Refactoring Assertion Roulette and Duplicate Assert test smells: a controlled experiment

Railana Santana[1], Luana Martins[1], Tássio Virgínio[2], Larissa Soares[1,3], Heitor Costa[4], Ivan Machado[1]

[1] Federal University of Bahia (UFBA) – Salvador, BA – Brazil
[2]Federal Institute of Tocantins (IFTO) – Paraíso do Tocantins, TO – Brazil
[3]State University of Feira de Santana (UEFS) – Feira de Santana, BA – Brazil
[4]Federal University of Lavras (UFLA) – Lavras, MG – Brazil

railana.santana@ufba.br, martins.luana@ufba.br,
lrsoares@uefs.br, tassio.virginio@ifto.edu.br,
heitor@ufla.br, ivan.machado@ufba.br

Abstract. *Test smells can reduce the developers' ability to interact with the test code. Refactoring test code offers a safe strategy to handle test smells. However, the manual refactoring activity is not a trivial process, and it is often tedious and error-prone. This study aims to evaluate* `RAIDE`*, a tool for automatic identification and refactoring of test smells. We present an empirical assessment of* `RAIDE`*, in which we analyzed its capability at refactoring Assertion Roulette and Duplicate Assert test smells and compared the results against both manual refactoring and a state-of-the-art approach. The results show that* `RAIDE` *provides a faster and more intuitive approach for handling test smells than using an automated tool for smells detection combined with manual refactoring.*

1. Introduction

Writing automated tests requires more than just understanding the business rules implemented in the source code, as the test engineer should be skilled enough to build well-structured test cases. In addition, the complexity of the system under test aligned with the lack of knowledge and experience may lead test engineers to use bad practices to either design or implement the test code [Garousi and Küçük 2018].

Bad design practices in the test code are commonly referred to as test smells [Van Deursen et al. 2001]. Test smells have recently gained importance given their effects on the performance of software testing activities, especially from a maintenance perspective [Virgínio et al. 2020b, Aljedaani et al. 2021]. For instance, *Empty Test* is a test smell that occurs when a test method does not contain any executable instructions. Since the method does not have a body, the test always passes. When developers introduce behavior-breaking changes, an empty test does not notify alternated outcomes [Van Deursen et al. 2001, Peruma et al. 2019].

While good testing practices and guidelines can prevent test smells, test engineers do not always follow them. In particular, whether a software project already comprises a large set of tests, it may not be cost-effective to create novel tests from scratch. An alternative is to employ test-specific refactoring strategies to improve the test code quality without changing its behavior [Van Deursen et al. 2001]. Due to the lack of test suites aimed to test themselves, there is a need for automated tools to refactor the test code and keep its behavior.

The literature has introduced a small set of automated tools to refactor test code [Aljedaani et al. 2021]. For example, `RTj` [Martinez et al. 2020] is a command-line tool that supports detecting and refactoring Rotten Green Tests, i.e., a test that passes during execution but has assertions rarely executed. `DARTS` [Lambiase et al. 2020] is an IntelliJ plugin that supports detecting and refactoring three test smells (General Fixture, Eager Test, and Lack of Cohesion of Test Methods). These tools address a small number of test smells when using the JUnit framework to develop test cases. [Aljedaani et al. 2021] point out that the existing tools do not provide details concerning the accuracy of their refactoring capabilities or usability. To expand the set of refactoring strategies for test smells, we previously presented a systematic approach to detect and refactor two test smells: *Assertion Roulette* (AR) and *Duplicate Assert* (DA). In addition, we implemented `RAIDE`, automated tool support for test smell refactoring [Santana et al. 2020]. `RAIDE` is an open-source tool with a user-friendly interface that can detect and refactor test smells with just a few clicks.

In this paper, we present an empirical study to evaluate `RAIDE`. We aimed to answer the Research Question: ***How does `RAIDE` support users to detect test smells and refactor the test code?*** As the related tools do not support refactoring strategies for the AR and DA test smells, we compared `RAIDE` with manual refactoring. We asked twenty test engineers to refactor test code from two projects. While using `RAIDE`, the test engineers had access to an interface integrated into the Eclipse IDE. They could detect and refactor the test smells. Otherwise, for the manual refactoring, the test engineers used the `tsDetect` to detect the test smells. This is a state-of-the-art test smell detection tool [Peruma et al. 2019], Given that `tsDetect` does not allow automated refactoring, the participants had to use their strategies to refactor the test smells.

In summary, we contributed with a controlled experiment to compare `RAIDE` with one state-of-the-art approach and discuss the usefulness of an IDE-integrated tool that automatically detects and refactors test smells. Controlled experiment findings can support the community of researchers and developers in building and maintaining intuitive tools to detect and refactor test code automatically. Besides, we collected the participants' perceptions of the tools, e.g., the `RAIDE` and `tsDetect` limitations, a feedback key to the continuity and evolution of these tools.

The remainder of this paper is structured as follows. Section 2 introduces the concept of test smells and the `RAIDE` and `tsDetect` tools. Section 3 details the experiment design and the main results. Section 4 discusses the evaluation results. Section 5 presents the threats to the validity. Section 6 presents related work. Section 7 concludes the paper.

2. Background

Test smells result from bad design choices implemented in the test code [Greiler et al. 2013b]. Smells in test code can affect its quality, mainly understandability and maintainability. Consequently, test smells can reduce the effectiveness of test cases to detect faults and the developers' ability to interact with the test code [Yusifoğlu et al. 2015].

