Journal of Software Engineering Research and Development, 2023, 11:13, doi: 10.5753/jserd.2023.3195

Test smell refactoring revisited: What can internal quality attributes and developers' experience tell us?

Humberto Damasceno [**Federal University of Ceará** | ***hdamasceno1998@gmail.com***]
Carla Bezerra [**Federal University of Ceará** | ***carlailane@ufc.br***]
Denivan Campos [**Federal University of Bahia** | ***denivan.campos@ufba.br***]
Ivan Machado [**Federal University of Bahia** | ***ivan.machado@ufba.br***]
Emanuel Coutinho [**Federal University of Ceará** | ***emanuel.coutinho@ufc.br***]

Abstract

Test smells represent a set of poorly designed tests, which can harm test code maintainability. Although fundamental steps to understand test smells have been taken, there is still an evident lack of studies evaluating the impact of test smell refactoring from the perspective of internal quality attributes, such as size, cohesion, coupling, and complexity. In addition, the literature still lacks research that addresses the difficulties developers face during test smell refactoring. This article investigates the impact of test smell refactoring from a developer's perspective, considering the internal quality attributes and the background experience. We investigated the perceptions and difficulties encountered by 20 developers while removing five types of test smells in four open-source projects over two months. Through this study, we analyzed: (i) the impact that test smell refactoring has on internal quality attributes; (ii) the developers' perception of test smells as actual problems within software systems; (iii) the main difficulties developers face during test smell refactoring; (iv) the influence of developers' experience on assertiveness and refactoring time of test smells, and (v) the effects of refactoring on the test smell density. Our findings can help developers design a prioritization scheme for test smell refactoring and make them aware of the real benefits of test smell refactoring.

Keywords: *Test Smells, Test Refactoring, Software Quality, Developers' Perception.*

1 Introduction

Software testing is an important activity that aims to ensure the quality of software systems. Testing might demand much effort and costly resources, especially when developed manually (Myers et al., 2011; Orso and Rothermel, 2014). In this context, automated tests, implemented via frameworks or testing tools, have become first-class citizens, playing a particularly important role in the software quality assurance process (Bertolino, 2007; Myers et al., 2011). Automated tests can assist in identifying and removing errors and ensuring that the production code is robust under various conditions of use (Candea et al., 2010).

Nevertheless, either in manually or automated developed tests, software developers should be careful about their test code as they are with production code (Berner et al., 2005; Beller et al., 2019). Recent studies show that developers understand and give more importance to the production code than the test code, thus causing problems in the test quality (Beller et al., 2015; Palomba et al., 2018). In fact, poorly designed tests might result in a lack of maintainability, and potential bugs in the software being tested (Yusifoğlu et al., 2015).

Van Deursen et al. (2001) introduced the concept of test smells, which represent a poorly designed test that can harm test code readability, maintainability, and quality. Since its conceptualization, test smell has become the focus of several studies in the software testing field due to its importance for the quality of the software test code (Bavota et al., 2012; Spadini et al., 2018; Garousi and Küçük, 2018; Kim, 2020; Virgínio et al., 2021; Santana et al., 2022; Soares et al., 2022). Some studies indicate that test smells negatively impact the maintenance and understanding of test code. Therefore, it is necessary to understand its likely effects and propose mechanisms, strategies, and tools to mitigate them. In addition, several authors claim it is essential to investigate developers' perceptions and difficulties in the impact of test smell refactoring (Spadini et al., 2018; Soares et al., 2020; Spadini et al., 2020).

Software refactoring consists of small code transformations to improve the quality of the source code without compromising its overall functionality and observable behavior (Alizadeh et al., 2020; Fowler, 1999; Paixão et al., 2020). Test smells can be removed through refactoring, which can positively impact the test code quality (Spadini et al., 2018; Campos et al., 2021). However, literature reviews indicate a lack of studies evaluating the impact of test smell refactoring on the quality attributes (Bavota et al., 2015).

In our previous study (Damasceno et al., 2022), we investigated the impact of refactoring five test smells: *Magic Number Test, Duplicate Assert, Eager Test, Assertion Roulette, and Sensitive Equality* in four open-source systems. We identified the perception and difficulties encountered by twenty software developers in refactoring test smells. We analyzed: (i) the impact that test smell refactoring has on internal quality attributes; (ii) the developers' perception of test smells as real problems within a software system; and (iii) the main difficulties that developers face during test smell refactoring. This article extends prior work by adding two new research questions, where we analyze whether test smell refactoring

can introduce new test smells, and the impact of developers' experience on assertiveness and refactoring time of *test smells*. We next summarize the key results from this study:

- After refactoring the test smells, cohesion increased respectively, 7.26%, 6.20%, 9.19%, and 9.47% in the four analyzed systems.
- After refactoring the test smells, the complexity decreased respectively, 20.16%, 7.55%, 28.53%, and 21% in the four analyzed systems.
- The developers considered the *Assertion Roulette* and *Magic Number Test* test smells the least harmful in a software project. On the other hand, the *Eager Test* and *Duplicate Assert* test smells were considered the most critical ones.
- Understanding the source code is one of the main difficulties in refactoring test smells.
- The larger the source code, the more effort it will take to complete the refactoring.
- Developers with less experience take twice as long to complete the test smells refactoring compared to more experienced ones.
- The density of test smells decreases as the projects evolve.

The remainder of this article is organized as follows. Section 2 presents the key definitions of this study; Section 3 introduce the step-by-step procedure for conducting the study; Section 4 presents the results found, followed by a discussion of the results; Section 5 discusses the main threats to validity; Section 6 discusses related work; and Section 7 concludes and suggests on future work.

2 Background

This background Section provides the necessary information to understand the concepts we addressed in our research. Hence, Section 2.1 introduces the concept of test smells, with an emphasis on its origins, and highlights the nature of the five test smells we address in the scope of this research, in particular their possible effects, detection, and refactoring strategies. Section 2.2 briefly addresses existing strategies to identify test smells. The last part, Section 2.3, briefly introduces the concepts behind Internal Quality Attributes, particularly those we used in this study.

2.1 Test Smells

Code smells are assumed to indicate bad design that leads to less maintainable code (Sjøberg et al., 2013). Smells could be found in both the software project source code and in the test code (Van Deursen et al., 2001). In this latter, they are often referred to as ***test smells***. Test smells refer to certain patterns in software tests that may indicate potential problems or weaknesses in the tests. They are the effect of poorly designed tests and may result in a lack of maintainability, and potential bugs in the software being tested (Yusifoğlu et al., 2015).

The literature has introduced various types of test smells. Van Deursen et al. (2001) documented an initial set of eleven test smells: *Assertion Roulette*, *Eager Test*, *For Testers Only*, *General Fixture*, *Indirect Testing*, *Lazy Test*, *Mystery Guest*, *Resource Optimism*, *Sensitive Equality*, *Test Code Duplication*, and *Test Run War*. Later, Peruma et al. (2019) cataloged 12 new types, as follows: *Conditional Test Logic*, *Constructor Initialisation*, *Default Test*, *Duplicate Assert*, *Empty Test* , *Exception Handling*, *Ignored Test*, *Magic Number Test*, *Redundant Assertion*, *Redundant Print*, *Sleepy Test, and Unknown Test*. Other researchers have introduced another set of test smells, mainly based on practical observations, such as Garousi and Küçük (2018), who proposed a diverse catalog of test smells, as a result of a multivocal literature review on smells in software test code.

In our research, we addressed five test smell types, which we detail next in this Section. We selected these test smells as they most commonly occurred in the selected, analyzed systems.It is worth mentioning such test smells are widely discussed in the literature (Spadini et al., 2018; Soares et al., 2020; Spadini et al., 2020; Virgínio and Machado, 2021; Santana et al., 2021; Campos et al., 2022). Nevertheless, such studies considered other quality aspects, such as test code maintainability and readability.

2.1.1 Assertion Roulette

It occurs when a test method has multiple undocumented assertions. JUnit assertions have an optional first argument for adding a message that should explain what each assertion checks for. This makes it easier to understand the different assertions in the same test. The absence of this parameter in the assertion structure can make it difficult to comprehend during maintenance and identify the assertion if the method fails (Van Deursen et al., 2001).

Possible effect: Multiple assertions statements in a test method without an explanatory message can affect test code readability, understandability, and maintainability.

Detection: A test method with more than one assertion statement without explanation (parameter in assertion method).

Refactoring: Include explanation for each assertion (refactoring technique **Add Assertion Explanation**).

2.1.2 Duplicate Assert

It occurs when a test method checks for the same condition more than once within its scope (Peruma et al., 2019).

Possible effect: Makes the test more difficult to read and maintain since identical assertions within the same method do not make the purpose of the test method explicit. *Duplicate Assert* often creates a scenario that violates the accountability of each method fulfilling a single objective.

Detection: A test method that contains two or more assertions with the same parameters.

