DANTES: Automating Detection and Refactoring of Test Smells

Daniel Bruno Ramos
Institute of Computing
Federal University of Bahia
Salvador, Bahia, Brazil
daniel.bruno@ufba.br

Railana Santana
Institute of Computing
Federal University of Bahia
Salvador, Bahia, Brazil
railana.santana@ufba.br

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

ABSTRACT

Test smells affect software quality and hinder code maintenance. Refactoring test code is the main way to address these problems. However, manually refactoring is not a simple task and is usually tedious and error-prone. This study aims to present DANTES, a tool for automatically detecting and refactoring eleven different types of test smells. We introduce eleven original refactoring algorithms. We conducted a study to evaluate the tool from the perspective of perceived usefulness and ease of use. Results show that DANTES is well received by professionals and academics in the field.
Demo video: https://youtu.be/uc5G6JMXtBs

KEYWORDS

Software Quality, Software testing, Test smells, Anti-patterns, Refactoring

1 Introduction

The evolution of software has increased its complexity [8], influencing a growing interest in strategies to ensure quality, identify defects, and facilitate maintenance [13, 16]. Among testing approaches, unit tests automatically evaluate individual functions based on specific inputs and expected outputs, ensuring code integrity [14, 17, 18]. However, writing unit tests is challenging, as it requires a deep understanding of the production code, mastery of good design practices, and rigor in implementation [9, 15, 26]. Often, this task is more complex than the software development itself.

Like production code, test code must also follow good design practices [6, 21, 31]. In this context, Van Deursen et al. [27] defined eleven antipatterns that compromise the quality of test code, known as test smells, and proposed refactoring techniques to correct them. Refactoring involves transforming code to improve its structure without altering its external behavior [12]. However, it is a complex process that requires an in-depth understanding of the software and care not to introduce new problems [6, 10], evidencing the need for tools capable of automatically detecting and applying refactorings to test code with test smells.

Several tools have been developed for detecting test smells [1], such as Test Smell Detector (TSDETECT [20]), JNOSE TEST [28], and RAIDE [22]. A few of these tools also offer the feature of automatically or semi-automatically refactoring test code, such as RAIDE.

These efforts are important for advancing test smell detection and refactoring tools. However, they have limitations, such as the lack of automated refactoring for most test smells, difficulty in locating the smells within the analyzed class, and the inability to refactor an entire class simultaneously.

This paper proposes the tool DANTES[1] (*Detection And Neutralization of Test Smells*), a web application that detects and automatically refactors eleven types of test smells in *Java* test code using the *JUnit* framework.

DANTES is an open-source[2] tool developed based on the source code of the RAIDE and TSDETECT tools, and contributes to the state of the art by providing eleven original algorithms for refactoring test smells. DANTES compares the original test code with the automatically refactored test code, allowing the user to track the significant changes made by the tool. Hence, this paper aims to provide a tool that supports developers and testers in enhancing their efficiency in daily activities.

To evaluate DANTES, we conducted an experimental study with 17 professionals experienced in software development and testing, specifically in *Java* and the *JUnit* framework. Through this study, we obtained valuable insights into the perceived usefulness and ease of use of the DANTES tool.

In summary, this study presents the following contributions: (i) we applied rule-based strategies to detect eleven types of test smells; (ii) we automated four literature-based refactorings to remove four test smells; (iii) we proposed seven new refactorings for seven test smells; (iv) we released an open-source web tool to detect and refactor eleven types of test semi-automatically smells in a test class; and (v) we conducted an experiment to evaluate the tool from the perspective of professionals regarding perceived usefulness and ease of use.

2 DANTES Tool

DANTES is a web tool that analyzes occurrences of test smells in test code. It performs text analysis in test code submitted by the user through the tool's UI. All test smells detected and their respective refactorings are provided in a graphical interface. After the refactoring process, the tool displays the comparison between the submitted code and the corrected code, helping the user understand the changes.

The current version of DANTES is intended exclusively for unit test code written in the *Java* language using the *JUnit* framework. The choice of language can be justified by its high popularity, in 2024 *Java* was among the five most used programming languages[7].

DANTES detects and provides semi-automated refactorings for eleven test smell types: *Assertion Roulette (AR); Constructor Initialization (CI); Duplicate Assert (DA); Empty Test (ET); Exception Handling (EH); General Fixture (GF); Ignored Test (IT); Magic Number Test (MNT); Mystery Guest (MG); Redundant Assertion (RA); and Unnecessary Print (UP).*

[1] Available at https://github.com/arieslab/DANTES
[2] DANTES tool is under the GNU General Public License.

