An Empirical Study on the Co-occurrence of Test Smells

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

Luana Martins
Department of Computer Science
University of Salerno
Salerno, Italy
lalmeidamartins@unisa.it

Márcio Ribeiro
Institute of Computing
Federal University of Alagoas
Maceió, Alagoas, Brazil
marcio@ic.ufal.br

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

ABSTRACT

The co-occurrence of test smells poses a challenge for refactoring test code, as these issues rarely appear in isolation. Consequently, developers often need to apply multiple transformations during refactoring, which increases complexity and effort. Therefore, it is essential to explore efficient strategies that reduce steps and practically allow the simultaneous refactoring of these problems. Although the literature already documents several test smells, the joint refactoring of these problems remains unaddressed. Therefore, this paper investigates the co-occurrence of test smells in test code (both at the class and test method level) and strategies to fix them efficiently, minimizing the number of transformations and preserving the behavior of the tests. To achieve this goal, we identified test smells using an automated tool in twenty-two open-source projects. We evaluated the co-occurrence between different types of test smells. Then, we ranked the thirty most frequent test smells pairs and suggested ways to refactor them in an integrated manner. Our findings can support developers in improving the quality of test cases, as our approach was designed with the industry's reality in mind, where test smells often appear simultaneously.

KEYWORDS

Software testing, Test code, Unit testing, Test smells, Anti-patterns, Controlled experiments

1 Introduction

Less than a decade ago, test code was often not prioritized within the software development cycle [4]. However, this view has changed significantly [35]. Testing is increasingly valued and encouraged, becoming an essential criterion and a competitive differentiator for candidates for software engineering positions [34].

This growing appreciation of testing has occurred due to several factors, such as the growth of agile methodologies [18], increased complexity of systems [15, 39], greater demand for software quality [5, 14], popularization of automation tools [8, 25], and change in corporate culture [23, 30]. Furthermore, although developing tests may be an 'additional task' for the software engineer [11], its benefits outweigh the cost of implementation. For instance, testing allows for the early identification of defects in production code, reducing the costs of corrections throughout the project life cycle [39], and detecting possible defects before delivering it to their end-users [7].

In this context, software testing has been increasingly recognized as a fundamental activity in software development. However, like source code, test code must meet a high-quality standard, respecting good practices and design patterns [13, 33]. According to Panichella et al. [20], writing test code ad hoc can compromise its maintainability and make it difficult to understand over time. Hence, when good coding choices [17] are not followed in the development of test code, so-called test smells [6, 23, 40] appear, indications that the quality of the test code is compromised and needs to be improved.

Test smells act as a warning mechanism [26, 27], similar to a fever in the human body, which signals the presence of infections or inflammations. Just as a fever indicates the need for medical investigation and treatment, test smells reveal bad practices in the test code that must be corrected. However, these poor practices rarely occur in isolation [19, 29, 38]; different types of test smells are often introduced simultaneously, making their identification and correction more challenging. What could be a single problem can quickly become a set of symptoms that harm the quality of the test code, "*developers can cope with one antipattern but that combinations of antipatterns should be avoided possibly through detection and refactoring*" [1].

A test smell instance can be removed through the refactoring process [6], in which small transformations are applied to maintain the test code's original behavior. In other words, refactoring removes the impurities that contaminate the code providing higher quality code [9, 10, 16, 31]. Resuming the analogy with the health sector, when a patient has multiple diseases, doctors must carefully choose the medications to treat one condition without aggravating another. Similarly, when correcting multiple test smells, an in-depth study is required to propose the best refactoring approach.

Although the literature presents an extensive list of test smells [12], the joint correction of these problems is still little explored despite their high frequency of occurrence. Our study seeks to delve deeper into and understand the different types of co-occurrence of test smells, both at the class and test method levels, since the nature of these co-occurrences can directly influence refactoring decisions and strategies.

We summarize our study contribution as follows: we analyzed the co-occurrence of 19 types of test smells at the class and method

levels in 22 open-source projects; we have created and made available scripts used to identify co-occurrences of test smells; we ranked the 15 most frequent co-occurrences of test smells at the class level; we ranked the 15 most frequent co-occurrences of test smells at the method levels; we evaluate and classify test smell co-occurrences into three categories (structural, statistical, and vicious); and we suggested 16 refactorings to remove co-occurrences of test smells.

2 Background

Test smells are poor strategies used when coding test cases, which can compromise the quality and maintainability of tests [6]. Test classes that present test smells tend to be more challenging to understand than those free of these problems [32].

Table 1 presents 19 test smells considered in this study, which are the types most frequently discussed in the literature [3] and also supported by tool [36]. The first column names the code smells. The remainder columns present: a short description (according to Deursen et al. [6], Peruma [22]) and the line where this type of test smell is represented in Listing 1, which exemplifies these smells in a toy example of test code written in Java, using the JUnit framework. We considered the types of test smells most frequently discussed in the literature [3, 6, 22–24, 38] and the types supported by JNose Test tool [36]. Listing 1 presents the CalculatorTest test class. Although it is an illustrative example created to demonstrate the presence and co-occurrence of test smells, it is common to find, in real systems, test classes with similar structures and with such maintenance and readability problems.

As demonstrated in Listing 1, it is possible to observe that practically all of the test smells analyzed co-occurred with other types. This overlap is common in poorly structured test classes and contributes significantly to the reduction of test code quality. For example, the *Mystery Guest* and *Resource Optimism* smells co-occur in the `testOperationsFromFile()` method, highlighting the dependence on external resources without proper verification of their state. Similarly, the Empty Test and Unknown Test smells appear together in the `testSquareRoot()` method, evidencing the absence of assertions and executable instructions. These co-occurrence patterns reinforce the importance of considering the interaction between different types of test smells, and of studying strategies to deal with this co-occurrence.

Table 1: Test class example with multiple test smells.

