Inconsistent Criticality Classifications. A further issue related to criticality is the substantial disagreement among studies regarding how harmful specific test smells are, even when assessing the same aspect (such as maintainability). For instance, *Eager Test* has been described as both "highly harmful" and "not harmful" in different studies, whereas *Assertion Roulette* has been rated anywhere from "irrelevant" to "medium severity."

These inconsistencies underscore the need for more rigorous and standardized research on the criticality of test smells. Future studies should employ larger sample sizes, adopt consistent evaluation frameworks, and use quantitative methodologies to improve the reliability and reproducibility of criticality assessments. Such methodological rigor is essential for building a more cohesive understanding of how test smells truly affect software quality.

Takeaway 5: The Detection–Refactoring Imbalance. Regarding tools related to test smells, our study reveals a significant imbalance between detection and refactoring solutions. While detection tools are abundant (31 identified), refactoring tools are scarce (only five identified). This disparity is concerning, as it shows that developers receive substantial support for detecting test smells but limited assistance for effectively addressing them. Bridging this gap enables developers to complement detection with actionable, automated, and reliable refactoring solutions.

Takeaway 6: Underutilization of AI-Based Solutions. Artificial intelligence (AI) has become an increasingly popular research topic in software engineering, yet its application within the context of test smells remains highly limited. Among all selected studies, only one (UTRefactor (Gao *et al.*, 2025)) has explored the use of LLMs to automate test smell refactoring. This work demonstrates the promising potential of AI-driven approaches in this domain. However, AI remains scarcely adopted when compared to more traditional techniques. Expanding the use of AI for both detection and refactoring could open new opportunities for more intelligent and adaptive test smell handling.

Takeaway 7: Limited Multi-Language Support. Our results also highlight the need for tools that support multiple programming languages. Currently, only a few recent tools attempt to achieve this. For example, SniffML (Lopes *et al.*, 2024) supports C++, C#, and Java, representing a significant step toward broader language coverage. Language-agnostic solutions, such as SniffML, offer a promising approach to promoting inclusiveness and generalizability across diverse programming ecosystems in test smell research.

4.5 Threats To Validity

In this section, we describe the threats to validity according to the classification proposed by Wholin *et al.* (2000).

Internal validity threats refer to the possibility of attributing the cause of an effect to a factor when actually a third, unconsidered factor is the actual cause. In the context of our SMS, this could occur if we incorrectly attribute patterns in test smell research to programming languages when other factors (such as research community focus, tool availability, or publication trends) might be the actual cause. To mitigate this threat, we carefully documented our search strategy and inclusion criteria, and multiple authors reviewed the study selection process to ensure consistent application of criteria.

Construct validity threats concern whether our measurements adequately represent the constructs we intend to measure. In our SMS, this relates to whether our search strategy and classification criteria accurately capture the concepts of "test smells," "programming languages," and "test frameworks" as intended. A potential threat is that different papers may define test smells differently, leading to inconsistent classification. To mitigate this, we conducted a pilot test with well-known reference works to validate our search string, and we established clear operational definitions for all key constructs. Additionally, we used consensus among multiple authors to resolve classification ambiguities.

External validity concerns the extent to which the study results can be generalized beyond the specific context of our research. In our SMS, this relates to whether our findings about test smell research patterns can be generalized to the broader software engineering community or to different time periods. A potential threat is that our search was limited to specific digital libraries and may not capture all relevant research. Additionally, the temporal scope of our study (ending in August 2025) may not reflect future trends. To mitigate this, we searched multiple major digital libraries (IEEE Xplore, ACM Digital Library, ScienceDirect, and SpringerLink), tested our search string iteratively across all databases, and used consensus among all authors to finalize the search strategy. We also documented our search process transparently to enable replication and extension.

Conclusion validity concerns whether the statistical tests and evidence adequately support the conclusions reached in the study. In our SMS, this relates to whether our analysis methods and data presentation support the claims we make about patterns in test smell research. A potential threat is that our descriptive analysis might not provide sufficient evidence for the conclusions drawn. To mitigate this, we used systematic data extraction procedures, provided detailed tables and figures to support our findings, and ensured that the extracted data directly support all conclusions. We also conducted a peer review of the analysis process to ensure consistency and accuracy.

4.6 Conclusions

In this work, we conducted a systematic mapping study to examine test smells from the perspective of programming languages. We analyzed 117 papers and extracted data to address six research questions. These questions focused on identifying which test smells are most frequently associated with specific programming languages or frameworks, and determining their prevalence across different contexts. Additionally, we explored which test smells are language-specific and which occur across multiple languages or frameworks. Our study also compiled information on refactoring techniques used to address test smells in various languages, as well as datasets related to test smells in specific languages and test frameworks. We further investigated the criticality of particular test smells as reported in the literature. Finally, we gathered information on tools proposed for detecting or managing test smells in different programming languages or frameworks.

Among the main findings of our work are:

1. Java is the most studied language in the context of test smells, with JUnit being the most widely used framework.
2. *Assertion Roulette* and *Eager Test* are the most frequently cited test smells related to Java and other languages and frameworks.
3. *Unknown Test* is the test smell observed in the most diverse set of languages and frameworks.
4. We identified several language-specific test smells for Java, TTCN-3, and the Robot Framework.
5. We identified a total of 94 refactorings for 56 different test smells.
6. We identified 16 test smell datasets across four different languages.
7. The test smells classified as most critical are *Sleepy Test*, *Ignored Tests*, *Resource Optimism*, and *Mystery Guest*.

8. A total of 36 tools related to test smells were found, with the majority designed for Java.

This study delivers substantial value to both the research community and industry practitioners. The comprehensive catalog of languages and test frameworks we present serves as a strategic foundation for future research, particularly enabling targeted investigations into less-explored programming environments to address existing knowledge gaps. Furthermore, our findings provide crucial guidance for developing more generalized approaches to test smell detection and refactoring that can operate across multiple languages and frameworks simultaneously. The criticality analysis we present offers researchers a valuable prioritization framework, allowing them to concentrate their efforts on the most impactful test smells. Additionally, our curated collection of datasets and tools serves as a vital benchmark for evaluating existing approaches and comparing them with novel solutions. Together, these contributions lay a strong foundation for advancing research in test smell detection and analysis.

5 AROMALIA: A LANGUAGE-INDEPENDENT APPROACH TO DETECT TEST SMELLS

5.1 Introduction

Implementing tests for software products is essential to ensure software quality, particularly in agile development environments where practices such as continuous delivery accelerate the transition from feature ideation to customer deployment (Tran *et al.*, 2021). Testing can be performed manually by humans or automated through scripts, which enhance repeatability and reduce effort once implemented (Garousi; Küçük, 2018).

However, similar to production code, test code is also susceptible to quality issues that can lead to additional maintenance effort and costs (Garousi; Küçük, 2018). Test engineers may make suboptimal design decisions when writing test code due to system complexity or limited experience (Santana *et al.*, 2024), resulting in code that is difficult to read, understand, and maintain (Junior *et al.*, 2021). These poor design choices are referred to as *test smells* (Aljedaani *et al.*, 2021). In recent years, test smells have garnered significant attention due to their negative impact on testing activities, particularly on test code maintainability (Virgínio *et al.*, 2020a; Bavota *et al.*, 2015; Peruma *et al.*, 2019).

The concept of test smells was first introduced by Deursen *et al.* (2001), who proposed an initial catalog of 11 test smells along with corresponding refactorings. Since then, researchers have identified many additional test smells and refactorings (Meszaros, 2007), as well as developed tools to automatically detect and refactor them across various programming languages (Aljedaani *et al.*, 2021). However, most of these studies and tools focus on a small number of languages, which significantly limits their applicability (Aljedaani *et al.*, 2021).

Similar challenges have been addressed in related fields. Schiewe *et al.* (2022) developed a language-independent static code analysis technique using language-agnostic Abstract Syntax Trees (ASTs), while Ducasse *et al.* (1999) proposed a language-independent approach for detecting code duplication. However, existing language-independent approaches for code smells are not directly applicable to test smells, as test smells are specific to test code and depend on testing conventions and patterns that general-purpose detectors are not designed to recognize.

To address this gap, we propose AromaLIA, a language-independent approach for detecting test smells. Our approach is based on the observation that test code typically follows similar structural patterns across different programming languages, and test smells tend to manifest in comparable ways regardless of the language (see examples in Listings 24 and 25).

In our approach, we leverage a language-agnostic Abstract Syntax Tree (LAAST), similar to Schiewe *et al.* (2022), from which we extract key information about test code (assertions, annotations, and other relevant elements) and organize it into a unified model that maintains a consistent structure across different programming languages. We then apply detection rules to this unified model to identify test smells without directly analyzing the original source code. Adding support for a new language does not require redefining or reimplementing detection rules, as these rules remain valid regardless of the language, reducing the effort required to extend our approach.

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

Listing 24 – Duplicate Assert example in Java

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

Listing 25 – Duplicate Assert example in Python

To validate our approach, we developed a tool that implements the AromaLIA methodology to detect ten test smells widely discussed in the literature (Soares *et al.*, 2023; Santana *et al.*, 2022; Schäfer *et al.*, 2024; Peruma *et al.*, 2020; Peruma; Newman, 2021; Liu; Yu, 2022; Martins *et al.*, 2024b; Campos *et al.*, 2023b; Fasolino; Tramontana, 2024; Chen *et al.*, 2023; Kiran *et al.*, 2019), namely: *Assertion Roulette*, *Conditional Test Logic*, *Duplicate Assert*, *Empty Test*, *Exception Handling*, *Ignored Test*, *Magic Number*, *Redundant Print*, *Sleepy Test*, and *Unknown Test*. The tool currently supports C#, Java, Python, JavaScript, and TypeScript. We then compared our AromaLIA-based tool with existing language-specific test smell detection tools for C#, Java, and Python.

For evaluation, we applied all tools to a pre-classified and manually validated dataset containing 830 test smell samples covering the ten selected test smells. We built this dataset from test cases extracted from public Java and Python projects on GitHub. To ensure coverage across all five target languages, we used ChatGPT-4o-mini to translate selected test samples between languages, followed by manual validation to confirm the correct presence or absence of each test smell.

Our approach demonstrated superior effectiveness compared to language-specific methods, achieving an overall precision of 97%, a recall of 96%, and an F1-score of 97%. These results surpassed those of the three existing language-specific smell detection test tools.

The remainder of this paper is organized as follows. Section 5.2 describes the AromaLIA approach in detail. Section 5.3 outlines the design and execution of our evaluation. Section 5.5 discusses potential threats to validity. Finally, Section 5.6 summarizes our findings and outlines directions for future research.

5.2 AromaLIA Approach

To develop our language-independent approach for detecting test smells, we first examined existing methods (Lambiase *et al.*, 2020; Delplanque *et al.*, 2019; Peruma *et al.*, 2020; Zhang *et al.*, 2014; Bell *et al.*, 2015; Virgínio *et al.*, 2020b; Santana *et al.*, 2020; Bleser *et al.*, 2019b; Palomba *et al.*, 2018; Koochakzadeh; Garousi, 2010; Greiler *et al.*, 2013a; Greiler *et al.*, 2013c; Baker *et al.*, 2006; Huo; Clause, 2014; Lopes *et al.*, 2024). As described in Chapter 2, current tools primarily employ four strategies for test smell detection: metrics, rules or heuristics, information retrieval, and dynamic tainting. For instance, TSDETECT (Peruma *et al.*, 2020), one of the most widely used tools in the field (Aljedaani *et al.*, 2021), follows the rules/heuristics approach. Its detection process involves using the `JavaParser` [1] library to generate the Abstract Syntax Tree (AST) of the test code, after which detection rules for each test smell are applied directly to the AST to identify occurrences.

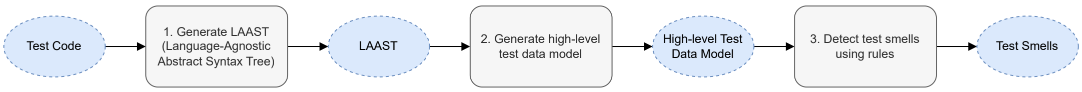
We also reviewed studies in related domains that propose language-independent techniques, such as code smell detection (Ducasse *et al.*, 1999) and static code analysis (Schiewe *et al.*, 2022). The work of Schiewe *et al.* (2022) was particularly influential because, like Peruma *et al.* (2020), it relies on AST-based analysis. However, whereas Peruma *et al.* (2020) use

[1] https://javaparser.org

a language-specific AST for Java, Schiewe *et al.* (2022) employ a language-agnostic AST (LAAST).

Building on insights from these works and introducing new concepts to address limitations of existing approaches, we developed our language-independent method for detecting test smells. Figure 8 illustrates the complete AromaLIA test smell detection process, which consists of three main steps described below. At the end of this section, we also provide a full example demonstrating the use of the AromaLIA approach across all three steps.

Figure 8 – AromaLIA Test Smell Detection Process



Source: prepared by the author.

5.2.1 *First Step: Generate LAAST*

The first step involves obtaining the test code and generating its LAAST. In our implementation, we use the same tool as Schiewe *et al.* (2022), which generates the LAAST via Mozilla's rust-code-analysis crate (Ardito *et al.*, 2020). This tool supports multiple programming languages, including Java, JavaScript, TypeScript, Python, Go, and Rust, and can be extended to accommodate additional languages as needed.

5.2.2 *Second Step: Generate High-level Test Data Model*

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

To address this, our approach first generates a high-level test data model from the LAAST. This model is significantly simpler than the full LAAST but retains all relevant metadata necessary for test smell detection. The model is designed to be language-independent, with no elements tied to a specific programming language. Test smell detection rules are then applied to this high-level model. As a result, adding support for a new language does not require reimplementing detection logic for existing test smells, substantially reducing development and maintenance effort.

Listing 26 shows the high-level test data model specification used in this work. The fields of this model were defined based on the detection rules for the test smells selected for this study, as detailed in Table 14. The model includes all fields necessary to detect these test smells, but can be extended to incorporate additional fields for detecting other smells. For example, to detect an *Assertion Roulette*, it is sufficient to determine whether a test contains multiple assert statements and whether at least one lacks an explanatory message. This can be implemented by counting the number of asserts in the `asserts` field of the `Test` entity and iterating through the list to check whether the `message` field is missing for any assert.

Table 14 – Test smells and their detection rules

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

Source: Adapted from Peruma *et al.* (2020).

This intermediary step clearly distinguishes our approach from existing test smell detection methods. One key advantage is that the resulting high-level test data model is simple enough for manual inspection, enabling researchers and developers to easily understand why a particular test smell was detected. Another benefit is that it significantly simplifies the detection algorithms compared to applying rules directly to the AST. This simplification also improves comprehension of the detection logic for each test smell, as will be illustrated in the next section.

```
interface TestSuiteFull {
  filePath?: string;              // Path to the test file
  name: string;                   // Suite name
  isExclusive: boolean;           // Only this suite runs
  isIgnored: boolean;             // Suite is skipped
  tests: {
    name: string;                 // Test name
    isExclusive: boolean;         // Only this test runs
    isIgnored: boolean;           // Test is skipped
    startLine: number;            // Source start line
    endLine: number;              // Source end line
    startColumn: number;          // Source start column
    endColumn: number;            // Source end column
    statements: {
      type: "assignment" | "call" | "condition" | "loop" |
          "exceptionHandling" | "exceptionThrowing" | "other";
      startLine: number;
      endLine: number;
      startColumn: number;
      endColumn: number;
    }[];
    events: {
      name: string;
      type: "assert" | "print" | "sleep" | "unknown";
      startLine: number;
      endLine: number;
      startColumn: number;
      endColumn: number;
    }[];
    asserts: {
      matcher: string;
      literalExpected?: string;
      literalActual?: string;
      message?: string;
      startLine: number;
      endLine: number;
      startColumn: number;
      endColumn: number;
    }[];
  }[];
}
```

Listing 26 – High Level Test Data Model

5.2.3 *Third Step: Detect Test Smells Using Rules*

In the third step, detection rules are applied to the high-level test data model to identify existing test smells. Algorithm 4 present a method for detecting the *Duplicate Assert* test smell. This algorithm was derived from the detection rules in Table 14 and the specification of the high-level test data model shown in Listing 26. The algorithm is simple, as much of the complexity has been abstracted away in the previous steps. Here, we present only the algorithm for one of the ten test smells addressed in this work. The full set of algorithms is available in our replication package.

Algorithm 1 Duplicate Assert detection algorithm

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

5.2.4 *Full Example of Implementing and Using the AromaLIA Approach*

Listing 24 presents an example of the *Duplicate Assert* test smell in Java. To detect this smell using the AromaLIA approach, the first step is to extract the LAAST (Language-Agnostic Abstract Syntax Tree). This extraction can be performed using the rust-code-analysis tool. Listing 27 illustrates a representation of the LAAST generated for the Java code by this tool.

After this step, the process proceeds to extract the high-level test data model from the LAAST. Listing 28 shows part of an implementation that performs this extraction for Java tests. While the implementation uses TypeScript, the same approach can be applied in any programming language. Due to space constraints, many details are omitted. However, the listing emphasizes the extraction of assert statements, which are critical for detecting the *Duplicate Assert* test smell. A key aspect of this implementation is its reliance on Java-specific conventions, such as identifying test methods by the presence of the `Test` modifier and recognizing assert methods by their `assert` prefix.

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

Listing 27 – LAAST for Duplicate Assert example in Java

One important aspect of this approach is its use of helper functions. Because the LAAST provides nodes that are largely consistent across programming languages (despite minor differences in tree structure), common tasks in test smell detection can be implemented once and reused. These tasks include locating nodes representing test method declarations, identifying method invocations, and finding class declarations, as some testing frameworks use classes to define test suites. Consequently, the AromaLIA approach streamlines and accelerates the addition of support for new programming languages.

After executing this code, it should generate the high-level test data model, as illustrated in Listing 27. Once again, many details are omitted due to space constraints.

```
class JavaHighLevelTestDataModelExtractor {
  extract(node: LAASTNode): TestSuite[] {
    const classes = helpers.findAllClassDeclarations(node);
    return classes.map(classDeclaration => ({
      // ...
      tests: this.extractTests(classDeclaration.node),
    }));
  }
  private extractTests(node: LAASTNode): Test[] {
    const methods = helpers.findAllMethodDeclarations(node);
    return methods
      .filter(method => method.modifiers.includes('Test'))
      .map(method => ({
        // ...
        asserts: this.extractAsserts(method),
      }));
  }
  private extractAsserts(node: LAASTNode): TestAssert[] {
    const invocations = helpers.findAllMethodInvocations(node);
    return invocations
      .filter(i => i.identifier.startsWith('assert'))
      .map(i => this.extractAssertData(i));
  }
  // ...
}
```

Listing 28 – Example of High-level Test Data Model Extractor for Java

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

Listing 29 – High Level Test Data Model for the Duplicate Assert example in Java

The next step in the process is to apply the algorithm presented in Algorithm 4 to detect the test smell within the high-level test data model. This algorithm can be implemented in any programming language of choice.

For a similar example in Python, such as the one shown in Listing 25, adding support for its detection in the existing AromaLIA-based solution requires updating only the second step, i.e., the extraction of the high-level test data model.

To generate the LAAST for Python, we simply use the rust-code-analysis tool again. The resulting LAAST is illustrated in Listing 30. It is evident that some nodes, such as the class node, are common to both the Java and Python examples. However, there are also differences in the structure of the LAAST for the Python code.

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

Listing 30 – LAAST for Duplicate Assert example in Python

Listing 31 shows part of the code for a new high-level test data model extractor, this time for Python. It is worth highlighting that, although this is a new implementation, several helper functions from the Java extractor are reused, reducing the need for redundant work.

5.3 AromaLIA Evaluation

This section describes the procedure used to evaluate our approach, including the research questions that guided the study.

```
class PythonHighLevelTestDataModelExtractor {
  extract(node: LAASTNode): TestSuite[] {
    const classes = helpers.findAllClassDefinitions(node);
    return classes.map(c => ({
      // ...
      tests: this.extractTests(c),
    }));
  }
  private extractTests(node: LAASTNode): Test[] {
    const functions = helpers.findAllFunctionDefinitions(node);
    return functions
      .filter(f => f.identifier.startsWith('test_'))
      .map(f => ({
        // ...
        asserts: this.extractAsserts(f),
      }));
  }
  private extractAsserts(node: LAASTNode): TestAssert[] {
    const assertStatements = helpers.findAllAssertStatements(node);
    return assertStatements
      .map(assert => this.extractAssertData(assert));
  }
  // ...
}
```

Listing 31 – Example of High-level Test Data Model Extractor for Python

5.3.1 Research Questions

To systematically evaluate the AromaLIA approach, we formulated the following research questions:

RQ$_1$: *How effective is AromaLIA in detecting test smells compared to existing language-specific test smell detection tools?* This question aims to evaluate the effectiveness of our approach relative to current language-specific methods. For this comparison, we employ the metrics of precision, recall, and F1-score (Chicco; Jurman, 2020).

RQ$_2$: *Which programming languages are most challenging for AromaLIA in detecting test smells?* This question focuses exclusively on AromaLIA's performance across different programming languages. We compare its effectiveness across languages to identify those that present the highest detection challenges. In addition to analyzing the metric values for each language, we also discuss how false positives and false negatives manifest in each case.

RQ$_3$: *Which test smells are most challenging for AromaLIA to detect, both overall and within specific programming languages?* This question investigates which test smells are the most difficult for AromaLIA to detect across all languages. We also examine how the difficulty of detecting specific test smells varies by language. Furthermore, we discuss the difficulty of detecting test smells by category, following the classification used in the test smell catalog of Soares *et al.* (2023).

RQ$_4$: *How does AromaLIA compare to other tools when analyzing multi-language projects?* One of the main advantages of language-independent approaches such as AromaLIA, when compared to conventional techniques, is their ability to be applied to multi-language projects. This research question aims to evaluate the effectiveness of AromaLIA in such contexts, in comparison with the use of multiple language-specific tools to detect test smells within the same project.

5.3.2 Steps

To compare our approach with existing methods for detecting test smells, we implemented a tool based on the AromaLIA detection approach.

5.3.2.1 Implementation of the AromaLIA-Based Detection Tool

We selected the target programming languages, focusing on languages frequently studied in test smell research with available detection tools, while also incorporating less commonly explored languages for novelty. Another criterion was compatibility with the rust-code-analysis tool (Ardito *et al.*, 2020), which we used to generate the LAAST.

Based on these criteria, we selected C#, Java, and Python due to their frequent appearance in test smell research and the availability of detection tools (Paul *et al.*, 2024; Virgínio *et al.*, 2020b; Santana *et al.*, 2020; Peruma *et al.*, 2020; Fernandes *et al.*, 2022; Wang *et al.*, 2021), along with JavaScript and TypeScript, which are less commonly studied. For test frameworks, we selected xUnit for C#, JUnit for Java, PyTest for Python, and Jest for both JavaScript and TypeScript.

