Journal of Software Engineering Research and Development, 2021, 9:8, doi: 10.5753/jserd.2021.1893

On the test smells detection: an empirical study on the JNose Test accuracy

Tássio Virgínio [**Federal Institute of Tocantins** | ***tassio.virginio@ifto.edu.br***]
Luana Martins [**Federal University of Bahia** | ***martins.luana@ufba.br***]
Railana Santana [**Federal University of Bahia** | ***railana.santana@ufba.br***]
Adriana Cruz [**Federal University of Lavras** | ***adriana.cruz@estudante.ufla.br***]
Larissa Rocha [**Federal University of Bahia / State Univ. of Feira de Santana** | ***larissa@ecomp.uefs.br***]
Heitor Costa [**Federal University of Lavras** | ***heitor@ufla.br***]
Ivan Machado [**Federal University of Bahia** | ***ivan.machado@ufba.br***]

Abstract

Several strategies have supported test quality measurement and analysis. For example, code coverage, a widely used one, enables verification of the test case to cover as many source code branches as possible. Another set of affordable strategies to evaluate the test code quality exists, such as test smells analysis. Test smells are poor design choices in test code implementation, and their occurrence might reduce the test suite quality. A practical and large-scale test smells identification depends on automated tool support. Otherwise, test smells analysis could become a cost-ineffective strategy. In an earlier study, we proposed the `JNose Test`, automated tool support to detect test smells and analyze test suite quality from the test smells perspective. This study extends the previous one in two directions: i) we implemented the `JNose-Core`, an API encompassing the test smells detection rules. Through an extensible architecture, the tool is now capable of accomodating new detection rules or programming languages; and ii) we performed an empirical study to evaluate the `JNose Test` effectiveness and compare it against the state-of-the-art tool, the `tsDetect`. Results showed that the `JNose-Core` precision score ranges from 91% to 100%, and the recall score from 89% to 100%. It also presented a slight improvement in the test smells detection rules compared to the `tsDetect` for the test smells detection at the class level.

Keywords: *Tests Quality, Test Evolution, Test Smells, Evidence-based Software Engineering*

1 Introduction

Ensuring end-user satisfaction, detecting software defects before go-live, and increasing software or product quality is among the most commonly reported software testing objectives, as written by the annual report of a global consulting firm (Capgemini, 2018). Recently published reports estimate over $ 2 trillion to quantify the impact of poor software quality on the United States economy, referencing publicly available source material for the year 2020 (CISQ, 2021).

Such data illustrates the need for employing software testing techniques in software development processes, as they could anticipate bug identification and fixing, thus reducing its likely effects still during implementation (or even when existing functionalities are under evolution) (Palomba et al., 2018; Spadini et al., 2018; Grano et al., 2019).

In a well-defined Software Engineering process, test code should co-evolve together with production code, as high-quality test code is essential to ease the maintenance and evolution of production and test code (Yusifoğlu et al., 2015; Guerra Calle et al., 2019). However, it might be time-consuming and cost-ineffective (Yusifoğlu et al., 2015; Guerra Calle et al., 2019).

Several approaches have been proposed in the literature to assess the quality of test suites. For example, code coverage measurement has been widely used to check the quality of automated tests. It measures the test suite quality based on how much a test covers structural elements, such as functions, instructions, branches, and lines of code (Gopinath et al., 2014). Nonetheless, even with high code coverage, the test code might encompass poor design choices in their implementation, the so-called test smells.

The presence of smells in test code may reduce the quality of test suites and, consequently, the production code quality (Deursen et al., 2001). Additionally, poorly-written tests can be challenging to comprehend and onerous for testers to maintain the code and detect faults (Bavota et al., 2015; Grano et al., 2019).

The software testing literature has introduced a set of tools focused on validating the quality of test suites, mainly through metrics analysis. For example, *CodeCover*[1] is an open-source Java tool for code coverage executed via a graphical user interface (with Eclipse IDE) and command-line; *tsDetect*[2] is a command-line tool for test smells detection. Other tools use code coverage results to predict test smells, such as *TeReDetect* (Negar and Garousi, 2010) and *TeCReVis* (Koochakzadeh and Garousi, 2010). Generally, these tools have many different data outputs, which might be hard for testers to establish a relationship between code coverage and internal test code quality. Moreover, several types of test smells have not been investigated in conjunction with code coverage yet, but could also provide opportunities to improve test code quality.

In previous studies (Virginio et al., 2019, 2020), we introduced the `JNose Test`, a tool to analyze the quality

[1] Available at: `https://codecover.org`
[2] Available at: `https://testsmells.github.io`

of test suites from the test smells perspective. The `JNose Test` provides an automated test strategy focused on *(i)* identifying possible test design flaws, *(ii)* analyzing the software project quality evolution, and *(iii)* reducing the effort for performing quality assurance of a test suite. The `JNose Test` integrates a conceptual framework which encompasses strategies for test smells prevention, identification, refactoring, and visualization to improve the test code quality. `RAIDE`[3] (Santana et al., 2020) and `TSVizzEvolution`[4] tools are part of this framework.

In this study, we proposed the `JNose-Core`, an API (Application Programming Interface) to detect test smells in the test code. It provides a flexible architecture to support the insertion of new test smells detection rules. The `JNose Test` implements the interface methods the `JNose-Core` provides and organizes the data flow in a web-based user interface. In this new version, our tool: i) detects test smells in different code granularities (line, method, block, and class); ii) detects test smells more accurately according to the literature definition; and iii) presents the outputs in a more user-friendly interface.

Additionally, we also extended our previous work by validating the test smells detection rules implemented in the `JNose Test` tool. We conducted an empirical evaluation to investigate two objectives: *(i)* verify the `JNose Test` accuracy compared with the `tsDetect` in terms of precision and recall at a class level, and *(ii)* verify the `JNose Test` accuracy compared with the manual analysis in terms of precision and recall at a fine-grained level. The results show that in a test class level, the `JNose Test` obtained slightly better results than the `tsDetect` for specific types of test smells, such as *Assertion Roulette*, *Lazy Test*, and *Eager Test*. When analyzing the test smells at a fine-grained level, our tool shows higher accuracy when detecting the test smells location.

The remainder of this paper is structured as follows. Section 2 introduces the test smells concept and types. Section 3 presents an overview of the `JNose-Core` API. Section 4 presents the `JNose Test`, a web application for test smells detection. Section 5 describes the empirical study to evaluate the `JNose Test` accuracy. Section 6 presents the results. Section 7 discusses related work. Section 8 presents the threats to the validity of our study. Finally, Section 9 draws concluding remarks.

2 Background

Test code development is not a trivial task (Palomba et al., 2018; Virginio et al., 2019). In real-world practice, developers are likely to use anti-patterns during test development (Bavota et al., 2012; Junior et al., 2020). Those anti-patterns may negatively impact the test code quality and maintenance and reduce its capability for detecting software faults (Bell et al., 2018; Spadini et al., 2020).

Several studies have investigated different types of test smells. Initially, Deursen et al. (2001) defined a catalog of 11 test smells and refactorings to remove them from the test code. Next, several authors extended this catalog and analyzed the test smells effects on the production and test code (Meszaros et al., 2003; Bavota et al., 2012; Greiler et al., 2013; Bavota et al., 2015; Bell et al., 2018; Virginio et al., 2019; Spadini et al., 2020). As a result of the researchers' efforts to identify anti-patterns, Garousi and Küçük (2018) listed more than 190 test smells in a literature review.