Test smell	Description	Lines in Listing 1
Assertion Roulette (AR)	A test method with two or more assertions [6].	29, 32, 35, 38, 50, 51, 54, 74, 77, 81, 84, 89
Conditional Test Logic (CTL)	A test method in which the result depends on a decision structure (e.g., if, else, for) [21].	20-42, 27-39
Constructor Initialization (CI)	A test class that contains the constructor method [21].	9-11
Duplicate Assert (DA)	A test method that has assertions with equal parameters [21].	77 and 81
Eager Test (EgT)	A test method that invokes different production methods [6].	34, 83
Empty Test (ET)	A test method that has no executable statements [21].	65-70
Exception Handling (EH)	A test method in which the result depends on an exception handling (e.g, try/catch) [21].	14-55
General Fixture (GF)	A test class with test methods that only access part of the class' fixture [6].	6, 7
Ignored Test (IT)	A test method ignored due to the use of the @Ignore() annotation [21].	57-64
Lazy Test (LT)	Two or more test methods that invoke the same production method [6].	28, 31, 34, 37 and 76, 80, 83, 84
Magic Number Test (MNT)	A test method that contains assertion with numeric literal as an argument [21].	23, 29, 32, 35, 38, 47, 51, 77, 81, 83, 84, 88, 89
Mystery Guest (MG)	A test method that uses external resources (e.g., files, database) [6].	15, 17, 49
Print Statement (PS)	A test method that invokes the print method or some similar method of the System class [21].	63
Redundant Assertion (RA)	A test method that contains assertions that force it to return true or false [21].	84
Resource Optimism (RO)	A test method that uses external resources and is so optimistic about the resource's state that it does not check before using it [6] .	16, 41, 50, 51
Sensitive Equality (SE)	A test method that invokes the toString method of some object [6].	89
Sleepy Test (ST)	A test method that invokes the *Thread.sleep()* method to break test execution for a period [21].	47
Unknown Test (UT)	A test method that does not contain assertion statements [21].	59-64, 66-70
Verbose Test (VT)	A test method that has many lines of code. We consider a method as verbose if it has 30 or more lines of code[36] .	13-56

3 Related Work

Test smell analysis has been the focus of several studies in software engineering, especially in the context of automated testing and code maintenance. Two relevant works that stand out in the theme of co-occurrence and detection of test smells are the works of Bavota et al. [3] and Palomba et al. [19].

Bavota et al. [3] investigated the diffusion and co-occurrence of test smells in JUnit test classes extracted from open-source and industrial projects. They analyzed both the production code and the test classes, identifying that some test smells appear together frequently, such as *Assertion Roulette* and *Eager Test*. One of the main contributions of this study was the introduction of a metric to calculate the co-occurrence between two smells, allowing us to observe test smell co-occurrence patterns and their correlations with other software metrics, such as LOC and the number of JUnit classes.

While Bavota et al. [3] mainly focused on test classes, our research goes further, investigating test smell co-occurrences at two levels of granularity: at the method level and at the class level. Moreover, our study covers a larger set of 19 test smells, compared to the 8 analyzed by Bavota et al. [3]. This novel approach allows for a more detailed analysis of the interactions between smells within individual classes and methods.

The study by Palomba et al. [19] also focuses on the detection and co-occurrence of test smells but with a particular focus on automatically generated tests using the EvoSuite tool. Palomba et al. [19] explore how to test smells often arise in automatically generated tests and how different test generation tools fail to mitigate design issues that lead to the introduction of test smells. Furthermore, they observe that test smells such as *Test Code Duplication* and *Indirect Testing* tend to appear together when tests reference intermediate classes or use the same set of declarations. In contrast, our paper is not restricted to automatically generated tests. We focus on analyzing 22 open-source projects with manually written test code, allowing us to observe co-occurrence patterns in a real development context, where coding practices can vary significantly between projects.

While both Bavota et al. [3]'s and Palomba et al. [19]'s studies have contributed to the understanding of test smells and their co-occurrences, our present study offers a complementary contribution. Our research, with its comprehensive analysis of a larger set of test smells and consideration of two levels of granularity, providing more details about each co-occurrence.

4 Study Settings

This section details the definition of our empirical study and how we designed the study to meet our objective, which is understanding

the most frequent co-occurrences of test smells in open source projects.

To conduct this investigation into connections and dependencies between different types of test smells, we define the following research question (RQ): **Which test smells co-occur together?** This RQ aims to understand the relationship between the different test smells, i.e., which test smell co-occur, both at the test method level and at the test class level. Our study followed these steps:

Step 1: Selection of the dataset for analysis. We selected 22 open-source projects written in the Java language and test classes using the JUnit framework by convenience through the SEART platform (https://seart-ghs.si.usi.ch/), utilizing keywords such as "Java" in the language field and "JUnit" in the label field. Furthermore, we considered only repositories with at least 5 stars and collaboration from two or more contributors. Next, we performed manual inspections to verify that the projects met our specifications, in addition to being Maven projects.

Step 2: Selection of test smell types. In our investigation, we considered the test smells listed in Section 2, such test smells are the most used in the literature [2, 6, 21, 28].

Step 3: Test smell detection. After obtaining the source code of each available project, we used the JNose Test to detect test smells, a tool developed and evaluated by Virgínio et al. [36], Virgínio et al. [37], which yielded precision results ranging from 91% to 100% and recall rates between 89% and 100%. JNose Test stands out from other tools for its broad coverage of supported test smells, in addition to indicating the line of each detected instance.