We prioritized general test smells that occur across multiple programming languages, are widely discussed in the literature, and are supported by existing detection tools. We selected 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*. These smells are supported by existing detection tools (Paul *et al.*, 2024; Peruma *et al.*, 2020; Bodea, 2022) and have been extensively studied in the selected languages (Campos *et al.*, 2023a; Virgínio *et al.*, 2019; Bodea, 2022; Wang *et al.*, 2021; Paul *et al.*, 2024). Some, such as *Ignored Test* and *Sleepy Test*, are considered particularly critical due to their negative impact on test code maintainability (Martins *et al.*, 2024a; Spadini *et al.*, 2020; Schvarcbacher *et al.*, 2019).

We implemented an AromaLIA-based tool capable of detecting the ten selected test smells across the five chosen languages. The tool is publicly available and open for use and extension [2].

5.3.2.2 Selection of Language-Specific Detection Tools

We identified tools from the literature that support the selected programming languages and test frameworks, as well as at least a subset of the chosen test smells. For Java, we selected *TSDETECT* (Peruma *et al.*, 2020), a well-known and widely adopted tool (Aljedaani *et al.*, 2021), which supports a comprehensive set of test smells, including all those selected for our study. For C#, we chose *xNose* (Paul *et al.*, 2024), which currently supports a broad range of test smells. Among these, it covers nine of the test smells we selected, missing only *Exception Handling*. For Python, we used *PyTest-Smell* (Bodea, 2022), which supports a subset of commonly observed test smells and includes all those relevant to our study. For JavaScript and TypeScript, we identified the *STEEL* tool (Jorge *et al.*, 2021). However, no available version or repository could be found. Consequently, our comparison focuses on the three tools mentioned above, while we analyze AromaLIA's performance on JavaScript and TypeScript separately.

In Section 4.3, we listed SniffML as a tool for detecting test smells that claims to be language-independent, making it an excellent candidate for comparison with our approach. However, when we attempted to use SniffML to detect test smells in languages other than Java, we were unable to do so. We also tried to contact the developer of the tool but did not receive a

[2] https://github.com/publiosilva/aromadr

response. Consequently, SniffML was not included in our comparative evaluation.

5.3.2.3 Preparation of Test Smell Dataset

To establish a ground truth for comparing the detections produced by the tools and to answer RQ_1-RQ_3, we constructed a dataset composed of test cases extracted from Java and Python projects hosted on GitHub. We used search queries targeting Java projects with JUnit tests and Python projects with pytest tests. The exact queries used to retrieve the test files from GitHub are provided in our replication package. To ensure a diverse set of test cases across different scenarios, we did not filter by project. Instead, we searched directly for individual test files using GitHub's code search. All selected files are included in our replication package, along with the URL of the original repository.

The data collection process involved executing search queries on GitHub, manually reviewing the retrieved files, and identifying occurrences of the ten selected test smells. We repeated this process until we obtained at least 20 instances of each test smell.

We collected test files from 58 Java projects and 47 Python projects, totaling 105 projects. The resulting dataset comprises 166 test cases, with 84 written in Java and 82 in Python. To further increase the number of test cases per language, we used `ChatGPT-4o-mini` to translate Java test cases into Python and vice versa. To evaluate the languages C#, JavaScript, and TypeScript, we applied the same translation process to convert all test cases into these languages. Our replication package includes the prompts used to convert code from Java to Python, Java to C#, Python to Java, and Python to C#. Only the original test cases were translated using the LLM, no test case was translated more than once.

After the translation process, we manually validated all generated code. We encountered minor issues, primarily related to syntax, such as extra characters or formatting problems, which we corrected. We also performed a manual re-detection of the ten test smells on the translated code. Most smells were preserved after translation, although some were removed and others were introduced. These changes do not affect the final results, as the final dataset for each language (used to compute the evaluation metrics for each tool) was fully manually validated.

Our final dataset comprises 830 test cases across five programming languages. Table 15 presents detailed statistics for the dataset. The complete dataset is available in our replication package.

To evaluate the AromaLIA approach on multi-language projects (RQ_4), we selected the Apache Beam [3] project. This project is publicly available on GitHub and contains the majority of its codebase in Java, including tests implemented using the JUnit framework. However, the repository also includes code written in other languages, such as Python and Go, as well as tests written in Python using the PyTest framework.

Table 15 – Test smell distribution, number of test files, and LOC statistics by language

Smell / Language	Python	Java	JavaScript	TypeScript	C#
Assertion Roulette	72	75	77	82	73
Conditional Test Logic	46	45	50	49	44
Duplicate Assert	28	29	28	29	30
Empty Test	23	22	22	22	22
Exception Handling	38	53	45	41	41
Ignored Test	26	23	29	29	23
Magic Number Test	46	47	49	49	47
Redundant Print	35	32	34	34	35
Sleepy Test	30	30	31	31	30
Unknown Test	53	48	53	48	48
Number of Files	166	166	166	166	166
LOC (Min)	4	7	6	7	9
LOC (Max)	95	149	127	127	156
LOC (Mean)	18.4	24.7	21.0	22.0	26.6

Source: prepared by the author.

5.3.2.4 *Execution of Detection Tools on the Dataset*

To run xNose, the user must download its source code and execute a command referencing the C# project solution file. The tool then generates a JSON file containing the detected test smells. TSDETECT requires preparing a CSV file with all test file paths and related metadata. This CSV file is provided as a parameter when executing the tool's `.jar` file, which produces another CSV file containing the detection results. For PyTest-Smell, the user installs the tool via `pip` and runs a command specifying the folder that contains the test files. The tool outputs a CSV file with the detected test smells.

Finally, for the AromaLIA-based tool, the process involves executing two `Docker` commands to start a container. Once running, the user accesses a web interface, provides the GitHub repository URL, and selects the programming language and corresponding test framework. The tool automatically identifies all test files, highlights detected test smells directly in the source code, and generates a downloadable JSON report for detailed analysis.

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

We executed the three language-specific tools only on the files in the dataset written in the programming language supported by each respective tool. The AromaLIA-based tool, on the other hand, was evaluated using the entire dataset. All detection results produced by the tools are available in our replication package.

For the multi-language project, we downloaded the Apache Beam repository and executed the AromaLIA-based tool, TSDETECT, and PyTest-Smell. We did not include the xNose tool in this comparison because the Apache Beam project does not contain C# tests. The detection results produced by each tool are available in our replication package.

5.3.2.5 Comparison and Analysis of Results

We analyzed and compared the results produced by the tools to address our research questions. To evaluate their effectiveness in detecting test smells, we computed the *precision*, *recall*, and *F1-score* metrics (Chicco; Jurman, 2020), as each metric captures a different aspect of performance. To ensure a fair comparison, we calculated these metrics considering only the test smells supported by each tool. For instance, when evaluating xNose, we excluded the *Exception Handling* smell because the tool does not support it. Since the tools were executed on different datasets, we did not perform direct comparisons among them. Instead, we compared each tool individually against AromaLIA, which was executed on the same dataset as the corresponding tool. In addition, we analyzed the effectiveness of AromaLIA by programming language and by test smell to identify the main challenges faced by our language-independent approach.

For the comparison on the multi-language project, we did not compute any metrics. Instead, we compared the number of test smells detected by each tool, both overall and by test smell. The goal of this comparison is to assess whether the AromaLIA tool is capable of detecting at least the same number of test smells as the other two tools combined.

5.4 Results and Discussion

In this section, we present and discuss the results of the evaluation of the AromaLIA approach, described in the previous section.

5.4.1 RQ_1: How effective is AromaLIA in detecting test smells compared to existing language-specific test smell detection tools?

In this research question, we aimed to compare the effectiveness of our language-independent approach with existing language-specific approaches. To address this objective, we computed precision, recall, and F1-score for all tools, considering only the ten selected test smells. As mentioned previously, when comparing with xNose, we excluded the *Exception Handling* test smell, since this tool does not support its detection and would therefore yield a metric value of zero.

Table 16 reports the values of the three metrics for each tool. As shown in the table, the AromaLIA-based tool achieved the highest scores across all metrics, with each exceeding 95%. These high precision and recall values indicate that our approach detects test smells with a very low rate of both false positives and false negatives. This observation is further supported by Table 17, which presents the confusion matrices for all tools, including true positives, false positives, true negatives, and false negatives.

In other words, AromaLIA correctly identifies nearly all instances of test smells while rarely misclassifying clean test code as smelly. This strong performance holds both for the overall results and for the results obtained per programming language.

Table 16 – Precision, Recall, and F1-score for all test smell detection tools

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

Source: prepared by the author.

Considering the F1-score, the xNose tool ranks second. Similar to the AromaLIA-based tool, xNose achieved strong precision and recall values, indicating that it is relatively robust to both false negatives and false positives. The TSDETECT tool follows, also exhibiting solid recall and F1-score values. However, its slightly lower precision suggests a higher susceptibility to false positives.

Finally, the PyTest-Smell tool achieved the lowest F1-score. Interestingly, it attained a relatively high precision value, indicating that it produces few false positives. This outcome, however, is attributable to its very low recall, which shows that the tool fails to detect the majority of test smell occurrences.

Table 17 – Confusion matrices for all test smell detection tools

Tool	TP	FP	TN	FN
AromaLIA overall	1,963	41	6,233	63
AromaLIA C#	382	8	1,259	11
AromaLIA Java	398	11	1,245	6
AromaLIA Python	383	6	1,257	14
AromaLIA JavaScript	406	7	1,235	12
AromaLIA TypeScript	394	9	1,237	20
pytest-smell	57	12	1,251	340
tsdetect	339	122	1,134	65
xNose	303	62	1,080	49

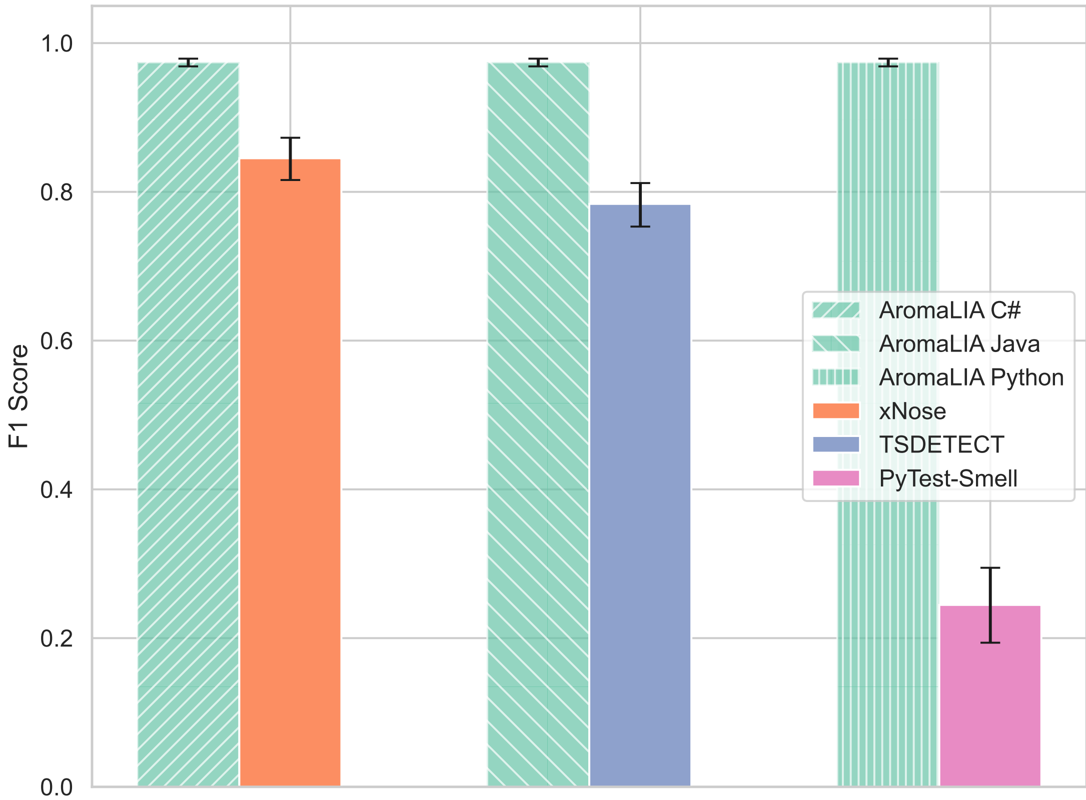
Source: prepared by the author.

Figure 9 compares the F1-score of AromaLIA with those of the other three tools, including their 95% confidence intervals. The AromaLIA F1-score values shown in the figure correspond to the programming language of the tool being compared (for example, when comparing with xNose, we report the AromaLIA F1-score for C#). As illustrated in the figure, the confidence interval for the AromaLIA-based tool is very narrow, indicating high reliability and stability of its metric values. In contrast, the other tools exhibit wider confidence intervals (particularly PyTest-Smell) suggesting greater variability in their detection performance.

We computed the bootstrap confidence intervals using the percentile method with 1,000 bootstrap iterations (Diciccio; Romano, 1988). When comparing tools, non-overlapping 95% confidence intervals indicate a statistically significant difference between results, whereas overlapping intervals do not allow us to conclude whether the difference is statistically significant (Cumming; Finch, 2005). As shown in Figure 9, the confidence intervals of AromaLIA do not overlap with those of the other tools, indicating a statistically significant difference in performance in favor of AromaLIA, and demonstrating that it outperforms the other approaches.

It is worth to mention, however, that while our tool supports 10 test smells, the xNose and TSDETECT tools support 16 and 19, respectively, which is a larger number. This also implies that if our AromaLIA-based tool were used on datasets containing test smells supported by those tools but not by ours, its detection effectiveness for those smells would be zero. Nevertheless, within the scope of this study, we focused on the 10 test smells previously mentioned and demonstrated that our AromaLIA-based tool performs effectively for all of them.

Figure 9 – Comparison of F1-Score for each tool (95% CI)



Source: prepared by the author.

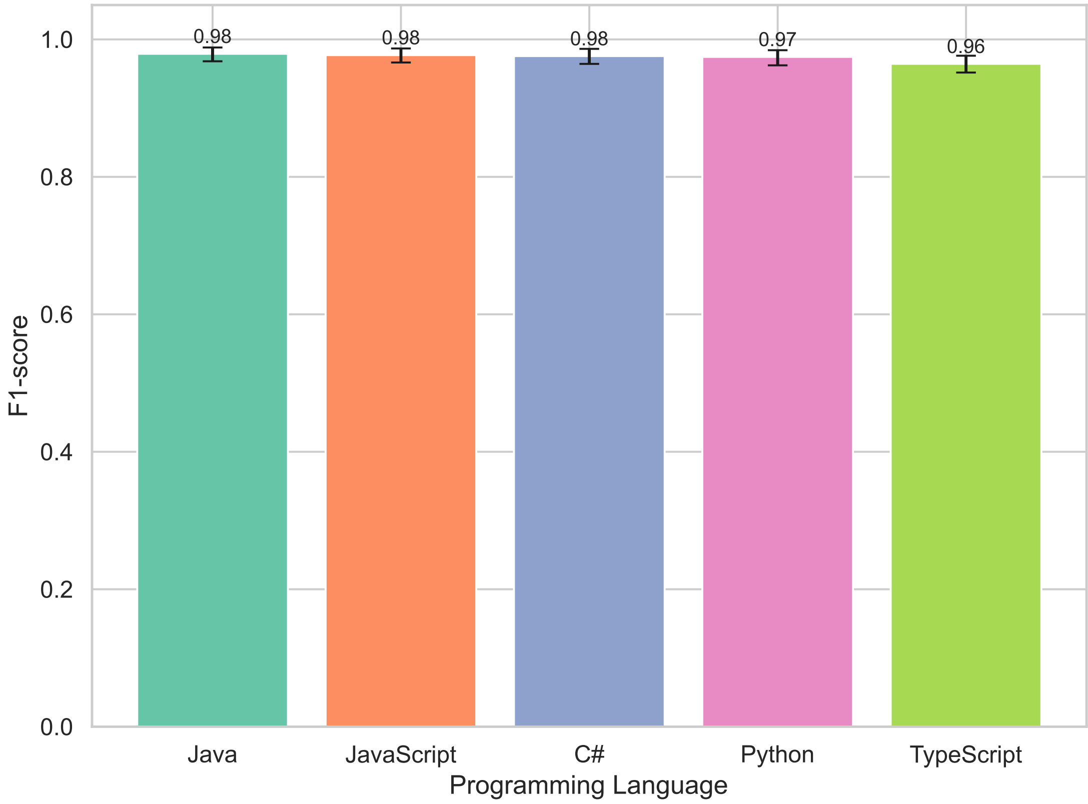
Understanding how our approach would behave in the presence of other test smells requires further investigation.

> **Answer to RQ$_1$:** The AromaLIA-based tool outperforms all language-specific tools, achieving precision, recall, and F1-score values above 95%. Moreover, it demonstrates a strong balance between precision and recall, indicating that it produces very few false positives and false negatives simultaneously.

5.4.2 *RQ$_2$: Which programming languages are most challenging for AromaLIA in detecting test smells?*

To answer this research question, we analyzed the results of the AromaLIA-based tool across the five programming languages included in our dataset (i.e., Java, C#, Python, JavaScript, and TypeScript). Figure 10 presents the F1-score of the AromaLIA-based tool for each language. As shown in the figure, the differences among the metrics are minimal, varying only in decimal digits. The languages Java, C#, and JavaScript achieved the highest F1-scores, followed by Python, while TypeScript obtained the lowest score.

Figure 10 – AromaLIA F1-score by Programming Language (95% CI)



Source: prepared by the author.

These results indicate that the detection performance of the AromaLIA-based tool does not vary significantly across different programming languages, which is consistent with its design as a language-independent approach for detecting test smells. Certain language-specific features that the second step of the AromaLIA detection process (the only step that deals with language-specific constructs) may not fully address, likely cause the minimal observed differences. We discuss these particularities in more detail in the answer to RQ_3.

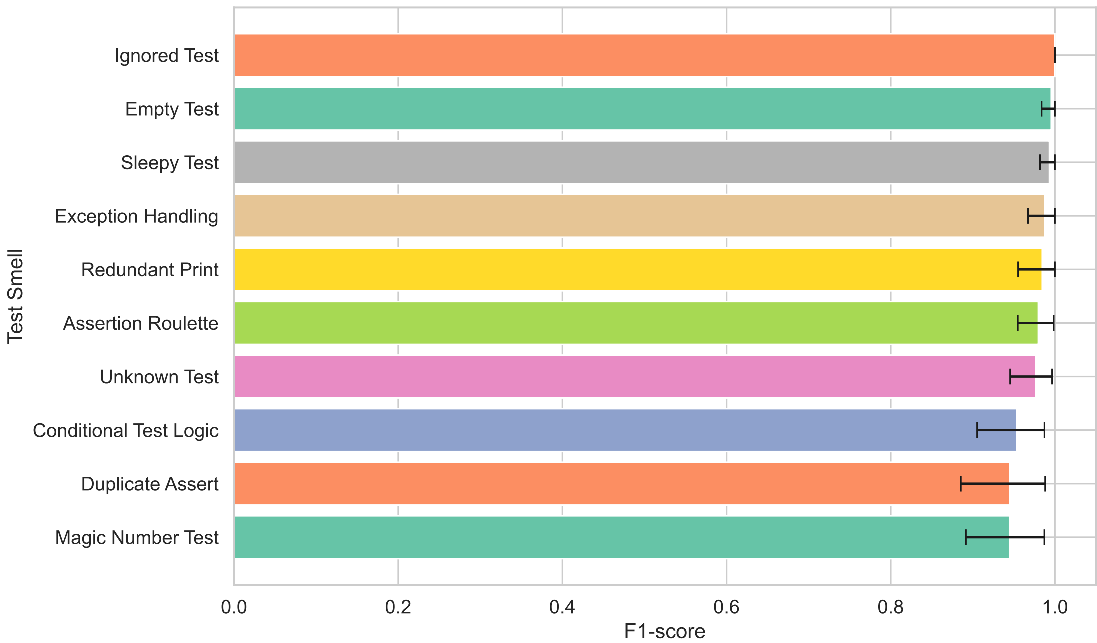
Answer to RQ_2: The language where the AromaLIA-based tool achieved the lowest F1-Score was TypeScript. Nevertheless, its overall performance remained strong and comparable to the other languages. Our approach demonstrates consistent effectiveness across all programming languages. As expected from its language-independent design, the variation in detection performance across languages is minimal.

5.4.3 *RQ$_3$: Which test smells are most challenging for AromaLIA to detect, both overall and within specific programming languages?*

To address this research question, we analyzed the effectiveness of the AromaLIA-based tool in detecting each of the ten selected test smells, examining its overall performance across all programming languages and conducting a per-language analysis.

Figure 11 presents the F1-scores of the AromaLIA-based tool for each test smell across all languages. Our approach successfully detected all instances of the *Ignored Test* smell in every language, achieving a perfect F1 Score of 100%. The tool experienced greater difficulty in detecting three specific test smells: *Magic Number Test*, *Duplicate Assert*, and *Conditional Test Logic*. However, because the corresponding confidence intervals overlap, it is not possible to conclude that the observed differences in F1-scores are statistically significant. Detailed values for each test smell are provided in our replication package.

Figure 11 – AromaLIA F1-score by Test Smell (95% CI)



Source: prepared by the author.

The difficulties observed in detecting the *Magic Number Test* and *Conditional Test Logic* smells may be related to the fact that their detection rules often need to handle code structures that vary significantly across programming languages.

For instance, detecting the *Conditional Test Logic* smell requires identifying conditional constructs in test code (such as `if`, `while`, or `for` loops). Some languages even provide multiple types of conditional constructs. For example, JavaScript and TypeScript both include several types of `for` loops, including the `for...of` loop. In several TypeScript test cases where the AromaLIA-based tool failed to detect the *Conditional Test Logic* smell, we found that the cause for this issue was the presence of a `for...of` loop, which the tool did not correctly map. We plan to address this issue in future versions of the tool. This limitation contributed to the lower F1-score observed for TypeScript. In contrast, when analyzing the equivalent JavaScript test cases, the code used conventional `for` loops, which the tool successfully handled.

A similar situation occurs with the *Magic Number Test* smell. Its detection rule involves identifying literal numbers used in assertions, but its manifestation can vary across languages. For example, in Python's PyTest framework, an assertion can check the equality between two numbers within a tolerance range using the `approx` function, which takes the number as a parameter. In such cases, a number appears in the assertion (thus characterizing a *Magic Number Test*), but it is passed as a parameter to a helper function and not used directly. In some of the analyzed cases, the AromaLIA-based tool failed to identify this smell due to this indirect usage pattern.