DANTES was designed as a *web tool*, meaning it is hosted on a website. After analyzing the tools available in the literature and evaluating their strengths and weaknesses, we built an IDE-independent tool to avoid limiting the use of DANTES.

Research on other test smell detection and refactoring tools greatly contributed to understanding how the tool should work. Few tools operate as a *web tool*, such as JNOSE TEST. Most tools are command-line based, while others work as IDE *plugins*. Thus, the fact that DANTES is a *web tool* was a factor in determining the choice of supported test smells.

2.1 Limitations

Being a web tool, DANTES offers advantages like *plug-and-play* functionality, allowing users to utilize it without complex installation. Users can access the website, paste code, quickly visualize, and refactor *smells*.

However, this design also introduced limitations; accessing the production code used by the test class became unviable, preventing the implementation of detection and refactoring for test smells such as *Eager Test* and *Lazy Test*. Furthermore, unlike tools such as DARTS and RAIDE, which use IDE plugin APIs (*IntelliJ* and *Eclipse*, respectively), DANTES lacks access to these. This necessitated a text manipulation-based refactoring approach.

In scenarios with a high density of test smells (for example, 100 or more in a single test class), the tool may experience slowness due to performance limitations inherent in its current version, taking more seconds to complete the refactoring.

Furthermore, refactoring the MNT test smells still presents limitations, the tool does not extract numeric values when embedded in function calls.

2.2 Development

The development planning for the DANTES tool was based on the study of other *test smell* detection and refactoring tools, such as RAIDE, JNOSE TEST, and TSDETECT. Based on the analysis of how these tools function, a development plan for DANTES was drawn up. The tool's structure consists of three integrated modules: the client, the server, and a *Java* application. Figure 1 presents a diagram illustrating the interaction between these three modules. These modules are described below.

2.2.1 ***Java Application.*** The RAIDE tool significantly influenced this tool's development. RAIDE, an *Eclipse IDE* plugin, was developed in *Java* using Eclipse-specific functionalities.

DANTES, a standalone web tool, does not utilize RAIDE's technologies. Its web interface and server primarily uses *HTML* and *JavaScript*. Consequently, RAIDE's code was adapt-ed into a standalone *Java* command-line application for DANTES, reusing much of RAIDE's code to detect test smells.

This module contains the complete functionality for detecting and refactoring all test smells supported by DANTES. Ten detection algorithms were reused from RAIDE, and one was newly implemented for DANTES. However, all refactoring code for the eleven test smells was developed specifically for this project.

The application operates by reading a `.java` file, generated by the server, which contains the code for test smell detection and refactoring. The program outputs structured *JSON* for client-side data handling. When code is refactored, the `.java` file is also updated to ensure consistency between the user display and the tool's processed data.

2.2.2 ***Client.*** DANTES has a client-server architecture, with the client handling user interaction and the server processing client requests. For this purpose, the *NodeJS* framework was chosen for server implementation. The client is responsible for the user interface. It includes the web page with a text box for user input and buttons to submit the code for analysis and to upload a file into the text box.

When the submission button is pressed, the code is sent to the server, which returns to the client a listing of all the test smells found in the provided code. Additionally, the code is displayed on the page with the lines containing test smells marked in red.

For each listed test smell instance, the client then generates a line describing the test smell it is, the method test, and on which line, along with a button to refactor that test smell. Furthermore, at the end of the listing of all test smells, the user can click a button to refactor all test smells at once.

When a button is pressed to refactor a *test smell*, the command is sent to the server, which returns the refactored code displayed alongside the original code with the refactored line(s) in green. DANTES also provides quality-of-life features such as a control for sorting the display of test smells, a day/night theme toggle button, and buttons to copy code.

2.2.3 ***Server.*** The server, in turn, is responsible for receiving the code from the client and returning the results of the operations. Upon receiving the code from the client, the server generates a `.java` file that will be read by the *Java* application to perform detection and refactoring of the test smells. The server then creates a process to execute the *Java* application with the necessary parameters for each functionality and receives the operation result.

Detection. The programming logic responsible for detecting test smells and applying appropriate refactorings to the code is encapsulated in a standalone *Java* application.

For ten of the eleven types of test smells covered by DANTES, the detection implementation was adapted from other well-known tools such as RAIDE [22] and TSDETECT [20]. In these cases, *test smell* detection is executed via Abstract Syntax Trees (ASTs). This approach enables a structured and detailed analysis of the examined code, identifying specific patterns associated with each smell.

However, the detection of the EH test smell was conceived as an original implementation in DANTES. In this particular case, the tool does not use ASTs. Instead, it employs a textual search strategy within the test source code, looking for keywords and specific information that characterize this *smell*. This approach was developed to allow precise and efficient detection of EH in tests, even without structural AST analysis.