Step 4: Co-occurrence detection. As far as we know from the literature, there are no tools for automatic analysis of test smell co-occurrences. Therefore, we rely on the study by [19], who present the following formula to identify smell co-occurrences in test code:

$$co\text{-}occurrence_{t_i,t_j} = \frac{|t_i \wedge t_j|}{|t_i|} \quad (1)$$

In detail, this formula calculates the percentage of times that each test smell t_i co-occurs with another test smell $t_j (i \neq j)$, where $|t_i \wedge t_j|$ is the number of times t_i co-occurs with t_j, and $|t_i|$is the number of occurrences of t_i. Formula 1 measures the influence of one test smell on the occurrence of another, being an asymmetric relationship. Therefore, $co\text{-}occurrence_{t_i,t_j} \neq co\text{-}occurrence_{t_j,t_i}$.

To assist in the process of detecting co-occurrences of test smells, we developed scripts that, based on the input from the projects' test smell reports, synthesize the required data for the co-occurrence analysis. The scripts were designed to process the data, allowing us to calculate co-occurrences more quickly in spreadsheets. Two experienced test smell researchers validated our scripts. Changes were made until the results of the scripts obtained 100% similarity with the results of our oracle.

Step 5: Analyses and Documenting. Our co-occurrence analysis consisted of both class and test method levels. We also classified co-occurrences into three categories: *i)* ***Structural***: Co-occurrence motivated predominantly by structural characteristics of the test smells themselves, that is, it results from a natural overlap between the elements that characterize each smell. This type of co-occurrence occurs, for example, when two smells are directly related to the same test structure, such as assertions. Example: The simultaneous presence of *Assertion Roulette* and *Duplicate Assert* is structural because both test smells are strictly associated with

Listing 1: Toy example of test class with multiple Test Smells.

```
import static org.junit.*;
...

class CalculatorTest {
    private Calculator calculator;
    private int operand1 = 2;
    private int operand2 = 4;

    public CalculatorTest() {
        this.calculator = new Calculator();
    }
    @Test
    public void testOperationsFromFile() {
        try {
            File inputFile = new File("operations.txt");
            BufferedReader reader = new BufferedReader(new FileReader(inputFile));
            FileWriter writer = new FileWriter("results.txt");

            String line;
            while ((line = reader.readLine()) != null) {

                String[] parts = line.split(" ");
                String operation = parts[0];
                int operand1 = Integer.parseInt(parts[1]);
                int operand2 = Integer.parseInt(parts[2]);
                int result = 0;
                if (operation.equals("add")) {
                    result = calculator.add(operand1, operand2);
                    assertEquals(34, result);
                } else if (operation.equals("subtract")) {
                    result = calculator.subtract(operand1, operand2);
                    assertEquals(42, result);
                } else if (operation.equals("multiply")) {
                    result = calculator.multiply(operand1, operand2);
                    assertEquals(100, result);
                } else if (operation.equals("divide")) {
                    result = calculator.divide(operand1, operand2);
                    assertEquals(36, result);
                }

                writer.write(operation + " " + operand1 + " " + operand2 + " = " + result + "\n");
            }

            reader.close();
            writer.close();

            Thread.sleep(500);

            File resultFile = new File("results.txt");
            assertTrue(resultFile.exists());
            assertTrue(resultFile.length() > 0);

        } catch (IOException | InterruptedException e) {
            fail("Error while reading/writing the file: " + e.getMessage());
        }
    }
    @Disabled
    @Test
    public void testExponent() {
        int operand1 = 9;
        int operand2 = 0;
        int result = calculator.Exponent(operand1, operand2);
        System.out.println( operand1 + "^" + operand2 + " = " + result);
    }
    @Test
    public void testSquareRoot() {
        //int operand1 = 100;
        //int result = calculator.squareRoot(operand1);
        //assertEquals(10, result);
    }
    @Test
    public void testFactorial() {
        int result;
        assertNotNull(calculator);
        operand1 = 0;
        result = calculator.factorial(operand1);
        assertEquals(result, 1);

        operand1 = 1;
        result = calculator.factorial(operand1);
        assertEquals(result, 1);

        int result2 = calculator.multiply(3, calculator.factorial(2));
        assertEquals(calculator.factorial(3), result2);
    }
    @Test
    public void testScientificNotation() {
        ScientificNotation result = calc.toScientificNotation(123456789);
        assertEquals("1.23457E8", result.toString());
    }
}
```

the assertion structure. *ii)* **Statistical**: Co-occurrence that occurs predominantly due to statistical factors, such as the high individual frequency of certain test smells in the analyzed data set. In these cases, it is not possible to identify a clear technical justification that explains the association between the smells. Example: The co-occurrence between *Assertion Roulette* and *Unknown Test* at method level is not possible, as the former indicates multiple non-documented assertions and the latter indicates the absence of assertions in the method. However, these test smells might co-occur at the class level due to the high number of methods exhibiting one or the other smell — i.e., the co-occurrence results from statistical aggregation rather than any relationship between their characteristics. *iii)* **Vicious** *or Indirect Structural:* Co-occurrence resulting from bad practices, development flaws or poorly planned decisions, which end up violating the same test design principles. In this case, although there is no direct structural link between the smells, they arise together as a consequence of recurring conceptual flaws. Example: The co-occurrence between *Eager Test* and *Lazy Test* may occur due to a programming flaw in not ensuring that each test method checks only one production method.

We performed this classification manually according to the defined criteria. To support the reproducibility, we created and made available an explanatory diagram. Section 5 details the analysis of the co-occurrences found. All materials used in our analysis were made available digitally in *Artifact Availability* section.

5 Results and Discussion

Our dataset is composed of broadly used open-source projects in different types of applications. In addition, the selected projects are maintained by recognized organizations, such as Apache Software Foundation and Facebook. Table 2 presents the selected projects and describes their characteristics: the commit version used, the number of Java files, the number of test classes, the number of smelly test classes, the number of test methods, the number of smelly test methods, and the total lines of code (LOC) in each project. As in the study by Palomba et al. [19], we chose not to perform a temporal analysis of the test smells. The inclusion of multiple commits from the same project could introduce bias in the database, as it overestimates certain types of co-occurrence due to the repetition of patterns already present.