Refactoring: To test the same condition with different values, a new test method must be implemented.

2.1.3 Eager Test

It occurs when a test method checks multiple methods of the tested object (Van Deursen et al., 2001).

Possible effect: Makes tests more difficult to understand and maintain.

Detection: A test method contains several calls to several methods of the same tested object.

Refactoring: Extract the method by separating the assertions into separate test methods.

2.1.4 Magic Number Test

It occurs when a test method contains unexplained, undocumented numeric literals as parameters or as values for identifiers (Peruma et al., 2019).

Possible effect: Similarly to production code, using undocumented numeric literals can break the test method, making tests difficult to understand and evolve.

Detection: Check if numerical literals are being used as parameters or as values for identifiers in the body of the test method.

Refactoring: Numeric literals must be replaced with constants or variables, thus providing a descriptive name for the value (refactoring technique: **Replace Magic Literal**).

2.1.5 Sensitive Equality

It occurs when a test method has equality checks using the `toString()` method. Generally, the actual result is calculated and converted into a string, which is compared with another string's value, representing the actual value (Van Deursen et al., 2001).

Possible effect: Equality checks using `toString()` depend on irrelevant factors such as commas, quotes, and spaces that can interfere with the test result. In this sense, it is common for tests to start to fail when the `toString()` method of an object is changed.

Detection: A test method comparing an object with an expected result via the `toString()` method.

Refactoring: Add an implementation of the `equals()` method to the object class and rewrite the tests that use `toString()` to use the `equals()` method (refactoring technique **Introduce Equality method**). This way, `toString()`'s equality checks are replaced by actual equality checks.

2.2 Identifying Test Smells

The literature presents several studies on tools, but a recent study by Aljedaani et al. (2021) presented 22 tools for dealing with test smells. In this study by Aljedaani et al. (2021), most tools support the Java programming language and the JUnit testing framework[1].

For this study, we adopted the *JNose Test Tool*[2] (Virgínio et al., 2020; Virgínio et al., 2021). *JNose tool* collects test smells in Java programming language and detects 21 types of test smells in the JUnit framework (Virgínio et al., 2020; Virgínio et al., 2021). The *JNose Tool* provides software developers with an advanced visual interface of aspects related to testing code, switching from statically computable indicators (such as quality attributes and test smells) to dynamic measures (such as code coverage and indicators). *JNose Tool* detects test smells through a set of rules and quantifies the types of test smells by class and by project version.

To detect test smells the *JNose tool* uses a set of rules and quantifies the types of test smells through four types of analysis, *TestClass*, *TestSmell*, *TestFile* and *Evolution*. As a result, *JNose* brings measures such as lines of code (LOC), number of methods, types of test smells and the amount of test smells in each test class of the project. Furthermore, *JNose* presents test quality from an evolutionary perspective.

2.3 Internal Quality Attributes

According to the ISO/IEC 25010/2011 standard (ISO, 2011), software quality can be defined as the compliance level of a software system, component, or process according to the needs and expectations of its stakeholders. Software quality can be measured by different quality attributes, which might be classified as (i) internal quality attributes and (ii) external quality attributes (Malhotra and Chug, 2016). External quality attributes are measured in how the process, resource, or software relates to the environment. Internal quality attributes such as size, cohesion, coupling, and complexity are measured by analyzing the software separately from its behavior (Fernandes et al., 2020). Measuring internal quality attributes is simpler than measuring external ones. For example, the method or class size can be measured using the LOC metric (Morasca, 2009). In our study, we used 9 widely-accepted metrics of internal quality attributes (see Table 3) (Morasca, 2009; Chidamber and Kemerer, 1991; Lorenz and Kidd, 1994; McCabe, 1976). Chidamber and Kemerer (1991) metrics are forerunners in object-oriented (OO) metrics and have a theoretical basis for measuring OO code. In this study, the metrics defined by Chidamber and Kemerer (1991) were chosen to analyze the internal quality of the systems (Malhotra and Chug, 2016; Dyer et al., 2012). To collect the metrics, we used the Understand tool (Chávez et al., 2017).

3 Study Settings

This section describes the empirical study settings, comprising its goal, research questions, and steps.

3.1 Goal and Research Questions

This article reports on a study on the impact of refactoring test smells on internal quality attributes and the perceptions and difficulties developers face during refactoring. In addition, we also analyzed the density of test smells before and after refactoring. The research questions (RQ_s) are stated next.

RQ_1: What impact does refactoring test smells have on the test code quality attributes? This question aims to assess the impact of refactoring test smells on the test code quality attributes. By answering $\mathbf{RQ_1}$, we can identify which test smells harm quality attributes and provide guidelines on eliminating such risks within a software project. Verifying which attributes improve or worsen after refactoring the test smells is possible.

RQ_2: Do developers perceive test smells as actual problems in a software project? This question aims to analyze users' perceptions through the impacts of test smells for a

[1] `https://junit.org/`

[2] `https://github.com/arieslab/jnose`

software project. By answering $\mathbf{RQ}_2$, we could warn and make developers aware of test smells' impacts on a software project. We can also emphasize the importance of mitigating such anomalies soon after detection, aiming to reduce their impacts.

$\mathbf{RQ}_3$: What are the main difficulties developers face during test smell refactoring? This question aims to identify developers' main difficulties during test smell refactoring. By answering $\mathbf{RQ}_3$, we can leverage and understand the main difficulties and what factors developers consider to justify the difficulties in refactoring the test smells.

$\mathbf{RQ}_4$: How does the experience of developers influence the assertiveness and refactoring time of *test smells?*. This RQ investigates whether the developer's experience influences the assertiveness of test smells refactoring. Furthermore, we also investigated whether experienced developers refactor test smells faster than inexperienced ones, and whether they face fewer difficulties. By answering $\mathbf{RQ}_4$, we can identify the profile of developers most likely to have difficulties during test smells refactoring.

$\mathbf{RQ}_5$: What are the effects of test code refactoring on the test smell density? This question aims to analyze test smell density before and after test code refactoring. By answering $\mathbf{RQ}_5$, we could analyze the likely changes in test smells density as a direct effect of refactoring. By density we consider the ratio of test smells inserted to the number of removed test smells.

3.2 Study Steps

Step 1: Select open-source systems for analysis. The first step involves selecting a set of candidate software systems. To perform this step, we considered the dataset from Pecorelli et al. (2021), which comprises eight open-source software systems from GitHub, with plenty of test smells. We selected four out of eight candidate software systems from such a dataset as they have the most significant test smells. The four software systems have 4147 commits, 53 releases, 300 production classes, and 155 test classes. Table 1 shows raw data of the selected software systems. The first column contains the identifiers of each system and their GitHub URLs. The remainder includes additional information about the systems, such as the number of commits, releases, production classes, production and test LoC, and test classes, as of the release date.

Table 1. Selected software systems.

Systems	Commits	Releases	PC	TC	LoC*	LoC**
S1	1702	45	26	31	4582	3850
S2	3091	49	270	139	9743	8248
S3	5400	54	100	47	6510	5620
S4	6395	65	804	404	8361	2198

Legend: [LoC*] Production code lines [LoC**] Test code line
[PC] Production Classes [TC] Test Classes

Step 2: Detect test smells. In this step, we first analyzed the outcomes of a recently published literature review on tool support for test smell detection Aljedaani et al. (2021). The authors leveraged twenty-two existing tools and analyzed their key features in such a study. Then, we defined the following criteria to select our study's test smell detection tool. The tool must: (i) be open-source; (ii) support the identification of our pre-defined set of test smells; (iii) provides the automated measurement of quality attributes in test code; (iv) handle Java projects developed with Maven support[3]; and (v) evaluate multiple test code files (classes and packages) simultaneously.

After analyzing the existing and available tools, we selected the **JNose Test**[4] for meeting all the pre-defined criteria. The selected tool automatically detects test smells and collects code coverage metrics (Virgínio et al., 2020). It holds an accuracy ranging between 91% and 100%, and a recall from 89% to 100%. The authors state that the test smells that use external resources have the worst precision and recall values, but this study did not evaluate them.

Table 2 shows the number of test smells detected in the four selected software systems. The first column shows the test smells analyzed in this study. We selected the *Magic Number Test, Duplicate Assert, Eager Test, Assertion Roulette, and Sensitive Equality* test smells as they are the ones that occur the most in the four selected systems. The following four columns show the number of test smells on each system. Finally, the last column shows each test smell's total number of occurrences.

Table 2. Number of test smells detected per system.

Test Smell	S1	S2	S3	S4	Total
Magic Number Test	265	1622	783	38	2708
Duplicate Assert	73	581	225	22	901
Eager Test	245	579	402	59	1285
Assertion Roulette	1908	4931	2697	135	9671
Sensitive Equality	27	191	153	49	420
Total	2518	7904	4260	303	-

Step 3: Measure quality attributes. This step consists of measuring the quality metrics of the test code. We used the **Understand for Java**[5] to calculate the metrics. This tool has been available for years (Lincke et al., 2008), and many studies have employed it due to its high precision in calculating OO metrics (Nilson et al., 2019; Martins et al., 2021).