Regarding the *Duplicate Assert* smell, its detection rule focuses on identifying assertions within test code that have identical parameters. Detection errors for this smell originated from an issue in rust-code-analysis, the tool responsible for generating the language-agnostic AST. Occasionally, this tool omits string values from the AST for some languages, replacing them with empty strings. Depending on how the assertions are structured, this can lead to high-level test data representations that incorrectly appear to contain *Duplicate Asserts*. We have reported this issue in the official repository of the rust-code-analysis project. Once it is solved, we will update the AromaLIA-based tool accordingly to improve detection accuracy.

We also analyzed the difficulty in detecting test smells according to their categories. The catalog proposed by Soares *et al.* (2023) classifies test smells into five categories, as shown in Table 18. Two belong to the *Test semantic – logic* category, five to the *Code-related* category, two to the *Test execution* category, one to the *Issues in test steps* category, and one to the *Design-related* category.

Figure 12 presents the F1-scores of the AromaLIA-based tool by category. The two categories with the lowest values are *Test semantic - logic* and *Code-related*, which aligns with the three test smells with the lowest F1-scores: *Conditional Test Logic* (under *Test semantic - logic*), and *Duplicate Assert* and *Magic Number Test* (under *Code-related*). The confidence

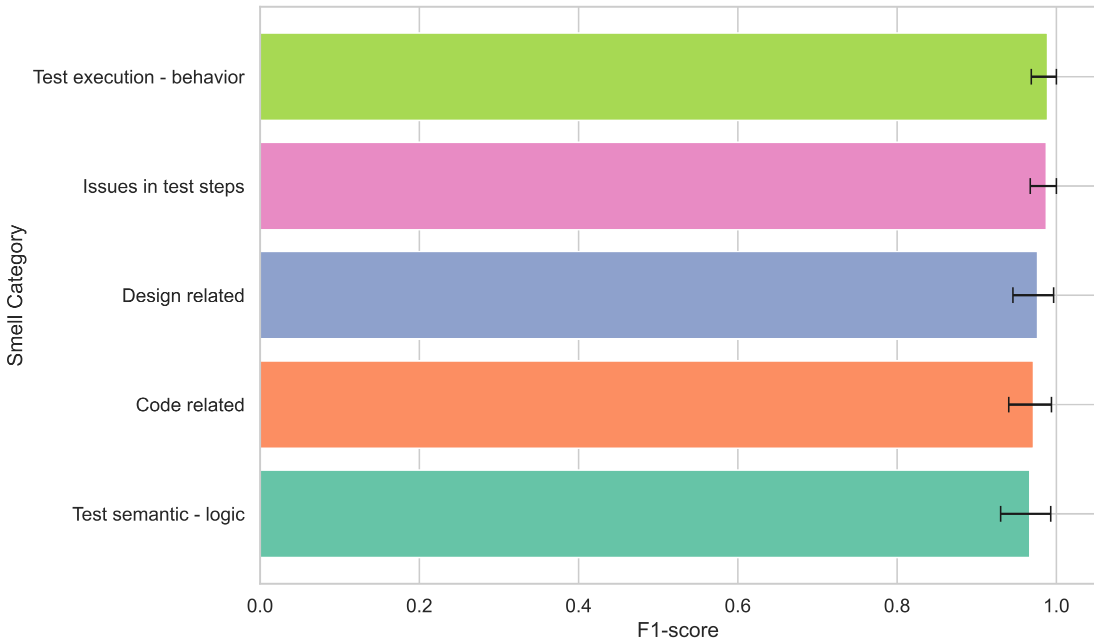
Table 18 – Test Smells Categories

Category	Test Smell
Test semantic – logic	*Assertion Roulette*
Test semantic – logic	*Conditional Test Logic*
Code related	*Duplicate Assert*
Code related	*Empty Test*
Code related	*Ignored Test*
Code related	*Magic Number Test*
Test execution – behavior	*Redundant Print*
Test execution – behavior	*Sleepy Test*
Issues in test steps	*Exception Handling*
Design related	*Unknown Test*

Source: prepared by the author.

interval for the *Test semantic - logic* category is the widest, which can be attributed to this category including one smell with lower detection effectiveness (*Conditional Test Logic*) and another where our tool performed very well (*Assertion Roulette*).

Figure 12 – AromaLIA F1-score by Smell Category (95% CI)



Source: prepared by the author.

Finally, we analyzed the relationship between programming languages and test smells with respect to the effectiveness of the AromaLIA-based tool. Figure 13 presents a heatmap illustrating this relationship. The *Magic Number Test*, *Duplicate Assert*, and *Conditional Test Logic* smells are the most challenging to detect across most of the five languages. The remaining seven test smells exhibit more consistent detection effectiveness across all languages. Even for the three more difficult smells, the differences in F1-score are minimal (only a few decimal digits), highlighting the overall stability and robustness of the AromaLIA-based tool in accurately detecting test smells across all five languages.

Figure 13 – Heatmap of AromaLIA F1-score by Test Smell and Language

Source: prepared by the author.

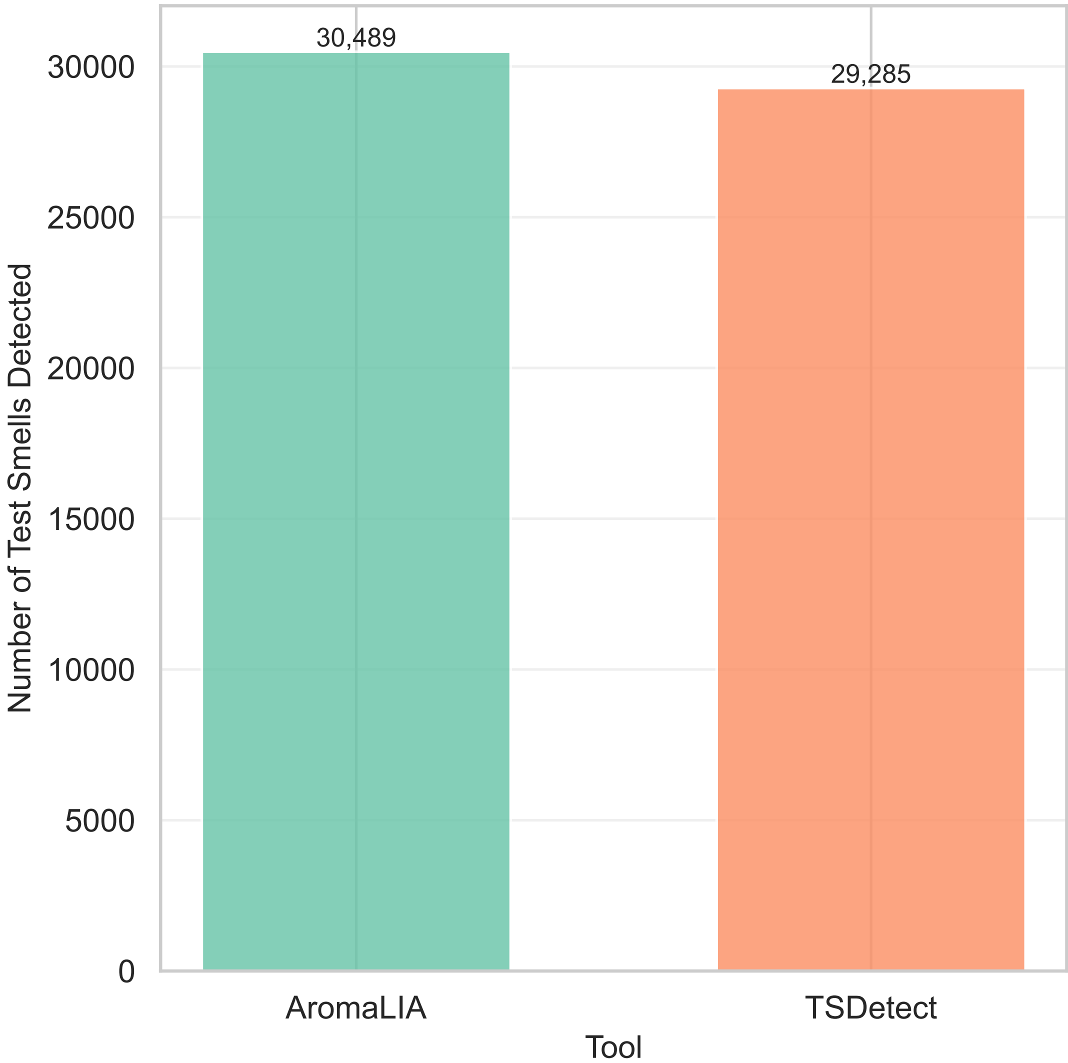
> **Answer to RQ$_3$:** The three test smells that were most difficult for the AromaLIA-based tool to detect, based on the F1-score values, were *Magic Number Test*, *Duplicate Assert*, and *Conditional Test Logic*. However, the differences in the metric values were minimal, indicating the robustness and stability of the AromaLIA-based tool's detection performance.

5.4.4 *RQ$_4$: How does AromaLIA compare to other tools when analyzing multi-language projects?*

To answer this research question, we executed the AromaLIA-based tool, TSDETECT, and PyTest-Smell to detect test smells in the Apache Beam project. We then computed the number of test smell instances detected by each tool. For this comparison, we only considered the 10 test smells analyzed in this work, which are the same test smells used for the previous research questions. TSDETECT was applied to detect test smells in the Java/JUnit tests, PyTest-Smell was applied to the Python/PyTest tests, and the AromaLIA-based tool was applied to detect test smells across both languages in the project.

Figure 14 shows the comparison between the total number of test smells found by the AromaLIA-based tool and TSDETECT on the Java/JUnit tests of the Apache Beam project. Both tools detected a substantial number of test smells; however, the AromaLIA-based tool identified over 1,200 more instances than TSDETECT. It is important to note that in this comparison, we do not account for the test smells detected by TSDETECT that the AromaLIA-based tool cannot detect. Thus, considering all the test smells supported by TSDETECT, it detects a higher total number of test smells. Nevertheless, when focusing only on the 10 test smells analyzed in this work, the AromaLIA-based tool detected more instances than TSDETECT.

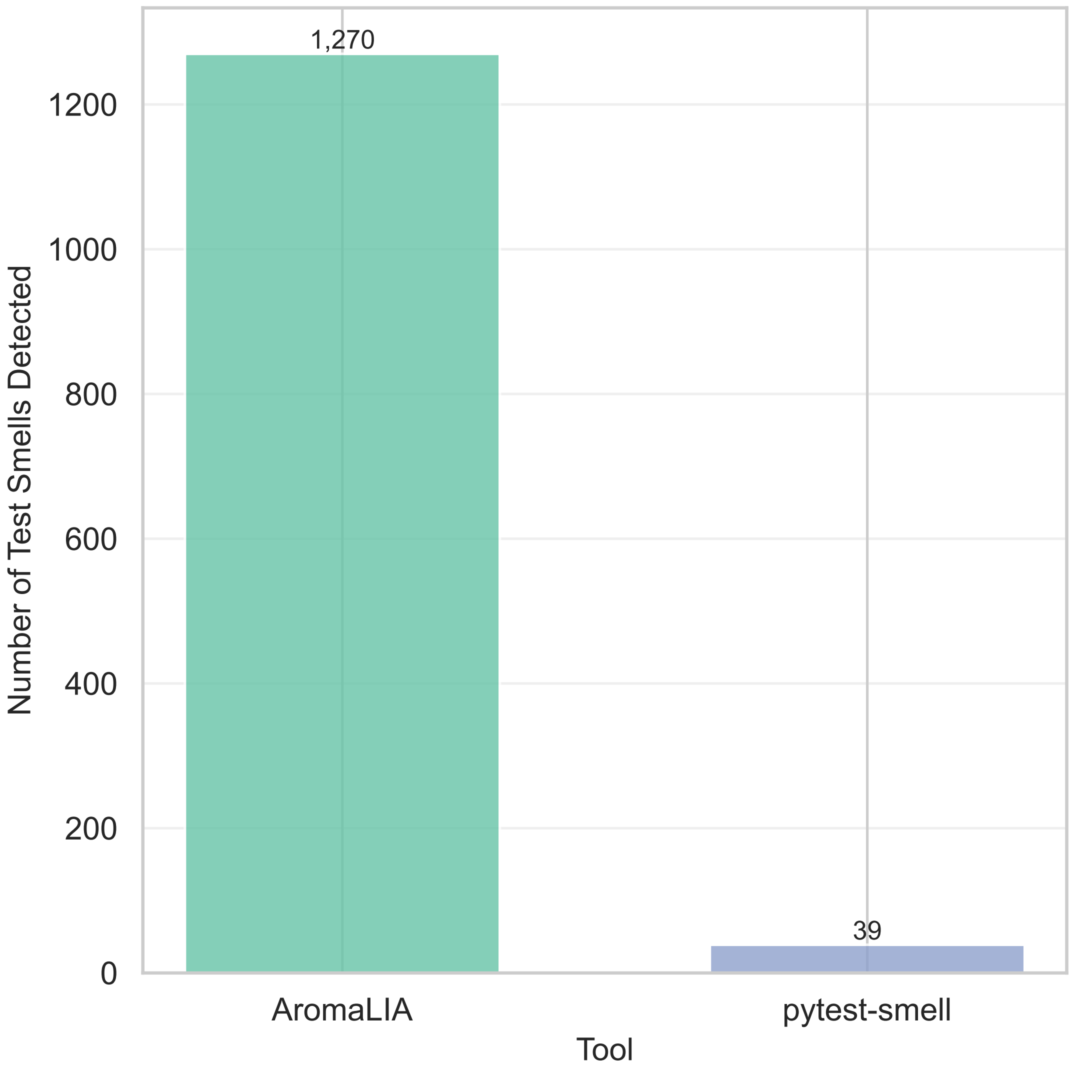
Figure 14 – Overall Test Smell Detection Comparison (AromaLIA vs TSDETECT)



Source: prepared by the author.

Figure 15 shows the same comparison, this time between the AromaLIA-based tool and the PyTest-Smell tool. As shown in the figure, the difference in the number of detected test smells is substantial. While the AromaLIA-based tool identified 1,270 instances of test smells, the PyTest-Smell tool detected only 39 instances. This result aligns with the findings from the other research questions, which demonstrated that the effectiveness of PyTest-Smell is significantly lower than that of the AromaLIA-based tool.

Figure 15 – Overall Test Smell Detection Comparison (AromaLIA vs PyTest-Smell)



Source: prepared by the author.

Table 19 presents the counts of test smells detected by each tool, broken down by test smell type. Although the AromaLIA-based tool achieved a higher total number of detected test smells in the Java tests of the Apache Beam project compared to TSDETECT, for certain test smells (namely *Exception Handling*, *Ignored Test*, and *Magic Number Test*)TSDETECT detected more instances than AromaLIA. The largest difference was observed for *Magic Number Test*. However, as discussed in the answer to RQ_3, this test smell is one of the more challenging for the AromaLIA-based tool to detect relative to the other nine test smells. It is also important to note that in our previous comparison for RQ_1, the precision of TSDETECT was lower than that of the AromaLIA-based tool, suggesting that some instances detected by TSDETECT may be false positives.

Regarding the comparison between the AromaLIA-based tool and PyTest-Smell, the table shows that AromaLIA detected more instances for all ten test smells considered, with some differences being substantial, for example, in *Unknown Test*.

Table 19 – Test Smell Counts Per Test Smell on Apache Beam

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

Source: prepared by the author.

When considering both languages together, the AromaLIA-based tool detected approximately 2,500 more instances than the other two tools combined, demonstrating its effectiveness in multi-language projects. In this comparison, we only considered Java and Python and compared AromaLIA with TSDETECT and PyTest-Smell. Future work could extend this comparison by including additional tools for other programming languages.

Answer to RQ_4: The AromaLIA-based tool detected more test smell instances in a multi-language project than two language-specific tools applied separately on the same project.

5.5 Threats to Validity

Internal Validity: To validate the test smell detection solutions analyzed in this study, we randomly selected test cases from public projects on GitHub. We also employed

ChatGPT-4o-mini to translate these samples, thereby constructing a dataset that includes test cases in all five programming languages considered in our work. Although all samples were manually validated, this process may introduce bias due to subjective interpretation of test smell definitions, particularly in ambiguous cases. To mitigate this threat, ambiguous instances were discussed among the authors to reach a consensus.

Construct Validity: A potential threat concerns our binary approach to labeling test instances with respect to the presence of a test smell (i.e., whether a smell exists or not). We did not account for the number of occurrences of each test smell within a test case, which could provide additional insights and a more fine-grained evaluation. As future work, we plan to investigate AromaLIA's capability to detect test smells while also quantifying their occurrences.

External Validity: Although this study covers a broader set of programming languages compared to most prior works on test smell detection, the performance of AromaLIA should be further evaluated in languages beyond the five included here. Moreover, our evaluation considered only one testing framework per language, while many languages support multiple frameworks with distinct characteristics. Future studies should therefore assess AromaLIA's effectiveness across different frameworks and additional programming languages to enhance generalizability. Furthermore, for the multi-language project comparison, future work should include additional languages and a broader set of tools to compare with the AromaLIA-based tool, which would further strengthen the generalizability of the results.

Conclusion Validity: One limitation of our comparative analysis is that not all evaluated tools support the same set of test smells. To address this, we calculated and compared metrics only for the smells supported by each respective tool. Another concern is the lack of available tools for JavaScript and TypeScript, which prevented us from including these languages in our comparison. Expanding the study to include such tools in future work would provide a more comprehensive evaluation of AromaLIA's performance across languages.

5.6 Conclusion

In recent years, several studies have addressed the detection of test smells. However, most existing approaches are language-specific. In this work, we propose AromaLIA, a language-independent approach for detecting test smells. We validated our approach by developing an AromaLIA-based tool and comparing it with existing test smell detection tools for C#, Java, and Python. The effectiveness of the tools was evaluated using a manually validated, pre-classified dataset containing 830 instances of test code encompassing 10 types of test smells. The AromaLIA-based tool achieved a precision of 97%, a recall of 96%, and an F1-score of 97%, outperforming all language-specific tools. Our approach lays the foundation for test smell detection solutions that are more reusable and broadly applicable, thereby facilitating the integration of new languages.

6 AROMADR: A LANGUAGE-INDEPENDENT TOOL FOR DETECTING TEST SMELLS

In this chapter, we present a tool named AromaDr, which implements the AromaLIA approach for detecting ten test smells across five different programming languages. The chapter is organized as follows. Section 6.1 provides the context and introduces the study. Section 6.2 describes the implementation of AromaDr, including its architecture and supported features. Section 6.3 illustrates a practical example of how the tool can be used. Section 6.4 presents a comparison between AromaDr and existing tools, highlighting its strengths and limitations. Finally, Section 6.5 concludes the chapter by summarizing the main contributions and outlining directions for future improvement.

6.1 Introduction

Ensuring the quality of developed software is essential for the success of any software project (Peruma *et al.*, 2020). One of the primary approaches to achieving this is software testing (Tran *et al.*, 2021). Testing can be performed manually or by automation (Garousi; Küçük, 2018). Automated tests offer significant advantages, including ease of execution and the ability to reproduce results consistently (Garousi; Küçük, 2018).

However, when using automated tests, it is equally important to assess the quality of the test code to ensure its continued usefulness over time (Tran *et al.*, 2021). The significance of maintaining high-quality test code is underscored by the fact that developers spend approximately one-quarter of their time writing tests (Beller *et al.*, 2015). Therefore, ensuring test quality is crucial to avoid unnecessary costs and additional development effort (Garousi; Küçük, 2018).

One of the key factors that can hinder the quality of test code is the presence of test smells (Panichella *et al.*, 2022). Test smells are poor design choices made during the development of test code (Aljedaani *et al.*, 2021). These suboptimal practices result in test code that is more difficult to read, maintain, and evolve (Junior *et al.*, 2021). Test smells may arise due to the inherent complexity of the system or the limited experience of the developers (Santana *et al.*, 2024).

This topic has gained increasing relevance in recent years, with numerous studies exploring it and proposing tools to detect and refactor test smells (Paul *et al.*, 2024; Virgínio *et al.*, 2020a; Santana *et al.*, 2024; Peruma *et al.*, 2020; Bodea, 2022; Wang *et al.*, 2021). Such tools are essential, as manual detection and refactoring of test smells are often impractical in large-scale software systems (Santana *et al.*, 2020).

However, most existing tools are language-specific, which makes it challenging to support additional programming languages in the context of test smell detection (Aljedaani *et al.*, 2021). To address this limitation, we proposed a language-independent approach for detecting test smells in a previous study (Silva *et al.*, 2024). 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.

6.2 AromaDr Tool

The AromaDr tool implements a language-independent approach for detecting test smells, as proposed in our previous work (Silva *et al.*, 2024). 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.

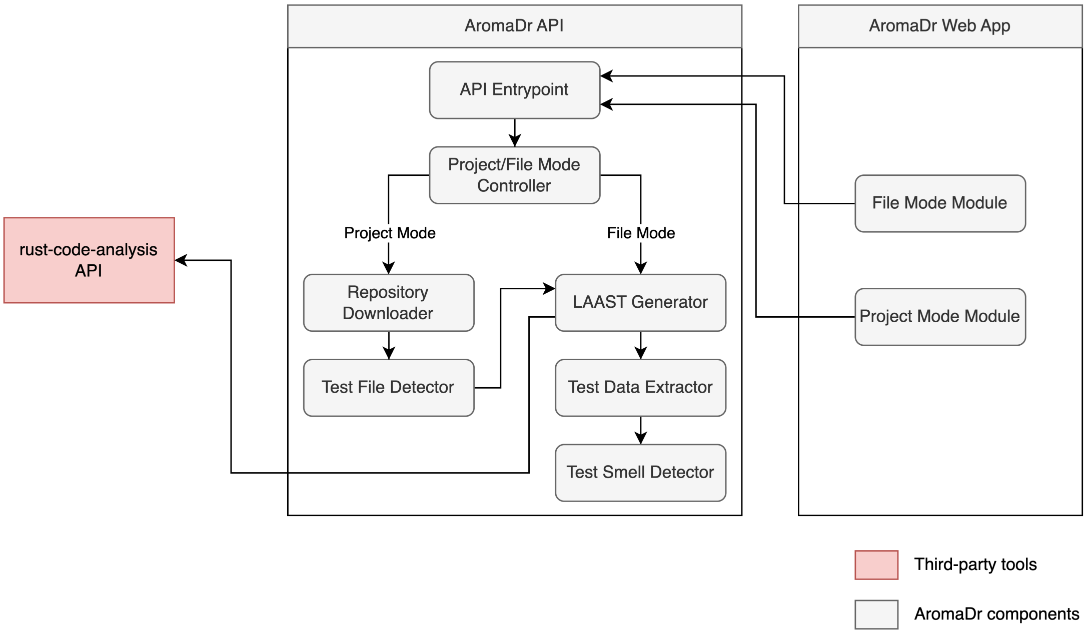
Currently, AromaDr supports five programming languages, each paired with a commonly used test framework: C# with xUnit, Java with JUnit, JavaScript or TypeScript with Jest, and Python with PyTest. The tool is capable of detecting ten types of test smells, namely: *Assertion Roulette*, *Conditional Test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

Figure 16 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 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 16). 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 16). To achieve this, we utilize Mozilla's rust-code-analysis crate (Ardito *et al.*, 2020). 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, Java, JavaScript, JavaScript (Firefox internals), Python, Rust, and

Figure 16 – AromaDr architecture



Source: prepared by the author.

