From Reactive to Preventive: Real-Time Test Smell Avoidance with Linting

Rafael Vieira
Instituto de Computação,
Universidade Federal da Bahia (UFBA)
Salvador, Brasil
rafaelvieira@ufba.br

Tassio Virginio
Instituto de Computação,
Universidade Federal da Bahia (UFBA)
Salvador, Brasil
tassio.virginio@ufba.br

Ivan Machado
Instituto de Computação,
Universidade Federal da Bahia (UFBA)
Salvador, Brasil
ivan.machado@ufba.br

Abstract

Test code plays a fundamental role in ensuring software quality, yet its own quality is often overlooked. Poor implementation practices, commonly known as test smells, can reduce the readability, maintainability, and reliability of test suites. Existing tools, such as tsDetect, JNose, and RAIDE, detect test smells only after the test code has been written, limiting their ability to support developers during implementation. To address this limitation, we present AriesLinter, a tool that detects seventeen types of test smells in real time as developers write test code. We evaluated AriesLinter through two complementary studies. First, we investigated developers' perceptions of real-time detection with respect to code quality, learning, and maintainability, obtaining positive feedback on its usefulness during development. Second, we evaluated the accuracy of AriesLinter against a manually constructed oracle. The results suggest that integrating test smell detection into the development process provides timely feedback while maintaining reliable detection, supporting continuous improvement in test code quality.

Keywords

Test Smells, Linter, Static Analysis, Test Code

Research Context

This work originated from an undergraduate research project conducted under a PIBIC/CNPq scholarship at the Federal University of Bahia (UFBA). This work was presented at the *Congresso UFBA 2025*, the university's undergraduate research symposium. Additional information about the event is available at: https://proext.ufba.br/congresso-ufba-2025-programacao-completa.

1 Introduction

Test code is now recognized as a key component of software quality, demanding the same level of attention as production code [12, 18]. Despite this recognition, test smells remain common in software projects and are often neglected during development [5, 6]. These poor implementation practices reduce the readability, maintainability, and reliability of test suites, making them more fragile, hindering software evolution, and increasing maintenance effort over time [3, 6].

Test smells remain widespread in industrial software projects. Previous studies have shown that nearly 80% of these issues remain unresolved for more than one thousand days after their introduction [3, 16]. Their persistence has been associated with unrealistic delivery deadlines [15], limited knowledge of testing best practices [6], and the longstanding tendency to prioritize production code over test code [3]. As a result, test smells contribute to fragile test suites, increasing test flakiness by up to 40% and raising maintenance effort in large-scale software projects [5, 16].

A variety of tools have been developed to detect test smells across different programming languages, including Python [1], JavaScript [10], Scala [4], and Java [17]. Despite this diversity, most of these tools analyze test code only after it has been written, requiring developers to explicitly invoke the detection process. This is the case for tools such as tsDetect [11], JNose [18], and even RAIDE [14], which integrates with the IDE but still depends on manual execution. Consequently, developers receive feedback late in the development process, often increasing the effort required to identify and remove test smells.

To address these limitations, we present AriesLinter, a lightweight and extensible tool for detecting test smells in Java test code. Built on the CheckStyle infrastructure, AriesLinter performs continuous static analysis and integrates directly with development environments such as VSCode[1] and IntelliJ IDEA[2], providing immediate feedback as developers write test code.

Unlike existing tools, AriesLinter provides continuous feedback while developers write test code instead of requiring manual, post hoc analysis. It currently detects 17 types of test smells and reports them in real time within the development environment. By making test smell detection part of the coding process, AriesLinter helps developers identify and address quality issues earlier, reducing the effort required to maintain test code.

This paper extends our previous work [13], which introduced AriesLinter. The extended version expands the tool's capabilities by increasing its detection coverage from 9 to 17 test smell types, evaluates its detection performance through a quantitative analysis, and reports the results of a survey with 17 developers on the perceived usefulness of the tool during software development.

The dataset and replication package are publicly available at https://zenodo.org/records/20945473.

2 Background

This section provides the background necessary to understand AriesLinter, covering static code analysis, CheckStyle, and test smells.

2.1 Linter

The first linter, Lint, was introduced by S. C. Johnson in 1988 for the C programming language [7]. It pioneered the use of static analysis to identify inconsistencies, type violations, and other potentially

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

erroneous constructs without executing the program. Since then, linters have evolved to support a wide range of programming languages and coding standards, providing automated feedback that helps developers detect defects and improve code quality during development [8].