In this study, we selected twenty-one types of test smells currently discussed in the literature (Peruma et al., 2019):

- **Assertion Roulette (AR).** It occurs when a test method contains non-documented assertions. If an assertion fails, it can be difficult to identify which one failed;
- **Conditional Test Logic (CTL).** It occurs when a test method contains conditional expression or loop structures. Conditions within the test method may alter its behavior which leads the test to fail;
- **Constructor Initialization (CI).** It occurs when a test method contains a constructor;
- **Default Test (DT).** It occurs when a test class is created by default;
- **Dependent Test (DepT).** It occurs when the test being executed depends on other tests' success;
- **Duplicate Assert (DA).** It occurs when a test method tests for the same condition multiple times within the same test method;
- **Eager Test (ET).** It occurs when a test method checks more than one method of the production class;
- **Empty Test (EpT).** It occurs when a test method does not contain executable statements;
- **Exception Catching Throwing (ECT).** It occurs when a test method is explicitly dependent on the production method throwing an exception;
- **General Fixture (GF).** It occurs when the test methods only access part of the test case fixture (setup method);
- **Ignored Test (IgT).** It occurs when a test method is suppressed from running;
- **Lazy Test (LT).** It occurs when several test methods check the same production method;
- **Magic Number Test (MNT).** It occurs when assert statements contain numeric literals;
- **Mystery Guest (MG).** It occurs when a test method utilizes external resources (*e.g.*, a file containing test data), and thus it is not self-contained;
- **Print Statement (PS).** It occurs when unit tests contains print statements;
- **Redundant Assertion (RA).** It occurs when the test method contains an assertion statement that always is true or false;
- **Resource Optimism (RO).** It occurs when a test method makes optimistic assumptions about the existence and state of external resources;
- **Sensitive Equality (SE).** It occurs in test methods that contains an equality check using a `toString()` method. The test may fail when the `toString()` method is changed;
- **Sleepy Test (ST).** It occurs when the execution of a test

[3] Available at `https://raideplugin.github.io`

[4] Available at `https://github.com/arieslab/TSVizzEvolution`

method is paused for a certain period (*e.g.*, simulate an external event) and then continues its execution;
- **Unknown Test (UT).** It occurs when a test method does not encompass an assertion statement.
- **Verbose Test (VT).** It occurs when the tests use too much code to do what they are supposed to do. In other words, the test code is not clean and simple.

3 JNose Core

In our previous work (Virginio et al., 2020), we introduced the first version of the `JNose Test`, a web application for the detection and coverage calculation of test smells. We reused and also expanded the test smells detection rules from the `tsDetect` (Peruma et al., 2020). Therefore, the `JNose Test` provides: (i) a graphical interface to facilitate the interaction between user and tool, (ii) the amount and location of the detected test smells, and (iii) support for the test smells analysis through several project versions.

When improving the detection rules from `tsDetect`, we faced some challenges regarding the coupling and dependency between the test framework and test code. The test frameworks, specifically the JUnit framework[5], require different implementations depending on the version used. For example, JUnit 4 uses a tag `@Ignore` to disable a test class or test method, while JUnit 5 uses the tag `@Disabled`. Regarding the assertions, JUnit 4 accepts an optional parameter for error message as the first argument, and JUnit 5 uses the last argument in the method signature.

Therefore, to facilitate the detection rules expansion and reuse of other tools, we implemented the `JNose-Core` API.[6] It is beneficial for the conceptual framework we are working on to evaluate the test code quality. The detection module is the framework base; the test smells detected are the same that should be removed by the refactoring module (`RAIDE tool`) and presented to the user by the visualization module (`TSVizzEvolution`).

3.1 Architecture

We designed the `JNose-Core` as a Maven[7] project to simplify and standardized the build process. Additionally, we provided a `JNose-Core` compiled version that can be imported by other projects built with Maven. The requirement to use the compiled version is to import the library in the `pom.xml` of the project, as Listing 1 shows. As a result, the `JNose-Core` provides methods to instantiate for the test smells detection. The `JNose-Core` is licensed under the GNU general public license, and its architecture comprises four packages, as follows (Figure 1):

- **core**. It implements the *JNoseCore*, a facade class that receives a instance of the *Config* interface. The *Config* interface contains the methods signature for the test smells detection;
- **detector**. It implements a structure to detect the smelly elements and contains classes to support a test code static analysis through an AST (Abstract Syntax Tree) generated by JavaParser[8].
- **smell**. It implements the detection rules for JUnit 4 and improves the detection rules from `tsDetect` (Section 2) to identify test smells at different granularity levels. Several classes are implemented (for each type of test smell) and use JavaParser to collect additional information on the location and number of test smells.
- **dto (data transfer object)**. It implements the classes responsible for transferring data among the packages.

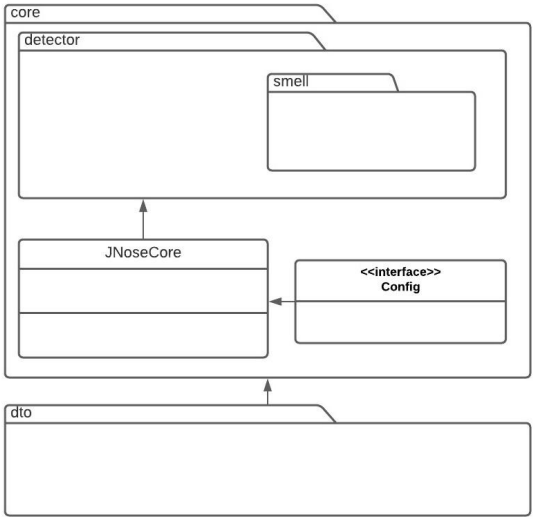


Figure 1. JNose-Core API internal architecture

```
<dependency>
    <groupId>br.ufba.jnose</groupId>
    <artifactId>jnose-core</artifactId>
    <version>0.7-SNAPSHOT</version>
</dependency>
```

Listing 1: pom.xml configuration to use JNose-Core

3.2 Detection Rules

We revisited the test smells definitions in the literature to identify how we should improve the detection rules from `tsDetect`. Table 1 shows the granularity levels that we defined to detect the exact test smells location in the test code, as follows: (i) **line**, test smells that occur in a specific line; (ii) **block**, test smells that occur in a statement block level, e.g., try/catch and conditional statements; (iii) **method**, test smells that occur in the method level; and (iv) **class**, test smells that occur in a test class level. Additionally, we made improvements in the test smells detection rules. We next detail the main modificationsw we performed:

- **Nested Structures.** We improved the rules for detecting the CTL, ECT, and MNT test smell to consider nested

[5] JUnit is a Java library for testing source code, which has advanced to the de-facto standard in unit testing. Available at `https://junit.org/`.

[6] Available at `https://github.com/arieslab/jnose-core`

[7] Maven is a software project management and comprehension tool. Maven can manage a project's build, reporting and documentation from a central piece of information. Available at `https://maven.apache.org/`

[8] Available at: `https://javaparser.org/`

Table 1. Test Smells detection rules.

Name	Detection Rule	Granularity
Assertion Roulette	A line with assertion statements without the explanation/message parameter	Line
Constructor Initialization	A method that is a constructor declaration	Method
Conditional Test Logic	A code block with conditional statements	Block
Duplicate Assert	A line with assertion whose parameters equal the other assertion inside of same test method	Line
Default Test	A method called `ExampleUnitTest()` or `ExampleInstrumentedTest()`	Method
Dependent Test	A method that depends on the previous execution of another test method	Method
Empty Test	A method that does not contain a single executable statement	Method
Eager Test	A line that contains a call to another production method	Line
Exception Catching Throwing	A block that contains either a throw statement or a catch clause	Block
General Fixture	A line with a field instantiated within the `setUp()` method that is not utilized by all test methods	Line
Ignored Test	A method that contains the `@Ignore` annotation	Method
Lazy Test	A line of method that calls the same production method that other test method	Line
Mystery Guest	A method that accessing object instances of files and databases classes	method
Magic Number Test	A line with assertion method that contains a numeric literal as an argument	Line
Print Statement	A line that invokes either the `print()` or `println()` or `printf()` or writes method of the `System` class	Line
Redundant Assertion	A line containing an assertion statement in which the expected and actual parameters are the same	Line
Resource Optimism	A method that uses an external resource without checking the state of the object	Method
Sensitive Equality	A method that contains an assertion that invokes the `toString()` method of an object	Method
Sleepy Test	A line that invokes the `Threadsleep()` method	Line
Unknown Test	A method that uses the `@Test` annotation but does not contain assertions statement	Method
Verbose Test	A method with more than 30 lines Counting non executable statements and annotations	Method

structures. When the tool reports a nested conditional structure as one test smell, it might be hard to identify which part of the test code needs refactoring at first glance. If the nested conditional is too long, the user may refactor parts of it. When rerunning the tool, the user will see that the problem is still there, making the refactoring process longer. Therefore, the tool presents one test smell for each structure;

- **Empty or Non-assertive.** The UT and EpT test smells present similar definitions. The UT test smell identifies methods without assertions, and the EpT test smell identifies methods with non-executable statements. Test methods without a body neither contain executable statements nor assertions. Therefore, we added another rule to separate both definitions; the UT test smell identifies methods that contain a body and does not identify asserts;
- **General Fixture.** The GF test smell occurs when test methods use only a setup method part, representing the cohesion among the test class's methods. Therefore, we improved the detection rules to show that all the test class methods are used with setup fixtures. It allows the user to identify the test method to which a fixture should be moved;
- **Missing Structures.** Each version of the test framework requires the static analysis of different code structures. The `assert` structures used in JUnit 3 is different from JUnit 4, which is also different from JUnit 5. Therefore, to improve the detection rules to JUnit 4, we added the code structures that were missing to detect the CTL, AR, DA, and ECT test smells;
- **Methods Overload.** Similar to the preceding item, there are differences among the JUnit versions regarding the overloaded methods. When analyzing test cases written with JUnit 3, we were not concerned about overloaded methods. However, to focus on the current detection rules for JUnit 4, we needed to improve the AR, and DA test smells to support the overloaded methods.