Although there are several test smells, some of them are more prevalent. [Palomba et al. 2016] empirically evaluated the diffusion of test smells in automatically generated JUnit test classes in 110 open-source software projects. The results showed that

83% of those classes are affected by at least one test smell. The most frequent test smells were the *AR* (54%) and *Test Code Duplication* (33%). In our study, we considered *DA*, once it is a representative type of *Test Code Duplication* test smell.

In addition, [Peruma 2018] conducted a large-scale empirical study on the test smells occurrence, distribution, and impact in the maintenance of open-source Android applications. They also observed that the *AR* test smell occurred in more than 50% of the test classes. In a multivocal literature review, [Garousi and Küçük 2018] reported the most extensive catalog of test smells and a summary of guidelines, techniques, and tools to handle test smells. The authors pointed out that test smells related to code duplication and code complexity (test redundancy and long test, respectively) have been the most discussed ones in the literature.

The recurring number of studies reporting on the occurrence of the *AR* test smell [Peruma 2018, Palomba et al. 2016] and the repercussion on test smells related to code duplication [Garousi and Küçük 2018] led us to investigate those two test smells and then propose `RAIDE`. We next introduce these test smells.

2.1. Assertion Roulette (AR)

In JUnit, *assertions* have an optional first argument of the String type to explain what each *assertion* is testing. AR occurs when a test method has several undocumented *assertions*, making understanding difficult during maintenance and challenging to detect the *assertion* if the method fails. Listing 1 presents a code excerpt with multiple *assertions* without a return message (lines 93 to 95). The example presents a method from the `TestAbstractPartial.java`[1] test class of the **Joda-Time** project.

```
90 public void testGetValues() throws Throwable {
91     MockPartial mock = new MockPartial();
92     int[] vals = mock.getValues();
93     assertEquals(2, vals.length);
94     assertEquals(1970, vals[0]);
95     assertEquals(1, vals[1]);
96 }
```

Listing 1. Test code with the Assertion Roulette test smell

Possible Effect: Multiple *assertion* statements in a test method without a descriptive message can affect test readability, comprehensibility, and maintainability. Multiple *assertions* make it difficult to detect which assertion gave an error in a test failure.

Detection: To check if a test method has *assertions* without explanation/message (parameter in the *assertion* method).

Refactoring: To include *assertion* explanations in each *assertion*. Listing 2 shows the Listing 1 code refactored with the appropriate explanations for each assert (text highlighted in yellow).

2.2. Duplicate Assert (DA)

DA occurs when a test method tests the same condition multiple times in the same test method [Peruma et al. 2019]. Listing 3 shows a code excerpt with two *assertions* with

[1] Available at `https://bit.ly/35Q56KV`

```
90  public void testGetValues() throws Throwable {
91      MockPartial mock = new MockPartial();
92      int[] vals = mock.getValues();
93      assertEquals("Vals size 2", 2, vals.length);
94      assertEquals("Year Equal 1970", 1970, vals[0]);
95      assertEquals("Month 1", 1, vals[1]);
96  }
```

Listing 2. Test code after refactoring the Assertion Roulette test smell

the same parameters (lines 361 and 363). The example presents one method from the `TestPeriodFormatterBuilder.java`[2] test class of the **Joda-Time** project.

```
356 public void testPluralAffixParseOrder() {
357     PeriodFormatter f = builder.appendDays()
358         .appendSuffix("day", "days").toFormatter();
359     String twoDays = Period.days(2).toString(f);
360     Period period = f.parsePeriod(twoDays);
361     assertEquals(Period.days(2), period);
362     period = f.parsePeriod(twoDays.toUpperCase(Locale.ENGLISH));
363     assertEquals(Period.days(2), period);
364 }
```

Listing 3. Test code with the Duplicate Assert test smell

Possible Effect: That test smell hinders test readability and maintenance, as there are repeated assertions (with the same parameters) without explaining the purpose/objective of the test method. In general, DA creates a scenario that violates the responsibility of each method to fulfill a single objective.

Detection: To check if the test method contains two or more *assertion* statements with the same parameters.

Refactoring: To create one test method for testing the same condition with different values. Listing 4 shows a code excerpt from Listing 3 refactored (text highlighted in yellow), which extracts the duplication for a new method (`testPluralAffixParseOrderExtracted`).

```
356 public void testPluralAffixParseOrder() {
357     PeriodFormatter f = builder.appendDays().
358         appendSuffix("day", "days").toFormatter();
359     String twoDays = Period.days(2).toString(f);
360     Period period = f.parsePeriod(twoDays);
361     assertEquals(Period.days(2), period);
362 }
363
364 /* Extracted Method */
365 public void testPluralAffixParseOrderExtracted() {
366     PeriodFormatter f = builder.appendDays().
367          appendSuffix("day","days").toFormatter();
368     String twoDays = Period.days(2).toString(f);
369     Period period = f.parsePeriod(twoDays.toUpperCase(Locale.ENGLISH));
370     assertEquals(Period.days(2), period);
371  }
```

Listing 4. Test code after refactoring the Duplicate Assert test smell

[2]Available at `https://bit.ly/3oriGL1`

2.3. tsDetect

tsDetect has been reported in a recently published literature review as a comprehensive tool for detecting test smells in Java projects [Aljedaani et al. 2021]. The tool covers 19 test smells with a precision score ranging from 85% to 100% and a recall score from 90% to 100% in open-source Android apps [Peruma et al. 2019]. Given those precision and recall scores, researchers built tsDetect-based tools [Virgínio et al. 2020a, Kim et al. 2021]. tsDetect tool indicates the existence of test smells in a test class based on a three-step detection process (Figure 1): 1) *Test File Detector* - reads the project test files; 2) *Test File Mapping* - links the test files to the production files under test; and 3) *Test Smell Detector* - detects smells in test code.

