AriesLinter: Sniffing Test Smells Before They Happen

Rafael Rocha
Federal University of Bahia
Salvador, Bahia, Brazil
rafaelvieira@ufba.br

Eronildo Junior
Federal University of Bahia
Salvador, Bahia, Brazil
eronildosilva@ufba.br

Tássio Virgínio
Federal University of Bahia
Salvador, Bahia, Brazil
tassio.virginio@ifto.edu.br

Larissa Rocha
State University of Bahia
Feira de Santana, Bahia, Brazil
larissabastos@uneb.br

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

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

ABSTRACT

Software quality depends not only on production code but also on the quality of test code. Among the factors that compromise this quality, test smells stand out, poor implementation patterns in tests that negatively affect the readability, maintainability, and reliability of the test suite. Although tools such as tsDetect and JNose support test smell detection, they operate post-hoc, requiring the code to be complete and analyzed outside the development environment. Other tools, like RAIDE, offer detection within the IDE via plugins; however, the analysis is not instantaneous and still occurs after the code is written. These reactive approaches result in delayed feedback and significantly increase the refactoring effort. In this context, we propose AriesLinter, a static analysis tool that can be integrated into major development environments and is capable of detecting test smells in real time, as the test code is being written. Built on top of the Checkstyle infrastructure, AriesLinter automatically detects 10 of the most recurrent test smells reported in the literature and commonly found in Java projects. The tool is cross-platform and compatible with multiple IDEs. By anticipating the detection of poor patterns and embedding it into the natural workflow of developers, AriesLinter acts proactively, reducing the likelihood of test smells being introduced into the codebase and fostering a culture of continuous attention to test quality.
Demo video: https://youtu.be/Cj6mQGGHlvk

KEYWORDS

Test Smells, Static Analysis, Linter, IDE Integration, Java, Checkstyle

1 Introduction

In software engineering, the pursuit of software quality is a continuous endeavor that spans not only production code but also test code. Among the elements that compromise test code quality, *test smells* stand out as significant factors. Test smells are poor design and implementation patterns in test code that reduce readability, maintainability, and trust in the test suite [14]. These anti-patterns may lead to brittle tests, hinder code evolution, and increase the long-term cost of software maintenance [10].

Modern software development increasingly recognizes test code quality as equally critical as production code quality [1, 22]. Test smells - problematic patterns in test implementation - directly undermine test effectiveness and maintainability [9, 10]. Empirical evidence shows these suboptimal patterns increase test flakiness by 40% and significantly raise maintenance costs in large-scale projects [9, 21]. The persistence of test smells in codebases presents particular concern, with studies documenting an 80% probability of smells remaining uncorrected even 1,000 days after introduction [7, 21]. This persistence stems from three key factors: unrealistic time pressures in development cycles [20], insufficient awareness of test quality best practices [10], and the historical prioritization of production over test code [7].

Numerous types of test smells have been cataloged over the years [11], and tools have been developed to detect them in several programming languages, including Python [5], JavaScript [15], Scala [8], and Java [22]. Despite the availability of these tools, most existing solutions operate under a post-hoc analysis model. Tools like tsDetect [16] and JNose [22] perform detection only after the code has been fully implemented and manually submitted for review. Others, such as RAIDE [19], improve on integration by functioning as IDE plugins but still rely on manual triggering and post-editing analysis. These delayed feedback mechanisms substantially increase refactoring effort and reduce developer productivity.

In contrast, early detection mechanisms—such as linters—are increasingly adopted in modern development environments due to their ability to provide real-time, in-editor feedback [3]. Linters analyze code as it is written, identifying violations of predefined rules, thereby preventing issues from being committed in the first place [4]. Their tight integration into IDEs makes them the first line of defense against the introduction of code and test anti-patterns.

However, although linters are widely available for production code, there is still a gap in the availability of equivalent tools specifically designed for test smells—especially those that offer real-time detection. This gap motivated the creation of **AriesLinter**, a lightweight and extensible linter for detecting test smells in tests written in Java. Built on top of the CheckStyle infrastructure, AriesLinter provides real-time static analysis and is compatible with IDEs such as Visual Studio Code[1] and IntelliJ IDEA[2]. It also includes a Maven plugin that prevents the project from being compiled while test smells are present.