4 JNose Test

The `JNose Test`[9] enables test code quality analysis through test smells detection and code coverage over several software project versions. Therefore, it is possible to compare whether a project test quality has either improved or declined throughout its life cycle. The `JNose Test` operation involves three key processes (Figure 2): (i) **Data Input**, receives the settings for the tool execution, i.e., the list of types of test smells, analysis mode (*By TestClass*, *By TestSmell*, *By TestFile*, and *Evolution*), and the project to be analyzed; (ii) **Project Analysis**, calls the `JNose-Core`, an API to perform the project analysis according to the analysis mode selected; and (iii) **Data Output**, shows the execution status and the analysis results.

4.1 Processes Description

Java Development Kit (JDK) 11 and Maven 3 (or superior) are necessary to install the `JNose Test`. Upon installation, the user would be able to use Jetty (embedded on Maven) and build and run the `JNose Test`.

[9] Available at `https://jnosetest.github.io`

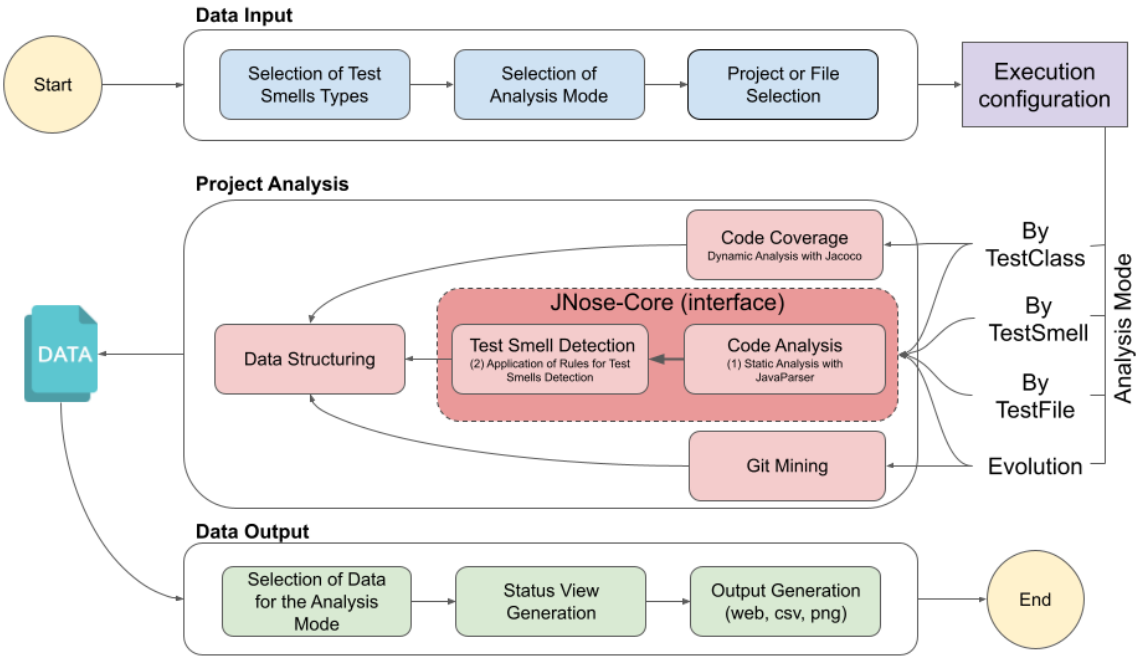


Figure 2. Schematic overview of the JNose Test tool and its main features

After starting the tool, the user must configure the **Data Input** (Figure 2). First, the user should import the projects to be analyzed (Figure 3a - Step 1). The `JNose Test` clones the repository directly from GitHub, and allows the user to manage it (Figure 3a - Step 2). Second, the user selects the analysis mode, i.e., *By TestClass*, *By TestSmells*, *By TestFile*, and *Evolution* (Figure 3a - Step 3). Each analysis mode provides a menu where the user chooses the repositories to be analyzed. By default, the tool detects twenty-one types of test smells, but the user could configure this feature as well (Figure 3a - Step 4).

After completing the project import and defining the settings detection, the tool starts the **Project Analysis** (Figure 2). For each analysis mode, the `JNose Test Tool` presents an interface with (i) a list of cloned projects (Figure 3b - Step 1), (ii) a menu with specific analysis mode settings (Figure 3b - Step 2), and (iii) a menu with the data output options (Figure 3b - Step 3). The **Project Analysis** considers the analysis mode selected by the user, described below.

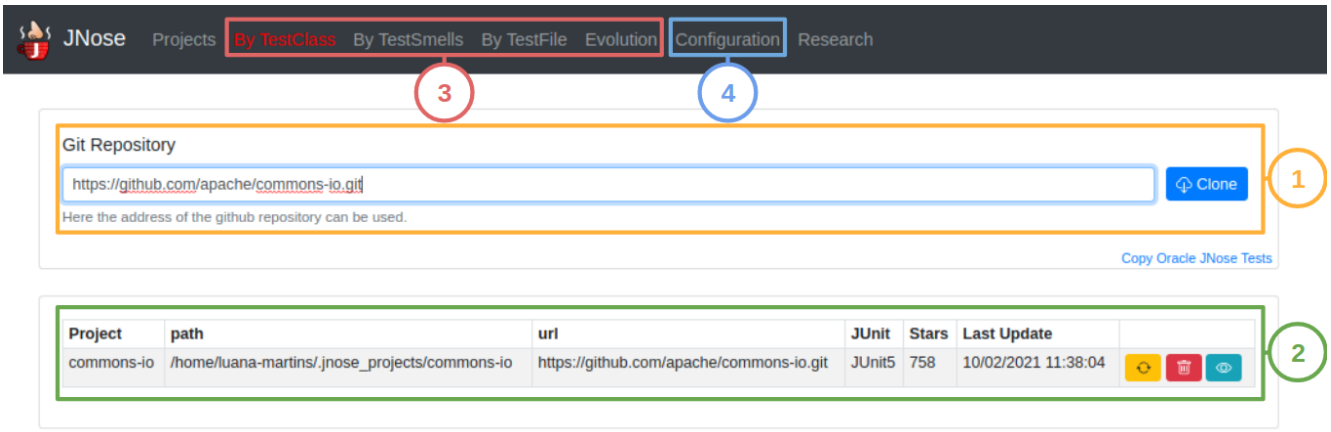
(1) By TestClass. In the **Data Input** process, the user could enable the coverage metrics calculation and select the projects to be analyzed. Then, to analyze the project by test class, the **Project Analysis** calls the *JNose-Core* and optionally executes the *Code Coverage* module. Finally, the **Data Output** process generates a view that contains a table with the number of test smells by test class. That table presents a row for each test class, and each column represents the type of parameter collected: project name, test class, and production class location, twenty-one columns for the types of test smells, the number of test class lines, the number of test methods, and five columns with coverage data. That table could be downloaded as a `.csv` file. Additionally, the user could view a chart or download it as a `.png` file with the amount of each test smell in the project.

(2) By TestSmells. The **Project Analysis** process only calls the *JNose-Core* to analyze the project by test smell. During the **Data Input** process, the user needs to select the projects to explore. Unlike the previous analysis, By TestSmells provides the exact location of a test smell. The last, the **Data Output**, offers a view with the data analysis results, which could also be downloaded as a `.csv` file. Each row of the table represents a test smell, and it has five columns to show the type of parameter collected: the project name, the test class location, the production class location, the test smell name, the test smell location.