Linters perform static analysis by parsing source code and evaluating it against a predefined set of rules [7, 8]. As illustrated in Figure 1, this process typically consists of lexical analysis, syntactic analysis, and the construction of an Abstract Syntax Tree (AST), which is traversed by a rule engine to identify violations. The detected issues are then reported to the developer, enabling immediate feedback during software development [8].

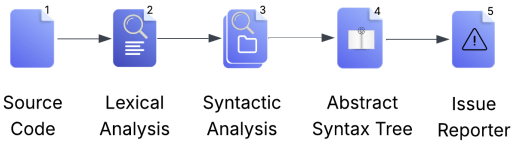

Figure 1: Conceptual architecture of a modern linter.

2.2 CheckStyle

CheckStyle is a widely adopted static analysis framework for Java. Originally developed by Oliver Burn in 2001 to enforce coding standards and improve code readability, it has evolved into an extensible platform for implementing rule-based analyses [2].

CheckStyle provides a modular architecture that supports the implementation of custom analysis rules and configurable severity levels [2]. Its rule engine traverses the Abstract Syntax Tree (AST), allowing developers to detect project-specific coding patterns beyond standard Java style conventions.

2.3 Test Smells

Test smells are poor design or implementation practices in test code that, although they do not prevent tests from executing, reduce their readability, reliability, and maintainability [12, 15]. Like code smells in production code, they indicate structural problems that make test suites harder to understand, maintain, and evolve, while also reducing their effectiveness in detecting defects [6, 14].

Although test smells are not faults themselves, removing them through refactoring can improve the internal quality of test code by reducing complexity and improving maintainability [3]. Addressing these issues early in the development process can also reduce the effort required to maintain test suites over time [5].

3 AriesLinter

AriesLinter is a static analysis tool that detects test smells in Java test code as developers write it. Unlike existing tools, such as tsDetect [11] and JNose [17], which perform detection after the code has been written, AriesLinter provides continuous feedback directly within the development environment. This approach allows developers to identify and address test smells earlier, reducing the need for later inspection and refactoring. In addition, while some existing tools are available only through the command line, AriesLinter integrates with modern IDEs, making test smell detection part of the regular development workflow.

AriesLinter was designed to address the lack of real-time support for test smell detection in Java. It is an open-source project released under the MIT License and is publicly available on GitHub[3].

AriesLinter addresses this limitation by providing real-time detection of test smells as developers write Java test code. Built on top of CheckStyle, it leverages its modular, AST-based architecture to implement custom detection rules for test smells. Because CheckStyle is already integrated with popular IDEs such as IntelliJ IDEA and VS Code, AriesLinter can provide immediate feedback without requiring additional analysis tools or changes to the development workflow. As illustrated in Figure 2, the tool analyzes the Abstract Syntax Tree (AST) generated by CheckStyle to identify test smell instances in Java test classes.

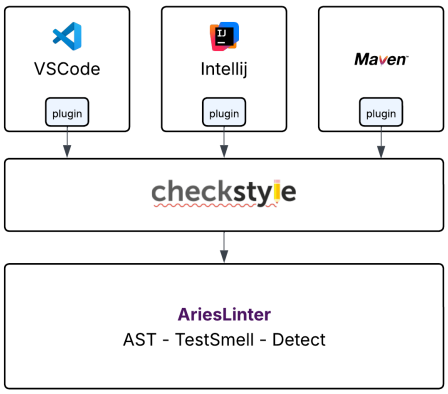

Figure 2: Architecture of AriesLinter.

Because AriesLinter is built on top of CheckStyle, it can be used in any development environment that supports the CheckStyle plugin. As developers write test code, the tool reports detected test smells directly in the editor, allowing issues to be addressed before the code is committed. This integration enables developers to incorporate test smell detection into their regular development workflow without requiring additional analysis steps.

AriesLinter currently detects 17 types of test smells. For each detected smell, the tool reports its location (class, method, and line) together with a descriptive message. The supported test smell types are listed below.

(1) **Assertion Roulette (AR):** Occurs when a test method contains more than one assertion statement without a message.
(2) **Conditional Test Logic (CTL):** Occurs when a test method contains one or more control statements, whereas it should be simple and execute all statements unconditionally.
(3) **Constructor Initialization (CI):** Occurs when a test class contains a constructor declaration. Initialization of fields should be in the `setUp()` method.
(4) **Duplicate Assert (DA):** A test method that contains more than one assertion statement with the same parameters.
(5) **Empty Test (ET):** Occurs when a test method does not contain executable statements.