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, language-independent format (Test Data Extractor in Figure 16). 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 32.

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 16). 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 2 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 (Peruma *et al.*, 2020).

```
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 32 – Test data model extract in the second step of the AromaDr test smell detection process

Using the test data model presented in Listing 32, 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.

Algorithm 2 Assertion Roulette detection algorithm

```
function DetectAssertionRoulette(test)
    hasMoreThanOneAssert ← length of test.asserts > 1
    hasSomeAssertWithoutMessage ← False
    for each assert in test.asserts do
        if assert.message is empty then
            hasSomeAssertWithoutMessage ← True
            break
        end if
    end for
    return hasMoreThanOneAssert and hasSomeAssertWithoutMessage
end function
```

Conditional Test Logic. Algorithm 3 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`) (Peruma *et al.*, 2020).

In the test data model shown in Listing 32, 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 3 Conditional Test Logic detection algorithm

```
function DetectConditionalTestLogic(test)
    for each statement in test.statements do
        if statement.type is condition or loop then
            return True
        end if
    end for
    return False
end function
```

Duplicate Assert. Algorithm 4 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 (Peruma *et al.*, 2020).

Using the test data model introduced in Listing 32, 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 4 Duplicate Assert detection algorithm

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

Empty Test. Algorithm 5 presents the procedure for detecting the *Empty Test* smell. This smell is characterized by the complete absence of executable statements within a test case (Peruma *et al.*, 2020).

Using the test data model described in Listing 32, detection is straightforward: the algorithm simply checks whether the `statements` list (representing all statements present in the test) is empty. If the list contains no elements, the test is considered empty and flagged accordingly.

Algorithm 5 Empty Test detection algorithm

```
function DetectEmptyTest(test)
    if test.statements is empty then
        return True
    end if
    return False
end function
```

Exception Handling. Algorithm 6 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 (Peruma *et al.*, 2020).

In the test data model introduced in Listing 32, 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 6 Exception Handling detection algorithm

```
function DETECTEXCEPTIONHANDLING(test)
    for each statement in test.statements do
        if statement.type is exceptionHandling or exceptionThrowing then
            return True
        end if
    end for
    return False
end function
```

Ignored Test. Algorithm 7 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 (Peruma *et al.*, 2020).

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 32) is set to `true`. Therefore, the algorithm to detect this smell is simple: it checks whether the `isIgnored` parameter is set to `true`.

Algorithm 7 Ignored Test detection algorithm

```
function DETECTIGNOREDTEST(test)
    return test.isIgnored
end function
```

Magic Number Test. Algorithm 8 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 (Peruma *et al.*, 2020), also known as “magic numbers.”

Using the test data model derived in the previous step (Listing 32), 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.

Redundant Print. Algorithm 9 outlines the process for detecting the *Redundant Print* test smell. This smell occurs when a test contains one or more unnecessary `print`

Algorithm 8 Magic Number Test detection algorithm

```
function DetectMagicNumberTest(test)
    if length of test.asserts > 1 then
        for each assert in test.asserts do
            if IsNumeric(assert.literalActual) or IsNumeric(assert.literalExpected) then
                return True
            end if
        end for
    end if
    return False
end function
```

statements (Peruma *et al.*, 2020).

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 32) 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 9 Redundant Print detection algorithm

```
function DetectRedundantPrint(test)
    for each event in test.events do
        if event = "print" then
            return true
        end if
    end for
    return false
end function
```

Sleepy Test. Algorithm 10 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 (Peruma *et al.*, 2020).

As part of the data model returned in the previous step (Listing 32), 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.

Algorithm 10 Sleepy Test detection algorithm