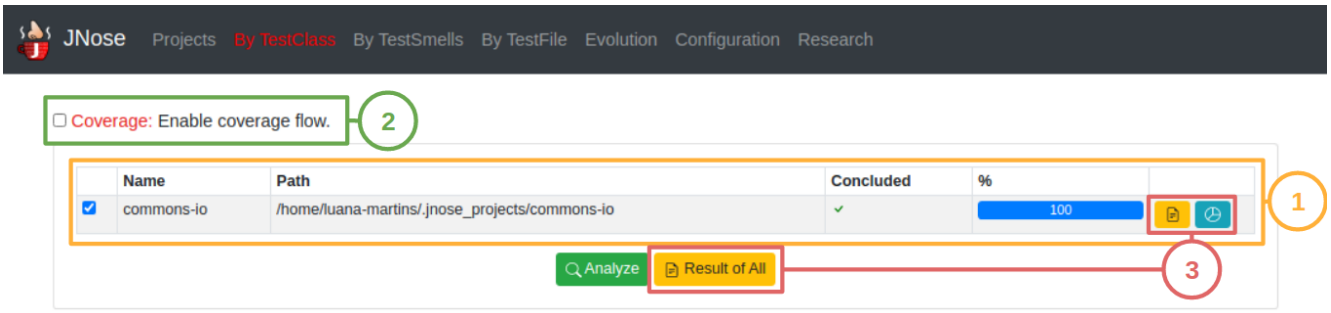
(3) By TestFile. The **Project Analysis** process only calls the *JNose-Core* to analyze the project by test file. During the **Data Input** process, the user should select a test class and optionally its respective production class. Besides the production class selection is an optional feature, the *Eager Test* and *Lazy Test* test smells are not detected without it. Then, the **Data Output** provides a view containing a row for each detected test smell and its location.

(4) Evolution. The **Project Analysis** process executes the *Git Mining* module and the *JNose-Core* to analyze the project by version. During the **Data Input**, the user should select projects to explore and search to be applied (by commits or by tags). This analysis provides the test smell detection for each project version, in addition to data about the author who committed the test smell. The **Data Output** process provides a view containing the data analysis results by test smells, downloadable as a `.csv` file. The table rows represent the test classes by commit. The columns encompass the following parameters: project name, test class and production class location, number of test smells, commit identification, authorship, date, and message. Additionally, the user could view a chart and download it as a `.png` file with the amount of test smells in each project version or the number of test smells committed by an author. The tool also automatically calculates the authorship of a test smell by guilt, i.e., the tester who last modified the method and did not fix it.

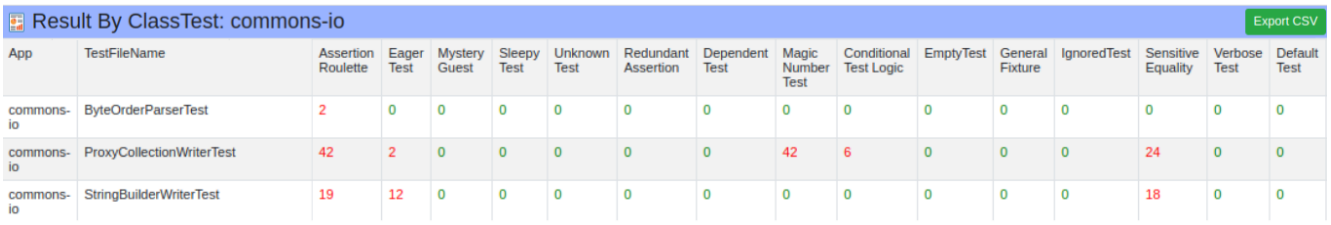
Different analysis mode allows other data visualization.



(a) Data Input: cloning projects from GitHub



(b) Project Analysis: configuring By TestClass analysis mode



App	TestFileName	Assertion Roulette	Eager Test	Mystery Guest	Sleepy Test	Unknown Test	Redundant Assertion	Dependent Test	Magic Number Test	Conditional Test Logic	EmptyTest	General Fixture	IgnoredTest	Sensitive Equality	Verbose Test	Default Test
commons-io	ByteOrderParserTest	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
commons-io	ProxyCollectionWriterTest	42	2	0	0	0	0	0	42	6	0	0	0	24	0	0
commons-io	StringBuilderWriterTest	19	12	0	0	0	0	0	0	0	0	0	0	18	0	0

(c) Data Output: an excerpt of the table with the By TestClass results

Figure 3. `JNose Test` - process execution

Therefore, the **Data Output** generates tables or charts depending on the analysis mode. Tables are generated for all analysis modes (Figure 3c). Charts are generated for *By TestClass* and *Evolution*. *By TestClass* charts present the total amount of test smells inserted in a project, and *Evolution* charts present the amount of test smells by project version or by author.

4.2 Tool Architecture

The `JNose Test` is implemented as a Java project and comprises five packages, as Figure 4 shows: (i) **base**, responsible for instantiating the *JNose-Core* interface implementation and calculating the coverage metrics; (ii) **page**, responsible for presenting the web pages and their content; (iii) **dtolocal**, responsible for encompassing the classes used in *dto*; (iv) **entity**, responsible for the domain objects persistence from the database; and (v) **business**, responsible for applying the business rules to present the results.

The **base** package implements the **Project Analysis** (Figure 3a), which was split into three other packages, as follows:

- **Coverage**. It applies the rules necessary to calculate

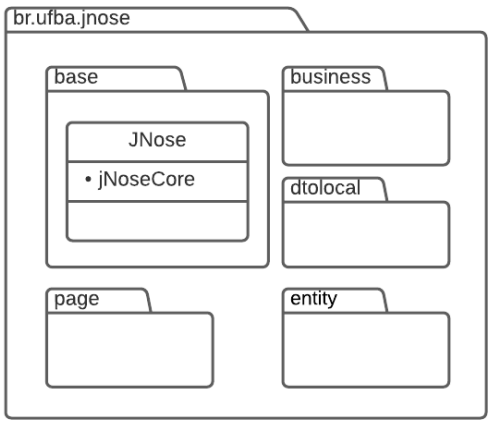


Figure 4. Packages of the JNose Test

coverage. It runs the JaCoCo library[10] to calculate code coverage in the Java language. It performs dynamic analysis of the production code branches (BC), instructions (IC), lines (LC), complexity (CC), and methods (MC) to determine which one is either missed or covered by the test (Virginio et al., 2019);

- **Git Mining**. It applies business rules for GitHub mining. It uses the GitHub API for Java library[11] to clone the projects from GitHub and extract information about the project's tags, commits, and authors;
- **JNose-Core**. It performs test code static analysis through an AST generated by JavaParser.[12] Then, it extracts information about the code structure to apply the rules for the test smells detection, and it collects additional information about the location and number of test smells. The detection rules were improved from the `tsDetect` tool (Section 2) to identify test smells at different granularity levels (Table 1).

The `JNose Test` interface was implemented in the **page** package based on the Apache Wicket[13], a framework for web application development in Java. We also used HTML5 and CSS3 to develop the web pages. This package implements the **Data Input** (Figure 2). The **business** implements utility classes responsible for generating the results. It is possible to generate a different type of report For each analysis mode. This package implements the **Data Output** (Figure 2). In the **dto** package, we have the classes used to transfer data among the project layers. That package implements the communication among **Data Input**, **Project Analysis**, and **Data Output** (Figure 2). Additionally, a local database stores the data generated by those processes, comprising persistence rules implemented in the **entity** package.

The `JNose Test` execution uses parallel processes, i.e., the tool creates threads for each uploaded project, for each test class, and so on. With parallel processing, the `JNose Test` could be used to analyze a massive set of projects in a short time (Virginio et al., 2019).

4.3 Running Example

We carried out an experimental study to verify the correlation between the coverage metrics and test smells in previous work. We selected eleven software projects to perform that study, in which we collected twenty-one test smells and five coverage metrics using the `JNose Test`.

This section presents an example considering the different types of analysis modes supported by the `JNose Test`. We used the `commons-io` project[14] (release 2.7-RC1), a library of utilities, to assist I/O development. We next discuss each supported method.

4.3.1 By TestClass Analysis

We ran the `JNose Test` by TestClass to analyze which type of test smells would achieve the highest diffusion over the `commons-io` project. Therefore, we took the following steps: (i) select all types of test smells; (ii) select the project path; and (iii) enable code coverage. The tool returned 58 test classes. We checked the number of classes where each test smell was present to understand the test smell type diffusion. For example, the ECT test smell was present in 23 classes, followed by AR test smell in 17 test classes, and ET test smell in 16 test classes. Each type of test smell could occur many times in a test class. Those three types of test smell presented the highest occurrence in the project, counting 316, 175, and 157 times, respectively.