Table 3 presents the four internal quality attributes we used to evaluate the quality of the selected systems. The first column lists the quality attributes. The second column shows the quality metrics related to each attribute. The third column provides a short description of each metric. We selected these metrics as they allow us to evaluate different particularities of the internal quality attributes (Bieman and Kang, 1995).

In (Tahir et al., 2016), the authors observed a relationship between production code quality metrics and test smells, reinforcing that some design aspects (e.g., complexity) are related to the appearance of failures in the test code.

Step 4: Select the developers to refactor the test smells. This step involves selecting software developers to refactor the test smells we detected in Step 2. To this end, we recruited

[3]Maven helps manage Java projects and automate application builds.
[4]Available at `https://GitHub.com/arieslab/jnose`.
[5]Understand for Java is a powerful static code analysis tool for Java source code. Available at `https://www.scitools.com/`.

Table 3. List of analyzed internal quality attributes.

Attribute	Metric	Description	Source
Size	Lines of code (LOC)	Measures the number of lines of code for classes and methods.	Morasca (2009)
Cohesion	Lack of Cohesion (LCOM)	Measures the cohesion of a class or method. The bigger this value the less cohesive the class/method.	Chidamber and Kemerer (1991)
	Lack Of Cohesion Modified (LCOM2)	Measures the cohesion of a class or method after suffering modifications. The higher this value, the less cohesive the class/method.	Chidamber and Kemerer (1991)
Coupling	Count Class Coupled (CCC)	Number of classes that a class is bound to. The higher this value, the greater the coupling of classes and methods.	Chidamber and Kemerer (1991)
	Count Class Coupled Modified (CCC2)	Measures the coupling of a class after modifications. The higher this value, the higher the complexity.	Chidamber and Kemerer (1991)
Complexity	MaxNesting (MN)	Maximum depth of the control constructs (if, while, etc.). The higher this value, the greater the complexity.	Lorenz and Kidd (1994)
	SumCyclomatic (SC)	Sum of the cyclomatic complexity of all functions and nested methods. The greater the value of this metric, the greater the complexity.	McCabe (1976)
	AvgCyclomatic (AC)	Average complexity of all nested methods. The higher this value, the more complex the class/method.	McCabe (1976)
	MaxCyclomatic (MC)	Maximum cyclomatic complexity of the methods nested. The higher this value, the more complex the class/method.	McCabe (1976)

a group 25 collaborating developers from other projects to participate in the experiment. All of them hold undergraduate degrees in CS or related areas and work as software developers in the industry. As a first task, each had to fill in a background questionnaire[6]. The questionnaire aimed to characterize the developer in terms of experience, knowledge of Java, and test smells. We carefully examined their responses to determine which were eligible to participate in the experiment. Out of the 25 developers invited, a total of 20 participants successfully completed and passed the questionnaire evaluation. These 20 developers were consequently selected to take part in the experiment. Table 4 presents the profile of the selected developers.

After selecting the developers, we held a two-hour training session. We discussed underlying concepts, such as (i) test smell detection, (ii) test code quality attributes, and (iii) test smell refactorings. We presented practical examples of test code refactoring. Afterward, we provided the participants with practical activities of test smell detection and refactoring. We sought to reduce bias by focusing on the main concepts and presenting practical and objective examples. Table 5 shows the refactoring techniques the participants employed in the experiment. The first column shows the technique's name. The second describes how the technique works. The third column shows the test smells refactored using the technique.

Step 5: Remove test smells through manual refactoring. This step consists of removing test smells through manual refactoring. We employed manual refactoring to gather developers' perceptions and difficulties more accurately, as they dealt directly with the test code. Our sample consisted of 20 examples of each of the 5 test smells randomly selected through the JNose Test tool, totaling 100 examples of test smels. Table 6 shows how we allocated the test smells for each participant. The first column shows the test smells. The second shows the occurrence of test smells. The third shows the number of test smells allocated to each participant. The fourth indicates the participants responsible for refactoring the test smells. The last column contains the total number of refactored test smells.

Table 4. Profile of experiment participants.

ID	Exp.	Educ. Level	QA	TS	Java
D1	2y	Undergrad.	Basic	Basic	Basic
D2	3y	Undergrad.	Int.	Basic	Int.
D3	3y	Undergrad.	Basic	Basic	Int.
D4	5y	Undergrad.	Basic	Basic	Adv.
D5	6y	Undergrad.	Basic	Basic	Adv.
D6	4y	Undergrad.	Adv.	Basic	Adv.
D7	4y	Undergrad.	Adv.	Basic	Adv.
D8	4y	Undergrad.	Int.	Basic	Adv.
D9	2y	Undergrad.	Int.	Basic	Basic
D10	4y	Undergrad.	Int.	Basic	Adv.
D11	8y	Undergrad.	Adv.	Int.	Adv.
D12	3y	Undergrad.	Basic	Basic	Int.
D13	5y	Undergrad.	Adv.	Int.	Adv.
D14	5y	Undergrad.	Adv.	Int.	Adv.
D15	2y	Undergrad.	Basic	Basic	Basic
D16	2y	Undergrad.	Basic	Basic	Int.
D17	4y	Undergrad.	Int.	Basic	Int.
D18	6y	Undergrad.	Int.	Int.	Adv.
D19	3y	Undergrad.	Basic	Basic	Int.
D20	4y	Undergrad.	Int.	Basic	Adv.

Legend: Exp. - Experience (in years); Educ. - Education Level; QA - (Knowledge Level of) Quality Attributes; TS - (Knowledge Level of) Test Smells; Int. - Intermediary Level; Adv. - Advanced Level

To assist the refactoring activity, we provided the participants with a list of classes and methods for identifying test smells in the detection step. In addition, we created a branch on GitHub for each developer to track the progress of refactorings individually. We held two weekly meetings to check the progress and check if the developers faced any difficulties. To maintain the organization of the experiment, each developer was responsible for performing a pull request with all the operations performed to refactor the test smells. We evaluated each pull request to see if the test smells were refac-

[6] https://GitHub.com/leanresearchlab/JSERD_2023

Table 5. Refactoring techniques used to remove the test smells

Refactoring	Description	Test smells
Extract Method	Extract a method, separating assertions into different methods.	Eager Test, Duplicate Assert
Add Assertion Explanation	Assertions usually have an optional first argument to provide an explanatory message to the user when the assertion fails. So, one way to make the most understandable test is by using this message to distinguish between different assertions that occur in the same test.	Assertion Roulette
Introduce Equality Method	If an object structure needs to be checked for equality in tests, add an implementation for the "equals" method for the object class.	Sensitive Equality
Replace Magic Literal	Substitute numeric literals without explanation by constants or descriptive variables.	Magic Number

Table 6. Allocation of developers to refactor test smells

Test Smells	Occurrences	Test smells allocated by dev.	Devs. allocated for refactoring	Total refactored test smells
Magic Number Test	2708	5	D7, D13, D14, D15	20
Duplicate Assert	901	5	D1, D8, D9, D20	20
Eager Test	1285	5	D2, D5, D11, D17	20
Assertion Roulette	9671	5	D4, D12, D16, D19	20
Sensitive Equality	420	5	D3, D6, D10, D18	20

tored correctly according to the procedures presented in the training session. Is it worth noting that we allowed the developers to have flexibility in terms of the frequency and time allocated for refactoring the test smells, considering that they also had work responsibilities to attend to. Therefore, we took into account only the number of days each developer took to complete the refactorings.

Step 6: Document developers' perceptions and difficulties in refactoring test smells. This step involves documenting developers' perceptions of the risks of test smells within a system and the difficulties encountered in refactoring such smells. We relied on the diary technique to document developers' perceptions of test smell refactoring. The Diary technique consists of a data collection method where participants record in a form their daily activities about some event that affected them positively or negatively. This technique is a way of understanding participant behavior, reducing the impact of researchers (França et al., 2020).

Hence, we asked the participants to document their activities whenever they refactored the test code. We did not establish a frequency at which they needed to work with the code because they could have other duties. Table 7 shows the questions answered by the participants using the diary technique.

Table 7. Questions answered by developers through the diary technique

Questions	Answers
Which test smell are you currently refactoring?	-
What were the main difficulties in refactoring this test smell?	-
How harmful is this test smell to the system?	0 to 10
What refactoring techniques do you use to remove this test smell?	-

Step 7: Validate test smells removal. We reviewed all pull requests to verify if the developers removed the test smells correctly. We also analyzed: (i) the impact of test smell refactoring on the internal quality attributes; and (ii) which test smells are the most harmful within the systems used in the study. As the developer completed the activity, we analyzed the refactored files on a first-come, first-served basis. After refactoring the test cases with test smells, the authors re-run the JNose Test on the projects to validate the removal and analyze the effects of test smell refactoring. The complete removal of test smells via manual refactoring took about two months. We completed our review in one month. We evaluated the metrics by comparing the sums of metrics before and after refactoring the test smells. To carry out the measurements, we considered each refactoring that was applied to each system. Finally, we measured the impact that developer experience can have during the test smells refactoring. To measure the developers' experience, we considered their experience in Java programming.