Refactoring. The refactorings for the eleven test smells in DANTES were originally implemented based on strategies from the literature. Some refactorings adapted documented strategies, like for AR (adding a *Lazy Assertion Message*) and MG (using *@TempDir*), both suggested by Soares et al. [25], by transforming suggestions into algorithms.

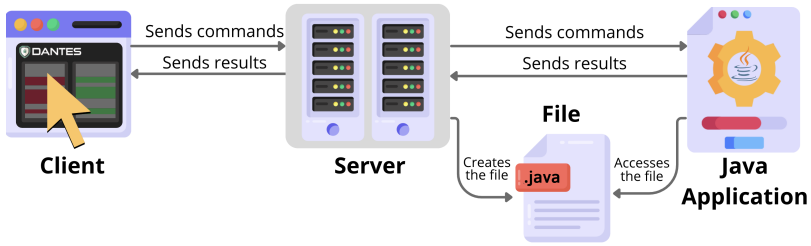

Figure 1: Architecture modules

Listing 1: Refactoring *EH* test smell

```
List<String> newLines = new ArrayList<>();
newLines.add(tabSpace + "assertThrows(" + tryCatchInfo.exception + ",");
if (tryCatchInfo.tryContent.size() == 1) {
    String line = tryCatchInfo.tryContent.get(0).trim();
    newLines.add(tabSpace + doubleTab + "() -> "
        + line.substring(0, line.length() - 1) + ");");
} else {
    newLines.add(tabSpace + doubleTab + "() -> {");
    for (int i = 0; i < tryCatchInfo.catchContent.size(); i++) {
        newLines.add(tabSpace + doubleTab + singleTab
            + tryCatchInfo.tryContent.get(i).trim());
    }
    newLines.add(tabSpace + doubleTab + "});");
}
for (int i = 0; i < tryCatchInfo.catchContent.size(); i++) {
    if (!tryCatchInfo.catchContent.get(i).trim()
            .toLowerCase().startsWith("assert")) {
        newLines.add(tabSpace + tryCatchInfo.catchContent.get(i).trim());
    }
}
```

Table 1: Original contributions in Test Smells

Test Smell	Detection	Refactoring	Algorithm
AR	RAIDE [22]	Van Deursen et al. [27]	Original
CI	RAIDE [22]	Original	Original
DA	RAIDE [22]	Original	Original
ET	RAIDE [22]	Original	Original
EH	Original	Soares et al. [25]	Original
GF	RAIDE [22]	Van Deursen et al. [27]	Original
IT	RAIDE [22]	Original	Original
MNT	RAIDE [22]	Original	Original
MG	RAIDE [22]	Soares et al. [25]	Original
RA	RAIDE [22]	Original	Original
UP	RAIDE [22]	Original	Original

For others, such as DA, ET, IT, RA, and UT, the refactoring strategy involved removing affected lines to simplify test code, chosen for efficiency.

Unlike the AST-based detection algorithms adapted from RAIDE and TSDETECT, refactoring strategies were implemented using algorithms based on textual code manipulation. Listing 1 exemplifies this with the *EH test smell* refactoring, where the algorithm gathers *try/catch* information and generates new code lines. The refactoring algorithms for other test smells operate similarly by removing, adding, and modifying code lines to eliminate the smell.

2.3 Test Smells

DANTES reuses code from RAIDE, which in turn uses code from TSDETECT. Table 1 describes the origins of the detection strategies, refactorings, and algorithms implemented in DANTES, distinguishing reused and original functionalities. For example, the GF test smell's detection was based on the RAIDE tool, its refactoring followed the recommendation of Van Deursen et al. [27], and the algorithm rule for refactoring was developed from scratch to address this test smell, i.e., originally by DANTES.

As shown in Table 1, most of the detection algorithms were adapted from the RAIDE tool [22], while all refactoring algorithms were implemented originally for DANTES, based on refactoring strategies of literature [25, 27].

AR. Detection of this test smell involves analyzing assertion calls in test methods to find those lacking explanatory messages. To refactor this smell, DANTES suggests the "*Add Assertion Explanation*" refactoring, which uses *JUnit* 5's *Lazy Assertion Message* feature. This feature allows inclusion of a lambda that supplies a detailed message explaining the purpose or behavior of each assertion. DANTES parses the line to be refactored and adds the message parameter to the assertion call.

CI. To detect this test smell, DANTES checks for the presence of a constructor method. This smell is refactored via the "*Refactor to setUp()*" strategy, in which the constructor is renamed and converted into a *setUp()* method.