Table 2 shows five test classes with the highest number of ECT, AR, and ET test smells. For example, the test class `ProxyCollectionWriterTest` contains the highest number of those test smells. Additionally, most test classes achieved good code coverage when considering the IC, LC, and MC coverage metrics (>70%). Therefore, even with high coverage, the test code might present low-quality.

4.3.2 By TestSmell

Once we found that the ECT, AR, and ET test smells had the highest diffusion numbers in the `commons-io` project test classes, we may improve the test code quality by fixing the problems. Then, we executed the `JNose Test` by TestSmell by taking the following steps: (i) select the ECT, AR, and ET test smells; and (ii) select the project. Table 3 shows a results excerpt filtered by the `ProxyCollectionWriterTest` test class.

4.3.3 By TestFile

In the previous example (By TestSmells), we filtered the results to present only the ones related to the `ProxyCollectionWriterTest` test class. In the By TestFile analysis, that class could be analyzed individually. Therefore, we executed the `JNose Test` by taking the following steps: (i) select the ECT, AR, and ET test smells; and (ii) select the `ProxyCollectionWriterTest` and `ProxyCollectionWriter` files. The results are the same as the filter presented in Table 3.

Listing 2 shows the `ProxyCollectionWriterTest` test class with the `testArrayIOExceptionOnAppendChar1()` test method (lines 39-53). We observed that the `assertEquals()` method is called twice within the test method (lines 50-51). Each one checks a different condition, but there is no explanation message for them. Thus, if the test method fails, there is no clue to identify which assertion caused the failure. That issue refers to the AR test smell. Moreover, those assertions are also related to the ECT test smell because they may fail when a specific exception occurs. Furthermore, a test method is supposed to check just one production class method; otherwise, the code has one ET test smell (`ProxyCollectionWriter()` on line 43 and `append()` on line 46).

4.3.4 Evolution Analysis

The evolution analysis might help us identify whether the `commons-io` has improved over time. We should take the following steps to perform this analysis: (i) select all test smells,

[10] Available at `https://www.eclemma.org/jacoco/`
[11] Available at `https://github-api.kohsuke.org/`
[12] Available at `https://javaparser.org/`
[13] Available at `https://wicket.apache.org/`
[14] Available at `https://github.com/apache/commons-io`

Table 2. Classes with high diffusion of test smell - by TestClass

TesFileName	...	LOC	Met	UT	IgT	RO	...	ST	LT	DA	ET	AR	CTL	CI	DT	EpT	ECT	GF	MG	PS	DpT	IC	BC	LC	CC	MC
ProxyCollectionTest	...	448	23	1	0	0	...	0	61	1	23	21	1	0	0	0	23	0	0	0	0	72	0	76	100	100
TreWriterTest	...	448	23	1	0	0	...	0	30	1	2	21	1	0	0	0	23	0	0	0	0	100	0	100	100	100
ProxyWriteTest	...	275	21	3	0	0	...	0	23	0	4	0	0	0	0	0	21	0	0	0	0	83	0	87	93	93
BoundedReaderTest	...	246	22	1	1	1	...	0	48	1	8	3	2	0	0	0	16	0	1	0	0	100	100	100	100	100
EndianUtilsTest	...	316	22	1	0	0	...	0	46	8	20	15	1	0	0	0	14	0	0	0	0	100	100	100	100	100

Table 3. Test Smells location in `ProxyCollectionWriterTest`

TesFileName	...	TestSmell	MethodLocationName	Lines
ProxyCollectionTest	...	AR	testArrayIOExceptionOnAppendChar1	50,51
ProxyCollectionTest	...	AR	testArrayIOExceptionOnAppendChar2	66,67
ProxyCollectionTest	...	AR	testArrayIOExceptionOnAppendCharSe	82,83
ProxyCollectionTest	...	ET	testArrayIOExceptionOnAppendChar1	50,51
ProxyCollectionTest	...	ET	testArrayIOExceptionOnAppendChar2	66,67
ProxyCollectionTest	...	ET	testArrayIOExceptionOnAppendCharSe	82,83
ProxyCollectionTest	...	ECT	testArrayIOExceptionOnAppendChar1	45-52
ProxyCollectionTest	...	ECT	testArrayIOExceptionOnAppendChar2	61-69
ProxyCollectionTest	...	ECT	testArrayIOExceptionOnAppendCharSe	77-84

Table 4. Classes with high diffusion of test smell - Evolution

TesFileName	...	TestSmell	CommitID	CommitName	CommitDate
ProxyCollectionWrite	...	153	b739ce7c	Adam Retter	03:39:47 2020
ProxyCollectionWrite	...	153	bcb36041	David Georg	00:09:03 2018
TreWriterTest	...	101	b739ce7c	Adam Retter	03:39:47 2020
TreWriterTest	...	101	bcb36041	David Georg	00:09:03 2018
ProxyWriteTest	...	59	b739cc7c	Adam Retter	03:39:47 2020
ProxyWriteTest	...	59	bcb36041	David Georg	00:09:03 2018
BoundedReaderTest	...	92	b739ce7c	Adam Retter	03:39:47 2020
BoundedReaderTest	...	96	51f13c84	Kristian Rose	15:36:15 2016
BoundedReaderTest	...	83	9a9b8385	Gary D. Greg	01:17:05 2014
EndianUtilsTest	...	118	b739ce7c	Adam Retter	03:39:47 2020
EndianUtilsTest	...	117	8940848G	Gary D. Greg	18:47:06 2018

```
37 public class ProxyCollectionWriterTest{
38
39   @Test
40   public void testArrayIOExceptionOnAppendChar1()
         throws IOException {
41     final Writer badW = new BrokenWriter();
42     final StringWriter goodW = mock(StringWriter.
         class);
43     final ProxyCollectionWriter tw = new
         ProxyCollectionWriter(badW, goodW, null);
44     final char data = 'A';
45     try {
46       tw.append(data);
47       fail("Expected "+IOException.class.getName
           ());
48     } catch (final IOExceptionList e) {
49       verify(goodW).append(data);
50       assertEquals(1,e.getCauseList().size());
51       assertEquals(0,e.getCause(0,
           IOIndexedException.class).getIndex());
52     }
53   }
```

Listing 2: `ProxyCollectionWriterTest` test class

(ii) select the analysis by commit, and (ii) select the project path. The project has 2,337 commits, 52 releases, and 56 contributors from the beginning until the release 2.7RC1. We filtered the five test class results with more ECT, ET, and AR test smells (Table 4).

Figure 5 shows the evolution of those classes and the project. The `ProxyCollectionWriterTest`, `TreWriterTest`, and `ProxyWriterTest` test classes are stable, as no test smell was either inserted or fixed. However, the `BoundedReaderTest` test class presented novel test smells during 2014-2016 and fixed them during 2016-2020. We could observe that the number of test smells increased over time, which might indicate that people involved in the project test suite development have not worked to get rid of test smells yet. In addition, authorship is calculated by fault, so the authors from that example might not have inserted all detected test smells.

5 Empirical Evaluation

This empirical evaluation aims to investigate the `JNose Test` accuracy in detecting test smells. We designed the empirical study in four steps, as Figure 6 shows: (i) **Dataset Selection**, in which we defined the test classes to analyze; (ii) **Oracle Definition**, in which we manually detected the test smells instances; (iii) **Data Collection**, where we applied the `JNose Test` and `tsDetect` to collect the test smells instances; and (iv) **Data Analysis**, in which we analyzed the data collected to investigate our objectives.

5.1 Dataset Selection

For this analysis, we used the dataset made available by Peruma et al. (2020), which contains 65 test classes extracted from GitHub projects. As we initially reused the `JNose Test` detection rules from the `tsDetect`, we decided to use the same dataset they used to perform a fair comparison between both tools and assess the `JNose test` effectiveness.

To build the dataset, Peruma et al. (2020) selected Android apps neither duplicated nor forked. Upon the smells identification in a test file, they randomly selected 65 test classes from the selected projects and followed the definitions to detect the test smells. Although the `tsDetect` implements detection rules for twenty-one types of test smells, only nineteen were validated. It did not detect the DT and DpT test smells. The same limitation applies to our study.