In total, the projects have 38,519 Java files, 7,066 test classes, 42,349 test methods, and 7,423,419 LOC. Cfx is the largest project considering the number of Java files (7,612), test classes (1,274), and test methods (7,775). Regarding LOC, the Hive project has the largest number (1,452,697). On the other hand, the jfnr project has the smallest size considering all the characteristics analyzed: only 4 Java files, 1 test class, 8 test methods, and 644 LOC. The ratio of test classes to Java files in the dataset is approximately 18.34%, while the average number of methods per test class is 5.99. The average project star rating is approximately 2,477 and the average contributor rating is 116.

Regarding the distribution of test smells in the classes, more than half of the test classes in our dataset (50.18%) presented at least one instance of test smell, evidencing the presence of these bad practices in the projects analyzed. The projects ASC and jfnr presented the highest rate of test smells in the classes, that is, bad design practices compromised all test classes (100.00%). Regarding the level of test methods, the jfnr project also stood out negatively, with 100.00% of its methods containing test smells. On the other hand, the jcef project presented the lowest overall compromise: only 8.33% of the test classes and 3.70% of the test methods presented test smells, being the project with the lowest incidence of test anti-patterns.

Table 3 presents the distribution of test smell instances detected in the 22 projects analyzed. The first two columns refer to the projects, while columns 3 to 21 correspond to each specific type of test smell. Column 22 displays the total number of instances detected per project, and the last column indicates the number of different types of test smells identified in each of them.

As often observed in similar studies, the *Assertion Roulette* test smell was the most recurrent among the projects, with a total of 56,286 instances, followed by the *Magic Number Test* test smell (16,649) and the *Eager Test* test smell (9,710). In contrast, the *Empty Test* test smell was the least frequent, with only 11 occurrences among the 22 projects evaluated.

Considering the total number of instances per project, Hive had the highest number (16,669), while the jcef project had the lowest number of instances, which is consistent with the size of each project (Hive is the largest in LOC, and jcef, the smallest).

Only three projects presented all types of test smells: Bookkeeper, Cxf and Hive. On average, each project presented 14 distinct types, which shows a wide variety of test anti-patterns adopted by engineers. The jcef project also stood out for having the lowest diversity of test smells, with only three types identified: *Assertion Roulette*, *Eager Test* and *Verbose Test*.

To answer the RQ, we calculated the co-occurrences at both the test class and method levels for a complete investigation. We then ranked the top co-occurrences. For co-occurrence cases where there was both $co\text{-}occurrence_{t_i,t_j}$ (where $i = a$ and $j = b$) and $co\text{-}occurrence_{t_i,t_j}$ (where $i = b$ and $j = a$), we consider the co-occurrence with the highest frequency.

To enable the analysis of co-occurrences at different levels, we initially investigated the test smell instances individually, considering their distribution across both test classes and methods. Our script then processed this data to count the number of test smell couples that occurred simultaneously in the same class. For example, after loading the test smell detection results, the script identified how many classes simultaneously presented the *Assertion Roulette* and *Conditional Test Logic* test smells. This procedure was performed for all possible combinations between all types of test smells. We repeated the same process at the method level.

We applied Formula 1 to calculate 342 class-level co-occurrences and 342 test-level co-occurrences, totaling 684 co-occurrences. To make the analysis more objective, we only considered the co- occurrences whose joint occurrence rate was higher than 50% (following the approach of [19]). This paper presents the RQ results in two parts: class-level and method-level co-occurrence analysis.

5.1 Co-occurrence in test classes

Table 4 presents co-occurrences of test classes level. Next, we present and explain the top 15 ranked co-occurrences, starting with the most frequent ones and moving on to the least frequent ones. At the test class level, the test smells that co-occurred the

Table 2: Details of subjects.

#	Project	Commit	Java Files	Test Classes	Smelly Test Classes (%)	Test Methods	Smelly Test Methods (%)	LOC
1	Accumulo	47ac68d	2,179	561	248 (44.21%)	2,054	1,046 (50.93%)	657,852
2	AMDP	e8c1a62	138	13	9 (69.23%)	61	56 (91.80%)	21,862
3	ASC	bc5867d	320	51	51 (100.00%)	260	220 (84.62%)	31,686
4	Bookkeeper	f233320	2,353	564	327 (57.98%)	2,760	1,691 (61.27%)	421,111
5	Cassandra	6b13426	3,749	1,058	482 (45.56%)	6,253	2,845 (45.50%)	839,494
6	Cayenne	7e15607	4,198	840	347 (41.31%)	4,310	1,438 (33.36%)	494,715
7	Cxf	6b27aec	7,612	1,274	645 (50.63%)	7,775	3,699 (47.58%)	1,045,091
8	DBeam	a8d3bc5	47	15	12 (80.00%)	101	86 (85.15%)	6,330
9	dble	e3a31b0	1,641	53	35 (66.04%)	307	170 (55.37%)	272,539
10	etcd-java	1fde9c0	40	9	2 (22.22%)	29	9 (31.03%)	9,845
11	fb-jb-sdk	be6bc7b	1,113	15	10 (66.67%)	62	46 (74.19%)	674,843
12	gctoolkit	4342691	276	54	13 (24.07%)	175	55 (31.43%)	29,872
13	Guice	b0e1d0f	621	166	23 (13.86%)	1,645	193 (11.73%)	106,762
14	Hive	5160d3a	5,088	752	391 (51.99%)	4,556	2,271 (49.85%)	1,452,697
15	jcef	a414114	274	12	1 (8.33%)	27	1 (3.70%)	158,387
16	jfnr	c9dbfee	4	1	1 (100.00%)	8	8 (100.00%)	644
17	Joda Time	290a451	330	135	65 (48.15%)	4,269	1,246 (29.19%)	148,962
18	JSqlParser	e47132a	284	56	37 (66.07%)	829	281 (33.90%)	27,969
19	kapua	56adf32	3,693	414	351 (84.78%)	2,531	2,112 (83.45%)	374,960
20	Neptune Export	632e0de	355	77	56 (72.73%)	395	248 (62.78%)	34,949
21	Wicket	48b1c3c	3,279	619	318 (51.37%)	2,510	1,513 (60.28%)	427,267
22	Zookeeper	e454e8c	925	327	122 (37.31%)	1,432	709 (49.51%)	185,582
		Total	**38,519**	**7,066**	**3,546 (50.18%)**	**42,349**	**19,943 (47.09%)**	**7,423,419**

Table 3: Distribution of instances Test Smell.