It is worth mentioning that we placed a strong emphasis on validating the successful removal of test smells and ensuring the preservation of test behavior. To achieve this, after completing the refactoring process, we executed the JNose tool to confirm the absence of test smells and verify the integrity of the refactored code. Nevertheless, while using the JNose tool provided a reliable means of validating the removal of test smells, it is important to note that the refactoring process itself is not completely immune to side effects. For example, new test smells may be introduced after removing a particular test smell through refactoring. This observation is consistent with the findings reported by Campos et al. (2021), which highlight the challenges associated with effectively addressing test smells and the potential for new smells to emerge during refactoring.

For additional details on the test smell detection process and the metrics collected before and after removing the test smells, a comprehensive documentation is available at the following GitHub repository: `https://GitHub.com/leanresearchlab/JSERD_2023`.

4 Results and Discussions

In this section, we present the results and discuss the results obtained, and show the main findings for each RQ.

4.1 Impact of test smell refactoring on test code internal quality attributes ($\mathbf{RQ_1}$)

In $\mathbf{RQ_1}$, we analyzed the impact that test smells refactoring has on the following internal quality attributes: (i) size, (ii) cohesion, (iii) coupling, and (iv) complexity. Twenty devel-

opers, who applied the refactoring techniques discussed in training via manual refactoring, performed the removal of test smells.

Table 8 contains data referring to the test smells refactoring from the perspective of the analyzed systems and the quality attributes. In turn, Table 9 shows data about the test code refactoring from the perspective of test smells and quality attributes. Both tables show the four quality attributes and the values of their respective metrics. To analyze the impact of test smells refactoring on attributes with more than one metric, we compared the sum of the metrics before and after refactoring using the Understand tool (Tarwani and Chug, 2016). The symbol ↑ represents an increase in the metric's value, the symbol ↓ represents a decrease in the metric's value, and the symbol – shows that the attribute value did not change after refactoring the test smells.

It is worth noting that if the cohesion value increases, this attribute has been improved due to the greater cohesion of the class/method, thus improving the system's quality. Attributes such as coupling and complexity must have low values to indicate an improvement in the system's quality. The size attribute can indicate improvement or deterioration in quality, depending on the context in which it is under evaluation.

Internal quality attributes greatly benefited by the refactoring of test smells. By analyzing data from Table 8, we could observe that the cohesion and complexity of all analyzed systems improved after refactoring the test smells. Regarding cohesion, we highlight systems S3 and S4, as they improved the attribute value by 9.19% and 9.47%, respectively.

> **Finding 1:** The refactoring of test smells resulted in an increase in cohesion in the test code by 7.26%, 6.20%, 9.19%, and 9.47% across the four analyzed systems. This finding suggests a positive relationship between the refactoring of test smells and improved cohesion.

Some related work has already studied the impact of complexity in OO systems (Alenezi and Almustafa, 2015). Prior studies relate complexity to problems such as worsening software maintainability, greater error proneness, and loss of quality (Darcy et al., 2005; Xie et al., 2009). Regarding the systems analyzed in this work, we can highlight the systems S1 (20.16%), S4 (21%), and S3 (28.53%), which considerably improved the value of complexity after removing the test smells.

> **Finding 2:** The refactoring of test smells resulted in a decrease in complexity in the test code by 20.16%, 7.55%, 28.53%, and 21% across the four analyzed systems. This finding suggests a positive impact of test smell refactoring on reducing complexity in the code.

Internal quality attributes are partially affected after test smell refactoring. Let us consider data from Table 8 from another perspective. It is possible to notice that the coupling quality attribute was not improved in all the systems. The systems S2 and S3 kept the values of the coupling-related metrics the same as those measured before the test smells were refactored. On the other hand, the systems S1 and S4 improved coupling by 4.3% and 2.97%, respectively. We may observe that refactoring test smells partially improves coupling, unlike the cohesion and complexity attributes, which significantly improve after refactoring.

Internal quality attributes are negatively affected by the refactoring of test smells. Table 8 shows that test smells refactoring harmed the quality attribute size. After refactoring, we observed that all systems worsened the value of the LOC, particularly the systems S4, S3, and S1, which increased the metric's value respectively by 150%, 157.77%, and 180.32%. We could infer that the size attribute is not directly related to the other attributes we analyzed in this study. It is worth mentioning that the cohesion, coupling, and complexity values improved, even with the worsening of the size.

Test smells that were removed and improved three quality attributes. Table 9 shows exciting data and results concerning such an issue. It is possible to notice that removing the test smells *Eager Test* and *Duplicate Assert* improved all quality attributes, except for size. Specifically, the *Eager Test* refactoring increased cohesion by 8.24% and decreased coupling and complexity by 3.72% and 21%. Removing *Duplicate Assert*, in turn, increased cohesion by 10.52%, decreased coupling by 3.56%, and complexity by 22.02%. These results suggest that the presence of these test smells causes damage to the system's quality since the values of quality attributes improved after refactoring both.

Test smells that partially impacted the internal quality attributes. By observing data from Table 9, it is possible to notice that the refactoring of all test smells, except for *Assertion Roulette* and *Sensitive Equality*, increased LOC. Specifically, *Eager Test* and *Duplicate Assert* increased the source code size by 320.33% and 125.77%, respectively. LOC increase in all systems occurred after refactoring due to the chosen techniques that imply an increase in LOC, e.g., Extract Method. Conversely, refactoring *Magic Number Test* and *Assertion Roulette* test smells did not change the coupling values. Despite the significant increase in LOC, it is possible to suggest that this is not such a major factor affecting the test code quality. Refactoring all test smells improved the values of other quality attributes, like cohesion, coupling, and complexity.

> **Finding 3:** The removal of the *Eager Test* and *Duplicate Assert* test smells had a positive impact on the internal quality attributes. Conversely, the *Magic Test Number* and *Assertion Roulette* test smells did not positively affect the size and coupling quality attributes. This suggests the presence of these latter test smells in terms of their impact on the code's quality attributes.

Implications of RQ_1. Our findings suggest that *Eager Test* and *Duplicate Assert* are particularly detrimental to system quality. Their removal resulted in a considerable improvement in internal quality attributes. In addition, *Magic Number Test*, *Assertion Roulette*, and *Sensitive Equality* improved certain internal quality attributes and worsened others. Therefore, the results obtained in $\mathbf{RQ_1}$ can help project managers and developers better decide to refactor a given

Table 8. Impacts of refactoring test smells grouped by system and internal quality attributes

System		Size	Cohesion		Coupling		Complexity			
		LOC	LCOM	LCOM2	CCC	CCC2	MN	SC	AC	MC
S1 with test smells		61	1164	943	277	277	39	414	41	96
	Total	61		2107		554				590
S1 Without test smells		171	1242	1018	260	270	18	338	35	80
	Total	171		2260		530				471
Results		**↑ 180.32%**		**↑ 7.26%**		**↓ 4.3%**				**↓ 20.16%**
S2 with test smells		38	673	569	375	367	45	1806	33	102
	Total	38		1242		742				1986
S2 without test smells		80	708	611	375	367	19	1697	22	98
	Total	80		1319		742				1836
Results		**↑ 110.52%**		**↑ 6.20%**		-				**↓ 7.55%**
S3 with test smells		45	910	863	266	265	48	243	63	70
	Total	45		1773		531				424
S3 without test smells		116	1025	911	266	265	21	187	44	51
	Total	116		1936		531				303
Results		**↑ 157.77%**		**↑ 9.19%**		-				**↓ 28.53%**
S4 with test smells		72	1001	857	308	298	79	363	80	116
	Total	72		1858		606				638
S4 without test smells		180	1123	911	302	286	55	297	69	83
	Total	180		2034		588				504
Results		**↑ 150%**		**↑ 9.47%**		**↓ 2.97%**				**↓ 21%**

Table 9. Impacts of refactoring test smells grouped by test smells and internal quality attributes