Since the authors did not have access to the test results from manual detection performed by Peruma et al. (2020), we created a new oracle using the same test and production classes for this study. Even if we had access to the Peruma et al. (2020) manual detection results, we would have to detect the test smells at a fine-grained level to validate the `JNose Test`. The reason for such assumption is that the `JNose Test` detects the test smells exact location, rather than just their presence (like the `tsDetect`).

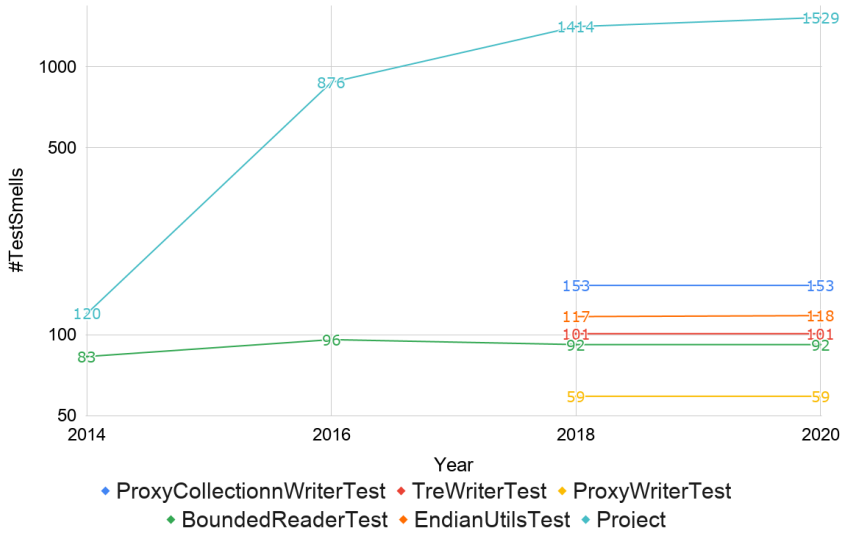


Figure 5. Evolution of the `commons-io` project and classes with high diffusion of test smells

5.2 Oracle Definition

To manually detect the test smells instances, we followed a design not fully crossed to assign coders to the subjects, i.e., different subjects are analyzed by different subsets of coders (Hallgren, 2012). The subjects are the 65 test classes, and four authors of this study served as coders. The coders are experts in test smells with at least three years of experience. Additionally, their Java programming development experience ranged from 4 to 15 years, including unit test development.

We organized the codes into two groups of two coders each, where one group analyzed 32 test classes and the other group 33 test classes. Two coders individually analyzed each test class. They collected data regarding the test smells type and location, following definitions from Table 1. As a result, each coder generated a document with all the test smells detected. Subsequently, the coders compiled the individual records into one document after discussing the divergences.

The review process of the test smells manually detected was time and effort-consuming (~60 minutes). The final oracle version supports the detection of eighteen types of test smells. In addition to the non-existence of the DpT and DT test smells in the dataset, previously reported by Peruma et al. (2020), we did not detect any IgT test instances smell. The analysis process of the test classes and the discussion about the classification divergences took about 60 hours.

5.3 Data Collection

Data collection consisted of detecting 65 test classes in two different analyses: detection with `tsDetect` and detection with `JNose Test` tool.

Detection with tsDetect. We downloaded the `tsDetect` version 2.0 to collect the data. It executes three modules: i) the `Test File Detector` to detect the test classes, ii) `Test File Mapping` to link the test classes to production classes, and iii) `tsDetect` to detect the test smells. All modules were executed by command line in the terminal sequentially. As a result, the `tsDetect` generates a file that contains a boolean value for each type of test smell detected in the test class. Therefore, the result provided by the `tsDetect` has a class-level granularity. The detection process took about 7 minutes, considering the tool execution time and the participants' expertise with the operating system terminal to exercise the necessary commands for its execution.

Detection with JNose Test. We use the `JNose Test` version 2.1 to detect the test smells. After running the tool, the output file with the result encompassed each test smell for each test class detected. The test smells detection granularity followed Table 1. The automated detection with `JNose Test` took about 1 minute due to the unified process to detect the test classes, production classes, and test smells. A friendly graphical interface makes this process easier.

5.4 Data Analysis

We used the oracle to calculate the `JNose test` and `tsDetect` accuracy against the manual analysis. Both tools present distinct granularity levels to detect test smells. `tsDetect` indicates whether a test class contains a test smell instance, i.e., returns a boolean value for each test smell in a class. `JNose Test` detects all instances of a test smell with its exact location (line, block, method, or class). Therefore, we carried out what follows:

1. We compared the `JNose Test` and `tsDetect` accuracy considering the class-level. We treated the `JNose Test` output to show boolean values at the class-level to compare with the `tsDetect`. As the `JNose Test` detection rules were reused from the `tsDetect`, our goal is to determine the extension we improved those detection rules. In this comparison, the accuracy is given at the class-level considering its precision and recall.
2. We compared the `JNose Test` and manual analysis accuracy considering a fine-grained level. For example, by evaluating the line-level of granularity, we can detect the AR test smell; therefore, we collected data at the line level to see it manually and automatically. Our goal is to show the `JNose Test` accuracy to indicate the test smells location. Therefore, we provide the accu-

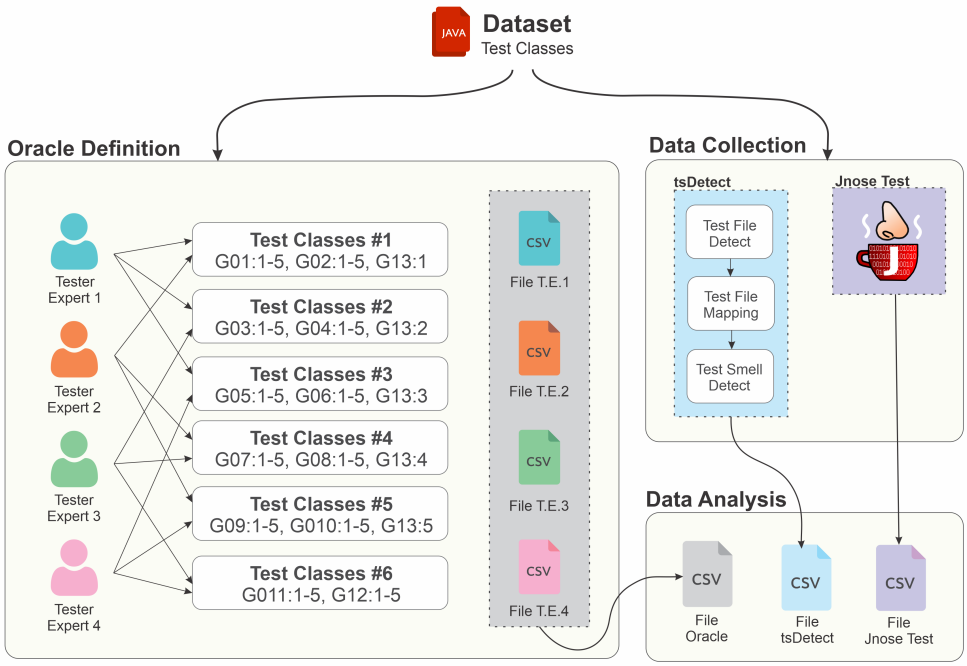


Figure 6. Steps to conduct the experiment

racy value at a fine-grained level in terms of precision and recall.

6 Results

This section reports the results of our empirical study. The data for replication purposes are available online (Virgínio et al., 2021).

6.1 Comparison between JNose and tsDetect

Table 5 reports precision and recall accuracy when detecting test smells with `JNose Test` and `tsDetect`. This comparison was made at the test class-level.

Table 5. `JNose Test` and `tsDetect` Comparison - Class-level

	Accuracy (%)		Precision (%)		Recall (%)		F1-Score (%)	
Test Smell	JNose	tsDetect	JNose	tsDetect	JNose	tsDetect	JNose	tsDetect
AR	100	75.38	100	90	100	75	100	78
CI	100	100	100	100	100	100	100	100
CTL	100	100	100	100	100	100	100	100
DA	98.46	96.92	99	98	98	97	99	97
ECT	100	46.15	100	92	100	46	100	55
ET	95.38	86.15	95	87	95	86	95	86
EpT	100	100	100	100	100	100	100	100
GF	98.46	98.46	99	99	98	98	99	99
LT	100	93.85	100	94	100	94	100	94
MG	90.77	90.77	92	92	91	91	89	89
MNT	95.38	90.77	96	92	95	91	95	90
PS	100	100	100	100	100	100	100	100
RA	100	100	100	100	100	100	100	100
RO	89.23	89.23	91	91	89	89	88	88
SE	100	100	100	100	100	100	100	100
ST	100	100	100	100	100	100	100	100
UT	100	93.85	100	94	100	94	100	94

The results obtained with the `tsDetect` diverges from those reported by Peruma et al. (2020). Such study yielded precision values from 85.71% to 100% and recall values from 95% to 100%. They could detect nineteen types of test smells. The `tsDetect` achieved a precision from 87.71% to 100% and recall from 46% to 100% for eighteen types of test smells when using our oracle. As we mentioned earlier, we did not detect any IgT test smell instances in none of the tools. Those divergences highlight the challenges of building an oracle due to different interpretations that a coder may have about the test smells definitions.