```
function DetectSleepyTest(test)
    for each event in test.events do
        if event = "sleep" then
            return true
        end if
    end for
    return false
end function
```

Unknown Test. Algorithm 11 outlines the procedure for detecting the *Unknown Test* test smell. This smell occurs when a test contains no assertions (Peruma *et al.*, 2020).

Detecting this test smell using the test data model derived in the previous step (Listing 32) 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 11 Unknown Test detection algorithm

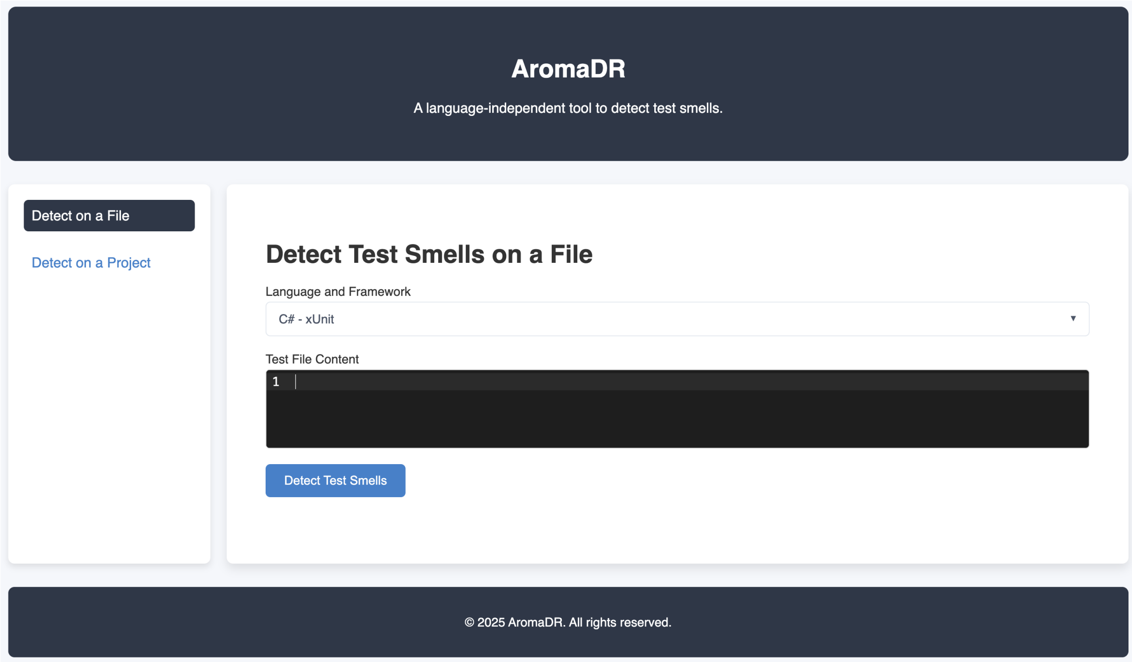
```
function DetectUnknownTest(test)
    if length of test.asserts = 0 then
        return True
    end if
    return False
end function
```

6.3 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 (Silva *et al.*, 2025). 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 17 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.

Figure 17 – AromaDr initial view (file mode)



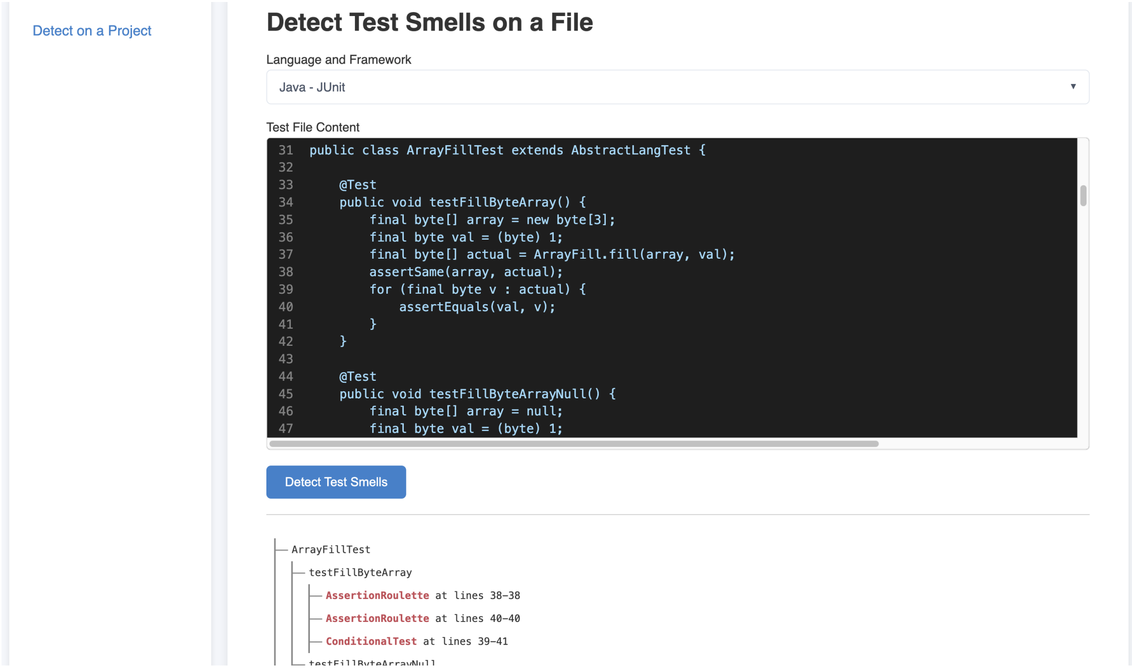
Source: prepared by the author.

After clicking the *Detect Test Smells* button in file mode, the identified test smells are displayed, as shown in Figure 18. 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.

Figure 19 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.

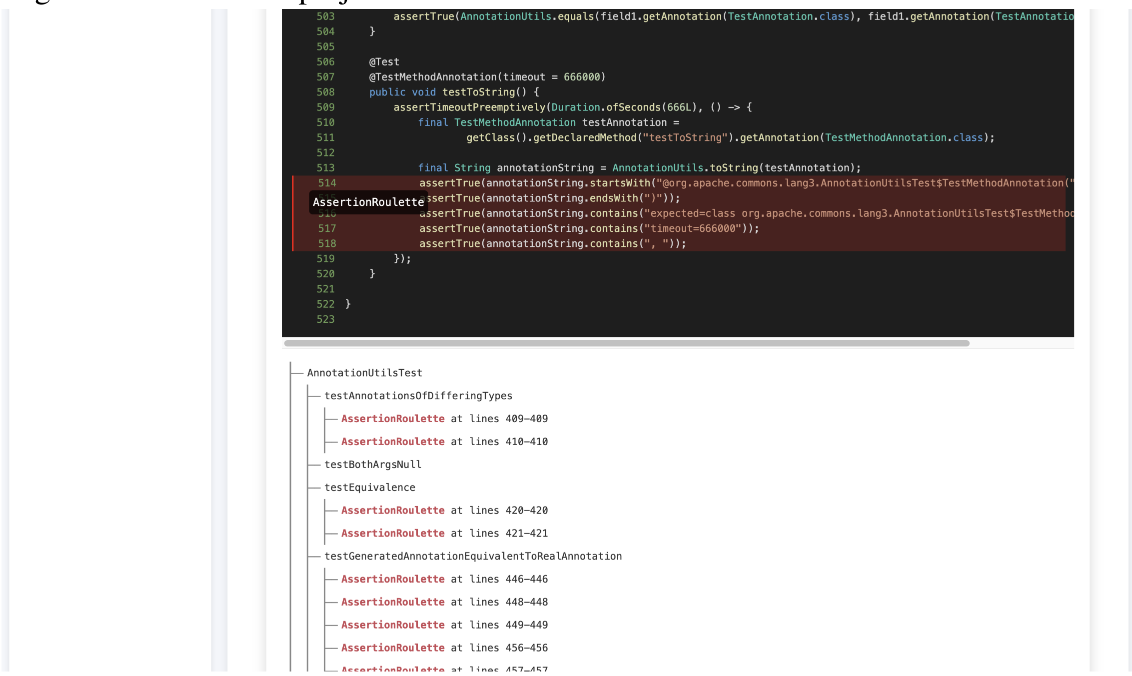
It is also possible to use the AromaDr tool through its REST API. The `README.md` file in the tool's GitHub repository provides detailed instructions for using the AromaDr API in both modes (file-based and project-based). This feature enables seamless integration of the tool into automated pipelines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

Figure 18 – AromaDr file mode test smell detection



Source: prepared by the author.

Figure 19 – AromaDr project mode test smell detection



Source: prepared by the author.

6.4 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 (Peruma *et al.*, 2020). 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 (Wang *et al.*, 2021). 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 (Paul *et al.*, 2024), 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 (Jorge *et al.*, 2021). 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 (Lopes *et al.*, 2024) 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. However, our tool supports more languages (five vs. four), detects all SniffML's smells plus three more, and offers a graphical user interface compared to SniffML's command-line interface.

6.5 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.

7 CONCLUSION AND FUTURE WORK

In this Master's dissertation, we propose AromaLIA, a language-independent approach for detecting test smells. This approach is a significant contribution for both researchers and developers because it not only enables test smell detection on code written in different programming languages using a single unified solution, but also simplifies the process of adding support for new languages by significantly reducing the required rework.

Test smells have become an increasingly relevant topic in recent years due to their potential to hinder the maintainability and evolution of test code (Tran *et al.*, 2021). Several studies have proposed techniques to detect and refactor test smells (Aljedaani *et al.*, 2021). However, most existing approaches are language-specific (Aljedaani *et al.*, 2021). This design makes it difficult to extend the approach to support additional programming languages. As a result, addressing test smells in a new language often requires developing a solution from scratch, a process that is both time-consuming and resource-intensive.

Despite differences in syntax and testing frameworks, many test smells manifest in similar ways across languages and can be detected using comparable patterns. This observation suggests the feasibility of developing a more general, language-independent solution for detecting test smells. However, researchers have not widely explored such solutions in the literature.

Indeed, Lopes *et al.* (2024) proposed a language-agnostic approach for detecting test smells. Their technique performs detection on an XML representation of the test code, generated independently of the programming language. To the best of our knowledge, the approach presented by Lopes *et al.* (2024) is the only existing language-agnostic method for detecting test smells. However, it has a key limitation: the tool used to extract the XML representation, *srcML*, supports only four programming languages. In our work, we perform detection on an AST generated by the *rust-code-analysis* tool (Ardito *et al.*, 2020), which currently supports eleven languages. Moreover, being an open-source tool maintained by Mozilla, it can be further extended to support additional languages.

Recent literature also shows an increasing adoption of LLM-based solutions across several domains, including test smell detection. This line of research is particularly appealing for achieving language-independent detection, as it avoids building an entire solution from scratch and instead leverages the knowledge embedded in pretrained LLMs. We identified several recent studies that employ LLM-based approaches to detect test smells across multiple languages (Lucas *et al.*, 2024; SANTANA JUNIOR *et al.*, 2025). However, the reported accuracy in these works suggests that more traditional approaches, including our own, still outperform current LLM-based methods. Therefore, further research is needed to fully explore the potential of LLM-based solutions in reducing the effort required for test smell detection while maintaining high accuracy.

This chapter presents a summary of the main contributions of this Master's dissertation, along with the publications that emerged during its development. We also outline future work directions to further advance this research.

7.1 Main Contributions

The main goal of this Master's dissertation was to propose a language-independent approach for detecting test smells. To achieve this, we first investigated how test smells are addressed in the literature across different programming languages through a SMS, presented in Chapter 4. Next, we introduced *AromaLIA*, our language-independent test smell detection approach, and compared it with existing solutions, as presented in Chapter 5. Finally, we developed a tool that implements AromaLIA, as described in Chapter 6. Below, we present the main contributions of this Master's dissertation.

Contribution 1: *An analysis of test smell prevalence across programming languages*. In Chapter 4, we conducted an SMS to explore how test smells manifest across programming languages. In this study, we analyzed which languages and testing frameworks are most frequently addressed in the context of test smells, as well as which test smells are most common within each language or framework. We also examined the prevalence of test smells based on their categories (e.g., code-related, dependency-related). This analysis provides a foundation for future research, enabling researchers to explore understudied languages or test smells that have received limited attention in the literature.

Contribution 2: *A catalog of test smells*. Also in Chapter 4, we present a catalog of 213 test smells encompassing both universal and language-specific instances. This catalog is particularly valuable for the development of future language-independent detection approaches, as such approaches should prioritize test smells that are broadly applicable across languages. Additionally, the catalog can support further investigation into whether test smells that appear to be language-specific (because they were reported only in one language) are truly exclusive to that language or could occur in other languages as well.

Contribution 3: *A catalog of test smell refactorings*. In Chapter 4, we also provide a catalog of 94 refactorings associated with 56 distinct test smells. This contribution can support future research focusing on test smell remediation and automated refactoring techniques.

Contribution 4: *A catalog of test smell datasets*. Chapter 4 also includes a catalog of 16 test smell datasets covering the languages C#, C++, Java, Python, and Robot Framework. Given that constructing a test smell dataset is a non-trivial and time-consuming task, the reuse of existing datasets is both useful and encouraged. These datasets can support future studies on test smell detection and refactoring, enabling consistent comparisons among different approaches.

Contribution 5: *A catalog of test smell detection tools*. We additionally provide, in Chapter 4, a catalog of 36 test smell detection tools, 22 of which remain available. This catalog can assist future researchers in selecting appropriate tools for comparative studies on test smell detection techniques.

Contribution 6: *A test smell criticality classification*. In Chapter 4, we also present a classification of test smell criticality based on data synthesized from multiple studies in the literature. This classification considers two perspectives: developers' perceptions and the impact of test smells on maintainability. It can guide future research by helping identify which test smells deserve priority, rather than selecting smells solely based on their frequency in past studies.

Contribution 7: *AromaLIA: a language-independent approach for detecting test smells*. The central contribution of this dissertation is our proposed language-independent approach for detecting test smells, detailed in Chapter 5. This approach benefits both developers and researchers, as it enables test smell detection in currently supported languages and facilitates the addition of new languages without requiring a complete solution to be developed from scratch. Furthermore, given AromaLIA's strong results in terms of precision, recall, and F1-score, it can serve as a baseline for evaluating emerging detection techniques, including LLM-based approaches.

Contribution 8: *A multi-language dataset of test smells*. Chapter 5 also introduces a dataset containing 830 test smell instances across C#, Java, Python, JavaScript, and TypeScript. This dataset can support future research efforts in both detection and refactoring tasks and is particularly valuable for evaluating language-independent approaches.

Contribution 9: *A tool for detecting test smells in multiple languages*. In Chapter 6, we present *AromaDr*, a tool that implements the AromaLIA approach. The tool supports the detection of ten test smells across five programming languages. To the best of our knowledge, it is currently the tool supporting the largest number of languages in this context. This tool is useful for both practitioners and researchers, and it can serve as a benchmark in future studies on test smell detection.

7.2 Publications

To date, two publications related to this work have been accepted, and two others have been submitted. Table 20 provides details regarding these publications. Additionally, Table 21 lists two other publications that are not directly related to this Master's dissertation.

Table 20 – Publications and submissions related to the dissertation

Publication	Description
SILVA, P.; BEZERRA, C.; MACHADO, I. Toward a Language-Agnostic Approach to Detect Test Smells. In: Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software. Porto Alegre, RS, Brasil: SBC, 2024. p. 686–692. ISSN 0000-0000. Available at: https://sol.sbc.org.br/index.php/sbes/article/view/30413. DOI: https://doi.org/10.5753/sbes.2024.3647.	In this study, we developed a proof of concept of our language-independent approach for detecting test smells, focusing on two test smells across two programming languages.
SILVA, P.; BEZERRA, C.; MACHADO, I.; RIBEIRO, M. AromaDr: A Language-Independent Tool for Detecting Test Smells. In: Anais do XXXIX Simpósio Brasileiro de Engenharia de Software. Recife/PE, Brasil: SBC, 2025. p. 914–920. ISSN 2833-0633. Available at: https://sol.sbc.org.br/index.php/sbes/article/view/37078. DOI: https://doi.org/10.5753/sbes.2025.11114.	This study corresponds to the work presented in Chapter 6, in which we proposed a tool implementing our language-independent approach for detecting ten test smells across five programming languages.
SILVA, P.; BEZERRA, C.; MACHADO, I.; RIBEIRO, M. *Exploring Test Smells Across Programming Languages: A Systematic Mapping Study*. Submitted to the *Journal of Systems and Software (JSS)*.	This study corresponds to Chapter 4, where we conducted a SMS to better understand how test smells are characterized in the literature across different programming languages.
SILVA, P.; BEZERRA, C.; MACHADO, I.; RIBEIRO, M. *AromaLIA: A Language-Independent Approach to Detect Test Smells*. Submitted to the *International Conference on Evaluation and Assessment in Software Engineering (EASE)*.	This study corresponds to Chapter 5, where we present and evaluate our language-independent approach for detecting test smells.

Source: prepared by the author.

7.3 Future Work

In this section, we propose several directions for future work to expand the research conducted in this Master's dissertation.

Future Work 1: *Extend AromaLIA to support language-independent test smell refactoring.* Beyond language-independent test smell detection, a largely unexplored topic is

Table 21 – Other publications and submissions not related to the dissertation

Publication	Description
ALVES, V.; BEZERRA, C.; MACHADO, I.; ROCHA, L.; VIRGÍNIO, T.; SILVA, P. Quality Assessment of Python Tests Generated by Large Language Models. arXiv preprint, 2025. Available at: https://arxiv.org/abs/2506.14297. arXiv:2506.14297 [cs.SE].	This study evaluates the quality of Python test code generated by several LLMs across different prompting contexts.
SILVA, R.; SILVA, P.; BEZERRA, C.; UCHÔA, A.; GARCIA, A. Unveiling the Relationship Between Continuous Integration and Code Review: A Study with 10 Closed-source Projects. Submitted to SBES 2025 – XXXIX Simpósio Brasileiro de Engenharia de Software.	This study investigates the relationship between Continuous Integration and Code Review practices in closed-source software development.

Source: prepared by the author.

language-independent test smell refactoring. LLM-based solutions seem promising, but as far as we know, no conventional approaches are addressing this problem. The AromaLIA approach could be extended to support test smell refactoring in a language-independent manner.

Future Work 2: *Investigate language-specific test smells.* In Chapter 4, we presented a list of language-specific test smells (i.e., those appearing in a single language). However, some of these test smells may simply be underexplored in the literature and may, in fact, appear in other languages. Investigating these test smells and their occurrence in less-studied languages represents an interesting path for future research.

Future Work 3: *Increase the number of test smells and languages supported by AromaDr.* The tool proposed in Chapter 6 currently supports the highest number of languages among similar tools. Future work could aim to increase the number of supported languages. Additionally, increasing the number of test smells supported by the tool (currently 10) would also be valuable, as some existing tools support a higher number of test smells.

Future Work 4: *Compare AromaLIA with LLM-based approaches.* LLM-based approaches appear to be a promising direction for future research on test smells. However, many aspects of such approaches remain underexplored, including how they compare with more conventional approaches across different types of test code (e.g., varying code sizes, languages, and frameworks). Comparing an LLM-based approach with AromaLIA would be an interesting direction for future work.

Future Work 5: *Evaluate the use of AromaDr in real-world scenarios.* Another valuable direction is to evaluate the usability and effectiveness of the AromaDr tool in real-world projects. This includes studying how developers use and perceive the tool, particularly in contexts with multi-language codebases.

BIBLIOGRAPHY

AFONSO, J.; CAMPOS, J. Automatic generation of smell-free unit tests. In: IEEE/ACM INTERNATIONAL WORKSHOP ON SEARCH-BASED AND FUZZ TESTING. **Proceedings [...]**. [*S. l.*], 2023. p. 9–16.

ALJEDAANI, W.; MKAOUER, M. W.; PERUMA, A.; LUDI, S. Do the test smells assertion roulette and eager test impact students' troubleshooting and debugging capabilities? In: 45TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING: SOFTWARE ENGINEERING EDUCATION AND TRAINING. **Proceedings [...]**. [*S. l.*], 2023. p. 29–39.

ALJEDAANI, W.; PERUMA, A.; ALJOHANI, A.; ALOTAIBI, M.; MKAOUER, M. W.; OUNI, A.; NEWMAN, C. D.; GHALLAB, A.; LUDI, S. Test smell detection tools: A systematic mapping study. In: 25TH INTERNATIONAL CONFERENCE ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2021. (EASE '21), p. 170–180. ISBN 9781450390538. Disponível em: https://doi.org/10.1145/3463274.3463335. Acesso em: 12 jun. 2025.

ALJOHANI, A.; DO, H. From fine-tuning to output: An empirical investigation of test smells in transformer-based test code generation. In: 39TH ACM/SIGAPP SYMPOSIUM ON APPLIED COMPUTING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2024. (SAC '24), p. 1282–1291. ISBN 9798400702433. Disponível em: https://doi.org/10.1145/3605098.3636058. Acesso em: 12 jun. 2025.

AMPATZOGLOU, A.; BIBI, S.; AVGERIOU, P.; VERBEEK, M.; CHATZIGEORGIOU, A. Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. **Information and Software Technology**, v. 106, p. 201–230, 2019. ISSN 0950-5849. Disponível em: https://www.sciencedirect.com/science/article/pii/S0950584918302106. Acesso em: 12 jun. 2025.

ANQUETIL, N.; DELPLANQUE, J.; DUCASSE, S.; ZAITSEV, O.; FUHRMAN, C.; GUéHéNEUC, Y.-G. What do developers consider magic literals? a smalltalk perspective. **Information and Software Technology**, v. 149, p. 106942, 2022. ISSN 0950-5849. Disponível em: https://www.sciencedirect.com/science/article/pii/S0950584922000908. Acesso em: 12 jun. 2025.

ARDITO, L.; BARBATO, L.; CASTELLUCCIO, M.; COPPOLA, R.; DENIZET, C.; LEDRU, S.; VALSESIA, M. rust-code-analysis: A rust library to analyze and extract maintainability information from source codes. **SoftwareX**, v. 12, p. 100635, 2020. ISSN 2352-7110. Disponível em: https://www.sciencedirect.com/science/article/pii/S2352711020303484. Acesso em: 12 jun. 2025.

ASAITHAMBI, S. P. R.; JARZABEK, S. Towards test case reuse: A study of redundancies in android platform test libraries. In: FAVARO, J.; MORISIO, M. (Ed.). **Proceedings [...]**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 49–64. ISBN 978-3-642-38977-1.

ASAITHAMBI, S. P. R.; JARZABEK, S. Pragmatic approach to test case reuse - a case study in android os biditests library. In: SCHAEFER, I.; STAMELOS, I. (Ed.). **Proceedings [...]**. Cham: Springer International Publishing, 2014. p. 122–138. ISBN 978-3-319-14130-5.

BAI, G. R.; SMITH, J.; STOLEE, K. T. How students unit test: Perceptions, practices, and pitfalls. In: 26TH ACM CONFERENCE ON INNOVATION AND TECHNOLOGY IN

COMPUTER SCIENCE EDUCATION. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2021. (ITiCSE '21), p. 248–254. ISBN 9781450382144. Disponível em: https://doi.org/10.1145/3430665.3456368. Acesso em: 12 jun. 2025.

BAKER, P.; EVANS, D.; GRABOWSKI, J.; NEUKIRCHEN, H.; ZEISS, B. Trex - the refactoring and metrics tool for ttcn-3 test specifications. In: TESTING: ACADEMIC & INDUSTRIAL CONFERENCE - PRACTICE AND RESEARCH TECHNIQUES. **Proceedings [...]**. [*S. l.*], 2006. p. 90–94.

BARNEY, S.; WOHLIN, C. Software product quality: ensuring a common goal. **Trustworthy Software Development Processes**, p. 256–267, 2009.

BARRAK, A.; EGHAN, E. E.; ADAMS, B.; KHOMH, F. Why do builds fail?—a conceptual replication study. **Journal of Systems and Software**, v. 177, p. 110939, 2021. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121221000364. Acesso em: 12 jun. 2025.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; LUCIA, A. D.; BINKLEY, D. Are test smells really harmful? an empirical study. **Empirical Software Engineering**, v. 20, n. 4, p. 1052–1094, Aug 2015. ISSN 1573-7616. Disponível em: https://doi.org/10.1007/s10664-014-9313-0. Acesso em: 12 jun. 2025.

BELL, J.; KAISER, G.; MELSKI, E.; DATTATREYA, M. Efficient dependency detection for safe java test acceleration. In: 10TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2015. (ESEC/FSE 2015), p. 770–781. ISBN 9781450336758. Disponível em: https://doi.org/10.1145/2786805.2786823. Acesso em: 12 jun. 2025.

BELLER, M.; GOUSIOS, G.; PANICHELLA, A.; ZAIDMAN, A. When, how, and why developers (do not) test in their ides. In: 10TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2015. (ESEC/FSE 2015), p. 179–190. ISBN 9781450336758. Disponível em: https://doi.org/10.1145/2786805.2786843. Acesso em: 12 jun. 2025.

BIAGIOLA, M.; STOCCO, A.; MESBAH, A.; RICCA, F.; TONELLA, P. Web test dependency detection. In: 27TH ACM JOINT MEETING ON EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2019. (ESEC/FSE 2019), p. 154–164. ISBN 9781450355728. Disponível em: https://doi.org/10.1145/3338906.3338948. Acesso em: 12 jun. 2025.

BISHT, S. **Robot framework test automation**. [*S. l.*]: Packt Publishing Ltd, 2013.

BLADEL, B. van; DEMEYER, S. A comparative study of test code clones and production code clones. **Journal of Systems and Software**, v. 176, p. 110940, 2021. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121221000376. Acesso em: 12 jun. 2025.

BLESER, J. D.; NUCCI, D. D.; ROOVER, C. D. Assessing diffusion and perception of test smells in scala projects. In: 16TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES. **Proceedings [...]**. [*S. l.*], 2019. p. 457–467.

BLESER, J. D.; NUCCI, D. D.; ROOVER, C. D. Socrates: Scala radar for test smells. In: TENTH ACM SIGPLAN SYMPOSIUM ON SCALA. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2019. (Scala '19), p. 22–26. ISBN 9781450368247. Disponível em: https://doi.org/10.1145/3337932.3338815. Acesso em: 12 jun. 2025.

BODEA, A. Pytest-smell: a smell detection tool for python unit tests. In: 31ST ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2022. (ISSTA 2022), p. 793–796. ISBN 9781450393799. Disponível em: https://doi.org/10.1145/3533767.3543290. Acesso em: 12 jun. 2025.

CAMARA, B.; SILVA, M.; ENDO, A.; VERGILIO, S. On the use of test smells for prediction of flaky tests. In: SIMPÓSIO BRASILEIRO DE TESTES SISTEMÁTICOS E AUTOMATIZADOS DE SOFTWARE, 6., 2021, Joinville, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2021. (SAST '21), p. 46–54. ISBN 9781450385039. Disponível em: https://doi.org/10.1145/3482909.3482916. Acesso em: 12 jun. 2025.

CAMPOS, D.; MARTINS, L.; BEZERRA, C.; MACHADO, I. Investigating developers' contributions to test smell survivability: A study of open-source projects. In: SIMPÓSIO BRASILEIRO DE TESTES SISTEMÁTICOS E AUTOMATIZADOS DE SOFTWARE, 8., 2023, Campo Grande, MS, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2023. (SAST '23), p. 86–95. ISBN 9798400716294. Disponível em: https://doi.org/10.1145/3624032.3624044. Acesso em: 12 jun. 2025.

CAMPOS, D.; MARTINS, L.; MACHADO, I. An empirical study on the influence of developers' experience on software test code quality. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, 21., 2023, Curitiba, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2023. (SBQS '22). ISBN 9781450399999. Disponível em: https://doi.org/10.1145/3571473.3571481. Acesso em: 12 jun. 2025.

CHEN, Z.; EMBURY, S. M.; VIGO, M. Who is afraid of test smells? assessing technical debt from developer actions. In: BONFANTI, S.; GARGANTINI, A.; SALVANESCHI, P. (Ed.). **Proceedings [...]**. Cham: Springer Nature Switzerland, 2023. p. 160–175. ISBN 978-3-031-43240-8.

CHEN, Z.; JIA, C.; CHEN, L. Evaluating test quality of python libraries for iot applications at the network edge. **Wireless Networks**, v. 30, n. 7, p. 6603–6618, Oct 2024. ISSN 1572-8196. Disponível em: https://doi.org/10.1007/s11276-023-03479-2. Acesso em: 12 jun. 2025.