DA. This detection involves comparing assert calls within a method to find identical invocations. To refactor, DANTES removes the duplicate call, a refactoring called "*Remove Line*." While some studies propose alternative refactorings, here we remove identical assertions in the same method, since duplicate assertions add redundancy and hinder readability.

ET. Detection of this *test smell* works as follows: the tool checks if the test method contains any executable statements; if not, the method is marked as containing the smell. To refactor this smell, DANTES removes the method via the "*Remove Method*" refactoring. Since an empty or commented-out test pollutes the test suite, removal is recommended.

EH. Detection of this *test smell* involves checking for the presence of a *try/catch* structure and verifying whether an *assert* call exists within the *catch* block. To refactor this smell, DANTES performs the "*Change to assertThrows*" refactoring, which consists of extracting the contents of the *try/catch* blocks and the expected exception, and then rewriting the code snippet using *JUnit*'s *assertThrows()* feature to verify the thrown exception.

GF. DANTES inspects all variables initialized in the *setUp()* method and determines which are used in every test method; those not used in all tests are marked as a GF. To refactor this test smell, it is necessary to extract all lines of code involving the initialized variable and then insert those lines only in the methods that actually

use the variable. We name this refactoring "*Make Fixture Unique*," as suggested by Van Deursen et al. [27].

IT. Detection of this *test smell* is simpler than for other smells, since DANTES checks if the method is annotated with *@Ignore*; if so, the method is classified as an IT. Once detected, refactoring is performed by removing the method using the "*Remove Method*" refactoring, because methods annotated with *@Ignore* are always skipped during test execution. Removing this method does not affect test outcomes.

MNT. This test smell involves the detection of literal numbers in assertion calls. To refactor this smell, DANTES applies the "*Create Local Variable*" refactoring, in which the tool extracts the literal value, infers its type, declares a variable with that type and value, and replaces the literal in the assertion call with the new variable.

MG. DANTES detects lines that read a file via a `File` call. It uses the "*Use Temporary Directory*" refactoring, leveraging the *@TempDir* annotation on the method to obtain a temporary file.

RA. Detection of this *test smell* involves checking if an assertion call verifies a tautology or contradiction. In such cases, refactoring is performed by removing the offending line via the "*Remove Line*" refactoring, since these tests contradict the purpose of verifying behavior changes in production code.

UP. Detection of this *test smell* identifies `print` calls in the code. Refactoring consists of removing these lines via the "*Remove Line*" refactoring, as such print statements are irrelevant in test code.

Refactored test code examples produced by DANTES are available in the *Artifact Availability* section.

3 Example of Use

Figure 2 presents a screenshot of the DANTES tool. The numbers highlighted in Figure 2 represent crucial elements or features of DANTES. Each of these elements/features is described in detail: **(1)** The primary text box of this tool represents the area where the user can enter or paste the desired test code to be submitted for analysis by DANTES ; **(2)** This button allows the user to switch between visualization themes, offering both light and dark theme options; **(3)** The "*Detect*" button must be clicked for the tool to perform the detection of test smells in the provided code; **(4)** The user can click the "*Upload a file*" button to select a specific file, allowing its contents to be directly inserted into the text box; **(5)** In this *dropdown* menu, the user can choose different ways to sort the display of the detected test smells: *(5.1)* When selecting "*Order by Position in Code*", the results will be sorted based on the sequence in which the *smells* appear in the code; *(5.2)* When choosing "*Order by Smell Type*", the sorting will be done alphabetically by the names of the test smells; **(6)** After detecting the test smells, this box will be updated with a list of all test smells found; **(7)** The following section will present the submitted code, highlighting in red the lines that contain test smells; **(8)** After applying any refactorings, the revised code will be displayed in this area, with the refactored lines highlighted in green; **(9)** The user can click this button to copy the entire content of the displayed code to the clipboard.

Figure 3 presents the result of a detection operation. The numbers in Figure 3 highlight some important elements of the tool's interface: **(1)** After executing the *test smell* detection analysis, the DANTES tool displays the number of *test smell* instances detected in the