#	Project	AR	CTL	CI	DA	EgT	ET	ECT	GF	IT	LT	MNT	MG	PS	RA	RO	SE	ST	UT	VT	Total	Smells types
1	Accumulo	3,226	382	5	243	533	0	131	88	19	541	700	34	4	33	33	150	14	198	187	6,521	18
2	AMDP	166	0	0	22	21	0	4	12	9	23	18	15	0	0	15	0	0	2	8	315	12
3	ASC	497	0	0	8	26	2	0	15	0	88	80	0	0	2	0	7	0	1	8	734	11
4	Bookkeeper	4,868	1,084	74	584	958	2	642	149	51	758	1,882	165	5	15	159	63	71	239	539	12,308	19
5	Cassandra	7,461	1,598	3	1,067	1,018	0	637	82	215	1,000	3,853	163	21	49	162	128	66	655	632	18,810	18
6	Cayenne	5,268	68	0	195	710	0	56	43	5	603	1,123	53	0	18	19	164	1	60	111	8,497	16
7	Cxf	8,513	510	34	518	1,759	2	511	188	37	2,384	1,934	133	7	8	83	463	51	517	426	18,078	19
8	DBeam	118	4	0	0	22	0	5	0	1	19	8	0	0	0	0	0	0	2	10	189	9
9	dble	566	20	0	45	69	2	5	2	3	47	218	0	4	1	0	15	1	29	9	1,036	16
10	etcd-java	44	13	0	5	8	1	7	0	1	9	12	0	9	0	0	0	16	1	5	131	13
11	fb-jb-sdk	126	5	0	0	21	0	0	0	0	66	17	0	1	0	0	1	0	21	14	272	9
12	gctoolkit	207	19	0	22	44	0	8	0	19	22	96	0	0	0	0	5	5	5	7	459	12
13	Guice	364	16	0	29	19	0	50	6	0	119	64	0	0	1	0	8	0	23	30	729	12
14	Hive	7,888	938	16	706	1,030	2	353	149	30	872	2,487	58	120	63	55	654	31	658	559	16,669	19
15	jcef	3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	5	3
16	jfnr	4	2	1	5	3	0	17	1	0	3	0	0	12	18	0	0	0	0	3	69	11
17	Joda Time	9,080	56	63	393	811	0	594	122	3	370	2,579	0	4	186	0	353	0	23	89	14,726	15
18	JSqlParser	807	30	16	43	87	0	8	1	6	72	184	0	5	0	0	102	0	25	20	1,406	14
19	kapua	824	1,113	6	291	1,244	0	466	795	5	1,045	225	1	2	2	0	459	0	68	52	6,598	16
20	Neptune Export	433	9	3	0	155	0	24	15	2	109	46	2	0	0	2	32	0	25	7	864	14
21	Wicket	4,321	160	7	378	825	0	120	37	7	703	657	17	13	11	17	417	0	194	160	8,044	17
22	Zookeeper	1,502	350	0	148	346	0	131	19	13	315	466	59	1	1	49	15	42	139	104	3,700	17
	Total (instances)	**56,286**	**6,377**	**228**	**4,702**	**9,710**	**11**	**3,769**	**1,724**	**426**	**9,168**	**16,649**	**700**	**208**	**408**	**594**	**3,036**	**298**	**2,885**	**2,981**	**120,160**	**19**

most (t_j) with other test smells were *Assertion Roulette*, *Eager Test*, *Lazy Test*, *Magic Number Test* and *Verbose Test*.

We evaluate and classify the co-occurrences of test smells into three categories, as described in Section 4. Each presented co-occurrence is accompanied by a colored label indicating its respective classification.

Resource Optimism & Mystery Guest (99%). Structural This test smells are relate to external resources, such as databases, files, and other components. The main difference between them is that *Mystery Guest* occurs whenever these external resources are present in the tests, while *Resource Optimism* manifests itself when these resources are used optimistically, without prior verification of their existence. Hence, the high co-occurrence between these test smells at the class level can be explained by structural factors and standard practices in test development, because the co-occurrence between *Mystery Guest* and *Resource Optimism* is 91%. The strong relationship between these smells suggests that in many situations, test cases that depend on external resources also tend to assume their availability without adequate verification, which reinforces the connection between the two.

Redundant Assertion & Lazy Test (92%). Statistical Before analyzing the co-occurrence between any test smells at the class level, it is essential to understand each one individually. The *Redundant Assertion* test smell occurs when test methods contain redundant assertions, that is, verifications that are always true or always false, usually because the expected and actual values are the same. The *Lazy Test* test smell is characterized by the existence of multiple test methods within the same class that invoke and test the same production method.

Other test smells & Assertion Roulette (85%). Structural Since the *Assertion Roulette* test smell presented a high number of instances compared to the others, we will evaluate it considering an average among the other test smells. That is, the existence of other types of test smells (t_i) influenced the *Assertion Roulette* test smell (t_j) on average 85% of the time. At the class level, practically all test smells occur with the *Assertion Roulette* test smell in at least 64%, corroborating findings from other studies in the literature. This occurs because *Assertion Roulette* is directly associated with the most minor test structure: assertions. In addition, as it is the most frequent test smell, it is understandable that it has a high relationship with the others. In classes where there is some type of test smell, *Assertion Roulette* is usually present, the other types of test smells occur simultaneously with *Assertion Roulette* in 85% of the classes, on average. The test smells with the highest co-occurrence with *Assertion Roulette* were *Magic Number Test*, *Duplicate Assert* and *Ignored Test*, all with 94%. On the other hand, the

Table 4: Test Smells co-occurrences in Test classes.