Regarding the results obtained with the `JNose Test`, the precision ranged from 91% to 100%, and the recall from 89% to 100% to detect eighteen types of test smells. As we reused the `tsDetect` detection rules, we showed the improvements we achieved. Considering the F1-Score metric, the `JNose Test` presented accuracy improvement of 45% for the ECT test smell, followed by 22% for the AR test smell, 11% for the VT test smell, 9% for the ET test smell, 6% for the LT, and UT test smells, 5% for the MNT test smell, and 2% for the DA test smell. Other test smells detection rules did not present any relevant improvement at the test class level.

Next, we showed the reason for the divergence between the results obtained by the tools for the ECT test smell detection. The `JNose Test` considers three compliant solutions to handle exceptions (Listing 3): i) the use of the tag `Test` with the expected parameter (lines 1-4), ii) the use of `assertThrows` statement (lines 6-9), or iii) throw the exception in the method signature (lines 11-14). As a non-compliant solution, it considers the `try/catch` structure within the method body (lines 16-23). The `tsDetect` considers the `try/catch` structure and the throw-in method signature as a non-compliant solution (lines 11-23).

We identified that the `tsDetect` does not consider the JUnit overloaded methods when using an assert statement regarding the AR test smell. For example, the `assertEquals`

asserts that (Listing 4) (i) two objects are equal (lines 1-9) or (ii) two objects are equal within a positive delta (lines 11-19). The optional value is a string that describes the assertion. The tool simplifies the number of parameters expected by the assert statement. It detects as a test smell only methods with two parameters (lines 14). The problem occurs because the tool always classifies the `assertEquals` as a non-test smell when the assert has three parameters. However, it is necessary to verify the fourth parameter to decide whether it is either a test smell or not. We improved the `JNose Test` in this direction.

Additionally, there was a conflict in the EpT, and UT test smells definition. The EpT test smell is a test method without executable statements (empty method). The UT test smell is a test method with executable statements but no assertions. The `tsDetect` considers methods without a body as both EpT and UT. Therefore, we implemented the rules necessary to differentiate those test smells. We performed some minor fixes to detect other types of test smells. For example, for the VT test smells, the `tsDetect` considers a class with more than 123 lines as one verbose test. As the `JNose Test` detects the test smells at a fine-grained level, we defined that a test method with more than 30 lines is verbose. Therefore, we found more instances because of our definition.

```
1  @Test(expected= Exception.class)
2  public void tag_usage(){
3      // Some code
4  }
5
6  @Test
7  void trows_statement_usage() {
8      assertThrows("Exception Message", Exception.
           class, parameter);
9  }
10
11 @Test
12 public void trows_signature_usage() throws
       Exception.class {
13     // Some code
14 }
15
16 @Test
17 public void try_catch_usage() {
18   try {
19     // Some code
20   } catch (MyException e) {
21     Assert.fail(e.getMessage());
22   }
23 }
```

Listing 3: (Non)Compliant Solutions for ECT considered by `JNose Test`

6.2 JNose and Manual Analysis Comparison

Table 6 reports accuracy through precision and recall values when detecting test smells with `JNose Test` and manual analysis. This comparison considered the granularity level for the test smells detection.

In a fine-grained level, the `JNose Test` precision score ranges from 84% to 100%, and the recall ranges from 47% to 100%. At the class level, the detection difficulties related

```
1  @Test
2  public void two_parameters(){
3      assertEquals(float expected, float actual)
4  }
5
6  @Test
7  public void three_parameters_with_message(){
8      assertEquals(String message, float expected,
           float actual)
9  }
10
11 @Test
12 public void four_parameters(){
13     assertEquals(String message, float expected,
           float actual, float delta)
14 }
15
16 @Test
17 public void three_parameters_no_message(){
18     assertEquals(float expected, float actual,
           float delta)
19 }
```

Listing 4: Solutions for AR considered by `JNose Test`

Table 6. `JNose Test` and Manual Analysis Comparison - Fine granularity level

Test Smell	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
AR	100	100	100	100
CI	100	100	100	100
CTL	100	100	100	100
DA	94.12	100	94	97
ECT	100	100	100	100
ET	89.13	100	89	94
EpT	100	100	100	100
GF	90	100	90	95
LT	96.55	100	97	98
MG	50	100	50	67
MNT	94.74	100	95	97
PS	100	100	100	100
RA	100	100	100	100
RO	47.06	84	47	60
SE	100	100	100	100
ST	100	100	100	100
UT	100	100	100	100
VT	100	100	100	100

to specific cases are not evident because it returns a Boolean value for test smells in the whole test class. However, when we performed a more detailed test smell detection, we noticed some test code-specific characteristics that the tool does not detect.

The most divergent results between the class- and fine granularity-level are the MG and RO test smells. At the class level, those test smells have the accuracy of 90.77% and 89.23%, respectively. However, those test smells present accuracy of 50% and 47.06%, respectively. Both the test smells to deal with external resources. A test method that makes optimistic assumptions about external resources' existence has the RO test smell (Listing 5, lines 10-21). The test method that uses external resources has the MG test smell (Listing 5, lines 2-5). As the `JNose Test` performs test code static analysis, we only considered the direct calls for external resources (Listing 5, lines 1-15). However, whether a test method calls a production class from any part of the project and that class calls for external resources, the test class uses

external resources indirectly (Listing 5, lines 17-21). In this scenario, the MG and RO test smells need additional work to determine the indirect calls.

We identified a specific characteristic that can detect other false positives instances using the DA test smell. That false positive occurs when one test method uses an assertion structure implemented by a JSON library similar to the assertion structure implemented by the JUnit. This is because the JUnit has the `assertThat(String reason, T actual, M matcher)` the other JSONAssert library implements the `assertThat(String).contains(String)`. When performing the static analysis, all the statements that start with `assert` were considered a JUnit assertion. Therefore, we may improve it by detecting the libraries imported in the test class. However, the tool might miss test smells instances if using a test class with another assert library.

Other types of test smell required minor fixes. The LT and ET test smell miss some instances due to default constructors. We considered that the same way a different test method should not call the same production class method, a class is instantiated several times in different test methods. If many test methods need to instantiate the same object, it should be moved to a setup method. Therefore, we need to improve the `JNose Test` to detect calls for the default constructors.

```
1  @Test
2  public void external_File(){
3      File file = openFile("config.xml");
4      if(file.exists()){
5          XmlPullParser config = XmlParserFactory.
               fromFile(file);
6          // Some code
7      }
8  }
9
10 @Test
11 public void external_File_Without_Checking(){
12     File file = openFile("config.xml");
13     XmlPullParser config = XmlParserFactory.
           fromFile(file);
14     // Some code
15 }
16
17 @Test
18 public void external_Resource_Indirectly(){
19     XmlReader reader = new XmlReader("xml/config.
           xml");
20     // Some code
21 }
```

Listing 5: `Mystery Guest and Resource Optimus`

7 Related Work

In large-sized test suites, software engineers barely perform manual detection of test smells. This practice is rather time-consuming and infeasible in many scenarios. Therefore, the research community has proposed automated tool support for detecting test smells.

The `Test Smell Detector (TSD)` detects nine types of test smells (Bavota et al., 2015). The `TSD` detection rules overestimate the presence of test smells in the code to ensure high recall (87%). It returns a list of candidate-affected classes. Similarly, `tsDetect`, the state-of-the-art tool to detect test smells, identifies twenty-one types of test smell (Section 2). It indicates whether a particular test smell appears in the test class with the precision score ranging from 85% to 100%, and recall score from 90% to 100% (Peruma et al., 2020).