AriesLinter currently detects 10 of the most frequent and disruptive test smells reported in the literature, providing developers with continuous feedback during the test development process. By embedding test smell detection directly into the development workflow, AriesLinter shifts quality control from a reactive to a proactive paradigm, helping developers maintain high test code quality and reducing the cost of future maintenance.

[1] https://code.visualstudio.com/
[2] https://www.jetbrains.com/idea/

Current test smell detection approaches exhibit fundamental limitations in feedback timing and workflow integration [1, 17]. Existing tools predominantly employ post-hoc analysis models requiring completed code and manual execution, resulting in delayed feedback cycles [1, 19, 22]. This reactive paradigm proves particularly costly, with late-stage refactoring requiring up to 3× more effort than immediate corrections [9]. The effectiveness of detection heuristics depends more on implementation choices and workflow integration than on theoretical capabilities [17]. Immediate feedback during development sessions has proven crucial for reducing smell introduction rates [13, 20], while developer education alone shows limited impact without tool support [10, 20].

These findings motivate **AriesLinter**'s core innovations:

- Real-time detection during active development
- Deep IDE integration matching developer workflows
- Balanced test smell coverage (10 critical smells) optimizing precision/recall tradeoffs

The tool specifically addresses the "quality awareness gap", where 68% of developers inadvertently introduce smells despite understanding test quality principles. By embedding detection in the coding workflow, AriesLinter transforms test quality from a retrospective concern to an integral part of the development process [9, 17].

2 Background

This background section establishes the necessary context for understanding AriesLinter by discussing the fundamental concepts of static code analysis and linting tools.

2.1 Linter

The evolution of static code analysis began with Stephen C. Johnson's original `lint` utility developed at Bell Labs during the late 1970s [2]. Designed specifically for the C programming language, this foundational tool introduced the paradigm of detecting "bugs and obscurities" through static pattern matching of source code [2]. Over four decades of technological advancement, linting tools have expanded far beyond their original capabilities, now supporting multiple programming paradigms and sophisticated analysis techniques while retaining the core principles established by Johnson's work [2, 12].

Modern linters operate through a multi-stage static analysis process that begins with lexical analysis, where source code is decomposed into interpretable tokens [12]. This token stream then undergoes syntactic analysis to construct an Abstract Syntax Tree (AST), forming a structural representation of the program's logic [2, 12]. The final phase applies configurable rulesets to this AST, evaluating code quality against predefined patterns and heuristics without requiring actual program execution [12]. This architecture enables comprehensive detection of diverse code quality issues ranging from basic syntax violations and type system inconsistencies to more subtle logical redundancies and security vulnerabilities [2, 12].

The theoretical foundation of linting tools positions them uniquely within the compilation pipeline [12]. Functioning as meta-compilers, linters prioritize defect detection over code generation, adopting more flexible analysis models than traditional compilers [2]. Unlike compilers that must reject invalid code, linters tolerate non-fatal errors during analysis to support gradual code quality improvement [12]. This tolerance, combined with configurable rule severity levels, explains why modern linters have become indispensable in professional software development workflows [2, 12]. Their value increases significantly when integrated directly into Integrated Development Environments (IDEs), where they provide real-time feedback during the coding process itself [2]. Contemporary implementations extend beyond basic style checking to incorporate sophisticated program analysis techniques, including data flow analysis and pattern-based vulnerability detection, while maintaining the original vision of preventing defects before code execution [2, 12].

Figure 1 illustrates this architecture by presenting the five core stages of a modern linter. The process begins with the source code, which is passed through lexical and syntactic analysis to produce an Abstract Syntax Tree (AST). The AST is then evaluated by a configurable rule engine that detects violations based on pattern matching and static heuristics. Finally, the issue reporter delivers structured feedback directly to the developer. This pipeline enables real-time, in-editor analysis and supports proactive quality assurance in contemporary development environments.

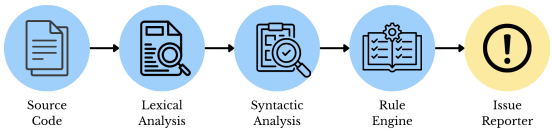

Figure 1: Conceptual architecture of a modern linter.