	AR	CTL	CI	DA	EgT	ET	ECT	GF	IT	LT	MNT	MG	PS	RA	RO	SE	ST	UT	VT
AR	-	0.27	0.07	0.32	0.59	0.00	0.26	0.14	0.06	0.62	0.48	0.07	0.02	0.03	0.06	0.15	0.03	0.18	0.36
CTL	0.74	-	0.06	0.39	0.67	0.00	0.41	0.22	0.12	0.70	0.54	0.11	0.04	0.04	0.10	0.17	0.07	0.30	0.54
CI	0.82	0.30	-	0.36	0.57	0.00	0.38	0.25	0.07	0.56	0.52	0.08	0.03	0.10	0.07	0.29	0.06	0.22	0.36
DA	0.93	0.43	0.08	-	0.69	0.00	0.43	0.20	0.09	0.71	0.67	0.09	0.03	0.06	0.09	0.21	0.07	0.23	0.66
EgT	0.83	0.34	0.06	0.33	-	0.00	0.28	0.17	0.07	0.75	0.46	0.07	0.02	0.04	0.06	0.16	0.04	0.20	0.36
ET	0.78	0.22	0.00	0.44	0.33	-	0.33	0.11	0.11	0.56	0.67	0.11	0.33	0.00	0.11	0.22	0.11	0.22	0.44
ECT	0.83	0.48	0.10	0.47	0.65	0.00	-	0.22	0.12	0.74	0.58	0.12	0.03	0.08	0.11	0.21	0.08	0.30	0.58
GF	0.68	0.40	0.10	0.34	0.60	0.00	0.34	-	0.07	0.76	0.50	0.07	0.02	0.06	0.07	0.20	0.05	0.24	0.37
IT	0.94	0.63	0.08	0.46	0.67	0.01	0.52	0.20	-	0.70	0.59	0.17	0.07	0.06	0.16	0.16	0.09	0.74	0.60
LT	0.83	0.34	0.06	0.32	0.72	0.00	0.31	0.21	0.06	-	0.48	0.07	0.02	0.05	0.07	0.17	0.04	0.22	0.35
MNT	0.93	0.39	0.08	0.44	0.63	0.00	0.35	0.19	0.08	0.69	-	0.08	0.02	0.05	0.08	0.17	0.05	0.20	0.48
MG	0.90	0.52	0.08	0.41	0.63	0.00	0.49	0.19	0.15	0.71	0.58	-	0.02	0.03	0.91	0.15	0.07	0.32	0.58
PS	0.86	0.69	0.12	0.44	0.68	0.05	0.49	0.24	0.22	0.73	0.61	0.08	-	0.10	0.08	0.19	0.10	0.36	0.71
RA	0.92	0.40	0.21	0.56	0.76	0.00	0.64	0.32	0.10	0.92	0.68	0.06	0.06	-	0.06	0.39	0.04	0.31	0.62
RO	0.90	0.53	0.08	0.42	0.64	0.01	0.49	0.20	0.16	0.74	0.59	0.99	0.03	0.03	-	0.16	0.07	0.35	0.60
SE	0.87	0.36	0.14	0.43	0.67	0.00	0.38	0.23	0.07	0.75	0.51	0.07	0.02	0.09	0.06	-	0.03	0.20	0.43
ST	0.92	0.71	0.12	0.60	0.77	0.01	0.64	0.27	0.17	0.72	0.75	0.13	0.06	0.04	0.13	0.13	-	0.30	0.75
UT	0.64	0.39	0.06	0.28	0.51	0.00	0.33	0.17	0.18	0.59	0.37	0.09	0.03	0.04	0.09	0.12	0.04	-	0.43
VT	0.90	0.49	0.07	0.56	0.64	0.00	0.44	0.19	0.10	0.65	0.61	0.11	0.04	0.06	0.10	0.18	0.07	0.30	-

test smell with the lowest co-occurrence with *Assertion Roulette* was *Unknown Test*, with 64%, which can be justified by the absence of assertions in some test class methods. Like *Assertion Roulette*, *Redundant Assertion* presented a high frequency in our dataset, being the second most recurrent test smell. Although there is no direct structural correlation between *Redundant Assertion* and *Lazy Test*, the high co-occurrence between them (92%) can be attributed to statistical factors, especially the high frequency of *Lazy Test* in the database. On the other hand, when we analyze the inverse relationship (the co-occurrence of *Lazy Test* with *Redundant Assertion*) we observe a significantly lower value, around 5%. This indicates that the presence of *Lazy Test* does not necessarily imply the occurrence of *Redundant Assertion*, reinforcing the idea that the co-occurrence between these test smells is not due to a direct structural connection, but instead to the high isolated incidence of *Lazy Test*. *Redundant Assertion* was identified in 105 test classes, while *Lazy Test* was present in practically all (2,111 classes). Thus, most of the classes that presented *Redundant Assertion* also contained *Lazy Test*, which explains the strong co-occurrence observed in the analyzed database.

Sleepy Test & Eager Test (77%). Vicious A *Sleepy Test* occurs when a test method pauses its execution for a specific time to simulate an external event, usually using commands such as `Thread.sleep()`. Considering that the *Eager Test* and *Sleepy Test* address distinct aspects of the test structure, we infer no direct structural relationship between them. Furthermore, the inverse co-occurrence between these test smells is low, with *Eager Test* and *Sleepy Test* occurring together in only 4% of cases. However, there may be an indirect correlation in scenarios where the *Eager Test* verifies production methods that depend on asynchronous operations. In such cases, if such operations require explicit waits, such as using `Thread.sleep()`, a *Sleepy Test* may arise as a consequence in the same class. However, it is important to note that the presence of one of these test smells does not automatically imply the occurrence of the other.