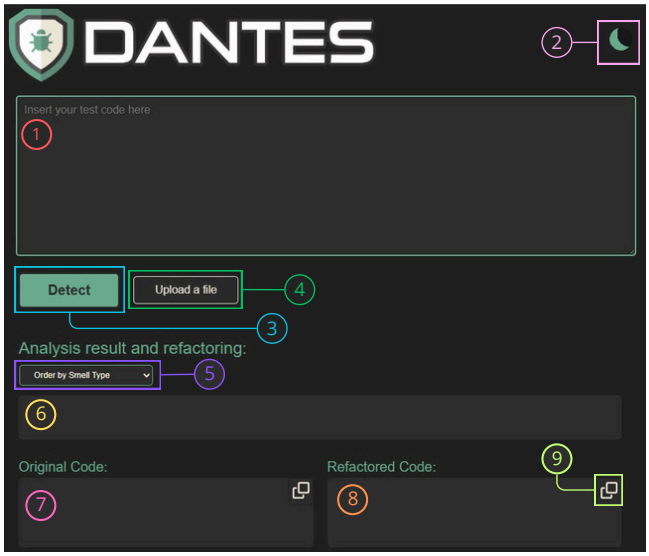

Figure 2: Steps to run the DANTES

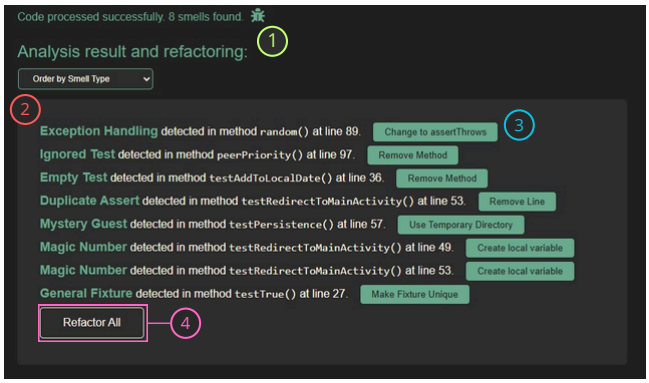

Figure 3: Example of detected test smells by DANTES

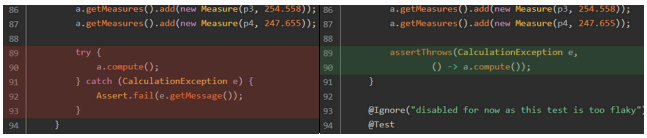

Figure 4: Refactoring example by DANTES

class; **(2)** Within the designated area, each line corresponds to an identified instance of a test smell present in the analyzed code. Each line details the specific type of *smell*, the method in which it was found, and the corresponding line of code where it occurs. **(3)** For each instance of a *smell*, a green-colored button is provided, associated with a specific refactoring action intended to mitigate that particular *smell*. By pressing this button, the tool will execute the corresponding refactoring action, as labeled on the button itself. **(4)** By clicking a specific button, all identified occurrences of test smells in the code will be refactored simultaneously, providing a comprehensive refinement approach.

Figure 4 illustrates a refactoring performed by DANTES to eliminates the EH test smell, showing the original code (in red) and the refactored version (in green).

4 Tool Evaluation

To evaluate our approach, we considered the users' perception of the DANTES tool, focusing specifically on the test smell detection and refactoring functionalities. The evaluation was guided by the Technology Acceptance Model (TAM) [5], recognized for explaining the acceptance behavior of end users about a specific technology. The TAM evaluates three primary constructs [3]: (1) *perceived usefulness*, (2) *perceived ease of use*, and (3) *self-predicted future use*.

We invited 17 industry practitioners to participate in the DANTES evaluation, all with experience in *Java*, *JUnit*, automated testing, and code refactoring. The selection of participants was made through social networks and direct contacts, e.g., LinkedIn. The participants were undergraduate students (35.3%), professionals (23.5%), masters (23.5%), and master's students (17.6%). Most had between 1 and 5 years of experience in *Java*, testing, and *JUnit*. The individual profiles of the participants are available in the *Artifact Availability* section.

The evaluation overview is illustrated in Figure 5. We applied two questionnaires: the first (Q1) dealt with demographic data and experience of the participants; the second (Q2) was divided between practical tasks (Q2.1) with the tool and an evaluation based on the TAM (Q2.2).

After receiving an invitation and accepting to participate in the study, participants received Q1. Next, Q2 was sent with instructions for installing the tool and performing the tasks. Participants used the tool with minimal intervention from researchers.

During Q2.1, participants performed four tasks with the DANTES tool using test files written in *Java* and *JUnit*. Each task included up to 4 objective questions (14 in total). The questions in Q2 were designed according to the execution of the tasks, that is, about the results obtained with the tool, such as: "*How many test smells were detected?*" or "*Which method had the most smells?*".

In the second part of Q2, participants answered questions about the perceived usefulness, ease of use, and self-predicted future use of the tool, following the six-point Likert scale, in which each answer represents the degree of probability of usefulness and ease of use of DANTES. Table 2 shows the questions and average scores assigned to each question.

One participant with no previous experience in *Java/JUnit* performed similarly to the others, highlighting that DANTES is an easy-to-use tool.