Test Smells		Size	Cohesion		Coupling		Complexity			
		LOC	LCOM	LCOM2	CCC	CCC2	MN	SC	AC	MC
Magic Number Test before refactoring		20	592	465	215	215	30	275	31	49
	Total	20		1057		430				385
Magic Number Test after refactoring		40	630	501	215	215	15	252	27	39
	Total	40		1131		430				333
Results		**↑ 100%**		**↑ 7%**		-				**↓ 13.50%**
Duplicate Assert before refactoring		97	847	674	197	196	68	427	48	61
	Total	97		1521		393				604
Duplicate Assert after refactoring		219	950	731	190	189	45	346	35	45
	Total	219		1681		379				471
Results		**↑ 125.77%**		**↑ 10.52%**		**↓ 3.56%**				**↓ 22.02%**
Eager Test before refactoring		59	1081	971	274	263	43	641	55	105
	Total	59		2052		537				844
Eager Test after refactoring		248	1187	1034	263	254	16	529	39	82
	Total	248		2221		517				666
Results		**↑ 320.33%**		**↑ 8.24%**		**↓ 3.72%**				**↓ 21%**
Assertion Roulette before refactoring		20	497	373	156	157	36	308	39	82
	Total	20		870		313				465
Assertion Roulette after refactoring		20	542	401	156	157	20	266	33	72
	Total	20		943		313				391
Results		-		**↑ 8.40%**		-				**↓ 15.91%**
Sensitive Equality before refactoring		20	731	749	384	376	34	1175	44	87
	Total	20		1480		760				1340
Sensitive Equality after refactoring		20	789	784	379	373	17	1117	36	74
	Total	20		1573		752				1244
Results		-		**↑ 6.28%**		**↓ 1.05%**				**↓ 7.16%**

test smell, taking into account the quality metric to improve. Finally, except for *Assertion Roulette* and *Sensitive Equality*, test smells refactoring led the source code to grow. Removing test smells demands more source code to implement. This is not necessarily harmful since the results show that removing test smells positively impacted the other internal quality attributes.

4.2 Developers' perception of test smells as actual problems in the projects (RQ_2)

We addressed $\mathbf{RQ_2}$ by analyzing the information collected through the diary technique, where each developer documented their impressions of the test smells. We also evaluated the opinions of developers during the initial training session. Table 10 presents the developers' perceptions about the impacts of test smells. The first column shows all reported insights. The second contains the developers who reported such insights. The third shows the occurrence of test smells according to the developers' perception.

We identified that all the developers who refactored *Eager Test* and *Duplicate Assert* considered such smells highly harmful. Some of them realized the relevance of these test smells during introductory training. Others, in turn, identified the harmful effects of these smells after performing the manual refactoring. We strengthen this evidence with some highlights, as follows:

Table 10. Developers' perceptions about the impacts of test smells

Perceptions	Developers	Test Smells
Highly harmful	D2, D5, D9, D11, D17, D20	Duplicate Assert, Eager Test
Harmful	D1, D8, D10	Duplicate Assert, Sensitive Equality
Reasonably harmful	D3, D6, D18	Sensitive Equality
Weakly harmful	D12, D14, D16	Assertion Roulette, Magic Number
Irrelevant	D4, D7, D13, D15, D19	Magic Number Test, Assertion Roulette

D2:"During the training, I noticed how harmful the test smell Eager Test could be in a software project."

D11:"Eager Test is undoubtedly harmful to the health of the test code."

D17:"The experience of refactoring the Eager Test test smell certainly changed my conception of how to develop test code."

On the other hand, we also encountered test smells that developers did not perceive as highly detrimental. Specifically, the test smells *Assertion Roulette* and *Magic Number Test*, which involve undocumented assertions and unexplained numeric literals, were found to be the least harmful in the analyzed scenarios. This observation is further supported by some reports:

D4:"In my view, the Assertion Roulette test smell does not negatively impact a software project."

D7:"I didn't consider the Magic Number Test smell as harmful inside the file I refactored."

D15:"I see no need to refactor the smell Magic number Test."

Finding 4: In the evaluation of test smell's perceived impact, developers identified that *Eager Test* and *Duplicate Assert* test smells as the most critical. Conversely, the *Assertion Roulette* and *Magic Number Test* were deemed the least harmful smells by developers. This discernment highlights the likely varying degrees of concern about different test smells in software projects.

Implications of $\mathbf{RQ_2}$. Our findings indicate that some developers might not see test smells as harmful within a software project. Let us recall that only eight out of twenty participants considered test smells irrelevant or weakly harmful within a system. We highlight the *Assertion Roulette* and *Magic Number Test* smells, which were the most cited ones. In contrast, all the participants who refactored *Eager Test* and *Duplicate Assert* considered them highly harmful or harmful within a software project. Due to the results obtained, we strengthen the claim that any test smell in a system can negatively impact it. In addition, test engineers must refactor such smells as soon as they are detected.

4.3 Difficulties identified by developers during test smell refactoring ($\mathbf{RQ_3}$)

We discussed $\mathbf{RQ_3}$ by analyzing the responses from the diaries. We performed qualitative analysis on the responses and identified four categories: (i) Difficulty in understanding the source code; (ii) Major refactoring effort; (iii) Difficulty in applying the refactoring technique; and (iv) a Large amount of source code. Table 11 shows the categories. The first column lists the categories found, and the second shows the number of participants that faced difficulties in refactoring the test smells.

From Table 11, it is possible to notice that there were developers who fit into more than one category of difficulty in refactoring the test smells. From this, we could identify relationships between the categories of difficulties. The first relationship was found between the categories *"difficulty understanding the source code"* and *"great refactoring effort."* We strengthened this relationship through the following reports:

Table 11. Categories of difficulties reported by the participants.

Categories	Developers
Difficulties understanding the source code	D1, D3, D7, D9, D13, D20
Huge refactoring effort	D1, D4, D7, D10, D13, D14, D16
Difficulty applying the refactoring technique	D3, D9, D20
Lots of source code	D4, D10, D14, D16

D1:"It took me longer than expected to refactor methods that required extraction."

D10:"Undoubtedly, the smell Sensitive Equality test was the most difficult to refactor, as I had complications applying the refactoring technique."

D2:"I had trouble refactoring the Eager Test because I couldn't fully understand the source code."

Our review reinforces that the difficulty of understanding the source code is directly proportional to the refactoring effort. Therefore, if the developer takes much time to understand the source code, his effort to perform the refactoring will be greater than usual.

We also identified a relationship between the categories *"great refactoring effort"* and *"a large amount of source code."* Some developer reports that confirm this relationship are highlighted next:

D17:"It took a lot of effort to complete the task, due to the large amount of source code."

D20:"Refactoring this gigantic method took a lot of work."

D16:"The test smells I received were very extensive, so I required a higher level of effort than necessary."

We may observe that the main difficulties in applying a given test smell refactoring technique concern the difficulty of understanding the source code. The reports below strengthen the relationship between these two categories of difficulties encountered:

D3:"I had difficulties applying the refactoring technique as I did not fully understand the source code."

D9:"There were times when I found the source code so complex to the point of not knowing how to apply the refactoring technique."

D11:"I didn't feel safe refactoring the test smell, as I had trouble understanding the source code."

Finding 5: The refactoring of test smells presents inherent challenges, particularly regarding the comprehension of the source code. As the size of the codebase increases, so does the effort required for successful refactoring. However, it is worth noting that well-written code significantly eases the process of refactoring test smells. These findings underscore the significance of code understandability and quality in enabling efficient and effective test smell refactoring.

Implications of $\mathbf{RQ}_3$. Our findings indicate that a large amount of source code and the difficulty of understanding the source code lead to a high refactoring effort. Developers do not feel safe applying a particular refactoring technique when they cannot fully understand the source code to refactor. Thus, good programming practices can positively impact the test smell refactoring. Furthermore, it is noticeable that developers still do not feel completely confident about identifying and refactoring test smells, given the difficulties documented in this RQ.

4.4 Impact of developer experience on assertiveness and refactoring time of test smells ($\mathbf{RQ}_4$)

We address $\mathbf{RQ}_4$ by analyzing the results obtained in $\mathbf{RQ}_1$ and $\mathbf{RQ}_3$, which range from the removal of test smells to the difficulties the developers faced when performing the refactoring. From such analyses, it was possible to investigate whether developers with more experience could refactor the test code faster and with fewer difficulties.

Table 12 presents the average time required (measured in days) for developers to refactor the passed test smells. The first column groups developers according to their programming experience. The second contains the days required to refactor the test smells. The third shows the standard deviations. It is worth reiterating that each developer was responsible for refactoring a single type of test smell.

Table 12. Average time needed to refactor the test smells, according to the developers' experience

Exp. (in years)	Ave. time to refactor (in days)	*Std. dev.*
2	4	1
3	4	0.71
4	3	1.15
5	3	0.82
6	2	0
8	2	0

From $\mathbf{RQ}_1$, all developers, experienced or not, refactored all the test smells during the study. However, from Table 12, we notice that less experienced developers (maximum 3 years) demanded twice as much time to refactor the test smells than experienced ones. The average time for experienced ones (minimum 6 years) to refactor the test smells was about two days. On average, developers with intermediate experience (between 4 and 5 years) took three days to complete the tasks. Therefore, we claim the importance of diversifying a team of developers in terms of programming experience. Bearing in mind that such diversification will be highly beneficial for less experienced developers, who will improve their skills and, consequently, will start to refactor test smells in a shorter period.