CHICCO, D.; JURMAN, G. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. **BMC Genomics**, v. 21, n. 1, p. 6, Jan 2020. ISSN 1471-2164. Disponível em: https://doi.org/10.1186/s12864-019-6413-7. Acesso em: 12 jun. 2025.

CUMMING, G.; FINCH, S. Inference by eye: Confidence intervals and how to read pictures of data. **American Psychologist**, v. 60, n. 2, p. 170–180, feb 2005. ISSN 0003-066X.

DAKA, E.; CAMPOS, J.; FRASER, G.; DORN, J.; WEIMER, W. Modeling readability to improve unit tests. In: 10TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2015. (ESEC/FSE 2015), p. 107–118. ISBN 9781450336758. Disponível em: https://doi.org/10.1145/2786805.2786838. Acesso em: 12 jun. 2025.

DAMASCENO, H.; BEZERRA, C.; COUTINHO, E.; MACHADO, I. Analyzing test smells refactoring from a developers perspective. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, 21., 2023, Curitiba, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2023. (SBQS '22). ISBN 9781450399999. Disponível em: https://doi.org/10.1145/3571473.3571487. Acesso em: 12 jun. 2025.

DELPLANQUE, J.; DUCASSE, S.; POLITO, G.; BLACK, A. P.; ETIEN, A. Rotten green tests. In: 41ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. **Proceedings [...]**. [*S. l.*], 2019. p. 500–511.

DEURSEN, A.; MOONEN, L. M.; BERGH, A.; KOK, G. **Refactoring test code**. NLD, 2001.

DICICCIO, T. J.; ROMANO, J. P. A review of bootstrap confidence intervals. **Journal of the Royal Statistical Society. Series B (Methodological)**, [Royal Statistical Society, Oxford University Press], v. 50, n. 3, p. 338–354, 1988. ISSN 00359246. Disponível em: http://www.jstor.org/stable/2345699. Acesso em: 12 jun. 2025.

DUCASSE, S.; RIEGER, M.; DEMEYER, S. 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)**. [*S. l.*: *s. n.*], 1999. p. 109–118.

FANG, Z. F.; LAM, P. Identifying test refactoring candidates with assertion fingerprints. In: PRINCIPLES AND PRACTICES OF PROGRAMMING ON THE JAVA PLATFORM. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2015. (PPPJ '15), p. 125–137. ISBN 9781450337120. Disponível em: https://doi.org/10.1145/2807426.2807437. Acesso em: 12 jun. 2025.

FASOLINO, A. R.; TRAMONTANA, P. Test smells learning by a gamification approach. In: 3RD ACM INTERNATIONAL WORKSHOP ON GAMIFICATION IN SOFTWARE DEVELOPMENT, VERIFICATION, AND VALIDATION. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2024. (Gamify 2024), p. 30–33. ISBN 9798400711138. Disponível em: https://doi.org/10.1145/3678869.3685687. Acesso em: 12 jun. 2025.

FAZZINI, M.; CHOI, C.; COPIA, J. M.; LEE, G.; KAKEHI, Y.; GORLA, A.; ORSO, A. Use of test doubles in android testing: An in-depth investigation. In: 44TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. **Proceedings [...]**. [*S. l.*], 2022. p. 2266–2278.

FERNANDES, D.; MACHADO, I.; MACIEL, R. Handling test smells in python: Results from a mixed-method study. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 35., 2021, Joinville, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2021. (SBES '21), p. 84–89. ISBN 9781450390613. Disponível em: https://doi.org/10.1145/3474624.3477066. Acesso em: 12 jun. 2025.

FERNANDES, D.; MACHADO, I.; MACIEL, R. Tempy: Test smell detector for python. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 36., 2022, Virtual Event, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2022. (SBES '22), p. 214–219. ISBN 9781450397353. Disponível em: https://doi.org/10.1145/3555228.3555280. Acesso em: 12 jun. 2025.

FUSHIHARA, Y.; AMAN, H.; AMASAKI, S.; YOKOGAWA, T.; KAWAHARA, M. A trend analysis of test smells in python test code over commit history. In: 49TH EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS. **Proceedings [...]**. [*S. l.*], 2023. p. 310–314.

FUSHIHARA, Y.; AMAN, H.; AMASAKI, S.; YOKOGAWA, T.; KAWAHARA, M. Fault-proneness of python programs tested by smelled test code. In: 50TH EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS. **Proceedings [...]**. [*S. l.*], 2024. p. 373–378.

GALINDO-GUTIERREZ, G.; CARVAJAL, M. N.; BLANCO, A. F.; ANQUETIL, N.; ALCOCER, J. P. S. A manual categorization of new quality issues on automatically-generated tests . In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION. **Proceedings [...]**. Los Alamitos, CA, USA: IEEE Computer Society, 2023. p. 271–281. Disponível em: https://doi.ieeecomputersociety.org/10.1109/ICSME58846.2023.00035. Acesso em: 12 jun. 2025.

GAO, Y.; HU, X.; YANG, X.; XIA, X. Automated unit test refactoring. **Proc. ACM Softw. Eng.**, Association for Computing Machinery, New York, NY, USA, v. 2, n. FSE, jun. 2025. Disponível em: https://doi.org/10.1145/3715750. Acesso em: 12 jun. 2025.

GAROUSI, V.; KUCUK, B.; FELDERER, M. What we know about smells in software test code. **IEEE Software**, v. 36, n. 3, p. 61–73, 2019.

GAROUSI, V.; KüçüK, B. Smells in software test code: A survey of knowledge in industry and academia. **Journal of Systems and Software**, v. 138, p. 52–81, 2018. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121217303060. Acesso em: 12 jun. 2025.

GRABOWSKI, J.; HOGREFE, D.; RéTHY, G.; SCHIEFERDECKER, I.; WILES, A.; WILLCOCK, C. An introduction to the testing and test control notation (ttcn-3). **Computer Networks**, v. 42, n. 3, p. 375–403, 2003. ISSN 1389-1286. ITU-T System Design Languages (SDL). Disponível em: https://www.sciencedirect.com/science/article/pii/S1389128603002494. Acesso em: 12 jun. 2025.

GRANO, G.; PALOMBA, F.; Di Nucci, D.; De Lucia, A.; GALL, H. C. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. **Journal of Systems and Software**, v. 156, p. 312–327, 2019. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121219301487. Acesso em: 12 jun. 2025.

GRANO, G.; SCALABRINO, S.; GALL, H. C.; OLIVETO, R. An empirical investigation on the readability of manual and generated test cases. In: 26TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION. **Proceedings [...]**. [*S. l.*], 2018. p. 348–3483.

GREILER, M.; DEURSEN, A. van; STOREY, M.-A. Automated detection of test fixture strategies and smells. In: SIXTH INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION. **Proceedings [...]**. [*S. l.*], 2013. p. 322–331.

GREILER, M.; DEURSEN, A. van; STOREY, M.-A. Automated detection of test fixture strategies and smells. In: SIXTH INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION. **Proceedings [...]**. [*S. l.*], 2013. p. 322–331.

GREILER, M.; ZAIDMAN, A.; DEURSEN, A. van; STOREY, M.-A. Strategies for avoiding text fixture smells during software evolution. In: 10TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES. **Proceedings [...]**. [*S. l.*], 2013. p. 387–396.

HADJ-KACEM, M.; BOUASSIDA, N. A multi-label classification approach for detecting test smells over java projects. **Journal of King Saud University - Computer and Information Sciences**, v. 34, n. 10, Part A, p. 8692–8701, 2022. ISSN 1319-1578. Disponível em: https://www.sciencedirect.com/science/article/pii/S1319157821002950. Acesso em: 12 jun. 2025.

HAJI, K. E.; BRANDT, C.; ZAIDMAN, A. Using github copilot for test generation in python: An empirical study. In: 5TH ACM/IEEE INTERNATIONAL CONFERENCE ON AUTOMATION OF SOFTWARE TEST. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2024. (AST ’24), p. 45–55. ISBN 9798400705885. Disponível em: https://doi.org/10.1145/3644032.3644443. Acesso em: 12 jun. 2025.

HASAN, M. A.; AHAMMED, T. Understanding the prevalence of test smells in open-source and industrial software: An empirical study on python projects. In: . Chongqing, China: [*S. n.*], 2024. v. 3864, p. 19 – 26. ISSN 16130073. Code quality;Industrial programs;Industrial software;Open source projects;Open-source softwares;Python testing;T-tests;Test code;Test code quality;Test smell;.

HASANAIN, W.; LABICHE, Y.; ELDH, S. An analysis of complex industrial test code using clone analysis. In: INTERNATIONAL CONFERENCE ON SOFTWARE QUALITY, RELIABILITY AND SECURITY. **Proceedings [...]**. [*S. l.*], 2018. p. 482–489.

HERCULANO, W. B. R.; ALVES, E. L. G.; MONGIOVI, M. Generated tests in the context of maintenance tasks: A series of empirical studies. **IEEE Access**, v. 10, p. 121418–121443, 2022.

HUO, C.; CLAUSE, J. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In: 22ND ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2014. (FSE 2014), p. 621–631. ISBN 9781450330565. Disponível em: https://doi.org/10.1145/2635868.2635917. Acesso em: 12 jun. 2025.

JOB, R.; HORA, A. How and why developers implement os-specific tests. **Empirical Software Engineering**, v. 30, n. 1, p. 8, Oct 2024. ISSN 1573-7616. Disponível em: https://doi.org/10.1007/s10664-024-10571-4. Acesso em: 12 jun. 2025.

JORGE, D.; MACHADO, P.; ANDRADE, W. Investigating test smells in javascript test code. In: SIMPÓSIO BRASILEIRO DE TESTES SISTEMÁTICOS E AUTOMATIZADOS DE SOFTWARE, 6., 2021, Joinville, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2021. (SAST ’21), p. 36–45. ISBN 9781450385039. Disponível em: https://doi.org/10.1145/3482909.3482915. Acesso em: 12 jun. 2025.

JUNIOR, N. S.; MARTINS, L.; ROCHA, L.; COSTA, H.; MACHADO, I. How are test smells treated in the wild? a tale of two empirical studies. **Journal of Software Engineering Research and Development**, v. 9, n. 1, p. 9:1–9:16, Sep. 2021. Disponível em: https://journals-sol.sbc.org.br/index.php/jserd/article/view/1802. Acesso em: 12 jun. 2025.

KIM, D. J. An empirical study on the evolution of test smell. In: 42ND INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING: COMPANION PROCEEDINGS. **Proceedings [...]**. [*S. l.*], 2020. p. 149–151.

KIM, D. J.; CHEN, T.-H. P.; YANG, J. The secret life of test smells - an empirical study on test smell evolution and maintenance. **Empirical Software Engineering**, v. 26, n. 5, p. 100, Jul 2021. ISSN 1573-7616. Disponível em: https://doi.org/10.1007/s10664-021-09969-1. Acesso em: 12 jun. 2025.

KIM, D. J.; YANG, B.; YANG, J.; CHEN, T.-H. P. How disabled tests manifest in test maintainability challenges? In: 29TH ACM JOINT MEETING ON EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2021. (ESEC/FSE 2021), p. 1045–1055. ISBN 9781450385626. Disponível em: https://doi.org/10.1145/3468264.3468609. Acesso em: 12 jun. 2025.

KINGSTON, S.; PUN, V. K. I.; STOLZ, V. Automated clone elimination in python tests. In: MARGARIA, T.; STEFFEN, B. (Ed.). **Proceedings [...]**. Cham: Springer Nature Switzerland, 2025. p. 97–114. ISBN 978-3-031-75387-9.

KIRAN, A.; BUTT, W. H.; ANWAR, M. W.; AZAM, F.; MAQBOOL, B. A comprehensive investigation of modern test suite optimization trends, tools and techniques. **IEEE Access**, v. 7, p. 89093–89117, 2019.

KOOCHAKZADEH, N.; GAROUSI, V. A tester-assisted methodology for test redundancy detection. **Adv. Soft. Eng.**, Hindawi Limited, London, GBR, v. 2010, jan 2010. ISSN 1687-8655. Disponível em: https://doi.org/10.1155/2010/932686. Acesso em: 12 jun. 2025.

KUHN, A.; ROMPAEY, B. V.; HAENSENBERGER, L.; NIERSTRASZ, O.; DEMEYER, S.; GAELLI, M.; LEEMPUT, K. V. Jexample: Exploiting dependencies between tests to improve defect localization. In: ABRAHAMSSON, P.; BASKERVILLE, R.; CONBOY, K.; FITZGERALD, B.; MORGAN, L.; WANG, X. (Ed.). **Proceedings [...]**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 73–82. ISBN 978-3-540-68255-4.

LAMBIASE, S.; CUPITO, A.; PECORELLI, F.; LUCIA, A. D.; PALOMBA, F. Just-in-time test smell detection and refactoring: The darts project. In: 28TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION. **Proceedings [...]**. [*S. l.*], 2020. p. 441–445.

LIMA, R.; COSTA, K.; SOUZA, J.; TEIXEIRA, L.; FONSECA, B.; D'AMORIM, M.; RIBEIRO, M.; MIRANDA, B. Do you see any problem? on the developers perceptions in test smells detection. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, 22., 2023, Brasília, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2023. (SBQS '23), p. 21–30. ISBN 9798400707865. Disponível em: https://doi.org/10.1145/3629479.3629485. Acesso em: 12 jun. 2025.

LIU, X.; YU, P. Randoop-tsr: Random-based test generator with test suite reduction. In: 13TH ASIA-PACIFIC SYMPOSIUM ON INTERNETWARE. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2022. (Internetware '22), p. 221–230. ISBN 9781450397803. Disponível em: https://doi.org/10.1145/3545258.3545280. Acesso em: 12 jun. 2025.

LIUTENKO, I.; GOLOSKOKOVA, A.; KURASOV, O.; LUKINOVA, D. Information solutions for evaluating the quality of software tests. **Science and Education a New Dimension**, VII(215), p. 28–31, 2019.

LOPES, G.; aO, D. R.; SOARES, E.; RIBEIRO, M.; AMARAL, G.; GHEYI, R.; MACHADO, I. A road to find them all: Towards an agnostic strategy for test smell detection. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, 23., 2024, Brasília, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2024. (SBQS '24), p. 231–241. ISBN 9798400717772. Disponível em: https://doi.org/10.1145/3701625.3701662. Acesso em: 12 jun. 2025.

LUCAS, K.; GHEYI, R.; SOARES, E.; RIBEIRO, M.; MACHADO, I. Evaluating large language models in detecting test smells. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 38., 2024, Curitiba/PR. **Anais [...]**. Porto Alegre, RS, Brasil: SBC, 2024. p. 672–678. ISSN 2833-0633. Disponível em: https://sol.sbc.org.br/index.php/sbes/article/view/30411. Acesso em: 12 jun. 2025.

MACHADO, I. do C.; MCGREGOR, J. D.; CAVALCANTI, Y. C.; de Almeida, E. S. On strategies for testing software product lines: A systematic literature review. **Information and Software Technology**, v. 56, n. 10, p. 1183–1199, 2014. ISSN 0950-5849. Disponível em: https://www.sciencedirect.com/science/article/pii/S0950584914000834. Acesso em: 12 jun. 2025.

MAIER, F.; FELDERER, M. Detection of test smells with basic language analysis methods and its evaluation. In: INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING. **Proceedings [...]**. [*S. l.*], 2023. p. 897–904.

MARTINS, L.; BEZERRA, C.; COSTA, H.; MACHADO, I. Smart prediction for refactorings in the software test code. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 35., 2021, Joinville, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2021. (SBES '21), p. 115–120. ISBN 9781450390613. Disponível em: https://doi.org/10.1145/3474624.3477070. Acesso em: 12 jun. 2025.

MARTINS, L.; CAMPOS, D.; SANTANA, R.; JUNIOR, J. M.; COSTA, H.; MACHADO, I. Hearing the voice of experts: Unveiling stack exchange communities' knowledge of test smells. In: 16TH INTERNATIONAL CONFERENCE ON COOPERATIVE AND HUMAN ASPECTS OF SOFTWARE ENGINEERING. **Proceedings [...]**. Los Alamitos, CA, USA: IEEE Computer Society, 2023. p. 80–91. Disponível em: https://doi.ieeecomputersociety.org/10.1109/CHASE58964.2023.00017. Acesso em: 12 jun. 2025.

MARTINS, L.; COSTA, H.; MACHADO, I. On the diffusion of test smells and their relationship with test code quality of java projects. **Journal of Software: Evolution and Process**, v. 36, n. 4, p. e2532, 2024. Disponível em: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2532. Acesso em: 12 jun. 2025.

MARTINS, L.; GHALEB, T. A.; COSTA, H.; MACHADO, I. A comprehensive catalog of refactoring strategies to handle test smells in java-based systems. **Software Quality Journal**, v. 32, n. 2, p. 641–679, Jun 2024. ISSN 1573-1367. Disponível em: https://doi.org/10.1007/s11219-024-09663-7. Acesso em: 12 jun. 2025.

MENDEZ, D.; GRAZIOTIN, D.; WAGNER, S.; SEIBOLD, H. Open science in software engineering. In: FELDERER, M.; TRAVASSOS, G. H. (Ed.). **Contemporary Empirical Methods in Software Engineering**. Cham: Springer International Publishing, 2020. p. 477–501. ISBN 978-3-030-32489-6. Disponível em: https://doi.org/10.1007/978-3-030-32489-6_17. Acesso em: 12 jun. 2025.

MESZAROS, G. **xUnit test patterns**: Refactoring test code. [*S. l.*]: Pearson Education, 2007.

MISU, M. R. H.; LI, J.; BHATTIPROLU, A.; LIU, Y.; de Almeida, E. S.; AHMED, I. Test smell: A parasitic energy consumer in software testing. **Information and Software Technology**, v. 181, p. 107671, 2025. ISSN 0950-5849. Disponível em: https://www.sciencedirect.com/science/article/pii/S0950584925000102. Acesso em: 12 jun. 2025.

NEUKIRCHEN, H.; BISANZ, M. Utilising code smells to detect quality problems in ttcn-3 test suites. In: PETRENKO, A.; VEANES, M.; TRETMANS, J.; GRIESKAMP, W. (Ed.). **Proceedings [...]**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 228–243. ISBN 978-3-540-73066-8.

NEUKIRCHEN, H.; ZEISS, B.; GRABOWSKI, J. An approach to quality engineering of ttcn-3 test specifications. **International Journal on Software Tools for Technology Transfer**, v. 10, n. 4, p. 309–326, Aug 2008. ISSN 1433-2787. Disponível em: https://doi.org/10.1007/s10009-008-0075-0. Acesso em: 12 jun. 2025.

OLIVEIRA, J.; MATEUS, L.; VIRGÍNIO, T.; ROCHA, L. Snuts.js: Sniffing nasty unit test smells in javascript. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 38., 2024, Curitiba/PR. **Anais [...]**. Porto Alegre, RS, Brasil: SBC, 2024. p. 720–726. ISSN 2833-0633. Disponível em: https://sol.sbc.org.br/index.php/sbes/article/view/30417. Acesso em: 12 jun. 2025.

PALOMBA, F.; NUCCI, D. D.; PANICHELLA, A.; OLIVETO, R.; LUCIA, A. D. On the diffusion of test smells in automatically generated test code: An empirical study. In: 9TH INTERNATIONAL WORKSHOP ON SEARCH-BASED SOFTWARE TESTING. **Proceedings [...]**. [*S. l.*], 2016. p. 5–14.

PALOMBA, F.; ZAIDMAN, A.; LUCIA, A. D. Automatic test smell detection using information retrieval techniques. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION. **Proceedings [...]**. [*S. l.*], 2018. p. 311–322.

PANICHELLA, A.; PANICHELLA, S.; FRASER, G.; SAWANT, A. A.; HELLENDOORN, V. J. Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION. **Proceedings [...]**. [*S. l.*], 2020. p. 523–533.

PANICHELLA, A.; PANICHELLA, S.; FRASER, G.; SAWANT, A. A.; HELLENDOORN, V. J. Test smells 20 years later: detectability, validity, and reliability. **Empirical Software Engineering**, v. 27, n. 7, p. 170, Sep 2022. ISSN 1573-7616. Disponível em: https://doi.org/10.1007/s10664-022-10207-5. Acesso em: 12 jun. 2025.

PAUL, P. P.; AKANDA, M. T.; ULLAH, M. R.; MONDAL, D.; CHOWDHURY, N. S.; TAWSIF, F. M. xnose: A test smell detector for c#. In: 46TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING: COMPANION PROCEEDINGS. **Proceedings [...]**. Los Alamitos, CA, USA: IEEE Computer Society, 2024. p. 370–371. Disponível em: https://doi.ieeecomputersociety.org/10.1145/3639478.3643116. Acesso em: 12 jun. 2025.

PECORELLI, F.; GRANO, G.; PALOMBA, F.; GALL, H. C.; LUCIA, A. D. Toward granular search-based automatic unit test case generation. **Empirical Software Engineering**, v. 29, n. 4, p. 71, May 2024. ISSN 1573-7616. Disponível em: https://doi.org/10.1007/s10664-024-10451-x. Acesso em: 12 jun. 2025.

PENG, Z.; CHEN, T.-H.; YANG, J. Revisiting test impact analysis in continuous testing from the perspective of code dependencies. **IEEE Transactions on Software Engineering**, v. 48, n. 6, p. 1979–1993, 2022.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. 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. **Proceedings [...]**. USA: IBM Corp., 2019. (CASCON '19), p. 193–202.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. tsdetect: an open source test smells detection tool. In: 28TH ACM JOINT MEETING ON EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2020. (ESEC/FSE 2020), p. 1650–1654. ISBN 9781450370431. Disponível em: https://doi.org/10.1145/3368089.3417921. Acesso em: 12 jun. 2025.

PERUMA, A.; ALOMAR, E. A.; ALJEDAANI, W.; NEWMAN, C. D.; MKAOUER, M. W. Insights from the field: Exploring students' perspectives on bad unit testing practices. In: 2024 INNOVATION AND TECHNOLOGY IN COMPUTER SCIENCE EDUCATION. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2024. (ITiCSE 2024), p. 101–107. ISBN 9798400706004. Disponível em: https://doi.org/10.1145/3649217.3653643. Acesso em: 12 jun. 2025.

PERUMA, A.; NEWMAN, C. D. On the distribution of "simple stupid bugs" in unit test files: An exploratory study. In: 18TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES. **Proceedings [...]**. [*S. l.*], 2021. p. 525–529.

PERUMA, A.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. An exploratory study on the refactoring of unit test files in android applications. In: IEEE/ACM 42ND INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING WORKSHOPS. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2020. (ICSEW'20), p. 350–357. ISBN 9781450379632. Disponível em: https://doi.org/10.1145/3387940.3392189. Acesso em: 12 jun. 2025.

PIZZINI, A.; REINEHR, S.; MALUCELLI, A. Automatic refactoring method to remove eager test smell. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, 21., 2023, Curitiba, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2023. (SBQS '22). ISBN 9781450399999. Disponível em: https://doi.org/10.1145/3571473.3571478. Acesso em: 12 jun. 2025.

PIZZINI, A.; REINEHR, S.; MALUCELLI, A. Sentinel: A process for automatic removing of test smells. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, 22., 2023, Brasília, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2023. (SBQS '23), p. 80–89. ISBN 9798400707865. Disponível em: https://doi.org/10.1145/3629479.3630019. Acesso em: 12 jun. 2025.

PONTILLO, V.; D'ARAGONA, D. A.; PECORELLI, F.; NUCCI, D. D.; FERRUCCI, F.; PALOMBA, F. Machine learning-based test smell detection. **Empirical Software Engineering**, v. 29, n. 2, p. 55, Mar 2024. ISSN 1573-7616. Disponível em: https://doi.org/10.1007/s10664-023-10436-2. Acesso em: 12 jun. 2025.

PONTILLO, V.; MARTINS, L.; MACHADO, I.; PALOMBA, F.; FERRUCCI, F. An empirical investigation into the capabilities of anomaly detection approaches for test smell detection. **Journal of Systems and Software**, v. 222, p. 112320, 2025. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121224003649. Acesso em: 12 jun. 2025.

QIAO, Y.; ROJAS, J. M. What's in a display name? an empirical study on the use of display names in open-source junit tests. In: THIRD ACM/IEEE INTERNATIONAL WORKSHOP ON NL-BASED SOFTWARE ENGINEERING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2024. (NLBSE '24), p. 17–24. ISBN 9798400705762. Disponível em: https://doi.org/10.1145/3643787.3648037. Acesso em: 12 jun. 2025.

QUSEF, A.; ELISH, M. O.; BINKLEY, D. An exploratory study of the relationship between software test smells and fault-proneness. **IEEE Access**, v. 7, p. 139526–139536, 2019.

RIO, A.; ABREU, F. B. e.; MENDES, D. Causal inference of server- and client-side code smells in web apps evolution. **Empirical Software Engineering**, v. 29, n. 5, p. 133, Aug 2024. ISSN 1573-7616. Disponível em: https://doi.org/10.1007/s10664-024-10478-0. Acesso em: 12 jun. 2025.

ROBILLARD, M. P.; NASSIF, M.; SOHAIL, M. Understanding test convention consistency as a dimension of test quality. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 34, n. 1, dez. 2024. ISSN 1049-331X. Disponível em: https://doi.org/10.1145/3672448. Acesso em: 12 jun. 2025.

ROMPAEY, B. V.; BOIS, B. D.; DEMEYER, S. Characterizing the relative significance of a test smell. In: 22ND IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE. **Proceedings [...]**. [*S. l.*], 2006. p. 391–400.

ROMPAEY, B. V.; BOIS, B. D.; DEMEYER, S.; RIEGER, M. On the detection of test smells: A metrics-based approach for general fixture and eager test. **IEEE Transactions on Software Engineering**, v. 33, n. 12, p. 800–817, 2007.

ROSLAN, M. F.; ROJAS, J. M.; MCMINN, P. Private-keep out? understanding how developers account for code visibility in unit testing. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION. **Proceedings [...]**. [*S. l.*], 2024. p. 312–324.

ROSLAN, M. F.; ROJAS, J. M.; MCMINN, P. Viscount: A direct method call coverage tool for java. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION. **Proceedings [...]**. [*S. l.*], 2024. p. 908–912.

RWEMALIKA, R.; HABCHI, S.; PAPADAKIS, M.; TRAON, Y. L.; BRASSEUR, M.-C. Smells in system user interactive tests. **Empirical Software Engineering**, v. 28, n. 1, p. 20, Dec 2022. ISSN 1573-7616. Disponível em: https://doi.org/10.1007/s10664-022-10251-1. Acesso em: 12 jun. 2025.