Other tools correlate test smells with structural and coverage metrics. The IntelliJ plug-in coined VITRuM (VIzualization of Test-Related Metrics) is an extension of `tsDetect`. It collects a set of seven types of test smells and structural metrics (Pecorelli et al., 2020). TeReDetect (Negar and Garousi, 2010) and TeCReVis (Koochakzadeh and Garousi, 2010) use code coverage analysis, held by CodeCover, to detect test smells related to code duplication.

Our tool uses a test smells rule-based detection instead of a metric- or coverage-based detection. It extends the `tsDetect` tool in several respects. For example, our tool provides the number of test smells identified in a test class and the method line and name with each test smell's location. Moreover, it supports the test suite analysis through several project versions, by mining Git for providing information about when and by who introduced the test smells.

Additionally, our tool supports other tools for test smells refactoring (`RAIDE`) (Santana et al., 2020) and visualization (`TSVizzEvolution`). The `RAIDE` is an Eclipse IDE plugin to detect and refactor the AR and DA test smells. The `TSVizzEvolution` is a test smells visualization tool that aims to help the user understand problems in the test code by using three visualization techniques (*Graph View*, *Treemap View*, and *Timeline View*). It represents the twenty-one types of test smells detected by `JNose Test`.

8 Threats to Validity

Internal Validity. In the manual analysis to construct the oracle, there may have been divergences among the researchers' analysis. We mitigated this threat by resolving disagreements collectively. After collecting data with the `JNose Test` and `tsDetect` tools, we checked if any test smells detected by the tools were not considered in the manual analysis.

External Validity. Our study results may not be generalized to other suites of test classes or other types of test smells. To mitigate this threat, we used the same dataset used in the study to validate the `tsDetect` tool (Peruma et al., 2020).

Conclusion Validity. Although the `JNose Test` detects twenty-one types of test smells, this study only validated eighteen ones because the dataset used did not have the DpT, DT, and IgT test smells. On the other hand, we used the same dataset used to evaluate `tsDetect` (Peruma et al., 2020).

Construct Validity. Although we used four coders to build the Oracle, they were experts with more than three years of experience with test smells. They were aware of the test code of the test smells detection tools.

9 Conclusion

This paper presents the `JNose Test` and its API, the `JNose-Core`. The API supports the detection of twenty-one types of test smells. It provides a flexible architecture to sup-

port the insertion of new test smells detection rules. The `JNose Test` tool is a web application to detect test smells and calculate coverage for Java projects.

To validate the detection rules implemented by `JNose-Core`, we conducted an empirical study to compare our tool's accuracy with the state-of-the-art tool and manual analysis. We built an oracle to detect test smells to perform the comparison. The oracle contains sixty-five test classes analyzed by specialists in the subject. The comparison between `JNose` and `tsDetect` was made at the class-level.

The results showed that `JNose` presented higher accuracy than `tsDetect`, in terms of precision and recall. As we reused the detection rules from the `tsDetect` to implement the `JNose Test`, the results indicated that we successfully improved them. Additionally, the `JNose` also detects test smells at a fine-grained level. As the `tsDetect` does not support this feature, we could only compare the fine-grained level detection against the manual analysis. Results showed a high accuracy to determine the exact line location, but it still needs further improvements.

There are many opportunities for other investigations. For example, it would be interesting to validate our tool efficiency in a real-world environment through a user study. Such a study could also consider significant usability concerns. There is open room for introducing new features in the `JNose Test` in terms of both detection and refactoring, and as necessary, in terms of how it behaves in practice considering quality attributes.

Acknowledgements

This research was partially funded by INES 2.0; CNPq grants 465614/2014-0 and 408356/2018-9 and FAPESB grants JCB0060/2016 and BOL0188/2020.

References

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., and Binkley, D. (2015). Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094.

Bavota, G., Qusef, A., Oliveto, R., Lucia, A., and Binkley, D. (2012). An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *28th IEEE International Conference on Software Maintenance (ICSM)*.

Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., and Marinov, D. (2018). DeFlaker: Automatically Detecting Flaky Tests. In *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 433–444.

Capgemini (2018). World Quality Report 2018-19. `https://www.capgemini.com/service/world-quality-report-2018-19/`. Accessed: March 1st, 2021.

CISQ (2021). The Cost of Poor Software Quality in the US: A 2020 Report. `https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf`. Acessed: March 1st, 2021.

Deursen, A., Moonen, L. M., Bergh, A., and Kok, G. (2001). Refactoring test code. In *Refactoring Test Code*, Amsterdam, The Netherlands, The Netherlands. CWI (Centre for Mathematics and Computer Science).

Garousi, V. and Küçük, B. (2018). Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52 – 81.

Gopinath, R., Jensen, C., and Groce, A. (2014). Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, New York, NY, USA. ACM.

Grano, G., Palomba, F., Di Nucci, D., De Lucia, A., and Gall, H. C. (2019). Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software*, 156:312–327.

Greiler, M., van Deursen, A., and Storey, M. (2013). Automated detection of test fixture strategies and smells. In *IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 322–331.

Guerra Calle, D., Delplanque, J., and Ducasse, S. (2019). Exposing Test Analysis Results with DrTests. In *International Workshop on Smalltalk Technologies*, pages 1–5, Cologne, Germany. HAL.

Hallgren, K. A. (2012). Computing inter-rater reliability for observational data: an overview and tutorial. *Tutorials in quantitative methods for psychology*, 8(1):23.

Junior, N. S., Rocha, L., Martins, L. A., and Machado, I. (2020). A survey on test practitioners' awareness of test smells. In *Proceedings of the XXIII Iberoamerican Conference on Software Engineering, CIbSE 2020*, pages 462–475. Curran Associates.

Koochakzadeh, N. and Garousi, V. (2010). TeCReVis: A Tool for Test Coverage and Test Redundancy Visualization. In Bottaci, L. and Fraser, G., editors, *Testing – Practice and Research Techniques*, pages 129–136, Berlin, Heidelberg. Springer Berlin Heidelberg.

Meszaros, G., Smith, S. M., and Andrea, J. (2003). The test automation manifesto. In Maurer, F. and Wells, D., editors, *Extreme Programming and Agile Methods - XP/Agile Universe 2003*, Berlin, Heidelberg. Springer Berlin Heidelberg.

Negar, K. and Garousi, V. (2010). A tester-assisted methodology for test redundancy detection. *Advances in Software Engineering*, 2010.

Palomba, F., Zaidman, A., and Lucia, A. D. (2018). Automatic test smell detection using information retrieval techniques. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 311–322, Madrid, Spain. IEEE.

Pecorelli, F., Di Lillo, G., Palomba, F., and De Lucia, A. (2020). VITRuM: A Plug-In for the Visualization of Test-Related Metrics. In *Proceedings of the International Conference on Advanced Visual Interfaces*, New York, NY, USA. ACM.

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

Engineering (CASCON), Riverton, NJ, USA. IBM.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2020). *TsDetect: An Open Source Test Smells Detection Tool*. ACM, New York, NY, USA.

Santana, R., Martins, L., Rocha, L., Virginio, T., Cruz, A., Costa, H., and Machado, I. (2020). RAIDE: A Tool for Assertion Roulette and Duplicate Assert Identification and Refactoring. In *Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES)*. ACM.

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., and Bacchelli, A. (2018). On the relation of test smells to software code quality. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12. IEEE.

Spadini, D., Schvarcbacher, M., Oprescu, A.-M., Bruntink, M., and Bacchelli, A. (2020). Investigating severity thresholds for test smells. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*. ACM.

Virginio, T., Martins, L., Soares, L. R., Railana, S., Costa, H., and Machado, I. (2020). An empirical study of automatically-generated tests from the perspective of test smells. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering (SBES)*, New York, NY, USA. ACM.

Virginio, T., Santana, R., Martins, L. A., Soares, L. R., Costa, H., and Machado, I. (2019). On the influence of test smells on test coverage. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering (SBES)*, pages 467–471, New York, NY, USA. ACM.

Virgínio, T., Martins, L., Santana, R., Cruz, A., Rocha, L., Costa, H., and Machado, I. (2021). On the test smells detection: an empirical study on the JNose Test accuracy [Dataset]. Available at: `https://doi.org/10.5281/zenodo.4570751`.

Yusifoğlu, V. G., Amannejad, Y., and Can, A. B. (2015). Software test-code engineering: A systematic mapping. *Information and Software Technology*, 58:123 – 147.