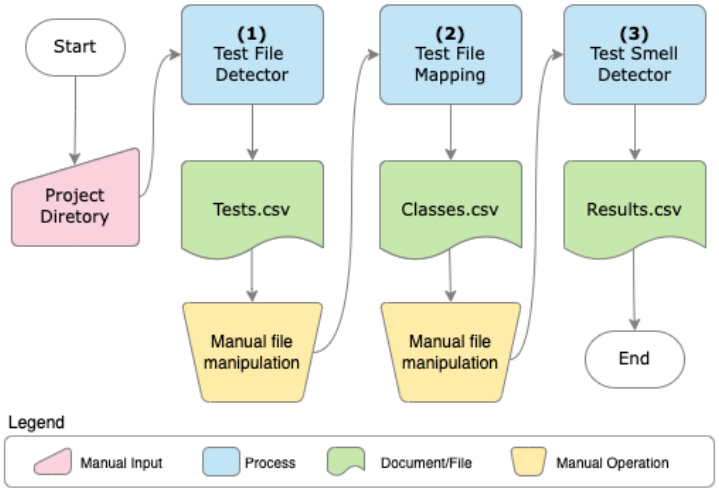

Figure 1. Process for running the tsDetect tool

In step (1), *Test File Detector* generates the Tests.csv file, which contains the path of the test classes from one software project. That file is input to *Test File Mapping*. Next, step (2) establishes the relationship between the test and production classes. It creates the Classes.csv file, which contains the project name, the path of each test class, and the production classes. That file is input to *Test Smell Detector*. Step (3) is responsible for analyzing test smells for the project based on the Classes.csv file. The output is the Results.csv file, which indicates the presence (true) or absence (false) of test smells in the test classes.

2.4. RAIDE

RAIDE is an AST (Abstract Syntax Tree)-based tool developed as an Eclipse open-source plugin to detect and refactor test smells [Santana et al. 2020]. We reused rule-based components from tsDetect and performed improvements to detect test smells. Whereas tsDetect works as a command-line tool that indicates the presence of test smells, RAIDE has a user-friendly Graphical User Interface (GUI), which identifies and indicates the exact location (code lines) of the AR and DA test smells. Besides, it includes one feature for automated refactoring of those test smells.

For the test code refactoring activity to succeed, the detection must be precise and explicit, pointing to the source code line where the test smell is located. However, not all tools report the exact location of test smells. For instance, the tsDetect only informs

whether a class is affected by a test smell. Thus, the users need to analyze the entire test class to identify the test smells and refactor them. Conversely, RAIDE exhibits the test smells exact location for users and provides a user-friendly GUI.

RAIDE uses graphical components from JDeodorant[3], an Eclipse plugin, to detect and refactor code smells in java code and reuses the components tsDetect tool: i) AST of the project, responsible for detecting test classes and code structure; and ii) AR test smell detection rules. In addition, we improved the detection rules to meet the scenario with an empty string ("") or space string (" ") in the explanation parameter and inform the line affected by the test smell. The way tsDetect implements the DA test smell detection does not allow code reuse for accurately detecting each test smell in RAIDE. Therefore, we built modules from scratch in RAIDE to detect and refactor the DA test smell and to refactor the AR test smell. Some limitations of RAIDE include: i) the implementation of detection rules is based on JUnit 4, other JUnit versions may require adaptations in such rules; and ii) the tool's execution detects the *Assertion Roulette* or *Duplicate Assert*, not both at the same time.

Figure 2 shows the RAIDE tool process to identify and refactor test smells. The user should provide as input: the **test package** of the project under analysis, and the test smell the plugin should detect and refactor. In step (1), the **test classes detection** identifies all JUnit classes in the **test package**. In step (2), the **test smells detection** detects a specific type of test smell and presents the **test smell detection results** in an Eclipse view. In step (3), **manual test smells selection** requires the user intervention to select which test smell instances(s) he would like to refactor. Then, in step (4), Test smells refactoring shows the user how to refactor the code, and the user can take the decision of accepting the **refactored test code**.

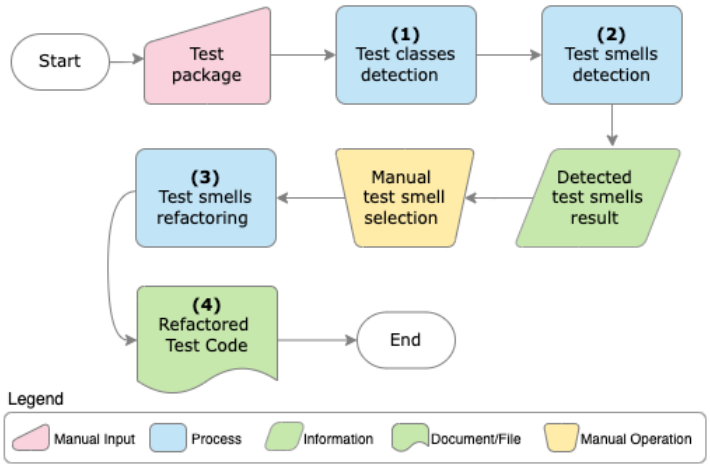

Figure 2. Process for running the RAIDE tool

Figure 3 shows a screenshot of the Eclipse IDE with the RAIDE plugin running on the Joda-Time[4] project. RAIDE refactored line 131 of method testGetFieldTypes() after detecting the AR test smell. RAIDE included the explain parameter "Add Assertion Explanation here" to correct that test smell. The user

[3]Available at https://github.com/tsantalis/JDeodorant
[4]Available at https://github.com/JodaOrg/joda-time

must replace the default string with an explanatory message about the assertion to remove the AR test smell from the code. In Figure 3, it is also possible to see that RAIDE also detected the AR test smell on line 132. After double-clicking on the detected test smell, the tool redirects the user to the highlighted line.