RWEMALIKA, R.; KINTIS, M.; PAPADAKIS, M.; TRAON, Y. L.; LORRACH, P. Ukwikora: continuous inspection for keyword-driven testing. In: 28TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2019. (ISSTA 2019), p. 402–405. ISBN 9781450362245. Disponível em: https://doi.org/10.1145/3293882.3339003. Acesso em: 12 jun. 2025.

SANTANA JUNIOR, E. G.; SANTOS JUNIOR, J. P.; ALMEIDA, E. P.; AHMED, I.; SILVEIRA NETO, P. A. M.; ALMEIDA, E. S. de. **Evaluating LLMs Effectiveness in Detecting and Correcting Test Smells**: An empirical study. 2025. Disponível em: https://arxiv.org/abs/2506.07594. Acesso em: 12 jun. 2025.

SANTANA, R.; FERNANDES, D.; CAMPOS, D.; SOARES, L.; MACIEL, R.; MACHADO, I. Understanding practitioners' strategies to handle test smells: a multi-method study. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 35., 2021, Joinville, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2021. (SBES '21), p. 49–53. ISBN 9781450390613. Disponível em: https://doi.org/10.1145/3474624.3474639. Acesso em: 12 jun. 2025.

SANTANA, R.; MARTINS, L.; ROCHA, L.; VIRGÍNIO, T.; CRUZ, A.; COSTA, H.; MACHADO, I. Raide: a tool for assertion roulette and duplicate assert identification and refactoring. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 34., 2020, Virtual Event, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2020. (SBES '20), p. 374–379. ISBN 9781450387538. Disponível em: https://doi.org/10.1145/3422392.3422510. Acesso em: 12 jun. 2025.

SANTANA, R.; MARTINS, L.; VIRGÍNIO, T.; SOARES, L.; COSTA, H.; MACHADO, I. Refactoring assertion roulette and duplicate assert test smells: a controlled experiment. In: CONGRESSO IBERO-AMERICANO EM ENGENHARIA DE SOFTWARE, 25., 2022, Córdoba. **Anais [...]**. Porto Alegre, RS, Brasil: SBC, 2022. p. 263–277. Disponível em: https://sol.sbc.org.br/index.php/cibse/article/view/20977. Acesso em: 12 jun. 2025.

SANTANA, R.; MARTINS, L.; VIRGÍNIO, T.; ROCHA, L.; COSTA, H.; MACHADO, I. An empirical evaluation of raide: A semi-automated approach for test smells detection and refactoring. **Sci. Comput. Program.**, Elsevier North-Holland, Inc., USA, v. 231, n. C, jan. 2024. ISSN 0167-6423. Disponível em: https://doi.org/10.1016/j.scico.2023.103013. Acesso em: 12 jun. 2025.

SANTANA, R.; MARTINS, L.; VIRGíNIO, T.; SOARES, L.; COSTA, H.; MACHADO, I. Refactoring assertion roulette and duplicate assert test smells: a controlled experiment. In: CONGRESSO IBERO-AMERICANO EM ENGENHARIA DE SOFTWARE, 25., 2022, Córdoba. **Anais [...]**. Porto Alegre, RS, Brasil: SBC, 2022. p. 263–277. ISSN 0000-0000. Disponível em: https://sol.sbc.org.br/index.php/cibse/article/view/20977. Acesso em: 12 jun. 2025.

SATTER, A.; AMI, A. S.; SAKIB, K. A static code search technique to identify dead fields by analyzing usage of setup fields and field dependency in test code. In: . Moscow, Russia: [*S. n.*], 2016. v. 1625, p. 60 – 71. ISSN 16130073. Code search;Dead field;Setup fields;Test code;Test fixture;.

SATTER, A.; NAHAR, N.; SAKIB, K. Automatically identifying dead fields in test code by resolving method call and field dependency. In: QUASOQ AT APSEC. **Proceedings [...]**. 2017. Disponível em: https://api.semanticscholar.org/CorpusID:38998320. Acesso em: 12 jun. 2025.

SCHIEWE, M.; CURTIS, J.; BUSHONG, V.; CERNY, T. Advancing static code analysis with language-agnostic component identification. **IEEE Access**, v. 10, p. 30743–30761, 2022.

SCHVARCBACHER, M.; SPADINI, D.; BRUNTINK, M.; OPRESCU, A. Investigating developer perception on test smells using better code hub - work in progress -. In: . Bolzano, Italy: [*S. n.*], 2019. v. 2510. ISSN 16130073. Code changes;Commercial projects;Detection accuracy;Detection tools;Monitoring code;Online environments;Open sources;Work in progress;.

SCHäFER, M.; NADI, S.; EGHBALI, A.; TIP, F. An empirical evaluation of using large language models for automated unit test generation. **IEEE Transactions on Software Engineering**, v. 50, n. 1, p. 85–105, 2024.

SHARMA, T.; GEORGIOU, S.; KECHAGIA, M.; GHALEB, T. A.; SARRO, F. Investigating developers' perception on software testability and its effects. **Empirical Software Engineering**, v. 28, n. 5, p. 120, Sep 2023. ISSN 1573-7616. Disponível em: https://doi.org/10.1007/s10664-023-10373-0. Acesso em: 12 jun. 2025.

SIDDIQ, M. L.; SANTOS, J. C. D. S.; TANVIR, R. H.; ULFAT, N.; RIFAT, F. A.; LOPES, V. C. Using large language models to generate junit tests: An empirical study. In: 28TH INTERNATIONAL CONFERENCE ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2024. (EASE '24), p. 313–322. ISBN 9798400717017. Disponível em: https://doi.org/10.1145/3661167.3661216. Acesso em: 12 jun. 2025.

SILVA, L. P. d.; VILAIN, P. Lccss: A similarity metric for identifying similar test code. In: SIMPÓSIO BRASILEIRO DE COMPONENTES, ARQUITETURAS E REUTILIZAcÃO DE SOFTWARE, 14., 2020, Natal, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2020. (SBCARS '20), p. 91–100. ISBN 9781450387545. Disponível em: https://doi.org/10.1145/3425269.3425283. Acesso em: 12 jun. 2025.

SILVA, P.; BEZERRA, C.; MACHADO, I. Toward a language-agnostic approach to detect test smells. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 38., 2024, Curitiba/PR. **Anais [...]**. Porto Alegre, RS, Brasil: SBC, 2024. p. 686–692. ISSN 0000-0000. Disponível em: https://sol.sbc.org.br/index.php/sbes/article/view/30413. Acesso em: 12 jun. 2025.

SILVA, P.; BEZERRA, C.; MACHADO, I.; RIBEIRO, M. Aromadr: A language-independent tool for detecting test smells. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 39., 2025, Brasília, Brazil. **Anais [...]**. Porto Alegre, RS, Brasil: SBC, 2025. p. 914–920. ISSN 2833-0633. Disponível em: https://sol.sbc.org.br/index.php/sbes/article/view/37078. Acesso em: 12 jun. 2025.

SOARES, E.; III, M. A.; ROMÃO, D.; RIBEIRO, M. **The Open Catalog of Test Smells**. 2023. Disponível em: https://test-smell-catalog.readthedocs.io/en/latest/index.html. Acesso em: 12 jun. 2025.

SOARES, E.; RIBEIRO, M.; AMARAL, G.; GHEYI, R.; FERNANDES, L.; GARCIA, A.; FONSECA, B.; SANTOS, A. Refactoring test smells: A perspective from open-source developers. In: SIMPÓSIO BRASILEIRO DE TESTES SISTEMÁTICOS E AUTOMATIZADOS DE SOFTWARE, 5., 2020, Natal, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2020. (SAST '20), p. 50–59. ISBN 9781450387552. Disponível em: https://doi.org/10.1145/3425174.3425212. Acesso em: 12 jun. 2025.

SOARES, E.; RIBEIRO, M.; GHEYI, R.; AMARAL, G.; SANTOS, A. Refactoring test smells with junit 5: Why should developers keep up-to-date? **IEEE Transactions on Software Engineering**, v. 49, n. 3, p. 1152–1170, 2023.

SPADINI, D.; SCHVARCBACHER, M.; OPRESCU, A.-M.; BRUNTINK, M.; BACCHELLI, A. Investigating severity thresholds for test smells. In: 17TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2020. (MSR '20), p. 311–321. ISBN 9781450375177. Disponível em: https://doi.org/10.1145/3379597.3387453. Acesso em: 12 jun. 2025.

STEENHOEK, B.; TUFANO, M.; SUNDARESAN, N.; SVYATKOVSKIY, A. Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation . In: INTERNATIONAL WORKSHOP ON DEEP LEARNING FOR TESTING AND TESTING FOR DEEP LEARNING. **Proceedings [...]**. Los Alamitos, CA, USA: IEEE Computer Society, 2025. p. 37–44. Disponível em: https://doi.ieeecomputersociety.org/10.1109/DeepTest66595.2025.00011. Acesso em: 12 jun. 2025.

STEFANO, M. D.; PECORELLI, F.; NUCCI, D. D.; LUCIA, A. D. A preliminary evaluation on the relationship among architectural and test smells. In: 22ND INTERNATIONAL WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION. **Proceedings [...]**. [*S. l.*], 2022. p. 66–70.

STRAUBINGER, P.; FULCINI, T.; GARACCIONE, G.; ARDITO, L.; FRASER, G. Gamifying testing in intellij: A replicability study. **Proc. ACM Softw. Eng.**, Association for Computing Machinery, New York, NY, USA, v. 2, n. ISSTA, jun. 2025. Disponível em: https://doi.org/10.1145/3728983. Acesso em: 12 jun. 2025.

STRAUBINGER, P.; GRELLER, T.; FRASER, G. Sojourner under sabotage: A serious testing and debugging game. In: 33RD ACM INTERNATIONAL CONFERENCE ON THE FOUNDATIONS OF SOFTWARE ENGINEERING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2025. (FSE Companion '25), p. 738–748. ISBN 9798400712760. Disponível em: https://doi.org/10.1145/3696630.3727231. Acesso em: 12 jun. 2025.

TANIGUCHI, M.; MATSUMOTO, S.; KUSUMOTO, S. Jtdog: a gradle plugin for dynamic test smell detection. In: 36TH INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING. **Proceedings [...]**. [*S. l.*], 2021. p. 1271–1275.

TOLL, D.; OLSSON, T. Why is unit-testing in computer games difficult? In: 16TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING. **Proceedings [...]**. [*S. l.*], 2012. p. 373–378.

TRAN, H. K. V.; ALI, N. bin; UNTERKALMSTEINER, M.; BöRSTLER, J. A proposal and assessment of an improved heuristic for the eager test smell detection. **Journal of Systems and Software**, v. 226, p. 112438, 2025. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121225001062. Acesso em: 12 jun. 2025.

TRAN, H. K. V.; UNTERKALMSTEINER, M.; BöRSTLER, J.; ALI, N. bin. Assessing test artifact quality—a tertiary study. **Information and Software Technology**, v. 139, p. 106620, 2021. ISSN 0950-5849. Disponível em: https://www.sciencedirect.com/science/article/pii/S0950584921000938. Acesso em: 12 jun. 2025.

TUFANO, M.; PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D.; POSHYVANYK, D. An empirical investigation into the nature of test smells. In: 31ST

INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING. **Proceedings [...]**. [*S. l.*], 2016. p. 4–15.

VAHABZADEH, A.; STOCCO, A.; MESBAH, A. Fine-grained test minimization. In: 40TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2018. (ICSE '18), p. 210–221. ISBN 9781450356381. Disponível em: https://doi.org/10.1145/3180155.3180203. Acesso em: 12 jun. 2025.

VELOSO, V.; HORA, A. Characterizing high-quality test methods: a first empirical study. In: 19TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2022. (MSR '22), p. 265–269. ISBN 9781450393034. Disponível em: https://doi.org/10.1145/3524842.3529092. Acesso em: 12 jun. 2025.

VIRGÍNIO, T.; BASTOS, L.; BEZERRA, C.; RIBEIRO, M.; MACHADO, I. How aware are we of test smells in quantum software systems? a preliminary empirical evaluation. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, 23., 2024, Brasília, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2024. (SBQS '24), p. 383–393. ISBN 9798400717772. Disponível em: https://doi.org/10.1145/3701625.3701676. Acesso em: 12 jun. 2025.

VIRGÍNIO, T.; MARTINS, L.; ROCHA, L.; SANTANA, R.; CRUZ, A.; COSTA, H.; MACHADO, I. Jnose: Java test smell detector. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 34., 2020, Virtual Event, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2020. (SBES '20), p. 564–569. ISBN 9781450387538. Disponível em: https://doi.org/10.1145/3422392.3422499. Acesso em: 12 jun. 2025.

VIRGÍNIO, T.; MARTINS, L. A.; SOARES, L. R.; SANTANA, R.; COSTA, H.; MACHADO, I. An empirical study of automatically-generated tests from the perspective of test smells. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 34., 2020, Virtual Event, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2020. (SBES '20), p. 92–96. ISBN 9781450387538. Disponível em: https://doi.org/10.1145/3422392.3422412. Acesso em: 12 jun. 2025.

VIRGÍNIO, T.; SANTANA, R.; MARTINS, L. A.; SOARES, L. R.; COSTA, H.; MACHADO, I. On the influence of test smells on test coverage. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 33., 2019, Salvador, Brazil. **Anais [...]**. New York, NY, USA: Association for Computing Machinery, 2019. (SBES '19), p. 467–471. ISBN 9781450376518. Disponível em: https://doi.org/10.1145/3350768.3350775. Acesso em: 12 jun. 2025.

WANG, H.; YU, S.; CHEN, C.; TURHAN, B.; ZHU, X. Beyond accuracy: An empirical study on unit testing in open-source deep learning projects. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 33, n. 4, abr. 2024. ISSN 1049-331X. Disponível em: https://doi.org/10.1145/3638245. Acesso em: 12 jun. 2025.

WANG, J.; ZHANG, W.; WANG, W.; ZHAO, R.; SHANG, Y. Predicting the Root Cause of Flaky Tests Based on Test Smells . In: 22ND INTERNATIONAL CONFERENCE ON SOFTWARE AND SYSTEMS REUSE. **Proceedings [...]**. Los Alamitos, CA, USA: IEEE Computer Society, 2025. p. 84–94. Disponível em: https://doi.ieeecomputersociety.org/10.1109/ICSR66718.2025.00015. Acesso em: 12 jun. 2025.

WANG, T.; GOLUBEV, Y.; SMIRNOV, O.; LI, J.; BRYKSIN, T.; AHMED, I. Pynose: A test smell detector for python. In: 36TH INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING. **Proceedings [...]**. Los Alamitos, CA, USA: IEEE Computer Society, 2021. p. 593–605. Disponível em: https://doi.ieeecomputersociety.org/10.1109/ASE51524.2021.9678615. Acesso em: 12 jun. 2025.

WEI, C.; XIAO, L.; YU, T.; WONG, S.; CLUNE, A. How do developers structure unit test cases? an empirical analysis of the aaa pattern in open source projects. **IEEE Transactions on Software Engineering**, v. 51, n. 4, p. 1007–1038, 2025.

WHOLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. Experimentation in software engineering: an introduction. **Massachusetts: Kluwer Academic Publishers**, 2000.

WU, H.; YIN, R.; GAO, J.; HUANG, Z.; HUANG, H. To what extent can code quality be improved by eliminating test smells? In: INTERNATIONAL CONFERENCE ON CODE QUALITY. **Proceedings [...]**. [*S. l.*], 2022. p. 19–26.

YANG, Y.; HU, X.; XIA, X.; YANG, X. The lost world: Characterizing and detecting undiscovered test smells. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 33, n. 3, mar. 2024. ISSN 1049-331X. Disponível em: https://doi.org/10.1145/3631973. Acesso em: 12 jun. 2025.

ZHANG, S.; JALALI, D.; WUTTKE, J.; MUsLU, K.; LAM, W.; ERNST, M. D.; NOTKIN, D. Empirically revisiting the test independence assumption. In: 2014 INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS. **Proceedings [...]**. New York, NY, USA: Association for Computing Machinery, 2014. (ISSTA 2014), p. 385–396. ISBN 9781450326452. Disponível em: https://doi.org/10.1145/2610384.2610404. Acesso em: 12 jun. 2025.

APPENDIX A – SELECTED PAPERS

Table 22 – Selected Papers

ID	Title
S1	The Lost World: Characterizing and Detecting Undiscovered Test Smells (Yang *et al.*, 2024)
S2	Analyzing Test Smells Refactoring from a Developers Perspective (Damasceno *et al.*, 2023)
S3	TEMPY: Test Smell Detector for Python (Fernandes *et al.*, 2022)
S4	Investigating Test Smells in JavaScript Test Code (Jorge *et al.*, 2021)
S5	Refactoring Test Smells: A Perspective from Open-Source Developers (Soares *et al.*, 2020)
S6	On the use of test smells for prediction of flaky tests (Camara *et al.*, 2021)
S7	Understanding practitioners' strategies to handle test smells: a multi-method study (Santana *et al.*, 2021)
S8	Handling Test Smells in Python: Results from a Mixed-Method Study (Fernandes *et al.*, 2021)
S9	Investigating Developers' Contributions to Test Smell Survivability: A Study of Open-Source Projects (Campos *et al.*, 2023a)
S10	On the distribution of test smells in open source Android applications: an exploratory study (Peruma *et al.*, 2019)
S11	On the influence of Test Smells on Test Coverage (Virgínio *et al.*, 2019)
S12	Do the Test Smells Assertion Roulette and Eager Test Impact Students' Troubleshooting and Debugging Capabilities? (Aljedaani *et al.*, 2023)
S13	SoCRATES: Scala radar for test smells (Bleser *et al.*, 2019b)
S14	PyTest-Smell: a smell detection tool for Python unit tests (Bodea, 2022)
S15	How Students Unit Test: Perceptions, Practices, and Pitfalls (Bai *et al.*, 2021)
S16	Characterizing high-quality test methods: a first empirical study (Veloso; Hora, 2022)
S17	Identifying Test Refactoring Candidates with Assertion Fingerprints (Fang; Lam, 2015)
S18	LCCSS: A Similarity Metric for Identifying Similar Test Code (Silva; Vilain, 2020)
S19	Fine-grained test minimization (Vahabzadeh *et al.*, 2018)
S20	An empirical investigation on the readability of manual and generated test cases (Grano *et al.*, 2018)
S21	Web test dependency detection (Biagiola *et al.*, 2019)
S22	Modeling readability to improve unit tests (Daka *et al.*, 2015)
S23	An empirical evaluation of RAIDE: A semi-automated approach for test smells detection and refactoring (Santana *et al.*, 2024)
S24	On the diffusion of test smells and their relationship with test code quality of Java projects (Martins *et al.*, 2024a)
S25	Automatic Refactoring Method to Remove Eager Test Smell (Pizzini *et al.*, 2023a)
S27	Refactoring Assertion Roulette and Duplicate Assert test smells: a controlled experiment (Santana *et al.*, 2022)
S28	Sentinel: A process for automatic removing of Test Smells (Pizzini *et al.*, 2023b)
S29	Hearing the voice of experts: Unveiling Stack Exchange communities' knowledge of test smells (Martins *et al.*, 2023)
S30	Do you see any problem? On the Developers Perceptions in Test Smells Detection (Lima *et al.*, 2023)
S31	PYNOSE: A test smell detector for python (Wang *et al.*, 2021)
S32	Machine Learning-Based Test Smell Detection (Pontillo *et al.*, 2024)
S33	JTDog: a Gradle Plugin for Detecting Dynamic Test Smells (Taniguchi *et al.*, 2021)
S34	Investigating developer perception on test smells using better code hub - Work in progress - (Schvarcbacher *et al.*, 2019)
S35	How and why developers implement OS-specific tests (Job; Hora, 2024)
S36	An Empirical Study on the Evolution of Test Smell (Kim, 2020)
S37	RAIDE: A tool for Assertion Roulette and Duplicate Assert identification and refactoring (Santana *et al.*, 2020)

ID	Title
S38	An Empirical Study of Using Large Language Models for Unit Test Generation (Schäfer *et al.*, 2024)
S39	An Exploratory Study on the Refactoring of Unit Test Files in Android Applications (Peruma *et al.*, 2020)
S40	Automatic Generation of Smell-free Unit Tests (Afonso; Campos, 2023)
S41	A manual categorization of new quality issues on automatically-generated tests (Galindo-Gutierrez *et al.*, 2023)
S42	On the detection of test smells: A metrics-based approach for general fixture and eager test (Rompaey *et al.*, 2007)
S43	How disabled tests manifest in test maintainability challenges? (Kim *et al.*, 2021b)
S44	To What Extent Can Code Quality be Improved by Eliminating Test Smells? (Wu *et al.*, 2022)
S45	Just-In-Time Test Smell Detection and Refactoring: The DARTS Project (Lambiase *et al.*, 2020)
S46	A Trend Analysis of Test Smells in Python Test Code Over Commit History (Fushihara *et al.*, 2023)
S47	A preliminary evaluation on the relationship among architectural and test smells (Stefano *et al.*, 2022)
S48	Assessing Diffusion and Perception of Test Smells in Scala Projects (Bleser *et al.*, 2019a)
S49	On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study (Palomba *et al.*, 2016)
S50	Automatic Test Smell Detection Using Information Retrieval Techniques (Palomba *et al.*, 2018)
S51	Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities (Panichella *et al.*, 2020)
S52	An Exploratory Study of the Relationship Between Software Test Smells and Fault-Proneness (Qusef *et al.*, 2019)
S53	An Analysis of Complex Industrial Test Code Using Clone Analysis (Hasanain *et al.*, 2018)
S54	Automated Detection of Test Fixture Strategies and Smells (Greiler *et al.*, 2013b)
S55	Generated Tests in the Context of Maintenance Tasks: A Series of Empirical Studies (Herculano *et al.*, 2022)
S56	Revisiting Test Impact Analysis in Continuous Testing From the Perspective of Code Dependencies (Peng *et al.*, 2022)
S57	Why is Unit-testing in Computer Games Difficult? (Toll; Olsson, 2012)
S58	Characterizing the Relative Significance of a Test Smell (Rompaey *et al.*, 2006)
S59	On the Distribution of “Simple Stupid Bugs” in Unit Test Files: An Exploratory Study (Peruma; Newman, 2021)
S60	Scented since the beginning: On the diffuseness of test smells in automatically generated test code (Grano *et al.*, 2019)
S61	Why do builds fail?—A conceptual replication study (Barrak *et al.*, 2021)
S62	A multi-label classification approach for detecting test smells over java projects (Hadj-Kacem; Bouassida, 2022)
S63	A comparative study of test code clones and production code clones (Bladel; Demeyer, 2021)
S64	What do developers consider magic literals? A smalltalk perspective (Anquetil *et al.*, 2022)
S65	The secret life of test smells - an empirical study on test smell evolution and maintenance (Kim *et al.*, 2021a)
S66	Use of Test Doubles in Android Testing: An In-Depth Investigation (Fazzini *et al.*, 2022)
S67	Investigating Severity Thresholds for Test Smells (Spadini *et al.*, 2020)
S68	Strategies for avoiding text fixture smells during software evolution (Greiler *et al.*, 2013c)
S69	Randoop-TSR: Random-based Test Generator with Test Suite Reduction (Liu; Yu, 2022)
S70	JNose: Java Test Smell Detector (Virgínio *et al.*, 2020a)
S71	An empirical study of automatically-generated tests from the perspective of test smells (Virgínio *et al.*, 2020b)
S72	Automatically identifying dead fields in test code by resolving method call and field dependency (Satter *et al.*, 2017)
S73	An empirical investigation into the nature of test smells (Tufano *et al.*, 2016)
S74	Smart prediction for refactorings in the software test code (Martins *et al.*, 2021)

ID	Title
S75	A comprehensive catalog of refactoring strategies to handle test smells in Java-based systems (Martins *et al.*, 2024b)
S76	An empirical study on the influence of developers' experience on software test code quality (Campos *et al.*, 2023b)
S77	Detection of test smells with basic language analysis methods and its evaluation (Maier; Felderer, 2023)
S78	Ukwikora: Continuous inspection for keyword-driven testing (Rwemalika *et al.*, 2019)
S79	TsDetect: An open source test smells detection tool (Peruma *et al.*, 2020)
S80	A static code search technique to identify dead fields by analyzing usage of setup fields and field dependency in test code (Satter *et al.*, 2016)
S81	Test smells 20 years later: detectability, validity, and reliability (Panichella *et al.*, 2022)
S82	Who Is Afraid of Test Smells? Assessing Technical Debt from Developer Actions (Chen *et al.*, 2023)
S83	Smells in system user interactive tests (Rwemalika *et al.*, 2022)
S84	Evaluating test quality of Python libraries for IoT applications at the network edge (Chen *et al.*, 2024)
S85	Investigating developers' perception on software testability and its effects (Sharma *et al.*, 2023)
S86	Are test smells really harmful? An empirical study (Bavota *et al.*, 2015)
S87	Pragmatic Approach to Test Case Reuse - A Case Study in Android OS BiDiTests Library (Asaithambi; Jarzabek, 2014)
S88	Towards Test Case Reuse: A Study of Redundancies in Android Platform Test Libraries (Asaithambi; Jarzabek, 2013)
S89	An approach to quality engineering of TTCN-3 test specifications (Neukirchen *et al.*, 2008)
S90	JExample: Exploiting Dependencies between Tests to Improve Defect Localization (Kuhn *et al.*, 2008)
S91	Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites (Neukirchen; Bisanz, 2007)
S92	Using Large Language Models to Generate JUnit Tests: An Empirical Study (Siddiq *et al.*, 2024)
S93	Understanding Test Convention Consistency as a Dimension of Test Quality (Robillard *et al.*, 2024)
S94	Beyond Accuracy: An Empirical Study on Unit Testing in Open-source Deep Learning Projects (Wang *et al.*, 2024)
S95	Using GitHub Copilot for Test Generation in Python: An Empirical Study (Haji *et al.*, 2024)
S96	From Fine-tuning to Output: An Empirical Investigation of Test Smells in Transformer-Based Test Code Generation (Aljohani; Do, 2024)
S97	Test Smells Learning by a Gamification Approach (Fasolino; Tramontana, 2024)
S98	Insights from the Field: Exploring Students' Perspectives on Bad Unit Testing Practices (Peruma *et al.*, 2024)
S99	XNose: A Test Smell Detector for C# (Paul *et al.*, 2024)
S100	What's in a Display Name? An Empirical Study on the Use of Display Names in Open-Source JUnit Tests (Qiao; Rojas, 2024)
S101	Toward granular search-based automatic unit test case generation (Pecorelli *et al.*, 2024)