Furthermore, the findings obtained in $\mathbf{RQ}_3$ align with the statement that developers with less experience require more time to refactor test smells. Table 11 presents the categories of difficulties reported by developers during the refactoring of test smells. From this information, it was possible to notice that categories, such as refactoring effort, problems in understanding the source code, and difficulty in applying the refactoring technique, were reported mainly by developers with less programming experience. Therefore, the test smells introduction training we carried out with all the developers was helpful for this experiment.

Finding 6: The experience level of developers significantly impacts the duration of test smell refactoring. We observed that less experienced developers took twice as long to complete the test smells refactoring compared to more experienced ones. In addition, the categories of difficulties identified in $\mathbf{RQ}_3$ were predominantly reported by less experienced developers. This suggests a direct relationship between the longer refactoring time and the challenges faced by developers with limited programming experience.

Implications of $\mathbf{RQ}_4$. Our findings infer that programming experience impacts the test smells refactoring. They bear in mind that developers reported virtually all difficulties collected with a maximum of four years of experience in programming. Consequently, the refactoring activity tends to take longer if performed by developers with less programming experience since they are more likely to experience difficulties when refactoring test smells. Therefore, a development team training process within a company/organization is essential. Such training should address theoretical and practical aspects of test smells to improve developers, increase the success rate in refactoring test smells, and reduce the time to implement such a process.

4.5 Test Smells Density ($\mathbf{RQ}_5$)

We ran the JNose Test to detect the existing test smells in the projects. Then, we asked the developers to refactor the five types of test smells (*Assertion Roulette, Eager Test, Magic Number Test, Sensitive Equality, Duplicate Assert*) in each selected software system. After refactoring, we reran the JNose Test and obtained a new .csv file with the test smells. As mentioned earlier in Section 3, the test smells refactored by the developers were all removed. We compared the test behavior before and after refactoring to answer this RQ.

Table 13 presents the density of test smells. The first column shows the test smells added to the project after refactoring; the second shows the number of test smells before refactoring. The third shows the number of test smells removed through refactoring in each project, the fourth shows the total number of test smells after refactorings, and the fifth shows

the number of added test smells after refactoring.

After test smells refactoring, we noticed that in project S1, all test smells were removed correctly, and during the refactoring, there was no change regarding the test smells. Project S2, after refactoring, added 6 test smells, 3 *Assertion Roulette* and 3 *Magic Number Test*. In project S3, the test smells that appeared after refactoring were 4 *Assertion Roulette* and 3 *Magic Number Test*. In project S4, after test smells refactoring, three more test smells were inserted (2 *Assertion Roulette* and 1 *Magic Number Test*.

Table 13. Density of test smells after refactoring

Test Smells	**S2 BR**	**RR**	**S2 AR***	**S2 Increased**
Assertion Roulette	4931	5	4929	3
Magic Number Test	1622	5	1620	3
Test Smells	**S3 BR**	**RR**	**S3 AR***	**S3 Increased**
Assertion Roulette	2697	5	2696	4
Magic Number Test	783	5	781	3
Test Smells	**S4 BR**	**RR**	**S4 AR***	**S4 Increased**
Assertion Roulette	135	5	132	2
Magic Number Test	38	5	34	1

Legend: Before Refactoring (BR), Refactoring and Removal (RR), After Refactoring (AR*), Magic Number Test (MNT)

The proposed test smells were refactored and removed correctly. While fixing these test smells (*Assertion Roulette, Eager Test, Magic Number Test, Sensitive Equality, Duplicate Assert*), new test smells were inserted. The test smells inserted are *Assertion Roulette* and *Magic Number Test*. The results show that the density of test smells decreases as the software evolves. Considering the four projects, 20 test smells were removed from the project; for example, the *Assertion Roulette*, 20 were removed, but during the refactoring, nine new *Assertion Roulette* were inserted.

Other results point out that there is a co-occurrence of test smells after test smells refactoring. In addition, other types of test smells were inserted in the projects, but there was no change concerning them, only about the test smells that were refactored. This may have occurred because the developers have reallocated or created new test methods.

Finding 7: Following the refactoring of test smells, increased density was observed in certain test smells. Specifically, the number of occurrences of *Assertion Roulette* increased by nine during the refactoring process. Similarly, the count of *Magic Number Test* instances increased by seven in the projects. This unforeseen outcome highlights the need for further investigation and potential adjustments in the refactoring process to address the unexpected rise in these specific test smells.

Implications of RQ$_5$. We compared the test behavior before and after refactoring the following test smells: *Assertion Roulette, Eager Test, Magic Number Test, Sensitive Equality, Duplicate Assert*. Out of the five refactored test smells, there was a co-occurrence of only two types of test smells (*Assertion Roulette and Magic Number Test*), distributed over three projects. The *Assertion Roulette* was the test smell with the most variation, and there was an addition of 9 insertions. In comparison, the test smells *Magic Number Test* varied with seven new insertions. These new insertions may occur due to the developer's effort to refactor the extensive test methods. These methods can have more than one test method that tests the same production method.

Furthermore, the test smells (*Assertion Roulette and Magic Number Test*) do not harm the project, so it may have been added, as reported by some of the developers. However, other test smells, such as *Sensitive Equality*, are more difficult to refactor, and new test smells can be inserted during refactoring. Therefore, further studies need to be conducted to find out whether other types of has a negative impact on the test code.

5 Threats to Validity

Internal validity. If a relationship between treatment and outcome is observed, we must be sure that it is a causal relationship and not the result of a factor over which we have no control or have not measured. Threats to internal validity concern issues that may indicate a causal relationship, even if none exists. Factors that impact internal validity are how subjects are selected and divided into different classes, how subjects are treated and compensated during the experiment, whether special events occur, etc. All these factors can cause the experiment to present behavior that is not due to the treatment but to the noise-causing factor.

In this sense, we might observe a potential threat in each developer's division of test smells. According to the study design, each developer only refactored one type of test smell. This might have influenced the results to some extent. However, taking from another perspective, this ensured a focused and dedicated approach to addressing specific test smells within the scope of the study.

Moreover, since the study involved professionals who had the freedom to perform the tasks at their discretion, there is a potential for a maturation effect on the results. This is attributed to the possibility of participants carrying out the tasks over an extended duration of time, potentially influencing the outcomes.

Construct validity. This validity concerns the relationship between theory and observation. If the relationship between cause and effect is causal, we must ensure that the treatment reflects the construct of cause well and that the outcome reflects the cause and effect well. It is about the extent to which the studied operational measures represent what the researcher has in mind and what is investigated according to the research questions.

Therefore, we could point out some threats that might have affected the results. For example, we only employed the JNose Test to identify test smells and the Understand for Java to gather code quality metrics. Although such tools are very representative of both the test smell detection field and the source code analysis field, if other tools were employed, the gathered results could have been different as well.

Another threat concerning construct validity is that the developers might subjectively fill the diary. However, to reduce the impacts of this threat, we have explained in detail the main reports written by the developers.

External validity. External validity refers to generaliz-

ability, the extent to which it is possible to generalize the findings, and the extent to which the findings interest others outside the investigated case. During the analysis, some questions must be asked, so the researcher tries to analyze the extent to which the findings are relevant to other cases. Threats to external validity concern the ability to generalize experiment results outside the experiment setting and are affected not only by the chosen experiment design but also by the chosen experiment objects and subjects.

In this sense, an important threat to consider in this study refer to the number of systems analyzed in our experiment. We only considered a small number of selected software systems. In addition, the results we gathered might be only valid for systems implemented in Java. We may not generalize the findings for every other scenario. Further empirical studies are necessary.

In addition, it is important to note that while we used the JNose tool for validating the removal of test smells, the refactoring process itself is not entirely free from side effects. We took rigorous measures to minimize these risks and maintain the desired behavior of the tests throughout the refactoring process. However, the complex nature of software systems introduces inherent uncertainties, and it is impossible to eliminate the possibility of side effects completely. We proactively monitored the test behavior, conducted thorough testing, and addressed any unintended changes. These considerations should be taken into account when interpreting and generalizing the results of our study.

6 Related Work

Over the past decade, researchers have made a remarkable effort to study (Palomba et al., 2014) and detect (Palomba et al., 2018) structural flaws in production code, the widely-known code smells (Van Rompaey et al., 2007). Analogously, problems concerning the test code design have been partially explored. This section discusses related work that has analyzed the impact of test smells on code maintenance, developer experience, and software quality.

6.1 Practitioners' perception of test smells

Campos et al. (2021) conducted an empirical study to investigate developers' perceptions about the severity of test smells and the behavior of test code after test smells refactoring. Most of the interviewed developers considered test smells to have low severity in their code. The developers report that test smells can negatively impact the project, particularly on the maintenance and evolution of the test code. Furthermore, the authors indicate that during the activity of refactoring test cases, it may induce the inclusion of new test smells.