[3]https://github.com/viRafael/arieslinter

(6) **Exception Handling (EH):** Occurs when the pass/fail status of a test depends on an exception being thrown by the production or test method, instead of using proper testing framework constructs.
(7) **General Fixture (GF):** Occurs when a test case fixture is too general and the test methods only access part of it. Not all fields instantiated within the `setUp()` method of a test class are utilized by all test methods in the same test class.
(8) **Ignored Test (IgT):** Occurs when test methods are deliberately excluded from execution using tags such as `@Ignore` or `@Disabled`.
(9) **Magic Number (MN):** Detected when assert statements contain numeric literals that lack semantic meaning, reducing test clarity.
(10) **Mystery Guest (MG):** Occurs when a test method utilizes external resources (e.g. files, database, etc.). Use of external resources in test methods will result in stability and performance issues.
(11) **Redundant Assertion (RA):** This smell occurs when test methods contain assertion statements that are either always true or always false.
(12) **Redundant Print (RP):** Occurs when a test method includes print statements, which are typically unnecessary and clutter test output.
(13) **Resource Optimism (RO):** This smell occurs when a test method makes an optimistic assumption that the external resource (e.g., File), utilized by the test method, exists.
(14) **Sensitive Equality (SE):** Detected when the `toString()` method is used within a test method for assertions, which may be sensitive to representation changes.
(15) **Sleepy Test (ST):** Occurs when a test method includes explicit thread sleep calls, which may cause flakiness due to differing processing times across environments.
(16) **Unknown Test (UT):** Occurs when a test method lacks assertions, causing JUnit to mark it as passed unless an exception is thrown.
(17) **Verbose Test (VT):** A test method that exceeds 30 lines of code.

4 Evaluation of AriesLinter Detection

This study evaluates the detection performance of AriesLinter across the 17 test smell types currently supported by the tool. To guide the evaluation, we investigate the following research question:

RQ1 How accurately does AriesLinter detect the supported test smell types in terms of precision, recall, F1-score, and accuracy?

4.1 Building a Curated Dataset of Test Smells

The empirical study was conducted on 10 open-source projects from the Apache Software Foundation. These projects were selected because they are actively maintained, widely adopted, and representative of mature Java software systems.

To construct the curated dataset, we first cloned the repositories of the selected Apache projects and analyzed their test suites using JNose [17]. For each of the 17 test smell types supported by AriesLinter, we randomly selected ten test methods containing the smell and ten methods without it based on JNose's detection results. Whenever possible, the selected methods were drawn from different projects to increase the diversity of the dataset.

Two smell types had fewer available instances across the selected projects. Consequently, the final dataset includes nine instances of *Mystery Guest* and six instances of *Redundant Print*, together with their corresponding non-smell examples. Overall, the curated dataset comprises 335 test methods.

Each selected example was independently inspected by three researchers using the test smell definitions reported in the literature [9, 11, 18]. JNose was used only to identify candidate instances, while the final classification of each example was established through manual inspection. The resulting dataset served as the reference oracle for evaluating the detection performance of AriesLinter.

4.2 Evaluation of the Accuracy of the Tool

Using the manually curated dataset earlier described in Section 4.1, we applied AriesLinter to each test method and compared its predictions with the reference oracle. For each test smell type, we computed the numbers of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). These values were then used to calculate accuracy, precision, recall, and F1-score to answer RQ1.

5 Qualitative Evaluation of AriesLinter by Developers

To evaluate developers' perceptions of AriesLinter's real-time detection capabilities, we formulated the following research question:

RQ2 How do developers perceive the usefulness of real-time test smell detection during software development?

To answer RQ2, we conducted a survey involving 17 developers with experience in Java and the JUnit testing framework.

5.1 Procedure and Instrumentation

The study involved 17 developers and was conducted in six stages. First, participants completed a questionnaire about their experience with Java, JUnit, and test smells. Next, they received an introduction to AriesLinter and to the 17 supported test smell types. They then installed the tool, explored its functionality through example cases, and completed the proposed activities. Finally, participants answered a questionnaire evaluating their perceptions of AriesLinter and its real-time detection capabilities.

Participants were recruited using convenience sampling. All were affiliated with the same university but represented different stages of academic training and varying levels of professional experience. Seven participants were undergraduate students, seven were master's students, and three were Ph.D. students. Participants reported between 0 and 144 months of professional experience. Industry and research experience were not mutually exclusive: 12 participants had industry experience, 14 had research experience, and several reported experience in both settings.