S102	A Road to Find Them All: Towards an Agnostic Strategy for Test Smell Detection (Lopes *et al.*, 2024)
S103	How Aware Are We of Test Smells in Quantum Software Systems? A Preliminary Empirical Evaluation (Virgínio *et al.*, 2024)
S104	Automated Unit Test Refactoring (Gao *et al.*, 2025)
S105	Gamifying Testing in IntelliJ: A Replicability Study (Straubinger *et al.*, 2025a)
S106	Sojourner under Sabotage: A Serious Testing and Debugging Game (Straubinger *et al.*, 2025b)
S107	Test smell: A parasitic energy consumer in software testing (Misu *et al.*, 2025)
S108	Predicting the Root Cause of Flaky Tests Based on Test Smells (Wang *et al.*, 2025)

ID	Title
S109	Understanding the Prevalence of Test Smells in Open-source and Industrial Software: An Empirical Study on Python Projects (Hasan; Ahammed, 2024)
S110	Fault-Proneness of Python Programs Tested By Smelled Test Code (Fushihara *et al.*, 2024)
S111	Private-Keep Out? Understanding How Developers Account for Code Visibility in Unit Testing (Roslan *et al.*, 2024a)
S112	Viscount: A Direct Method Call Coverage Tool for Java (Roslan *et al.*, 2024b)
S113	Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation (Steenhoek *et al.*, 2025)
S114	How Do Developers Structure Unit Test Cases? An Empirical Analysis of the AAA Pattern in Open Source Projects (Wei *et al.*, 2025)
S115	A proposal and assessment of an improved heuristic for the Eager Test smell detection (Tran *et al.*, 2025)
S116	An empirical investigation into the capabilities of anomaly detection approaches for test smell detection (Pontillo *et al.*, 2025)
S117	Automated Clone Elimination in Python Tests (Kingston *et al.*, 2025)

APPENDIX B – TEST SMELLS THAT APPEAR IN ONLY ONE LANGUAGE OR FRAMEWORK

Table 23 – Test Smells That Appear In Only One Language Or Framework

Language/Framework	Test Smell
C++	*Fragile Test*
Java	*Assertion In Loop*
Java	*Dependent Assert*
Java	*Fire And Forget*
Java	*Multithreading*
Java	*Non-Deterministic Data Test*
Java	*Ordered Hypothesis*
Java	*Parameter Order Wrong*
Java	*Poor Test Organization*
Java	*Return In Test*
Java	*Time Acquisition*
Java/JUnit	*Ambiguous Display Name*
Java/JUnit	*Anal Probe*
Java/JUnit	*Annotation Repetition*
Java/JUnit	*Anonymous Test*
Java/JUnit	*Arrange&Quit*
Java/JUnit	*Assert Pre-Condition*
Java/JUnit	*Asserting Constants*
Java/JUnit	*Asserting Object Initialization Multiple Times*
Java/JUnit	*Assertions With Not Related Parent Class Method*
Java/JUnit	*Chain Gang*
Java/JUnit	*Code Organization*
Java/JUnit	*Coupling Between Test Methods*
Java/JUnit	*Coupling With Production Code*
Java/JUnit	*Dependencies On Test Containers*
Java/JUnit	*Doppelgänger*
Java/JUnit	*Duplicate Display Names*
Java/JUnit	*Duplicated Setup*
Java/JUnit	*Exceptions Due To External Dependencies*
Java/JUnit	*Exceptions Due To Incomplete Setup*
Java/JUnit	*Exceptions Due To Null Arguments*
Java/JUnit	*For Testers Only*
Java/JUnit	*Friction*
Java/JUnit	*God Test Class*
Java/JUnit	*Happy Path Only*
Java/JUnit	*Hard To Co-Evolve*
Java/JUnit	*Hardcoded Dependency*
Java/JUnit	*Hidden Dependency*

Language/Framework	Test Smell
Java/JUnit	*Inconsistent Display Name*
Java/JUnit	*Inconsistent Style*
Java/JUnit	*Interacting Test Suites*
Java/JUnit	*Irrelevant Information*
Java/JUnit	*Is Mockito Working Fine*
Java/JUnit	*Lack Comments*
Java/JUnit	*Likely Ineffective Object Comparison*
Java/JUnit	*Messy Tests*
Java/JUnit	*Missing Assert*
Java/JUnit	*Missing Display Name*
Java/JUnit	*Mock Call Verification*
Java/JUnit	*Mock Makes The Test Always Passes*
Java/JUnit	*Mock Naming*
Java/JUnit	*Multilingual Display Name*
Java/JUnit	*Multiple Aaa*
Java/JUnit	*Multiple Acts*
Java/JUnit	*Naming Convention Violation*
Java/JUnit	*No Assertions*
Java/JUnit	*Not Asserted Return Values*
Java/JUnit	*Not Asserted Side Effects*
Java/JUnit	*Obscure Assert*
Java/JUnit	*Obscure Inline Setup*
Java/JUnit	*Obsolete Tests*
Java/JUnit	*Overreferencing*
Java/JUnit	*Redundant Display Name*
Java/JUnit	*Redundant Not Null Assertion*
Java/JUnit	*Redundant Test*
Java/JUnit	*Refused Bequest*
Java/JUnit	*Replicating Disadvantages Of The Code*
Java/JUnit	*Resource Leakage*
Java/JUnit	*Resource Optimist*
Java/JUnit	*Second Class Citizens*
Java/JUnit	*State Leak*
Java/JUnit	*Suppressed Exception*
Java/JUnit	*Test Logic In Production Code*
Java/JUnit	*Test Run War*
Java/JUnit	*Testing Only Field Accessors*
Java/JUnit	*Testing The Same Exception Scenario*
Java/JUnit	*Testing The Same Void Method*
Java/JUnit	*The Butterfly*
Java/JUnit	*The Conjoined Twins*
Java/JUnit	*The Dead Tree*
Java/JUnit	*The Environmental Vandal*
Java/JUnit	*The Forty Foot Pole Test*

Language/Framework	Test Smell
Java/JUnit	*The Giant*
Java/JUnit	*The Sleeper*
Java/JUnit	*The Test With No Name*
Java/JUnit	*Tooling Details In Test Case*
Java/JUnit	*Ugly Mirror*
Java/JUnit	*Unclear Display Name*
Java/JUnit	*Unecessary Test*
Java/JUnit	*Ungrammatical Display Name*
Java/JUnit	*Unrelated Assertions*
Java/JUnit	*Unused Inputs*
Java/JUnit	*Using More Than One Mock For A Test*
Java/TestNG	*Duplicate Test Runs Caused By Inheritance*
Java/TestNG	*Scattered Test Fixtures Caused By Inheritance*
Java/TestNG	*Using Test Case Inheritance To Test Source Code Polymorphism*
Java/TestNG	*Vague Header Setup Smell*
Python/PyTest	*Test Clone*
Python/Unittest	*Lack Of Cohesion Of Test Cases*
Python/Unittest	*Suboptimal Assert*
Robot Framework	*Army Of Clones*
Robot Framework	*Complicated Setup Scenarios*
Robot Framework	*Conditional Assertions*
Robot Framework	*Conspiracy Of Silence*
Robot Framework	*Data Creep*
Robot Framework	*Dependencies Between Tests*
Robot Framework	*Directly Executing Ui Scripts*
Robot Framework	*Duplicate Check*
Robot Framework	*Duplicated Keyword*
Robot Framework	*Duplicated Variable*
Robot Framework	*Hardcoded Environment*
Robot Framework	*Hiding Test Data*
Robot Framework	*Implementation Dependent*
Robot Framework	*Inconsistent Hierarchy*
Robot Framework	*Inconsistent Wording*
Robot Framework	*Lack Of Early Feedback*
Robot Framework	*Lack Of Encapsulation*
Robot Framework	*Lifeless*
Robot Framework	*Long Test Steps*
Robot Framework	*Middle Man*
Robot Framework	*Missing Assertion*
Robot Framework	*Missing Definition*
Robot Framework	*Narcissistic*
Robot Framework	*Noisy Logging*
Robot Framework	*On The Fly*

Language/Framework	Test Smell
Robot Framework	*Over-Checking*
Robot Framework	*Pointless Scenario Descriptions*
Robot Framework	*Sensitive Locators*
Robot Framework	*Sneaky Checking*
Robot Framework	*Stinky Synchronization*
Robot Framework	*Test Data Loss*
Robot Framework	*Testing Data Not Code*
Robot Framework	*Transitive Dependency*
Robot Framework	*Unnecessary Navigation*
Robot Framework	*Unrealistic Data*
Robot Framework	*Unsecured Test Data*
Robot Framework	*Unsuitable Naming*
Smaltalk	*Magic Literals*
TTCN-3	*Activation Asymmetry*
TTCN-3	*Bad Comment Rate*
TTCN-3	*Bad Documentation Comment*
TTCN-3	*Complex Conditional*
TTCN-3	*Constant Actual Parameter Value*
TTCN-3	*Disorder*
TTCN-3	*Duplicate Alt Branches*
TTCN-3	*Duplicate Component Definition*
TTCN-3	*Duplicate In-Line Templates*
TTCN-3	*Duplicate Local Variable/Constant/Timer*
TTCN-3	*Duplicate Statements*
TTCN-3	*Duplicate Template Fields*
TTCN-3	*Duplicated Code In Conditional*
TTCN-3	*Fully-Parametrised Template*
TTCN-3	*Goto*
TTCN-3	*Idle Ptc*
TTCN-3	*Insufficient Grouping*
TTCN-3	*Long Parameter List*
TTCN-3	*Long Statement Block*
TTCN-3	*Magic Values*
TTCN-3	*Missing Log*
TTCN-3	*Missing Variable Definition*
TTCN-3	*Missing Verdict*
TTCN-3	*Nested Conditional*
TTCN-3	*Over-Specificrunson*
TTCN-3	*Short Template*
TTCN-3	*Singular Component Variable/Constant/Timer Reference*
TTCN-3	*Singular Template Reference*
TTCN-3	*Stop In Function*
TTCN-3	*Unreachable Default*

Language/Framework	Test Smell
TTCN-3	*Unrestricted Imports*
TTCN-3	*Unused Definition*
TTCN-3	*Unused Imports*
TTCN-3	*Unused Parameter*
TTCN-3	*Unused Variable Definition*
TTCN-3	*Wasted Variable Definition*
Typescript/Jest	*Empty Tests*
Typescript/Jest	*Timeouts*

APPENDIX C – TEST SMELLS REFACTORINGS BY LANGUAGE AND FRAMEWORK

Table 24 – Test Smells Refactorings by Language and Framework

Test Smell	Refactoring	Language/Framework	Papers
Anonymous Test	Rename method	Java/JUnit	S55
Army Of Clones	Ensure that the user keyword that is called by a test and have at least one clone in the test suite is removed	Robot Framework	S83
Arrange&quit	Replace the if-return block by an assume for the precondition	Java/JUnit	S114
Assert Pre-condition	Replace the assert pre-condition by the junit assume	Java/JUnit	S114
Assertion In Loop	Create test cases for the edge cases and vulnerable values	Java	S1
Assertion In Loop	Utilize repeated test feature	Java	S1
Assertion Roulette	Add assertion explanation	Java	S2
Assertion Roulette	Add assertion explanation	Java/JUnit	S23
Assertion Roulette	Add assertion explanation	Java/JUnit	S26
Assertion Roulette	Add assertion explanation	Java/JUnit	S27
Assertion Roulette	Add assertion explanation	Java/JUnit	S28
Assertion Roulette	Add assertion explanation	Java/JUnit	S37
Assertion Roulette	Add assertion explanation	Scala/ScalaTest	S48
Assertion Roulette	Add assertion explanation	Java/JUnit	S49
Assertion Roulette	Add assertion explanation	Java/JUnit	S5
Assertion Roulette	Add assertion explanation	Java/JUnit	S60
Assertion Roulette	Add assertion explanation	Java/JUnit	S7
Assertion Roulette	Add assertion explanation	Java	S73
Assertion Roulette	Add assertion explanation	Java/JUnit	S75
Assertion Roulette	Add assertion explanation	Java/JUnit	S86
Assertion Roulette	Add assertion explanation	Java/JUnit	S93
Assertion Roulette	Add assertion explanation	Java/JUnit	S104
Assertion Roulette	Extract method	Java/JUnit	S23
Assertion Roulette	Extract method	Java/JUnit	S55
Assertion Roulette	Extract method	Java/JUnit	S7
Assertion Roulette	Extract method	Java/JUnit	S75
Assertion Roulette	Extract method	Python/Unittest	S84
Assertion Roulette	Extract method	Python/Nose	S84
Assertion Roulette	Extract method	Python/PyTest	S84
Assertion Roulette	Single-condition tests	Java/JUnit	S26
Assertion Roulette	Single-condition tests	Java/JUnit	S5
Assertion Roulette	Surround assertions with assertall	Java/JUnit	S75
Assertion Roulette	Use junit5 assertall	Java/JUnit	S107
Assertion Roulette	Use junit5 grouped assertions	Java/JUnit	S26
Assertion Roulette	Wrap assert on withclue	Scala/ScalaTest	S48
Bad Naming	Rename test classes	Java/JUnit	S75
Conditional Assertions	Ensure that the conditional assertion node is removed from the call graph	Robot Framework	S83

Test Smell	Refactoring	Language/Framework	Papers
Conditional Test Logic	Add guarded assertion	Java/JUnit	S26
Conditional Test Logic	Encapsulating this test logic in a test utility method with an intent-revealing name	Java/JUnit	S26
Conditional Test Logic	Extract method	Java/JUnit	S5
Conditional Test Logic	Extract method	Java/JUnit	S7
Conditional Test Logic	Refactoring tests into separate modules (e.g., one for each operating system), making each test module fully executable on the target os	Python/PyTest	S35
Conditional Test Logic	Refactoring tests into separate modules (e.g., one for each operating system), making each test module fully executable on the target os	Python/Unittest	S35
Conditional Test Logic	Transforming os-specific tests that check the os in the test code into their test decorators counterparts, whenever possible	Python/PyTest	S35
Conditional Test Logic	Transforming os-specific tests that check the os in the test code into their test decorators counterparts, whenever possible	Python/Unittest	S35
Conditional Test Logic	Use junit 5 conditional test execution	Java/JUnit	S26
Conditional Test Logic	Use junit 5 repeated tests feature	Java/JUnit	S26
Dead Field	Extract class	Java/JUnit	S54
Dead Field	Extract class	Java/TestNG	S54
Duplicate Assert	Extract method	Java	S2
Duplicate Assert	Extract method	Java/JUnit	S26
Duplicate Assert	Extract method	Java/JUnit	S27
Duplicate Assert	Extract method	Java/JUnit	S28
Duplicate Assert	Extract method	Java/JUnit	S37
Duplicate Assert	Extract method	Java/JUnit	S7
Duplicate Assert	Extract method	Java/JUnit	S104
Duplicate Assert	Use parameterized test	Java/JUnit	S23
Duplicate Assert	Use parameterized test	Java/JUnit	S26
Duplicate Assert	Use parameterized test	Java/JUnit	S104
Duplicate Test Runs Caused By Inheritance	Avoid inheritance from non-abstract test cases	Java/JUnit	S56
Duplicate Test Runs Caused By Inheritance	Avoid inheritance from non-abstract test cases	Java/TestNG	S56
Duplicated Test Code	Extract method	Java/JUnit	S17
Duplicated Test Code	Extract method	Java/JUnit	S26
Duplicated Test Code	Extract method	Java/JUnit	S49
Duplicated Test Code	Extract method	Java/JUnit	S5
Duplicated Test Code	Extract method	Java/JUnit	S86
Duplicated Test Code	Implicity setup	Java/JUnit	S18
Duplicated Test Code	Use junit 5 repeated tests	Java/JUnit	S26
Duplicated Test Code	Use parameterized test	Java/JUnit	S26
Eager Test	Code addition	Java/JUnit	S75
Eager Test	Code removal	Java/JUnit	S75

Test Smell	Refactoring	Language/Framework	Papers
Eager Test	Extract method	Java	S2
Eager Test	Extract method	Java/JUnit	S25
Eager Test	Extract method	Java/JUnit	S28
Eager Test	Extract method	Java	S45
Eager Test	Extract method	Scala/ScalaTest	S48
Eager Test	Extract method	Java/JUnit	S49
Eager Test	Extract method	Java/JUnit	S55
Eager Test	Extract method	Java/JUnit	S60
Eager Test	Extract method	Java	S73
Eager Test	Extract method	Java/JUnit	S86
Eager Test	Extract method	Java/JUnit	S104
Exception Handling	Extract method	Java/JUnit	S7
Exception Handling	Replace try/catch block with @test expected annotation	Java/JUnit	S75
Exception Handling	Replace try/catch block, @test expected, or @rule annotations with the assertthrows	Java/JUnit	S75
Exception Handling	Use ""teardown"" instead of ""finnaly""	Java/JUnit	S5
Exception Handling	Use @rule someexception e = someexception.none()	Java/JUnit	S5
Exception Handling	Use assertthrows and assertexcpetion	Java/JUnit	S7
Exception Handling	Use assumptions	Java/JUnit	S7
Exception Handling	Use catchthrowable	Java/JUnit	S5
Exception Handling	Use expected=myexception.class	Java/JUnit	S5
Exception Handling	Use unit 5 exception handling feature	Java/JUnit	S26
Exception Handling	Using framework-specific features to handle exception testing	Java/JUnit	S26
For Testers Only	Extract subclass	Java/JUnit	S86
For Testers Only	Move method	Java/JUnit	S86
General Fixture	Creating a minimal fixture, which covers only the setup code common for all test methods	Java/JUnit	S54
General Fixture	Creating a minimal fixture, which covers only the setup code common for all test methods	Java/TestNG	S54
General Fixture	Extract class	Java	S45
General Fixture	Extract method	Java	S45
General Fixture	Extract method	Java	S73
General Fixture	Extract method	Java/JUnit	S86
General Fixture	Extract method	Scala/ScalaTest	S48
General Fixture	Inline method	Java/JUnit	S86
General Fixture	Remove fixture	Scala/ScalaTest	S48
General Fixture	Removing trait beforeandafter from the test class and demoting the fields referenced in the argument to method before to local, immutable variables in the appropriate test case	Scala/ScalaTest	S48
General Fixture	Splitting the fixture into multiple smaller fixtures	Scala/ScalaTest	S13
General Fixture	Splitting the fixture into multiple smaller fixtures	Scala/ScalaTest	S48

Test Smell	Refactoring	Language/Framework	Papers
Hardcoded Environment	Ensure the hardcoded argument in a call to a library keyword annotated as "configuration" is replaced with a variable	Robot Framework	S83
Hiding Test Data	Ensure the call to the library keywords annotated as "getter" in the setup of test t is removed	Robot Framework	S83
Ignored Test	Code removal	Java/JUnit	S75
Inappropriate Assertion	Replace reserved words with an appropriate assertion	Java/JUnit	S75
Inappropriate Assertion	Replace the not operator within the assertions	Java/JUnit	S75
Inappropriate Assertion	Split conditional expressions into two different parameters	Java/JUnit	S75
Indirect Testing	Extract method	Java/JUnit	S49
Indirect Testing	Extract method	Java/JUnit	S60
Indirect Testing	Extract method	Java/JUnit	S86
Indirect Testing	Move method	Java/JUnit	S49
Indirect Testing	Move method	Java/JUnit	S60
Interacting Test Suites	Use junit 5 temporary directory	Java/JUnit	S26
Lack Of Cohesion Of Test Methods	Extract class	Java	S45
Lack Of Cohesion Of Test Methods	Extract class	Java/JUnit	S54
Lack Of Cohesion Of Test Methods	Extract class	Java/TestNG	S54
Lack Of Cohesion Of Test Methods	Extract method	Java	S45
Lack Of Encapsulation	Ensure the direct call to a library keyword is removed from the acceptance criteria	Robot Framework	S83
Lazy Test	Inline method	Java/JUnit	S86
Lazy Test	Merge the individual test cases that execute the same method into a single one	Scala/ScalaTest	S48
Lazy Test	Use parameterized test	Java/JUnit	S26
Long Test Steps	Ensure the number of actions performed on the sut by a long step sees its value pass under the threshold l	Robot Framework	S83
Magic Number	Replace magic literal	Java	S2
Magic Number	Replace magic literal	Java/JUnit	S5
Magic Number	Replace magic literal	Java	S65
Magic Number	Replace magic literal	Java/JUnit	S104
Magic Number	Replace magic literal	Java/JUnit	S107
Middle Man	Ensure the delegate keyword call is replaced with another call which is not simply delegating its actions	Robot Framework	S83
Missing Assertion	Ensure library keyword annotated as "assertion" is introduced in test that is missing any assertion	Robot Framework	S83
Multiple Aaa	Extract method	Java/JUnit	S114
Multiple Acts	Break the test case into separate ones, each focusing on one act and add separate assert	Java/JUnit	S114
Mystery Guest	Inline resource	Java/JUnit	S86

Test Smell	Refactoring	Language/Framework	Papers
Mystery Guest	Manage resources explicitly in a fixture	Scala/ScalaTest	S48
Mystery Guest	Setup external resource	Java/JUnit	S26
Mystery Guest	Setup external resource	Java/JUnit	S49
Mystery Guest	Setup external resource	Java/JUnit	S60
Mystery Guest	Setup external resource	Java	S73
Mystery Guest	Setup external resource	Java/JUnit	S86
Mystery Guest	Use junit 5 temporary directory	Java/JUnit	S26
Narcissistic	Ensure the name of a symptomatic "test steps" is changed so that it does not contain a personal pronoun anymore	Robot Framework	S83
Non-deterministic Data Test	Adopt hashing algorithm to generate deterministic randomness	Java	S1
Non-deterministic Data Test	Create a class and store n random data beforehand and then hand them out without synchronization	Java	S1
Non-deterministic Data Test	Set a seed number for random generators to produce deterministic random data	Java	S1
Non-functional Statement	Remove the empty scope or implement the expected functionality	Python/PyTest	S8
Non-functional Statement	Remove the empty scope or implement the expected functionality	Python/Unittest	S8
Obscure Assert	Eliminate unnecessary control flow to simply the logic of assert	Java/JUnit	S114
Obscure In-line Setup	Extract method	Java/JUnit	S68
Obscure In-line Setup	Extract method	Java/TestNG	S68
Obscure In-line Setup	If the in-line setup is common to all tests, one can use an implicit setup	Java/JUnit	S54
Obscure In-line Setup	If the in-line setup is common to all tests, one can use an implicit setup	Java/TestNG	S54
Obscure In-line Setup	Setup code be moved into delegate setup methods	Java/JUnit	S54
Obscure In-line Setup	Setup code be moved into delegate setup methods	Java/TestNG	S54
Parameter Order Wrong	Correct the wrong parameter order in assertions	Java	S1
Poor Test Organization	Separate source and test code and use parallel package structures	Java	S1
Programming Paradigms Blend	Fields located outside the test class can be initialized in the setup method or in a helper method inside the test class	Python/PyTest	S8
Programming Paradigms Blend	Fields located outside the test class can be initialized in the setup method or in a helper method inside the test class	Python/Unittest	S8
Redudant Print	Code removal	Java/JUnit	S75
Resource Leakage	Use junit 5 temporary directory	Java/JUnit	S26
Resource Optimism	Add mock	Java/JUnit	S7
Resource Optimism	Check before use	Java/JUnit	S7
Resource Optimism	Create the resource in the setup	Java/JUnit	S7
Resource Optimism	Make resource unique	Java/JUnit	S86
Resource Optimism	Setup external resource	Java/JUnit	S49

Test Smell	Refactoring	Language/Framework	Papers
Resource Optimism	Setup external resource	Java/JUnit	S5
Resource Optimism	Setup external resource	Java/JUnit	S60
Resource Optimism	Setup external resource	Java/JUnit	S86
Resource Optimism	Use junit 5 temporary directory	Java/JUnit	S26
Resource Optimism	Use temp files (in case the resource is a file)	Java/JUnit	S7
Return In Test	Remove the return statement in test cases	Java	S1
Scattered Test Fixtures Caused By Inheritance	Manage test fixtures individually and independently in each test case	Java/JUnit	S56
Scattered Test Fixtures Caused By Inheritance	Manage test fixtures individually and independently in each test case	Java/TestNG	S56
Scattered Test Fixtures Caused By Inheritance	Use junit's @rule annotation to refactor the code that needs to be executed before and after a test	Java/JUnit	S56
Scattered Test Fixtures Caused By Inheritance	Use junit's @rule annotation to refactor the code that needs to be executed before and after a test	Java/TestNG	S56
Scattered Test Fixtures Caused By Inheritance	Use test utility classes to manage test fixtures instead of using inheritance	Java/JUnit	S56
Scattered Test Fixtures Caused By Inheritance	Use test utility classes to manage test fixtures instead of using inheritance	Java/TestNG	S56
Sensitive Equality	Compare the members of the object states structurally instead of relying on tostring() of the wholes	Scala/ScalaTest	S48
Sensitive Equality	Introduce equality method	Java	S2
Sensitive Equality	Introduce equality method	Java/JUnit	S49
Sensitive Equality	Introduce equality method	Java/JUnit	S5
Sensitive Equality	Introduce equality method	Java/JUnit	S60
Sensitive Equality	Introduce equality method	Java	S73
Sensitive Equality	Introduce equality method	Java/JUnit	S86
Sleepy Test	Add mock	Java/JUnit	S7
Sleepy Test	Code addition	Java/JUnit	S75
Sleepy Test	Refactor the test smell using java's future library	Java	S65
Sleepy Test	Reorder tests	Java/JUnit	S7
Sleepy Test	Use intelligent waiting library	Java/JUnit	S7
Sleepy Test	Use waitfor() condition in the java awaitility library	Java	S65
Suppressed Exception	Remove the catch and keep the try	Java/JUnit	S114
Test Maverick	Extract class	Java/JUnit	S54
Test Maverick	Extract class	Java/TestNG	S54
Test Run War	Make resource unique	Java/JUnit	S26
Test Run War	Make resource unique	Java/JUnit	S86
Test Run War	Reorder tests	Java/JUnit	S7
Test Run War	Use junit 5 resource lock feature	Java/JUnit	S26
Undefined Test	Rename the test method using the prefix "test" or remove it	Python/PyTest	S8
Undefined Test	Rename the test method using the prefix "test" or remove it	Python/Unittest	S8
Unknown Test	Add assertion	Java/JUnit	S7
Unknown Test	Add assertion	Java/JUnit	S114

Test Smell	Refactoring	Language/Framework	Papers
Unknown Test	Code addition	Java/JUnit	S75
Unknown Test	Remove method	Java/JUnit	S7
Using Test Case Inheritance To Test Source Code Polymorphism	Use parameterized test	Java/JUnit	S56
Using Test Case Inheritance To Test Source Code Polymorphism	Use parameterized test	Java/TestNG	S56
Vague Header Setup Smell	Field initializations should be placed in an implicit setup to specify the behavior and the places to inspect within a class	Java/JUnit	S54
Vague Header Setup Smell	Field initializations should be placed in an implicit setup to specify the behavior and the places to inspect within a class	Java/TestNG	S54
Verifying In Setup Method	Instead of using it in the setup method that runs before each test method, use it in the setupclass method that runs only once	Python/PyTest	S8
Verifying In Setup Method	Instead of using it in the setup method that runs before each test method, use it in the setupclass method that runs only once	Python/Unittest	S8