The results of the execution of the tasks (Q2.1) are in Table 3. Most of the questions related to the execution of the functionalities, which involved detection and refactoring, had a correct answer rate above 80%, except for questions 3 and 12, which can be justified by the writing of the statement and the alternatives similar to the correct option. In addition, the results of Q2.2 also showed positive results regarding the tool. Table 2 shows the average evaluations of the tool, all above 5.0, where the maximum score is 6.0.

Through the open-ended questions, we discovered that the tool's intuitive interface, ease of use, and the ability to view the refactored code next to the original code (code highlighted in red and green) are some of the main positive points of the DANTES tool. Among the negative points, participants mentioned the limitation of working with only one class at a time, the lack of integration with the system, and the limited number of test smell types. To adapt to the reality of the industry, participants listed the following improvements: support for more than one type of refactoring for each type of test smell, and the possibility of accessing the system's production code.

Table 2: Qualitative Questions

	Questions related to Perceived Usefulness:	**Score**
PU1	Using the DANTES tool in my work, I would be able to detect/refactor test smells more quickly.	5.5
PU2	Using the DANTES tool would improve my performance in detecting/refactoring test smells.	5.5
PU3	Using the DANTES tool to detect/refactor test smells would increase my productivity.	5.2
PU4	Using the DANTES tool would increase my effectiveness in detecting/refactoring test smells.	5.5
PU5	Using the DANTES tool would make it easier to detect/refactor test smells.	5.6
PU6	I would consider the DANTES tool useful for performing detection/refactoring of test smells.	5.6
	Questions related to Perceived Ease of Use:	**Score**
PEOU1	Learning to operate the DANTES tool would be easy for me.	5.8
PEOU2	I would find it easy to get the DANTES tool to do what I want.	5.4
PEOU3	My interaction with the DANTES tool would be clear and understandable.	5.6
PEOU4	It would be easy to become skillful at using the DANTES tool.	5.8
PEOU5	It would be easy to remember how to detect/refactor test smells using the DANTES tool.	5.8
PEOU6	I would find the DANTES tool easy to use.	5.8
	Questions related to Self-Predicted Future Use:	**Score**
SPFU1	Assuming the DANTES tool was available in my work, I predict I would use it regularly in the future.	5.0
SPFU2	I prefer using the DANTES tool to detect/refactor test smells than not using it.	5.6

5 Related Work

Aljedaani et al. [2] mapped tools and publications related to test smell detection and refactoring, indicating growing interest in the area. This section highlights tools developed to address anti-patterns in testing, all of which support automated detection of test smells in Java code, some of which also perform automated refactoring.

TRex was the first tool identified by Aljedaani et al. [2]. TRex was released in 2006 by Baker et al. [4], and analyzes test code with the *TTCN-3* framework, detecting problems in the test code. Later in 2019, Peruma et al. [19] published the tsDetect tool, a command-line tool that detects 19 test smells in *Java* tests with *JUnit*, using syntax tree and production code. tsDetect only presents boolean results indicating the presence or absence of test smells in a test class, does not specify the affected location, and does not assist in refactoring the test code. In the same year, Virgínio et al. [29] made available JNose Test, a tool that extends tsDetect. JNose Test detects 21 test smells and provides other metrics, such as test coverage. The current version of JNose Test displays the location of the smells and has a web interface [28, 30].

In the context of test code refactoring, some tools are available, e.g., DARTS and RAIDE. DARTS is an IntelliJ IDE plugin, released in