2.2 CheckStyle

Checkstyle was originally developed in 2001 by Oliver Burn as an automated verification tool for Java coding standards, designed to enforce consistency and improve code readability [6]. Over two decades, it has evolved into one of the most widely-adopted static analysis tools in the Java ecosystem, serving both individual developers and large enterprise teams [6].

As a specialized Java linter, Checkstyle's core functionality involves automated verification of code compliance against configurable style rules and coding standards [6]. Its modular architecture enables extensible rule customization through:

- Predefined checks for common Java conventions
- Configurable severity levels
- Custom rule development via AST parsing

The tool's current maintenance involves an active open-source community, with the original creator no longer directly contributing [6]. Modern implementations support:

- IDE integration (IntelliJ, VS Code)
- Build tool plugins (Maven, Gradle, Ant)
- Continuous integration pipelines

Figure 2 illustrates the overall architecture and workflow of Checkstyle, including its interaction with style rules, Java source code, and the generation of feedback.

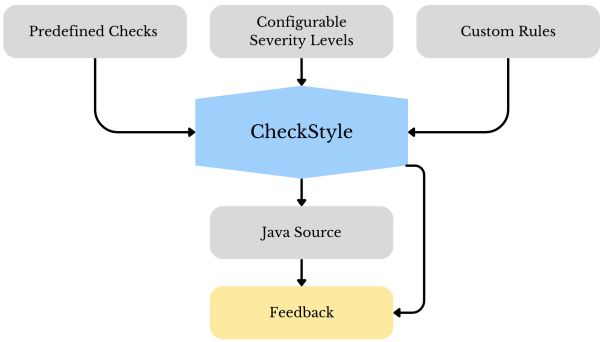

Figure 2: Overview of Checkstyle architecture and analysis flow.

2.3 Test Smells

Test Smells (TS) are suboptimal design or implementation choices in test code that can negatively affect code readability, comprehension, and maintainability[17, 20]. Their presence often results in less effective tests, which may fail to detect bugs in the production code[10, 16]. Moreover, they increase the cost of maintaining the test suite and often serve as early indicators of poor programming practices in the codebase[19].

The term Test Smell originates from the broader concept of Code Smell and refers to symptoms in test code that[9, 18], while not preventing test execution, can degrade the ability to detect regressions, inflate maintenance effort, and erode developers' trust in the test suite [10]. As with code smells, test smells are not bugs themselves but rather signs of structural weaknesses that can lead to future failures.

Refactoring test smells has a positive impact on test code quality: negative attributes such as code size, coupling, and complexity are significantly reduced, while positive qualities like readability, reusability, and maintainability are greatly enhanced[7]. Addressing test smells early—particularly in agile development environments—can reduce long-term maintenance costs by up to threefold[9].

Test smells are frequently introduced in the initial versions of test code and tend to persist over time. Studies show that there is an 80% probability a test smell will remain unaddressed even after 1,000 days of its introduction[21].

3 AriesLinter

AriesLinter was created to fill a gap in the state of the art, bringing the benefits of real-time static code analysis. As the code is written, it is analyzed — a functionality also known as a linter. Our library was developed based on CheckStyle, a static code checker for the Java language. We chose CheckStyle due to its wide range of plugins for execution and integration with the main IDEs available on the market.

The advantage of using a linter lies in the detection of errors as the code is being written, allowing for early assessment of code quality and preventing problematic code from being committed to repositories — and especially from being included in a system release.

When analyzing the state of the art, we could not find a linter focused on detecting Test Smells in Java. In fact, there are several good tools for test smell detection, including tsDetect [16] and JNose [22]. However, both require additional steps to be used after the code has been written. Moreover, tsDetect is a command-line level tool, which further hinders its usability.

It was in this context that the idea to create a tool that addresses this gap emerged. From its conception, we decided that it should follow a few principles: it should run in the most widely used IDEs, be easy to install, simple to use, provide clear messages, and be fast during test writing. Therefore, we developed our tool as a module of CheckStyle — already well-established in Java application development and supported by various stable plugins for most IDEs.