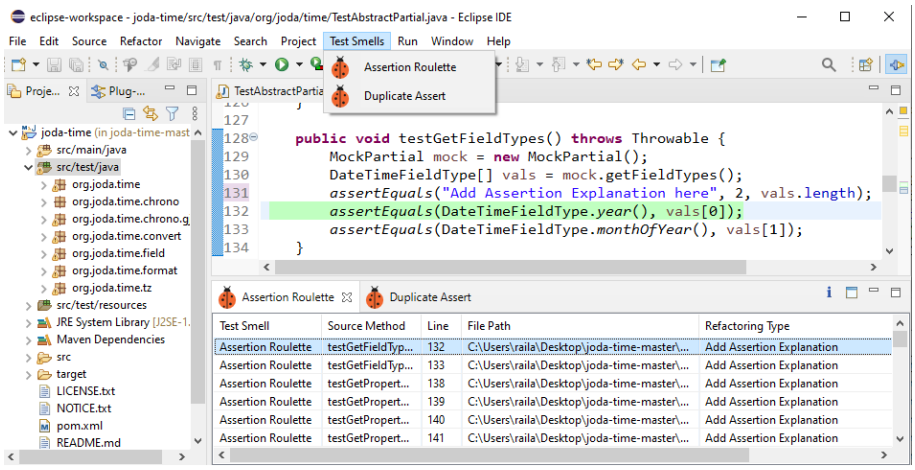

Figure 3. Screenshot of the RAIDE plugin under execution

3. Empirical Assessment

In this study, we carried out a controlled experiment to evaluate how the automated process proposed by RAIDE assists the test engineers to (i) detect test smells and (ii) refactor test code. To answer the main research question (RQ), we split it into two sub-RQs: **RQ_1)** How does RAIDE facilitate the AR and DA test smells detection compared to tsDetect? **RQ_2)** How does RAIDE facilitate the smelly test code refactoring with the AR and DA test smells compared to manual refactoring? For RQ_1, we compared RAIDE and tsDetect concerning how both detect test smells and show the results. For RQ_2, we compared refactorings with RAIDE to manual refactorings. Thus, we measured the participants' time taken to identify the test smells in the test code with the support of RAIDE and tsDetect. After, we measured the refactorings time performed manually and with RAIDE.

Experiment Overview. Before joining the experiment session, the participants filled out an online form with questions regarding their experience in software programming and testing, Java language, JUnit framework, Eclipse, and test smells. Next, we presented the concepts about test smells (AR and DA, particularly) and detection and refactoring processes. We also informed the objective of the experiment. Figure 4 shows an overview of the experiment steps. The participants performed two tasks, one for RAIDE and the other for tsDetect. Each task encompassed the analysis of a different software project. Before running the experiment, the participants received training on using and executing the tool shortly after. We used the same project in the training sessions of both tools (*Project 1* - Figure 4). After training, the participants performed the actual experiment tasks, as follows: (i) they used RAIDE and tsDetect to detect two test smells (AR and DA) and informed the location (code lines) of the test smells, and (ii) they refactored the code to remove the two test smells. Tasks 1 and 2 are similar, but the participants used different tools (either *Tool 1* or *Tool 2*) and projects (either *Project 2* or *Project 3*) (Figure

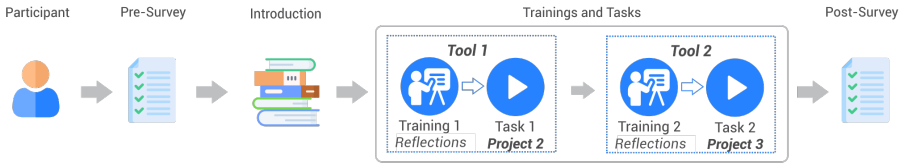

Figure 4. Experiment Flow

4). In the end, the participants answered an online post-survey[5], comprising questions about the experiment execution and their perception of each tool.

Pilot Study. We performed a pilot study with five participants, two graduate students, two practitioners, and another acting in both roles. We found that they took 47 seconds, on average, using `RAIDE` and 15 minutes, on average, using `tsDetect` to detect the test smells. The pilot study was critical in reviewing the concepts and standardizing the training, especially the commands needed to use `tsDetect`. It also was helpful to assess whether the participants could understand the tasks and tools. Data gathered in this pilot study was not considered in the final analysis.

Participants. We recruited twenty participants, where ten participants (50%) were from the academy, five participants (25%) were from industry, and five participants (25%) acted in both roles. All participants held at least a B.Sc. degree, seven participants (35%) were M.Sc. students, and seven participants (35%) were Ph.D. students. They come from sixteen different Brazilian institutions, eight different universities (U1 to U8), and eight different software development companies (C1 to C8). Some of them also belong to more than one institution (P1, P7, P12, P15, P17, P18, and P20). In addition, they have different roles: Agile Coach (AC), Developer (D), Database Admin (DA), Lecturer (L), Researcher (R), and Requirements Analyst (RA). Most participants (80%) had already heard about test smells. Six participants (30%) had specific knowledge about test smell in either industry or academia. Table 1 shows detailed information. Although the participant's profile was collected, we did not investigate the relationship between their experience and performance in carrying out the tasks.