Participants reported an average of 61 months of software development experience, including 31 months of Java development

and 28 months of automated testing experience. Regarding familiarity with test smells, 81.8% reported prior knowledge of the concept, while the remaining 18.2% indicated partial familiarity. Consequently, all participants had at least a basic understanding of test smells before using AriesLinter.

To keep the study within a manageable duration, we selected nine representative test smell types from those supported by AriesLinter. For each smell, participants were presented with a code example together with its definition and a brief explanation. After reviewing each example, they answered the same set of questions to assess their perceptions of the impact of the corresponding test smell on test code quality:

- **A1.** "Do you believe this test smell can compromise the quality of test code?"
- **A2.** "If you encountered this test smell while maintaining a test suite, how would you respond?"
- **A3.** "On a scale from 0 (no impact) to 10 (maximum impact), how would you rate the impact of this test smell on test code quality?"

After completing the first questionnaire, participants watched a short demonstration showing how to install and use AriesLinter. They were then asked to evaluate the tool by answering a second questionnaire covering usability, perceived usefulness, and the benefits of real-time test smell detection:

- **A4.** "How would you rate your experience installing and configuring AriesLinter?"
- **A5.** "Do you consider real-time test smell detection during code writing useful?"
- **A6.** "Do you believe real-time test smell detection improves the development of test cases?"
- **A7.** "Would immediate feedback from AriesLinter encourage you to refactor test code as you write it?"
- **A8.** "Do you consider real-time detection to be one of the main advantages of AriesLinter?"
- **A9.** "Do you consider real-time test smell detection more useful than post hoc detection performed after the code has been written?"
- **A10.** "Would you use AriesLinter as part of your regular development workflow?"

5.2 Understanding the Developers' Perceptions

Responses to A1 indicate that participants consistently perceived the presented test smells as harmful to test code quality. Although agreement was unanimous for some examples, a small number of participants disagreed for EX2, EX3, EX7, EX8, and EX9, with 2, 1, 3, 1, and 1 negative responses, respectively. Overall, these results suggest a broad consensus regarding the negative impact of the evaluated test smells.

Responses to A2 show that most participants would refactor the test code when encountering the presented test smells. Even for the example with the lowest level of agreement, 64.1% of participants indicated that they would perform a refactoring, suggesting broad acceptance of the need to address these issues.

Responses to A3 exhibited greater variability. Participants assigned different impact scores to the presented test smells, indicating that the perceived severity of individual smells is not uniform across developers.

In the qualitative analysis, all responses were quite positive. We obtained the following data:

- (A4) 5.9% rated the installation as "very easy", 29.4% as "easy", 47.1% were neutral, and the rest considered the process "difficult" or "very difficult";
- (A5) 64.7% responded "very useful", 29.4% "useful", and 5.9% "neutral";
- (A6) 76.5% answered "definitely yes", 17.6% "probably yes", and 5.9% "neutral";
- (A7) 70.6% answered "definitely yes", 23.5% "probably yes", and 5.9% "neutral";
- (A8) 88.2% said "yes", and 11.8% answered "no";
- (A9) 88.2% said they were "in favor", and 11.8% were "uncertain";
- (A10) 88.2% said they would use it, and 11.8% said they might use it.

6 Discussion

In this section, we discuss the results obtained from the two studies presented here, aiming to answer the research questions.

RQ1 How accurately does AriesLinter detect the supported test smell types in terms of precision, recall, F1-score, and accuracy?

Table 1 summarizes the detection performance of AriesLinter for the 17 supported test smell types using the manually curated oracle described in Section 4.1. Performance is reported in terms of true positives (TP), false positives (FP), true negatives (TN), false negatives (FN), accuracy, precision, recall, and F1-score.