Thus, AriesLinter is an extension of CheckStyle, aimed at analyzing Java projects, specifically those with tests written using the JUnit framework. It enables the analysis of test code quality through the detection of test smells during the coding process itself. The tool identifies classes, methods, and lines affected by a specific test smell in real time.

As explained in the background section, CheckStyle is a linter for Java code available as a plugin in various IDEs, such as Visual Studio Code. Since AriesLinter is built upon CheckStyle, it can be configured in any environment where the CheckStyle plugin is available.

Our tool detects 10 types of test smells, which we describe below:

(1) Conditional Test Logic: It occurs when a test method contains one or more control statements, whereas it should be simple and execute all statements.
(2) Default Test: When an IDE automatically generates a template to be modified for a test, but the developer leaves the example without making any changes.
(3) Exception Handling: It occurs when a test method for which pass or failure depends on the production/test method to throw an exception instead of using the testing framework constructs.
(4) Verbose Test: A test method with more than 30 lines.
(5) Ignored Test: It occurs when developers suppress test methods from running using the tags @Ignore or @Disabled.
(6) Magic Number Test: It occurs when assert statements in a test method contain numeric literals as parameters that cannot provide their meaning.
(7) Sensitive Equality Test: Occurs when the toString method is used within a test method.
(8) Redundant Print: It occurs when a test method contains print statements.
(9) Sleepy Test: Explicitly causing a thread to sleep can lead to unexpected results as the processing time for a task can differ on different devices.
(10) Unknown Test: It occurs when a test method does not contain assertions. As a result, JUnit shows the test method as passing unless the statements raise an exception in the test method.

The selection of test smells implemented in the tool was guided by two main criteria. First, we considered smells that are already detected by well-established tools in the literature, such as tsDetect and JNose, aiming to align our approach with recognized practices. Second, during the internal testing phase, we implemented the

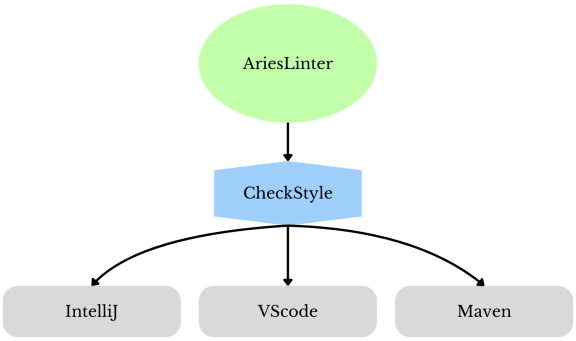

Figure 3: Architecture of AriesLinter.

detection of all smells that were technically feasible. However, we chose to retain only those whose detection did not achieve 100% accuracy in our test suite. This decision aimed to maximize the tool's value as a support for the test review process, prioritizing smells that are relevant in both the literature and industry.

As previously discussed in the background section, Checkstyle is a Java linter available as a plugin for various IDEs, such as Visual Studio Code and Intellij IDEA. Because AriesLinter is built on top of Checkstyle, it can be configured in any development environment that supports the Checkstyle plugin.

Regarding its test smell detection capabilities, AriesLinter leverages Checkstyle's abstract syntax tree (AST) construction mechanism to analyze source code and identify smells. Each test smell is defined through a dedicated detection module, which implements the specific algorithm used to flag the corresponding code pattern.

Figure 3 presents the overall design of AriesLinter, highlighting its main components and how they interact to perform real-time test smell detection.

3.1 Usage Examples

As previously discussed, AriesLinter can be used in any IDE that supports the Checkstyle plugin. To demonstrate this cross-IDE compatibility, we present usage examples in the most popular Java development environments.

Figure 4 shows a test smell detection case using AriesLinter in Visual Studio Code. The test method `should_be_verbose_test_0` is underlined in red, indicating a violation detected by the linter. The IDE highlights the issue and displays the message *"Verbose Test detected: method with 33 lines"*, providing concise feedback about the nature of the detected test smell. Notably, the issue is flagged in real time, even before the developer saves or runs the code.