Similarly, Santana et al. (2021) conducted a survey of 87 developers and interviewed another eight of these developers intending to understand how often and what strategies developers use to refactor test code to fix test smells. The results indicated that most participants consider it relevant to refactor test smells but do not refactor frequently. What differentiates our study from that of Campos et al. (2021); Santana et al. (2021) is that we investigate the density of test smells and the developers' experience with refactorings.

Soares et al. (2020) conducted a study to assess open-source developers' awareness of the existence of test smells and their appropriate refactoring strategies. To validate the research, the authors performed a two-part mixed study: (i) a survey of 73 open-source developers with extensive experience; and (ii) elicited 50 *pull requests* to assess the developers' acceptance of the refactoring proposals. As a result, the authors found that most developers chose to refactor proposals for 78% of the investigated test smells. Moreover, that *pull requests* had an average acceptance of 75% among the developers who were interviewed. In general, developers could identify the negative impact that test smells have on test code. Our experiment differs from this one because we evaluate the impact of test smell refactoring on the internal quality attributes size, cohesion, coupling, and complexity.

Spadini et al. (2020) investigated the severity rating for four test smells and their respective impacts on the maintainability of the developers who were selected to perform such activity. The authors analyzed about 1,500 open-source projects to obtain the severity limits for test smells to carry out the study. After that, they conducted a study with experienced developers to assess the limits of the severity of test smells. The results obtained after the application of the study were: (i) the set of rules for the detection of certain test smells was considered very restrictive by the developers, and (ii) the newly defined severity thresholds are in line with participants' perception of how test smells impact the sustainability of test files. In our study, we increased the number of test smells evaluated. Furthermore, we analyze the risks of test smells through the impact of refactoring such smells on the internal quality attributes.

Bavota et al. (2015) carried out an empirical study divided into two parts to analyze the prevalence and impacts that test smells have on test code. The first part of the study found a high occurrence of test smells in open-source and industrial systems. 86% of JUnit tests contain at least one test smell and six tests with six different test smells. The second part, in turn, shows that test smells negatively impact quality criteria such as the maintenance and understanding of the test code. The highlights of the second study point out that the understanding of the test code is 30% better in the absence of test smells. Our study found that in addition to negatively impacting test code understanding, test smells also negatively impact quality attributes, such as coupling, cohesion, and complexity.

6.2 Developers' Experience and Refactoring

Campos et al. (2022) conducted an empirical study to analyze developer experience and test code quality from the perspective of test smells. The authors investigated whether developer experience on the project influenced the insertion or removal of test smells in 4 Java language projects. In the study, the authors identified 18 types of test smells and their authorship. The authors of the test smells were classified as core and peripheral. The results indicate that core developers insert and remove more test smells than peripheral developers. Also, most test smells are removed due to the removal of classes or test methods, which may indicate that core and peripheral developers are unaware of test smells in the test

code. The difference in our study is that we want to know if developer experience can impact refactorings of test cases with test smells. We selected developers with skills other than project experiences, such as Java programming experience and quality attributes ranging from basic to advanced.

AlOmar et al. (2020, 2021) conducted an empirical study to identify contributors to production code and test code and associated a score for each contributor's experience with the number of recent commits (across commit authors and source files that were changed) over the past three years in refactoring activities across 800 open source projects hosted on GitHub. As a result, refactoring does not limit only a subset of developers. Furthermore, developers with higher scores performed more refactorings than other developers. Although, the authors report no correlation between experience and motivation behind refactoring. The difference in our study is focused on the relationship between developer experience and the difficulties in the activity of refactoring test smells.

Gall et al. (2010) investigated the relationship between different code ownership (known as developer experience in the project) and software failures in different domains (Windows Vista, Eclipse Java IDE, and the Firefox browser). The authors classified this experience as core or peripheral, core developers are the more experienced developers, and peripheral developers are the developers with less experience in the projects. As a result, the authors identified that the core developer has a relationship with pre-release and post-release failures. The difference between our study and Gall et al. (2010)'s study is that our study is focused on the developer's experience concerning their programming knowledge and length of experience from a test smells perspective. In this study, we observe whether developers with a long time of experience and knowledge in Java language programming refactor test cases with test smells more easily.

Spadini et al. (2018) investigated the link between the existence of test smells and the propensity for changes and defects in the test code. The authors collected data about 221 releases of ten software systems to conduct the research. They analyzed over one million test cases to investigate the relationship between six test smells and their co-occurrence with software quality. The results obtained in the research show that: (i) test smells are more subject to refactoring and defects; (ii) the *Indirect Testing*, *Eager Test* and *Assertion Roulette* test smells are the most biased towards refactoring; and (iii) the probability of production code having defects is much higher when tested by code that has test smells in its structure. However, this study does not assess the main difficulties encountered by developers during the test smell refactoring. Our study found that developers experience several difficulties in refactoring test smells, the main one being understanding the source code.

Kim et al. (2021) performed an empirical study on 12 open-source systems from projects considered relevant to understanding the evolution and maintenance of test smells and how such smells are linked to software quality. The results obtained by the authors highlight that: (i) although the number of test smell instances increases, the density of test smells decreases as systems evolve; (ii) 83% of removed test smells are a by-product of resource maintenance activities; (iii) 45% of the removed test smells were transferred to other test cases; and, (iv) most test smells have a minimal effect on post-release defects. Despite analyzing factors related to test smells and software quality, the authors did not investigate the impacts of test smell refactoring on internal quality attributes. Our experiment found that refactoring test smells impacts the values of internal quality attributes in several ways.

7 Concluding Remarks

Our study investigated the impact of test smell refactoring from developers' perspectives and the internal quality attributes. Our experiment considered 100 examples of five types of test smells present in four open-source systems and four internal quality attributes: (i) size, (ii) cohesion, (iii) coupling, and (vi) complexity. Twenty developers refactored the test smells, and their perceptions and difficulties were qualitatively collected using the diary technique.

Our main findings were: (i) after refactoring the test smells, cohesion increased respectively, 7.26%, 6.20%, 9.19%, and 9.47% in the four analyzed systems; (ii) after refactoring the test smells, the complexity decreased respectively, 20.16%, 7.55%, 28.53% and 21% in the four analyzed systems; (iii) the developers considered the *Assertion Roulette* and *Magic Number Test* test smells to be the least harmful smells within a software project. On the other hand, the *Eager Test* and *Duplicate Assert* tests were considered the most critical ones; (iv) understanding the source code is one of the main difficulties in refactoring of test smells; (v) the larger the source code, the more effort it will take to complete the refactoring; (vi) developers with less experience (maximum 3 years) took twice as long to complete the test smells refactoring compared to more experienced developers (minimum 6 years); (vii) by refactoring large methods there can be a co-occurrence of test smells; and, (viii) after refactoring, there was an addition of 16 test smells (9 *Assertion Roulette* and 7 *Magic Number Test*), but as the project evolves the density of these test smells decreases.

Based on the results obtained, we developed the following recommendations for software development teams: (i) prioritize the refactoring of the Eager Test and Duplicate Assert test smells; (ii) implement test smells identification and refactoring training; (iii) implement test smell detection and refactoring tools; and, (iv) establish a refactoring hierarchy of test smells, according to the internal quality attribute that needs to be improved.

In future work, we intend to: (i) increase the number of systems to be used; (ii) increase the number of test smells to be evaluated; (iii) use other tools to identify test smells and measure quality attributes; (iv) analyze systems developed in other programming languages; (v) increase the number of developers; (vi) analyze the phenomenon of co-occurrences of test smells in test files; and, (vii) evaluate the developers' ability to detect test smells.

Acknowledgements

This work is partially supported by the Ceará Foundation for Scientific and Technological Support (FUNCAP-CE); INES

(www.ines.org.br), CNPq grant 465614/2014-0, FACEPE grants APQ0399-1.03/17 and APQ/0388-1.03/14, CAPES grant 88887.136410/2017-00; the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001; and FAPESB grants BOL0599/2019 and PIE0002/2022.

References

Alenezi, M. and Almustafa, K. (2015). Empirical analysis of the complexity evolution in open-source software systems. *International Journal of Hybrid Information Technology*, 8(2):257–266.

Alizadeh, V., Kessentini, M., Mkaouer, M. W., Ó Cinnéide, M., Ouni, A., and Cai, Y. (2020). An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, 46(9):932–961.

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*, EASE 2021, page 170–180, New York, NY, USA. Association for Computing Machinery.

AlOmar, E. A., Peruma, A., Mkaouer, M. W., Newman, C. D., and Ouni, A. (2021). Behind the scenes: On the relationship between developer experience and refactoring. *CoRR*, abs/2109.11089.

AlOmar, E. A., Peruma, A., Newman, C. D., Mkaouer, M. W., and Ouni, A. (2020). On the relationship between developer experience and refactoring: An exploratory study and preliminary results. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 342–349, New York, NY, USA. ACM.