Experiment Material and Tasks. We selected three open-source projects for this experiment: *Reflections* project for training and *Joda-Time* and *Commons-collections* projects for executing tasks 1 and 2 (*Project 2* and *Project 3* - Figure 4, but not necessarily in that order). We chose the projects considering the limitations of `RAIDE` and `tsDetect`, which support Maven projects and tests with JUnit (version 4). Due to the size of the projects, we decided to consider only two test classes for the experiment. Therefore, the *Reflections*, *Joda-Time*, and *Commons-collections* projects would have a similar complexity level, number of methods, and number of test smells. Regarding the test smells in the projects, one method has the AR test smell (4 *assertions* without explanation), and another one has two pairs of the DA test smell (4 *assertions* in the same method).

Design and Procedure. In our experiment, we used a crossover design [Vegas et al. 2015] to avoid the learning effect, as the participants performed two tasks in a row. The projects and the tools are the independent variables, and time is the dependent

[5]Available at `https://zenodo.org/record/5978022#.Yf55hOrMKUk`

Table 1. Participants' profile and experience (in years)

ID	Group	Education	Profile	Institution	Programming	JAVA	Eclipse	Testing	JUnit	Test Smells
P01	1	Ph.D. student	R/L	U1/U2	10+	10+	10+	10+	1⊢ 5	✓
P02	2	M.Sc. student	R	U1	10+	0-1	0-1	5⊢ 10	0-1	✳
P03	3	Ph.D. student	R	U1	0-1	0	1⊢5	1 ⊢5	1⊢ 5	✓
P04	4	B.Sc.	RA	C1	1⊢5	1⊢5	1⊢5	1⊢5	0-1	✗
P05	1	B.Sc.	D	C2	5⊢10	1⊢5	0-1	0-1	0-1	✓
P06	2	M.Sc. student	R/L	U1	5⊢10	1⊢5	1⊢5	1⊢5	1⊢5	✓
P07	3	Ph.D. student	R/L	U1/U3	1⊢5	1⊢5	1⊢5	5⊢10	0-1	★
P08	4	B.Sc.	L	U4	5⊢10	1⊢5	1⊢5	0-1	0	✗
P09	1	B.Sc.	D	C1	1⊢5	1⊢5	0	0-1	0-1	✓
P10	2	B.Sc.	D	C3	1⊢5	0-1	0-1	0-1	0	✗
P11	3	M.Sc. student	R/DA	U5	1⊢5	1⊢5	1⊢5	0	0	✳
P12	4	Ph.D. student	R/L	U1/U6	5⊢10	1⊢5	1⊢5	0-1	0-1	★
P13	1	Ph.D. student	R	U7	10+	1⊢5	1⊢5	0	0	★
P14	2	Ph.D. student	R	U1	1⊢5	1⊢5	0-1	0-1	0	☆
P15	3	M.Sc. student	R/D	U1/C4	1⊢5	1⊢5	0-1	0	0	✓
P16	4	B.Sc.	D	C5	5⊢10	1⊢5	1⊢5	1⊢5	1⊢5	✓
P17	1	M.Sc. student	R/AC	U5/C6	5⊢10	5⊢10	5⊢10	0-1	0-1	✓
P18	2	M.Sc. student	R/D	U1/C7	5⊢10	1⊢5	1⊢5	0	0	★
P19	3	Ph.D. student	R	U8	10+	10+	10+	10+	10+	✓
P20	4	M.Sc. student	R/D	U1/C8	1⊢5	1⊢5	1⊢5	0	0	✗

Labels: (✗) Never heard about them; (✓) Already heard anything about itthem, but had never worked with them; (☆) Knew a little bit about them; (★) Heard about test smells and had already worked with them; and (✳) Researcher investigating the topic of test smells.

variable of the experiment. During the experiment with each participant, we captured the audio and the computer screen to count later the time spent by them to detect and refactor the test smells in each task of the experiment. When a participant inaccurately identified or refactored the test smell, the researcher reported that the task had not been completed yet, and the time continued counting until the task be completed correctly.

Data Analysis. We performed the Shapiro-Wilk test, with a significance level of 5%, to verify the data distribution for the data analysis. As a result, the data distribution is not normal. Then, we selected the Mann-Whitney paired test [Mann and Whitney 1947], with a significance level of 5%, to answer RQ_1 and RQ_2 (sub-RQs). We defined the null hypothesis to investigate the RQ_1: *The detection time of AR and DA test smells with* `RAIDE` *is similar to detection with* `tsDetect`. Regarding the RQ_2, we defined the null hypothesis: *The refactoring time of AR and DA test smells with* `RAIDE` *is similar to manual refactoring.*

4. Results and Discussion

This section presents and discusses the results gathered from the empirical assessment.

4.1. Detection of Test Smells (RQ1)

We collected and analyzed the time spent by the participants to complete the task of detecting and locating the AR and DA test smells with `RAIDE` and `tsDetect` to answer RQ_1. The participants completed that task in 63.95 seconds (s) on average with a standard deviation (*sd*) of 28.12s using `RAIDE` and 679.20s on average with *sd* = 248.71s using `tsDetect`. Thus, those values suggest that detecting and locating the AR and DA test smells with `tsDetect` is slower than with `RAIDE`.