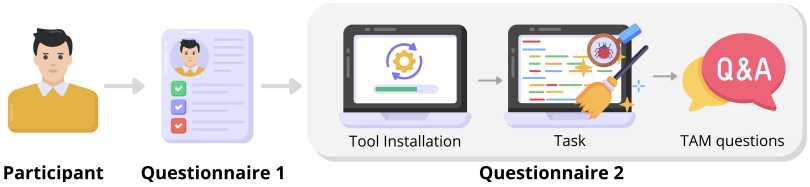

Figure 5: Evaluation Procedure

Table 3: Evaluation tasks

#	Questions	Result
1	How many test smells did the DANTES tool detect in this class?	100.00%
2	In which method and line, respectively, did the DANTES tool detect an instance of the CI test smell?	100.00%
3	Refactor the AR on line 44 INDIVIDUALLY and identify which alternative corresponds to the code of line 44 refactored.	76.47%
4	What is the suggested refactoring for the CI test smell present on line 6?	88.24%
5	What type of test smell was detected on line 18? In which method was this test smell detected?	94.12%
6	The DANTES tool detected two test smells in this class. Mark the alternative that corresponds to one of the test smell occurrences and its respective method found in this class.	82.35%
7	Besides AR, what was the other type of test smell detected in this class?	100.00%
8	Click on "Refactor All" to refactor all test smells, then mark the alternative that contains all the lines that were refactored by the tool:	94.12%
9	After applying "Refactor All", which of the snippets below was removed from the test code?	100.00%
10	Copy the refactored code and then paste it into the tool's text box and click "Detect". How many test smells were found in the class in the second analysis?	88.24%
11	Which method in this class had the most test smells?	100.00%
12	Considering the order of the test code, what are the first and last refactorings suggested by DANTES?	64.71%
13	What were the two different types of test smells that had the same refactoring suggestion in this class?	100.00%
14	Refactor all test smells INDIVIDUALLY, except DA. Mark the alternative that corresponds to the lines that start and end the random() method.	100.00%

Table 4: Comparison of the main tools

Tool	*Framework*	Platform	*Test Smells*	Refactors?
TREX	*TTCN-3*	*Plugin*	27	No
TSDETECT	*JUnit* 4	*CMD*	19	No
JNOSE TEST	*JUnit* 4	*GUI Web*	21	No
DARTS	*JUnit* 4	*Plugin*	3	Yes
RAIDE	*JUnit* 4	*Plugin*	2	Yes
DANTES	*JUnit* 5	*GUI Web*	11	Yes

2020 by Lambiase et al. [11], that detects and refactors three test smells: GF, ET, and Lack of Cohesion of Test Methods. RAIDE [22–24] is an open-source and IDE-integrated tool. RAIDE is a plugin for the *Eclipse IDE* designed to detect and refactor test smells in code written with the *JUnit* framework in *Java* projects. RAIDE supports two test smells: AR and DA.

The DANTES tool evolves from RAIDE, leveraging its code to expand detection and include automated refactoring of eleven types of test smells. Table 4 summarizes and compares the tools mentioned in terms of supported framework, platform, number of test smells detected, and refactoring support. Among the tools mentioned, DANTES stands out for its ability to automatically refactor an entire class with just a few clicks, in addition to allowing the user to compare the test code with smells to the refactored test code.

6 Threats to Validity

Conclusion validity. The lack of comparison with the state of the art represents a limitation in the validity of the conclusions of this study, since there are no automated tools that refactor the same test smells as DANTES. However, the evaluation was based on the performance of the participants in the tasks and on the perspective of perceived usefulness and ease of use of the tool.

Internal validity. A detailed tutorial was provided since all participants used the tool for the first time. The interface was designed based on studies of similar tools to facilitate use, which allowed participants to perform the tasks without assistance (41.2% got all tasks right and 58.8% got three of 14 questions wrong at most).

Construct validity. A pilot study allowed us to improve the evaluation instruments, detailing tasks and questions better for greater clarity for the participants.

External validity. Diversity was considered when selecting participants from different regions of Brazil, with varied profiles (industry, academia, or both) and different levels of education and experience, increasing the generalizability of the results.

7 Conclusion

Test code refactoring is essential, but rarely practiced due to its cost. Automated tools, such as DANTES, become valuable by detecting and refactoring multiple types of test smells, allowing complete refactorings with just one click while offering a user-friendly interface. The tool was evaluated from a usability perspective with 17 participants, and our results indicate good acceptance. We intend to expand its functionality in the future by providing greater coverage of test smells and the possibility of refactoring multiple classes simultaneously. In addition, we consider validating the accuracy of its functionalities.

ARTIFACT AVAILABILITY

The tool evaluation artifact is publicly available at https://zenodo.org/records/16247032.

ACKNOWLEDGMENTS

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

REFERENCES

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

[2] 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. https://doi.org/10.1145/3463274.3463335

[3] Muhammad Ali Babar, Dietmar Winkler, and Stefan Biffl. 2007. Evaluating the Usefulness and Ease of Use of a Groupware Tool for the Software Architecture Evaluation Process. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, Madrid, Spain, 430–439. https://doi.org/10.1109/ESEM.2007.48

[4] Paul Baker, Dominic Evans, Jens Grabowski, Helmut Neukirchen, and Benjamin Zeiss. 2006. TRex - The Refactoring and Metrics Tool for TTCN-3 Test Specifications.. In *Testing: Academic & Industrial Conference - Practice And Research Techniques (TAIC PART'06)*. IEEE, Windsor, UK, 90–94. https://doi.org/10.1109/TAICPART.2006.35

[5] Fred D. Davis. 1989. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly* 13, 3 (1989), 319–340. http://www.jstor.org/stable/249008

[6] Martin Fowler. 2018. *Refactoring*. Addison-Wesley Professional, Boston.