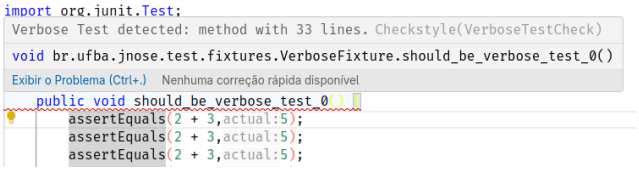

Figure 4: Verbose Test detection in VSCode

In Figure 5, we observe a similar case within IntelliJ IDEA. Again, the test method is underlined, and a tooltip message clearly reports the detection: *"Verbose Test detected: method with 31 lines."* This reinforces AriesLinter's consistent behavior across different IDEs, maintaining the same user experience and detection feedback.

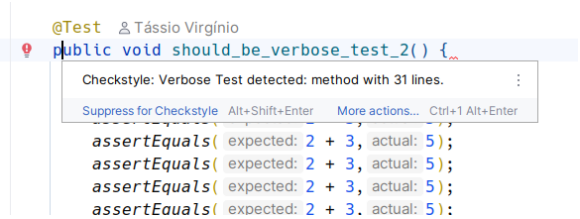

Figure 5: Verbose Test detection in IntelliJ (inline view)

Figure 6 provides a broader view of how AriesLinter integrates with IntelliJ's Checkstyle plugin panel. In the left sidebar, the Checkstyle scan results are accessible via a dedicated tab. The "Rules" dropdown shows that the selected ruleset is "AriesLinter". The panel displays the detection of three verbose test methods in the file `VerboseFixture.java`, with each entry providing line-specific feedback, such as the number of lines and the exact smell type. This allows developers to quickly locate and fix problematic test methods.

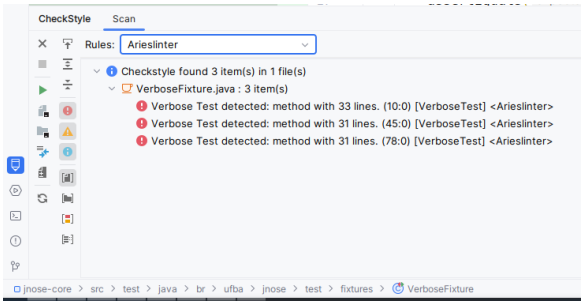

Figure 6: Verbose Test detection in IntelliJ (Checkstyle panel)

In Figure 7, we now see the detection of a Magic Number. The line where the smell is identified is underlined in red, and an error message is displayed with a brief description: "Magic Number detected: use a variable with a self-explanatory name."

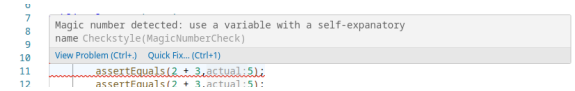

Figure 7: Magic Number detection in VSCode (in line view)

In Figure 8, still using an example in VSCode, we can observe the detection of the Ignored Test smell. Upon analysis, we see the entire test highlighted in red, indicating a syntax-related issue, along with a context-aware error message: "Ignored test detected: method with an '@Ignore' annotation."

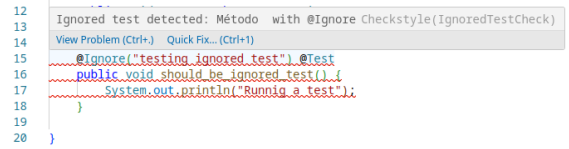

Figure 8: Ignored Test detection in VSCode (in line view)

4 Related Works

The literature presents several tools for test smell detection, each with distinct characteristics. In this section, we will analyze some existing solutions and their limitations, always in comparison with AriesLinter.

First, we have **tsDetect**[3], a well-established open-source tool for detecting test smells in Java code [16]. Developed by an international research team, it was designed to integrate test quality verification into modern code reviews. Implemented as a command-line JAR file, tsDetect operates through three sequential phases: test file detection (identifying unit test files), test file mapping (linking tests to corresponding production classes), and test smell detection (identifying 21 distinct smell types using JavaParser for AST construction). The tool generates CSV reports with binary results indicating the presence of smells. While it demonstrates high detection accuracy, tsDetect has critical limitations: it requires complete code for analysis, involves complex setup procedures with significant time overhead, and lacks integration with IDEs.