The difference between RAIDE and tsDetect is the first one to be an IDE-integrated tool, in which participants select the test smell they want to analyze and report the lines with the smells highlighted by the tool. In tsDetect, the participants need to run three different tools, open a .csv file to check which classes have test smells, and manually inspect the source code. Therefore, the participants spend most of the time using the Test File Detector and Test File Mapping tools and performing adjustments in the .csv files. Although the tools are similar in execution time, they present a wide difference in their efficiency regarding how fast they place the user in front of the test smells.

We also analyzed whether there is a statistically significant difference in identifying the AR and DA test smells. Since our data did not have a normal distribution (*p-value* = 7.748e-05), we performed the non-parametric Mann-Whitney test. The test indicated a significant difference (*p-value* = 9.537e-07) between the average time spent using tsDetect and RAIDE. Thus, we refuted the null hypothesis ($H_0 1$) and answered RQ_1. We also observed that data resulting from the time measures with tsDetect is more dispersed than the time using RAIDE. Figure 5 shows boxplots on a logarithmic scale comparing data gathered from RAIDE and tsDetect. There is no overlap between the boxes of RAIDE and tsDetect, which means a difference between them. In addition, the longest reported time for detecting the AR and AD test smells with RAIDE was much shorter than the shortest time to detect them with tsDetect. The participants using RAIDE achieved similar results in terms of efficiency. However, we can not say the same when they used tsDetect, which indicates that RAIDE standardizes how participants detect the AR and AD test smells, regardless of their experience.

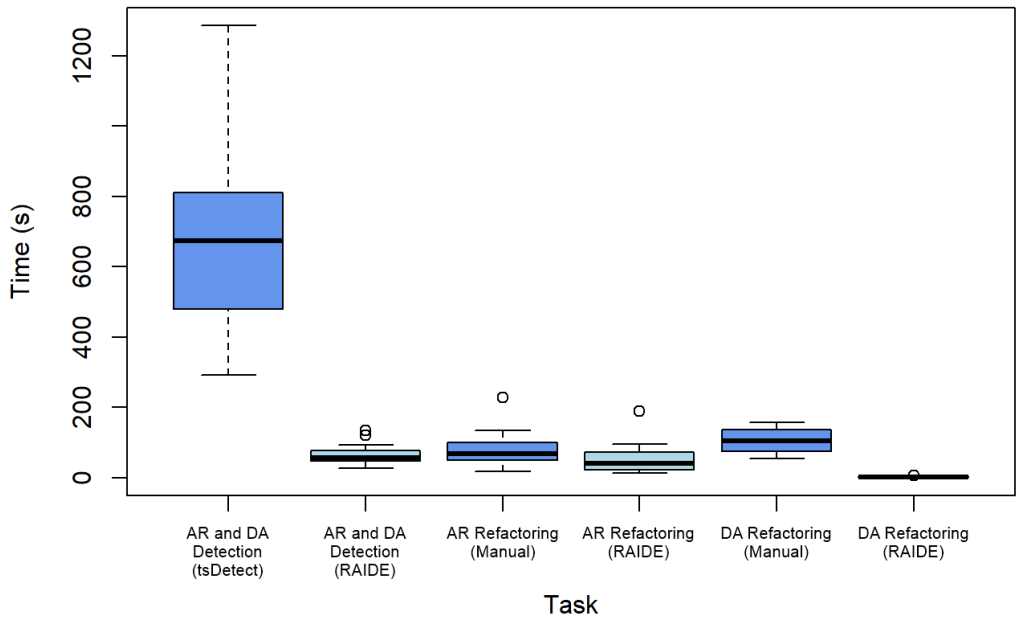

Figure 5. Comparison of the test smell detection and refactoring times between RAIDE and tsDetect

Our findings show that all participants encountered the AR and DA test smells faster with RAIDE than using tsDetect. The mean difference between the time of each task was more than 615s, considering all the steps needed to run tsDetect. Moreover,

the participants pointed out many advantages that RAIDE would have over tsDetect related to identifying the AR and DA test smells. For example, according to P02, RAIDE stood out because *"The learning process is easier and faster, it has fewer steps to take. It integrates into a development tool with a graphical interface, facilitating its use. It also identifies the lines that each test smell occurs and suggests refactorings."*

Summary: The AR and DA test smells could be identified ten times faster with RAIDE than with the state-of-the-art tool. RAIDE offers a better user experience, e.g., it enables detecting test smells with just one click and highlights their location in the source code.

4.2. Test Code Refactoring (RQ2)

We collected and analyzed the time spent by the participants to refactor test code with the AR and DA test smells using RAIDE and tsDetect.

AR Refactoring. The participants could refactor the test code with RAIDE on an average of 49.9s (*sd* = 41.35s), while the manual refactoring took 78.30s (*sd* = 46.39s). As the refactoring of RAIDE is by line, we need to count the time spent by the user to select each test smell of the method correctly and refactor them individually. For the statistical analysis, we performed the Shapiro-Wilk normality test. We found that the refactoring time of the AR test smell does not have a normal distribution (*p-value* = 6.607e-05). Therefore, we also used the Mann-Whitney test. Although the refactoring with RAIDE has shown similar dispersion to manual refactoring, the results indicated a significant difference between automated refactoring and manual refactoring (*p-value* = 0.022). It leads us to understand that RAIDE has a shorter detection time, which refutes the null hypothesis (H_02). Figure 5 shows boxplots with such comparisons, the median line of the box of RAIDE is outside the box of manual refactoring, which confirms a difference between them.