Table 1: Performance of the AriesLinter tool in Test Smell detection

TS	TP	FN	FP	TN	Accuracy	Precision	Recall	F1-Score
AR	10	0	0	10	1.00	1.00	1.00	1.00
CTL	10	0	0	10	1.00	1.00	1.00	1.00
CI	10	0	0	10	1.00	1.00	1.00	1.00
DA	10	0	0	10	1.00	1.00	1.00	1.00
ET	10	0	0	10	1.00	1.00	1.00	1.00
EH	10	0	0	10	1.00	1.00	1.00	1.00
GF	10	0	0	10	1.00	1.00	1.00	1.00
IgT	10	0	0	10	1.00	1.00	1.00	1.00
MN	10	0	0	10	1.00	1.00	1.00	1.00
MG	9	1	0	10	0.95	1.00	0.90	0.95
RA	10	0	0	10	1.00	1.00	1.00	1.00
RP	6	0	0	10	1.00	1.00	1.00	1.00
RO	10	0	0	10	1.00	1.00	1.00	1.00
SE	10	0	0	10	1.00	1.00	1.00	1.00
ST	10	0	0	10	1.00	1.00	1.00	1.00
UT	10	0	0	10	1.00	1.00	1.00	1.00
VT	10	0	0	10	1.00	1.00	1.00	1.00

AriesLinter achieved perfect precision, recall, and F1-score for 16 of the 17 evaluated test smell types. The only exception was *Mystery Guest*, for which one false negative resulted in a recall of 0.90 and an F1-score of 0.95. Overall, these results indicate that AriesLinter accurately detects the supported test smell types on the curated dataset used in this study.

The high detection performance is largely due to the rule-based architecture adopted by AriesLinter. Built on top of CheckStyle, the tool analyzes the Abstract Syntax Tree (AST) of Java test code to identify well-defined syntactic and structural patterns. For example, smells such as *Verbose Test* are detected by analyzing the size of test methods, whereas *Redundant Print* is identified through explicit calls to `System.out.println()`. Because these smells can be characterized by deterministic structural rules, the tool produced no false positives for the evaluated dataset.

These results suggest that, for the syntactic patterns currently supported by AriesLinter, real-time detection can be achieved without compromising detection performance.

In summary, the results provide a positive answer to RQ1. AriesLinter achieved perfect detection performance for 16 of the 17 evaluated test smell types, producing no false positives and only a single false negative, observed for the *Mystery Guest* smell. These findings indicate that the AST-based detection rules implemented in AriesLinter can accurately identify the supported test smell types while providing real-time feedback during test development.

RQ2 How do developers perceive the usefulness of real-time test smell detection during software development?

The results provide a positive answer to RQ2. Overall, participants perceived real-time test smell detection as a useful feature that can support the development and maintenance of test code. Most participants reported that immediate feedback would encourage them to address test smells during development and indicated that they would be willing to incorporate AriesLinter into their regular development workflow.

The responses consistently indicate a positive perception of real-time test smell detection. Most participants (94.1%) considered the feature useful or very useful during code writing, while 76.5% stated that it positively influences the development of test cases. Furthermore, 88.2% preferred real-time detection over post-development analysis, and 70.6% reported that immediate feedback would encourage them to refactor test code as soon as a smell is detected.

Participants' comments reinforce these findings. Several respondents emphasized that immediate feedback helps prevent test smells from being introduced in the first place, avoiding the need for later refactoring. Others highlighted the benefit of identifying quality issues while writing tests rather than after execution, integrating quality assurance more naturally into the development workflow. Representative comments include:

- "It prevents the inclusion of smells and eliminates the need to refactor the test code later."
- "Not having to run the tests to realize something was done wrong, and already being able to write the tests correctly from the beginning."
- "It is essential to have tools that enable the detection of issues in test code. In this regard, I find the proposal excellent."

This study has some limitations that should be considered when interpreting its results. The survey employed convenience sampling and involved 17 participants, all affiliated with the Federal University of Bahia (UFBA), which limits the generalizability of the findings to other populations and industrial settings. Nevertheless, the participant pool included individuals at different stages of academic training and with varying levels of professional experience, including both industry and research experience. Although the results cannot be generalized, they provide encouraging evidence that developers perceive value in integrating real-time test smell detection into their development workflow. Further studies involving larger and more diverse samples, particularly from industry, are needed to strengthen these findings.

7 Related Work

One of the most widely used tools for test smell detection is **tsDetect** [11]. The tool identifies up to 21 test smell types by analyzing Java test code through an Abstract Syntax Tree (AST) generated with JavaParser. Although tsDetect provides comprehensive support for test smell detection, it operates as a standalone command-line tool and analyzes only completed source code. Consequently, developers receive feedback only after implementation, rather than during the coding process.

JNose extends the functionality of tsDetect by providing a web-based interface and additional analyses, such as code coverage and the evolution of test smells across software repositories [17]. Despite these improvements, JNose remains a standalone analysis tool that requires manual execution on completed source code. Consequently, it supports post-development quality assessment rather than providing immediate feedback during test implementation. In contrast, AriesLinter integrates with the development environment and reports test smells as the code is written.