Next, we consider **JNose**[4], an extension of tsDetect developed by Brazilian researchers [22], which introduced a web-based graphical interface with code coverage metrics. The tool processes projects in three stages: input configuration, project analysis (operating in ClassTest, TestSmell, or Evolution modes for different perspectives), and CSV report generation. JNose improved upon tsDetect by offering more intuitive interaction and faster processing times. However, it maintains similar constraints, requiring fully implemented test code and operating outside development environments, which demands manual intervention to execute analyses.

Finally, we have **RAIDE**[5], an Eclipse plugin [19] specialized in detecting and refactoring two specific smells (Assertion Roulette and Duplicate Assert). Its workflow combines test class detection, smell identification through Eclipse views, and semi-automated refactoring with change previews. Although RAIDE advances IDE integration by operating within Eclipse and supporting automated fixes, its applicability remains narrow due to Eclipse exclusivity, limited smell coverage, and the requirement for manual approval of each refactoring operation.

To conclude the section, Table 1 presents a systematic comparison between the existing tools and our proposed solution, AriesLinter.

5 Conclusion

This work presented *AriesLinter*, a static analysis tool designed to detect test smells in real time during the development of automated tests in Java. Built as an extension of the Checkstyle plugin,

[3]https://testsmells.org/
[4]https://testsmells.org/pages/testsmelldetector.html
[5]https://raideplugin.github.io/RAIDE/

Table 1: Comparative Analysis of Test Smell Detection Tools

Feature	tsDetect	JNose	RAIDE	AriesLinter
Detection Timing	Post-hoc	Post-hoc	Post-hoc	**Real-time**
IDE Integration	None	Web Interface	Eclipse-only	**Multi-IDE**
Automated Correction	No	No	Partial	**No**
Test Smells Detected	21	21	2	10
Analysis Trigger	Manual	Manual	Manual	**Continuous**
Setup Complexity	High	Medium	Medium	**Low**
Feedback Latency	Hours-minutes	Minutes	Minutes	**Seconds**
Refactoring Support	No	No	Yes	**No**
Language Support	Java	Java	Java	**Java**

AriesLinter distinguishes itself from traditional approaches by introducing a proactive perspective on test quality assurance, shifting the detection of anti-patterns from post-development analysis to the moment the test code is written.

The motivation behind the tool is grounded in empirical evidence from the literature, which highlights the persistence and harmful impact of test smells in codebases, as well as the limitations of existing tools in terms of response time, usability, and integration with modern development environments. In this context, AriesLinter emerges as a proposal to mitigate the "quality awareness gap"—a phenomenon in which developers, despite knowing test best practices, still introduce smells due to the lack of adequate tools that offer timely feedback.

Throughout the development of AriesLinter, we adopted guiding principles such as ease of use, lightweight integration, and multi-IDE support, ensuring not only technical effectiveness but also practical viability for adoption in everyday development workflows. The tool currently detects 10 recurring types of test smells, using specialized detection algorithms built on top of Checkstyle's abstract syntax tree (AST).

However, we acknowledge that the journey toward continuous improvement in test quality does not end here. As future work, we highlight four main directions: (i) expanding the coverage of *smells*, including more complex and context-dependent categories; (ii) incorporating automated fix mechanisms and contextualized refactoring suggestions; (iii) conducting controlled empirical studies and evaluations in industrial environments to measure the tool's impact on developer productivity and the long-term quality of the test suite; and (iv) performing a new comparative analysis of AriesLinter with other tools, such as *SonarLint* and customized *CheckStyle* rules for test code, in order to provide a broader and more contextualized view of the tool's capabilities.

Ultimately, AriesLinter represents more than a technical contribution: it proposes a paradigm shift in which test code quality ceases to be a retrospective concern and becomes a continuously monitored, addressed, and valued aspect from the earliest stages of software development.

ARTIFACT AVAILABILITY

The authors declare that the research artifacts supporting the findings of this study are accessible at:
https://zenodo.org/records/16998677

The AriesLinter source code is licensed under the MIT license and is available on GitHub at https://github.com/viRafael/arieslinter.