[7] Innovation Graph. 2025. *Programming Languages - Global Metrics*. https://innovationgraph.github.com/global-metrics/programming-languages

[8] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. 2016. Software Testing Techniques: A Literature Review. In *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. IEEE, Jakarta, Indonesia, 177–182. https://doi.org/10.1109/ICT4M.2016.045

[9] Vladimir Khorikov. 2020. *Unit Testing Principles, Practices, and Patterns*. Manning Publications, New York.

[10] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167 (2020), 110610.

[11] Stefano Lambiase, Andrea Cupito, Fabiano Pecorelli, Andrea De Lucia, and Fabio Palomba. 2020. Just-In-Time Test Smell Detection and Refactoring: The DARTS Project. In *Proceedings of the 28th International Conference on Program Comprehension* (Seoul, Republic of Korea) *(ICPC '20)*. Association for Computing Machinery, New York, NY, USA, 441–445. https://doi.org/10.1145/3387904.3389296

[12] Luana Martins, Denivan Campos, Railana Santana, Joselito Mota Junior, Heitor Costa, and Ivan Machado. 2023. Hearing the voice of experts: Unveiling Stack Exchange communities' knowledge of test smells. In *2023 IEEE/ACM 16th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 80–91.

[13] Alok Mishra and Ziadoon Otaiwi. 2020. DevOps and software quality: A systematic mapping. *Computer Science Review* 38 (2020), 100308.

[14] Sanjay Misra, Adewole Adewumi, Rytis Maskeliūnas, Robertas Damaševičius, and Ferid Cafer. 2018. Unit testing in global software development environment. In *Data Science and Analytics: 4th International Conference on Recent Developments in Science, Engineering and Technology, REDSET 2017, Gurgaon, India, October 13-14, 2017, Revised Selected Papers 4*. Springer, Springer, Singapore, 309–317.

[15] Priyadarshi Naik, Kshirasagar e Tripathy. 2016. *Software testing and quality assurance* (2 ed.). John Wiley & Sons, Nashville, TN.

[16] Padmalata Nistala, Kesav Vithal Nori, and Raghu Reddy. 2019. Software Quality Models: A Systematic Mapping Study. In *2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)*. IEEE, Montreal, QC, Canada, 125–134. https://doi.org/10.1109/ICSSP.2019.00025

[17] Tauhida Parveen, Scott Tilley, Nigel Daley, and Pedro Morales. 2009. Towards a distributed execution framework for JUnit test cases. In *2009 IEEE International Conference on Software Maintenance*. IEEE, Edmonton, AB, Canada, 425–428. https://doi.org/10.1109/ICSM.2009.5306292

[18] Zedong Peng, Xuanyi Lin, and Nan Niu. 2020. Unit tests of scientific software: A study on SWMM. In *Computational Science–ICCS 2020: 20th International Conference, Amsterdam, The Netherlands, June 3–5, 2020, Proceedings, Part VII 20*. Springer, Springer, Amsterdam, 413–427.

[19] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study. In *29th Annual International Conference on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '19)*. IBM Corp., USA, 193–202.

[20] 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. https://doi.org/10.1145/3368089.3417921

[21] Railana Santana, Daniel Fernandes, Denivan Campos, Larissa Soares, Rita Maciel, and Ivan Machado. 2021. *Understanding Practitioners' Strategies to Handle Test Smells: A Multi-Method Study*. ACM, New York, NY, USA, 49–53.

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

[23] 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.

[24] 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. In *Anais do XXV Congresso Ibero-Americano em Engenharia de Software* (Córdoba). SBC, Porto Alegre, RS, Brasil, 263–277. https://doi.org/10.5753/cibse.2022.20977

[25] Elvys Soares, Márcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André Santos. 2023. Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date? *IEEE Transactions on Software Engineering* 49, 3 (2023), 1152–1170. https://doi.org/10.1109/TSE.2022.3172654

[26] Daniel Toll and Tobias Olsson. 2012. Why is Unit-testing in Computer Games Difficult?. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, Szeged, Hungary, 373–378. https://doi.org/10.1109/CSMR.2012.46

[27] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 92–95.

[28] 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. https://doi.org/10.1145/3422392.3422499

[29] 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 - Tools Track* (Virtual Conference) *(SBES 2020)*. ACM, New York, NY, USA, 6 pages.

[30] 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 (2021), 8–1.

[31] Tássio Virgínio, Railana Santana, Luana Almeida Martins, Larissa Rocha Soares, Heitor Costa, and Ivan Machado. 2019. On the influence of Test Smells on Test Coverage. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering* (Salvador, Brazil) *(SBES '19)*. Association for Computing Machinery, New York, NY, USA, 467–471. https://doi.org/10.1145/3350768.3350775