RAIDE is an Eclipse plugin that supports the detection and semi-automated refactoring of two test smell types: *Assertion Roulette* and *Duplicate Assert* [14]. Unlike standalone tools such as tsDetect and JNose, RAIDE integrates test smell detection into the development environment. However, it is limited to Eclipse, supports only two test smell types, and relies on user interaction to apply refactoring suggestions. In contrast, AriesLinter supports 17 test smell types and can be used in any IDE that provides CheckStyle integration, offering immediate feedback during test development.

8 Threats to Validity

Internal Validity. The main threat to internal validity concerns the construction of the reference oracle used to evaluate AriesLinter. Candidate examples were initially identified with JNose and subsequently inspected manually by three researchers based on established test smell definitions, reducing the risk of relying on a single labeler's judgment. Disagreements among researchers were resolved through discussion before a final classification was assigned. Even so, manual classification of test smells remains inherently subject to human judgment, and residual labeling noise may still affect the reported performance metrics.

External Validity. The external validity of this study is limited by the evaluation setting. The quantitative evaluation was conducted on 10 Apache Software Foundation projects, chosen for being actively maintained and representative of mature Java systems, but this sample may not capture the diversity of industrial codebases, testing conventions, and team practices found in professional settings. Moreover, AriesLinter currently supports only 17 test smell types, preventing conclusions from being generalized to other smells not covered by the tool. Finally, the developer study involved a convenience sample of 17 participants from a single

university; although the sample spanned different academic levels and included participants with industry experience, it may not fully reflect the perceptions of professional developers working under real-world constraints such as deadlines and legacy codebases. Extending the quantitative evaluation to a larger and more diverse set of projects, and replicating the developer study with industry practitioners, are necessary steps to strengthen the generalizability of these findings and are part of our planned future work.

Construct Validity. The implemented detection rules may not fully capture the conceptual definitions of test smells reported in the literature. Although the implementation was guided by established definitions and manually inspected by researchers, differences between conceptual descriptions and executable detection rules may still exist, particularly for smells whose definitions allow room for interpretation. In addition, participants' responses in the qualitative study may have been influenced by the wording of the questionnaire and by the presentation materials used during the experiment, a risk we mitigated by using the same materials and question set for all participants.

9 Conclusion

This paper presented AriesLinter, a tool for real-time detection of test smells in Java test code, and evaluated it through two complementary empirical studies. The first study assessed the detection performance of the tool using a manually curated oracle, while the second investigated developers' perceptions of real-time test smell detection.

The quantitative evaluation showed that AriesLinter achieved perfect detection performance for 16 of the 17 supported test smell types, with only one false negative observed for *Mystery Guest*. The developer study also produced encouraging results: most participants considered real-time detection useful during test development, preferred it over post-development analysis, and indicated that immediate feedback would encourage earlier refactoring and the adoption of the tool in their regular development workflow.

Taken together, these findings suggest that integrating test smell detection into the development environment is both technically feasible and positively perceived by developers. Rather than replacing existing post-development analysis tools, AriesLinter complements them by providing immediate feedback while test code is being written.

As future work, we plan to expand the tool to support additional test smell types, investigate automated refactoring recommendations, and conduct larger empirical studies in industrial settings to evaluate the long-term impact of real-time test smell detection on software quality and developer practices.

Artifact Availability

The AriesLinter source code is licensed under the MIT License and is available on GitHub at https://github.com/viRafael/arieslinter. The dataset and replication package are publicly available at https://zenodo.org/records/20945473. The video is available on Zenodo: https://doi.org/10.5281/zenodo.15484942.

Previous Work

This paper is an extended version of a previous publication presented at the *SBES Tools Track 2025*. The original publication is available at: https://doi.org/10.5753/sbes.2025.11576.

References

[1] 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.

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

[3] 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

[4] 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.

[5] 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

[6] 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

[7] Stephen C. Johnson. 1988. Lint, a C Program Checker.

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

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

[10] 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.

[11] 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

[12] 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

[13] Rafael Rocha, Eronildo Junior, Tássio Virgínio, Larissa Rocha, Carla Bezerra, and Ivan Machado. 2025. AriesLinter: Sniffing Test Smells Before They Happen. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 990–995. doi:10.5753/sbes.2025.11576

[14] 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

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

[16] 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

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

[18] 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