DA Refactoring. Likewise the former, this analysis also considered the Shapiro-Wilk normality test (*p-value* = 7.69e-06) beforehand. Data normality test indicated no normal distribution, then used the non-parametric Mann-Whitney test. Our results showed a significant statistical difference between the refactoring of DA (*p-value* = 4.764e-05), confirming a significant difference between the refactoring with RAIDE and manual refactoring. Therefore, we refute the null hypothesis (H_02). The average time was 107.2s (*sd* = 35.86) and 2.55s (*sd* = 1.57s) for manual refactoring and using RAIDE, respectively. Figure 5 shows that the median line of the box of RAIDE is outside the box of manual refactoring, which confirms a difference between them.

In addition, from a more qualitative standpoint, we also considered the participants' comments on the refactoring with and without RAIDE. P01, P13, and P16 indicated that they would use RAIDE in their projects due to detecting and refactoring the AR and DA test smells. According to P17, RAIDE can increase productivity in the identification and refactoring process, and such resources try to decrease the chance of human error. According to P19, RAIDE is convenient because it visually shows the AR and DA test smells, and their removal is automated, which allows the user to understand the steps taken.

Summary: The difference between RAIDE and the manual process for refactoring the AR and DA test smells is significant. RAIDE was more than forty-two times faster than the manual process for refactoring the DA test smell. Also, there is a consensus regarding the participants' opinion about RAIDE. There is an indication that RAIDE makes it easier to refactor the AR and DA test smells.

4.3. Discussion

After analyzing the gathered data, we could infer that detecting the AR and DA test smells with tsDetect was the most time-consuming task. That result is related to the fragmentation of the tsDetect, which has three intermediary steps until it presents the results. We also highlight the tool's low usability and the laborious process to detect the location of the test smells in the test code.

The participants' feedback includes essential data we should consider. For example, P01 and P02 reported that although tsDetect treats more test smells than RAIDE, detecting test smells in the former is more cumbersome. They stated that it is necessary to manipulate several files, and the command-line interaction might make it difficult for the identification process. P03 and P06 highlighted that the test smells detection process with tsDetect takes longer than with the RAIDE. P04 claimed that the process performed by tsDetect is counterintuitive, i.e., users would need to repeat the steps more often to learn the process (the order of entry and exit of the files). P06 also mentioned that the results of tsDetect are not very expressive because it only reports the existence or not of test smells, which can lead to misinterpretation, especially when dealing with long test classes.

Furthermore, the tasks performed with RAIDE took an average of 1.94 minutes, against an average of 14.42 minutes to perform the tasks with tsDetect. In addition, we limited the number of classes and tested the smells of the projects used to experiment. Therefore, gathered data indicate that manual test code refactoring would take longer in real-world environments, even without RAIDE. Indeed, it is necessary to carry out further studies in this direction to either confirm or refute this statement.

5. Threats to Validity

Internal Validity. It refers to the effects of the treatments on the variables due to uncontrolled factors in the environment [Wohlin et al. 2012]. We used system training to introduce the tools and participants' tasks to mitigate this threat. We used randomization to assign the order of participants to the tasks to mitigate the learning effect. However, in realizing the assigned tasks, some participants presented difficulties in manually accomplishing the refactoring task (i.e., using the tsDetect). In this case, we guided them to find a solution that may positively influence those participants' performance using tsDetect.

External Validity. It concerns whether the results can be generalized outside the experimental settings [Wohlin et al. 2012]. To mitigate this threat, we counted on experts with different backgrounds. Although there was not a big difference in the number of practitioners and academics, we introduced the detection and refactoring concepts with the tools using a training system. Although the participants had no experience in the context of the experiment, they were skilled enough to perform the tasks.

Construct Validity. It represents the cause and effect concepts to measure in the experiment through dependent and independent variables [Wohlin et al. 2012]. In this study, we did not compare RAIDE with a tool that presents the same features. tsDetect tool detects several test smells but does not assist in refactoring them. Conversely, RAIDE detects two test smells and support their refactoring. In the end, we could compare the effects of both manual and automated refactoring. Also, we conducted a pilot study that helped us improve the experiment design and materials.

Conclusion Validity. It refers to the extension of the conclusions about the relationship between the treatments and the outcomes [Wohlin et al. 2012]. The main threat to conclusion validity is the small size of the sample. Although we carried out the controlled experiment with 20 participants, we selected most of them with some prior knowledge about test smell.

6. Related Work

[Greiler et al. 2013a] introduced the TestHound tool, which performs a test code static analysis to detect the smells: Dead Field, General Fixture, Lack of Cohesion of Test Method, Obscure In-Line Setup, Test Maverick, and Vague Header Setup. The tool suggests refactoring candidates for removing test smells. The authors conducted a study with users and showed that the tool helps developers understand and adjust the test code.

[Baker et al. 2006] proposed TRex, an Eclipse plugin that detects refactoring opportunities to Standardized Tree and Tabular Combined Notation (TTCN-3) test suites. TRex calculates metrics to measure the overall quality of a TTCN-3 test suite and applies a pattern-based analysis to suggest candidate test refactorings. Its accuracy in detecting/refactoring test smells in TTCN-3 test suites, and evaluation with practitioners are unknown.

[Martinez et al. 2020] released RTj, a command-line tool that performs static and dynamic analysis to detect rotten green test smells. RTj also suggests to developers candidate test refactorings. The authors pointed out that RTj detects some false positives due to using conditionals or multiple test contexts. However, there are no details concerning the accuracy of its refactoring capabilities nor experiments validating its usability.