Redundant Assertion & Eager Test (76%). Statistical The *Eager Test* test smell occurs when a single test method verifies multiple production methods, as opposed to the *Lazy Test*, which tests the same production method on several different test methods. Similar to the co-occurrence between the *Redundant Assertion* and the *Lazy Test*, we believe that the relationship between the *Eager Test* and the *Redundant Assertion* at the class level is predominantly statistical. This is evidenced by the fact that the inverse co-occurrence (*Eager Test* & *Redundant Assertion*) is only 4%, indicating that there is no strong structural correlation between these test smells.

General Fixture & Lazy Test (76%). Statistical The *General Fixture* test smell occurs when a test class uses generic fixtures, that is, fixtures that are not necessary for all test methods. On the other hand, the *Lazy Test* arises for distinct structural reasons. Therefore, there is no evidence of a direct structural relationship between these two test smells at the class level. However, it is important to highlight that there may be an indirect correlation in scenarios where a set of test methods uses a single set of fixtures and, as a result, ends up repeatedly invoking the same production method, characterizing *Lazy Test*. Although this correlation is possible, it is not guaranteed since the co-occurrence between the *Lazy Test* and the *General Fixture* is relatively low, with only 21% occurrence.

Sleepy Test & Verbose Test (75%). Vicious The *Verbose Test* is a test smell that occurs when there are too many instructions in a test method; the threshold used by the —- tool is 30 lines. Regarding the structure of both test smells, we do not see any direct similarity with the structure. However, it is worth noting that it is typical for test methods that have *Sleepy Test* also to be extensive and tend to be robust in terms of the number of instructions. This is because commands such as `Thread.sleep()` and other explicit waits are usually associated with tests that involve asynchronous events, such as I/O operations, network communication, concurrent processing or integration with external systems. These types of tests usually require more setups, manipulations and checks, which can increase the total size of the test method. In addition, to ensure that an event occurred successfully after the wait, the tests can include extra checks, such as polling, multiple assertions and intermediate logs. However, the reverse is not true - the co-occurrence between the *Verbose Test* and the *Sleepy Test* is 7%. This is because a test method can be extended for N reasons, including asynchronous waits.

Eager Test & Lazy Test (75%). Vicious In principle, the *Eager Test* and *Lazy Test* represent opposite approaches to test design. However, when one of these practices is used, the other is usually also introduced. This is because both test smells focus on a single problem: the lack of isolated tests for each production method, that

is, the absence of a design principle where each production method is tested independently within a single test case. In these cases, test methods that verify multiple production methods (*Eager Test*) and methods that excessively test a single production method (*Lazy Test*) can coexist. The reverse co-occurrence (*Lazy Test* and *Eager Test*) is also true (72%). Therefore, the common factor between these test smells is the lack of standardization in test writing, which contributes to the simultaneous presence of these test smells.

Sensitive Equality & Lazy Test (75%). Statistical The *Sensitive Equality* test smell occurs when the `toString()` method is used inside a test method. When comparing the two types of test smells, we see that this method-level relationship is likely to be more statistical than structural, resulting from specific implementation practices. This can be seen more as a reflection of how the tests are structured, rather than an intrinsic relationship between the test smells. A likely assumption is that there may be test methods where values are checked by invoking `toString()` and coincidentally are also called by more than one test method, resulting in the co-occurrence of the *Sensitive Equality* & *Lazy Test* test smells.

Sleepy Test & Magic Number Test (75%). Structural The *Magic Number Test* test smell occurs due to the use of literal numbers (also known as "magic numbers") directly in the test code, without an adequate explanation or definition, which can harm the readability and maintenance of the tests. Although these test smells have distinct causes and detections, they can occur together in some situations. For example, a test that uses magic numbers to define waiting times (such as pauses or delays in the test) may characterize both the *Sleepy Test* and the *Magic Number Test*. However, this does not mean that these test smells are always related. The high correlation is justified by using magic numeric values in contexts where pauses are implemented.

Ignored Test & Unknown Test (74%). Vicious Generally, incomplete tests or tests that still need to be improved are marked with the `@Ignore` annotation, with the intention of adding assertions and making them functional in the future. This practice may explain the co-occurrence of *Ignored Test* and *Unknown Test*, since both reflect early-stage testing.

Resource Optimism & Lazy Test (74%). Statistical Although there is no direct justification for this co-occurrence, it may exist when a production method that depends on external resources (such as databases, files, or remote services) is repeatedly tested by different test methods. The repetition of these calls in different test methods may cause *Resource Optimism* and *Lazy Test*.

Exception Handling & Lazy Test (74%). Statistical No direct justification was identified for this co-occurrence. However, it is possible that production methods that throw exceptions are being repeatedly tested by several different test methods, improperly causing these two test smells.

***Print Statement* & *Lazy Test* (73%).** Statistical Although there is no direct justification, we believe that developers may have the practice of inserting print commands (`System.out.print`) for test debugging purposes. If these methods are tested repeatedly in different test methods, without proper cleaning afterwards, this co-occurrence may occur.

***Sleepy Test* & *Lazy Test* (72%).** Statistical There is also no direct explanation for this co-occurrence. However, it is possible that envious methods are making asynchronous calls repeatedly, which may coincide with *Lazy Test*.

As for the type, we classified eight co-occurrences as statistical, four as vicious or indirect structural and three as structural.

5.2 Co-occurrence in test methods

Table 5 presents the co-occurrences of test smells at the test method level. Next, the 15 most frequent co-occurrences are described and analyzed, in decreasing order of occurrence.

At the test method level, co-occurrences did not show as high frequencies as at the class level. This is because the scope of a test method is more restricted, which allows observing co-occurrences that make sense from a semantic point of view, and not just statistical associations.