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. D., and Binkley, D. W. (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 2012, Trento, Italy, September 23-28, 2012*, pages 56–65. IEEE Computer Society.

Beller, M., Gousios, G., Panichella, A., Proksch, S., Amann, S., and Zaidman, A. (2019). Developer testing in the IDE: patterns, beliefs, and behavior. *IEEE Trans. Software Eng.*, 45(3):261–284.

Beller, M., Gousios, G., Panichella, A., and Zaidman, A. (2015). When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 179–190, New York, NY, USA. Association for Computing Machinery.

Berner, S., Weber, R., and Keller, R. K. (2005). Observations and lessons learned from automated testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 571–579, New York, NY, USA. Association for Computing Machinery.

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In Briand, L. C. and Wolf, A. L., editors, *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 85–103. IEEE Computer Society.

Bieman, J. M. and Kang, B.-K. (1995). Cohesion and reuse in an object-oriented system. *SIGSOFT Softw. Eng. Notes*, 20(SI):259–262.

Campos, D., Martins, L., and Machado, I. (2022). An empirical study on the influence of developers' experience on software test code quality. In *Proceedings of the XXI Brazilian Symposium on Software Quality*, SBQS '22, New York, NY, USA. Association for Computing Machinery.

Campos, D., Rocha, L., and Machado, I. (2021). Developers' perception of the severity of test smells: an empirical study. pages 192–205.

Candea, G., Bucur, S., and Zamfir, C. (2010). Automated software testing as a service. SoCC '10, page 155–160, New York, NY, USA. Association for Computing Machinery.

Chávez, A., Ferreira, I., Fernandes, E., Cedrim, D., and Garcia, A. (2017). How does refactoring affect internal quality attributes? a multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES'17, page 74–83, New York, NY, USA. Association for Computing Machinery.

Chidamber, S. R. and Kemerer, C. F. (1991). Towards a metrics suite for object oriented design. *SIGPLAN Not.*, 26(11):197–211.

Damasceno, H., Bezerra, C., Coutinho, E., and Machado, I. (2022). Analyzing test smells refactoring from a developers perspective. In *Proceedings of the XXI Brazilian Symposium on Software Quality*, SBQS '22, New York, NY, USA. Association for Computing Machinery.

Darcy, D. P., Kemerer, C. F., Slaughter, S., and Tomayko, J. E. (2005). The structural complexity of software: An experimental test. *IEEE Trans. Software Eng.*, 31(11):982–995.

Dyer, R., Rajan, H., and Cai, Y. (2012). An exploratory study of the design impact of language features for aspect-oriented interfaces. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*, AOSD '12, page 143–154, New York, NY, USA. Association for Computing Machinery.

Fernandes, E., Chávez, A., Garcia, A., Ferreira, I., Cedrim, D., da Silva Sousa, L., and Oizumi, W. N. (2020). Refactoring effect on internal quality attributes: What haven't they told you yet? *Inf. Softw. Technol.*, 126:106347.

Fowler, M. (1999). *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley.

França, A. C. C., da Silva, F. Q. B., and Sharp, H. (2020). Motivation and satisfaction of software engineers. *IEEE Trans. Software Eng.*, 46(2):118–140.

Gall, H., Devanbu, P., Murphy, B., Bird, C., and Nagappan, N. (2010). An analysis of the effect of code ownership on software quality across windows, eclipse, and firefox.

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.

ISO (2011). Iec 25010: 2011 systems and software engineering–systems and software quality requirements and evaluation (square)–system and software quality models. *International Organization for Standardization*, 34:2910.

Kim, D. J. (2020). An empirical study on the evolution of test smell. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, page 149–151, New York, NY, USA. Association for Computing Machinery.

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 Softw. Engg.*, 26(5).

Lincke, R., Lundberg, J., and Löwe, W. (2008). Comparing software metrics tools. In Ryder, B. G. and Zeller, A., editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 131–142. ACM.

Lorenz, M. and Kidd, J. (1994). *Object-oriented software metrics - a practical guide*. Prentice Hall.

Malhotra, R. and Chug, A. (2016). An empirical study to assess the effects of refactoring on software maintainability. In *2016 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2016, Jaipur, India, September 21-24, 2016*, pages 110–117. IEEE.

Martins, J., Bezerra, C., Uchôa, A., and Garcia, A. (2021). How do code smell co-occurrences removal impact internal quality attributes? a developers' perspective. In *Brazilian Symposium on Software Engineering*, SBES '21, page 54–63, New York, NY, USA. Association for Computing Machinery.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Morasca, S. (2009). A probability-based approach for measuring external attributes of software artifacts. In *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, October 15-16, 2009, Lake Buena Vista, Florida, USA*, pages 44–55. IEEE Computer Society.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The art of software testing*. John Wiley & Sons.

Nilson, M., Antinyan, V., and Gren, L. (2019). Do internal software quality tools measure validated metrics? In Franch, X., Männistö, T., and Martínez-Fernández, S., editors, *Product-Focused Software Process Improvement*, pages 637–648, Cham. Springer International Publishing.

Orso, A. and Rothermel, G. (2014). Software testing: A research travelogue (2000–2014). In *Future of Software Engineering Proceedings*, FOSE 2014, page 117–132, New York, NY, USA. Association for Computing Machinery.

Paixão, M., Uchôa, A., Bibiano, A. C., Oliveira, D., Garcia, A., Krinke, J., and Arvonio, E. (2020). Behind the intents: An in-depth empirical study on software refactoring in modern code review. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 125–136, New York, NY, USA. Association for Computing Machinery.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., and Lucia, A. D. (2014). Do they really smell bad? A study on developers' perception of bad code smells. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 101–110. IEEE Computer Society.

Palomba, F., Zaidman, A., and Lucia, A. D. (2018). Automatic test smell detection using information retrieval techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pages 311–322. IEEE Computer Society.

Pecorelli, F., Palomba, F., and Lucia, A. D. (2021). The relation of test-related factors to software quality: A case study on apache systems. *Empir. Softw. Eng.*, 26(2):18.

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. pages 193–202.

Santana, R., Fernandes, D., Campos, D., Soares, L., Maciel, R., and Machado, I. (2021). Understanding practitioners' strategies to handle test smells: A multi-method study. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, SBES '21, page 49–53, New York, NY, USA. Association for Computing Machinery.

Santana, R., Martins, L., Virgínio, T., Soares, L., Costa, H., and Machado, I. (2022). Refactoring assertion roulette and duplicate assert test smells: a controlled experiment. *arXiv preprint arXiv:2207.05539*.

Sjøberg, D. I., Yamashita, A., Anda, B. C., Mockus, A., and Dybå, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156.

Soares, E., Ribeiro, M., Amaral, G., Gheyi, R., Fernandes, L., Garcia, A., Fonseca, B., and Santos, A. (2020). Refactoring test smells: A perspective from open-source developers. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, SAST 20, page 50–59, New York, NY, USA. Association for Computing Machinery.

Soares, E., Ribeiro, M., Gheyi, R., Amaral, G., and Santos, A. L. M. (2022). Refactoring test smells with junit 5: Why should developers keep up-to-date? *IEEE Trans. Software Eng.*, 49(3):1152–1170.

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., and Bacchelli, A. (2018). On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pages 1–12. IEEE Computer Society.

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 '20, page 311–321, New York, NY, USA. Association for Computing Machinery.

Tahir, A., Counsell, S., and MacDonell, S. G. (2016). An empirical study into the relationship between class features

and test smells. In Potanin, A., Murphy, G. C., Reeves, S., and Dietrich, J., editors, *23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, December 6-9, 2016*, pages 137–144. IEEE Computer Society.

Tarwani, S. and Chug, A. (2016). Sequencing of refactoring techniques by greedy algorithm for maximizing maintainability. In *2016 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2016, Jaipur, India, September 21-24, 2016*, pages 1397–1403. IEEE.

Van Deursen, A., Moonen, L., Van Den Bergh, A., and Kok, G. (2001). Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95. Citeseer.

Van Rompaey, B., Du Bois, B., Demeyer, S., and Rieger, M. (2007). On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817.

Virgínio, T., Martins, L., Rocha, L., Santana, R., Cruz, A., Costa, H., and Machado, I. (2020). Jnose: Java test smell detector. In *Proceedings 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., 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. *Journal of Software Engineering Research and Development*, 9:8–1.

Virgínio, T. G. A. and Machado, I. (2021). Avaliação empírica da geração automatizada de testes de software sob a perspectiva de test smells. In *Anais Estendidos do XII Congresso Brasileiro de Software: Teoria e Prática*, pages 112–126. SBC.

Xie, G., Chen, J., and Neamtiu, I. (2009). Towards a better understanding of software evolution: An empirical study on open source software. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 51–60. IEEE Computer Society.

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