[Lambiase et al. 2020] released DARTS (Detection And Refactoring of Test Smells), a plugin for IntelliJ that utilizes information retrieval to detect three smells (General Fixture, Eager Test, and Lack of Cohesion of Test Methods). They built the tool on top of TASTE [Palomba et al. 2018], a textual-based detector with an overall f-measure of 67%, 76%, and 72% for the three test smells mentioned above, respectively. The tool also offers refactoring support. As for the refactoring support available in the tool, DARTS exploits the IntelliJ APIs to ensure that the refactored code is compilable and error-free.

Compared with the related tools, RAIDE expands the support to detect and refactor other test smells [Santana et al. 2020]. During plugin development researchers with experience in test smells validated the automated refactorings applied by RAIDE.

7. Conclusion

Software test code refactoring is highly dependent on automated support, so it might be cost-effective. Current literature encompasses a few tools supporting automated test code

refactoring. However, there is little evidence of automated support to handle test smells refactoring. In prior work, we introduced RAIDE, an Eclipse IDE plugin to automate the test smells detection and refactoring from the JUnit test code. That tool can handle Assertion Roulette and Duplicate Assert test smells in the current version. In this paper, we presented an empirical assessment of the RAIDE. We carried out a controlled experiment with twenty participants to evaluate the tool. The results indicate that RAIDE can detect test smells faster than a comparable state-of-the-art approach. Also, RAIDE was able to refactor a test method in a tiny fraction of time. In particular, the results were very favorable compared to the state-of-the-art approach. Future work directions include extending the RAIDE to consider other test smells and refactoring techniques. Furthermore, there is a need to conduct further empirical studies to validate whether the tool is valuable and effective in real-world practice.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001; and FAPESB grants JCB0060/2016 and BOL0188/2020.

References

Aljedaani, W., Peruma, A., Aljohani, A., Alotaibi, M., Mkaouer, M. W., Ouni, A., Newman, C. D., Ghallab, A., and Ludi, S. (2021). Test smell detection tools: A systematic mapping study. In *Evaluation and Assessment in Software Engineering*, page 170–180, New York, NY, USA. ACM.

Baker, P., Evans, D., Grabowski, J., Neukirchen, H., and Zeiss, B. (2006). Trex-the refactoring and metrics tool for ttcn-3 test specifications. In *Testing: Academic & Industrial Conference-Practice And Research Techniques*, pages 90–94, UK. IEEE.

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.

Greiler, M., van Deursen, A., and Storey, M.-A. (2013a). Automated detection of test fixture strategies and smells. In *60th International Conference on Software Testing, Verification and Validation*, pages 322–331, Luxembourg. IEEE.

Greiler, M., Zaidman, A., Deursen, A. v., and Storey, M.-A. (2013b). Strategies for avoiding text fixture smells during software evolution. In *Proc. of the 10th Working Conference on Mining Software Repositories*, pages 387–396, San Francisco, CA, USA. IEEE Press.

Kim, D. J., Chen, T.-H. P., and Yang, J. (2021). The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empirical Software Engineering*, 26(5):1–47.

Lambiase, S., Cupito, A., Pecorelli, F., De Lucia, A., and Palomba, F. (2020). Just-in-time test smell detection and refactoring: The darts project. In *Proc. of the 28th International Conference on Program Comprehension*, page 441–445, New York, NY, USA. ACM.

Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. In *The annals of mathematical statistics*, volume 18, pages 50–60, Michigan. JSTOR.

Martinez, M., Etien, A., Ducasse, S., and Fuhrman, C. (2020). Rtj: A java framework for detecting and refactoring rotten green test cases. In *Proc. of the 42nd International Conference on Software Engineering: Companion Proceedings*, page 69–72, New York, NY, USA. ACM.

Palomba, F., Di Nucci, D., Panichella, A., Oliveto, R., and De Lucia, A. (2016). On the diffusion of test smells in automatically generated test code: An empirical study. In *Proc. of the 9th international workshop on search-based software testing*, pages 5–14, Austin, TX, USA. ACM, IEEE.

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

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 *Proc. of the 29th Annual International Conference on Computer Science and Software Engineering*, page 193–202, USA. IBM Corp.

Peruma, A. S. A. (2018). *What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications*. Ph.d. thesis, Rochester Institute of Technology, Rochester, New York.

Santana, R., Martins, L., Rocha, L., Virgínio, T., Cruz, A., Costa, H., and Machado, I. (2020). RAIDE: a tool for assertion roulette and duplicate assert identification and refactoring. In *Proc. of the XXXIV Brazilian Symposium on Software Engineering - Tools Track*, pages 374–379, NY, USA. ACM.

Van Deursen, A., Moonen, L., Van Den Bergh, A., and Kok, G. (2001). Refactoring test code. In *Proc. of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95, NLD. CWI (Centre for Mathematics and Computer Science).

Vegas, S., Apa, C., and Juristo, N. (2015). Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering*, 42(2):120–135.

Virgínio, T., Martins, L., Rocha, L., Santana, R., Cruz, A., Costa, H., and Machado, I. (2020a). Jnose: Java test smell detector. In *Proc. of the 34th Brazilian Symposium on Software Engineering*, SBES '20, page 564–569, New York, NY, USA. Association for Computing Machinery.

Virgínio, T., Martins, L. A., Soares, L. R., Santana, R., Costa, H., and Machado, I. (2020b). An empirical study of automatically-generated tests from the perspective of test smells. In *Proc. of the 34th Brazilian Symposium on Software Engineering*, page 92–96, New York, NY, USA. ACM.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., and Regnell, B. (2012). *Experimentation in Software Engineering*. Springer.

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