The video are available on Zenodo:
https://doi.org/10.5281/zenodo.15484942.

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001; CNPq grants 140587/2024-1, 315840/2023-4 and 403361/2023-0; and FAPESB grant PIE0002/2022.

REFERENCES

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

[2] Fr ans Kunst. 1988. Lint, a C Program Checker. (1988).

[3] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008), 22–29.

[4] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2017. Developer testing in the ide: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering* 45, 3 (2017), 261–284.

[5] Alexandru Bodea. 2022. Pytest-Smell: a smell detection tool for Python unit tests. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 793–796.

[6] Checkstyle Development Team. [n. d.]. Checkstyle - A development tool to help programmers write Java code that adheres to a coding standard. https://checkstyle.sourceforge.io/. Accessed: 2025-05-10.

[7] Humberto Damasceno, Carla Bezerra, Denivan Campos, Ivan Machado, and Emanuel Coutinho. 2023. Test smell refactoring revisited: What can internal quality attributes and developers' experience tell us? *Journal of Software Engineering Research and Development* 11, 1 (Oct. 2023), 13:1 – 13:16. doi:10.5753/jserd.2023.3195

[8] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2019. Socrates: Scala radar for test smells. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*. 22–26.

[9] Vahid Garousi, Baris Kucuk, and Michael Felderer. 2019. What We Know About Smells in Software Test Code. *IEEE Software* 36, 3 (2019), 61–73. doi:10.1109/MS.2018.2875843

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

[11] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. 2012. Test confessions: A study of testing practices for plug-in systems. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 244–254.

[12] P. Louridas. 2006. Static code analysis. *IEEE Software* 23, 4 (2006), 58–61. doi:10.1109/MS.2006.114

[13] Luana Martins, Taher A. Ghaleb, Heitor Costa, and Ivan Machado. 2024. A comprehensive catalog of refactoring strategies to handle test smells in Java-based systems. *Software Quality Journal* 32, 2 (March 2024), 641–679. doi:10.1007/s11219-024-09663-7

[14] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code*. Pearson Education.

[15] Jhonatan Oliveira, Luigi Mateus, Tássio Virgínio, and Larissa Rocha. 2024. SNUTS. js: Sniffing Nasty Unit Test Smells in Javascript. In *Simpósio Brasileiro de Engenharia de Software (SBES)*. SBC, 720–726.

[16] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. tsDetect: an open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1650–1654. doi:10.1145/3368089.3417921

[17] Valeria Pontillo, Luana Martins, Ivan Machado, Fabio Palomba, and Filomena Ferrucci. 2025. An empirical investigation into the capabilities of anomaly detection approaches for test smell detection. *Journal of Systems and Software* 222 (2025), 112320. doi:10.1016/j.jss.2024.112320

[18] Railana Santana, Luana Martins, Tássio Virgínio, Larissa Soares, Heitor Costa, and Ivan Machado. 2022. Refactoring assertion roulette and duplicate assert test smells: a controlled experiment. *arXiv preprint arXiv:2207.05539* (2022).

[19] Railana Santana, Luana Martins, Tássio Virgínio, Larissa Rocha, Heitor Costa, and Ivan Machado. 2024. An empirical evaluation of RAIDE: A semi-automated approach for test smells detection and refactoring. *Science of Computer Programming* 231 (2024), 103013. doi:10.1016/j.scico.2023.103013

[20] Nildo Silva Junior, Luana Martins, Larissa Rocha, Heitor Costa, and Ivan Machado. 2021. How are test smells treated in the wild? A tale of two empirical studies. *Journal of Software Engineering Research and Development* 9, 1 (Sep. 2021), 9:1 – 9:16. doi:10.5753/jserd.2021.1802

[21] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 4–15. doi:10.1145/2970276.2970340

[22] Tássio Virgínio, Luana Martins, Railana Santana, Adriana Cruz, Larissa Rocha, Heitor Costa, and Ivan Machado. 2021. On the test smells detection: an empirical study on the JNose Test accuracy. *Journal of Software Engineering Research and Development* 9, 1 (Sep. 2021), 8:1 – 8:14. doi:10.5753/jserd.2021.1893