Next, we will detail the co-occurrences identified at the test method level. For the types of co-occurrences that have already been analyzed at the class level, only the frequency will be presented. Thus, this section focuses on co-occurrences not previously explored.

Resource Optimism & Mystery Guest (98%). Structural See co-occurrence in the test class.

Sleepy Test & Eager Test (72%). Vicious See co-occurrence in the test class.

Redundant Assertion & Eager Test (57%). Statistical See co-occurrence in the test class.

Duplicate Assert & Eager Test (56%). Vicious The co-occurrence of *Duplicate Assert* and *Eager Test* smells is frequent in scenarios where there is a violation of fundamental test design principles: Test a single behavior per test method and division of responsibilities per test method. The *Duplicate Assert* test smell occurs when two or more assertions are repeated with identical parameters. Generally, test engineers who do not respect the use of a single test method for a production method possibly check more than one production method in a test method (*Eager Test*). Hence, a good practice principle, when broken, can introduce different types of test smells.

Print Statement & Eager Test (56%). Vicious The relationship between these two test smells can be observed in scenarios where a test method, when verifying multiple methods of the production class (*Eager Test*), demands debugging actions by the developer. In these situations, it is common to use commands such as `System.out.print` inappropriately to track execution behavior. As a result, the test method ends up presenting the *Eager Test* and *Print Statement* smells simultaneously.

Verbose Test & Eager Test (54%). Vicious A test considered eager usually wants to validate different production methods in a single test method. This type of poor choice usually leaves the test method extensive, consequently implying in *Verbose Test*.

Other test smells & Assertion Roulette (53%). Structural *Assertion Roulette* test smell coexists with virtually all other method-level test smells, with the exception of the *Constructor Initialization*, *Empty Test*, *General Fixture*, and *Unknown Test* test smells. *Constructor Initialization* refers to the test setup method and therefore has no assertions; *Empty Test* contains no executable instructions, which prevents it from containing assertions. The null co-occurrence between *General Fixture* and *Assertion Roulette* indicates that, although

the setup method does not directly contain assertions, a test method that uses the generalized fixture may present the *Assertion Roulette* smell. Similarly, the lack of co-occurrence between *Lazy Test* and *Assertion Roulette* occurs because the tool output groups all the "lazy" methods, making it difficult to map directly to the lines containing the *Assertion Roulette* assertions.

Sleepy Test & Verbose Test (53%). Vicious See co-occurrence in the test class.

Sleepy Test & Exception Handling (53%). Vicious As mentioned above, test engineers often use `Thread.sleep()` (*Sleep Test*) to wait for certain operations to be executed, simulating asynchronous behaviors or behaviors that depend on external events. In these scenarios, it is common for the use of `Thread.sleep()` to be accompanied by the introduction of try-catch blocks, with the aim of catching and handling exceptions thrown by production class methods during or after the forced pause in execution.

Sleepy Test & Conditional Test Logic (53%). Vicious This co-occurrence can occur in scenarios where the test method depends on dynamic conditions to decide whether to wait (`Thread.sleep()`) or continue execution. In these cases, conditional structures such as if or switch are used to check the state of variables before applying the pause in execution, in order to simulate response times or external events.

Print Statement & Verbose Test (53%). Vicious Long test methods tend to be difficult to understand and maintain. For this reason, it is common for developers to resort to console output commands, such as `System.out.print`, to aid in debugging. Therefore, the cause of this co-occurrence is similar to the co-occurrence of the *Print Statement & Eager Test* test smells, as both co-occurrences can arise due to the need to debug poorly structured tests.

Conditional Test Logic & Eager Test (52%). Vicious The co-occurrence of *Conditional Test Logic* and *Eager Test* can arise when a test method is designed to encompass multiple test behaviors or scenarios. The use of conditional logic to control the flow of test execution can, in turn, lead to the invocation of multiple production methods within the same test method, characterizing *Eager Test*. This occurs when, in each conditional path, different aspects of the production class are selected.

Sensitive Equality & Eager Test (52%). Statistical The *Sensitive Equality* and *Eager Test* test smells can co-occur when a test method has calls to different methods of the same production class, including assertions of the textual representation of the object (`toString()`). The presence of *Eager Test* characterizes an increase in the complexity, while *Sensitive Equality* contributes to the fragility of the test, since any change in `toString()` may affect the test behavior.

Magic Number Test & Eager Test (52%). Structural When a test method is characterized as *Eager Test* and verifies multiple production methods of a given class (especially in cases where these methods perform operations with parameters, mathematical calculations or transformations of numeric data), it is common for the body of the test method to contain executable instructions or assertions with numeric literals (*Magic Number Test*) used directly, without the use of descriptive names.

Print Statement & Conditional Test Logic (51%). Vicious The use of conditional structures (*Conditional Test Logic*) increases the complexity of test methods, making them difficult to understand, maintain and evolve. Similar to the co-occurrences between the *Print Statement & Eager Test* and *Print Statement & Verbose Test* test smells, the presence of commands such as `System.out.print` with *Conditional Test Logic* is usually associated with the need to debug methods with complex logic. In this context, the introduction of prints serves as an auxiliary resource to track the behavior of the test in situations that are difficult to understand.

As for the type, we classified three co-occurrences as structural, two as statistical, and ten as vicious or indirect structural.

5.3 Fixing of test smells co-occurrence

To complement our RQ, we conducted additional investigations to propose refactoring strategies targeting test smell co-occurrences classified as *Structural* or *Vicious*. To this end, we used both the technical knowledge accumulated by researchers in the field and the definitions consolidated in the literature on test smells. The suggested approaches are based on principles of good software testing practices to promote greater maintainability and readability of tests.

Table 6 presents 16 co-occurrences between test smells, along with their respective correction strategies. The first column lists the refactorings (#), while the second correspond to the test smells involved in the co-occurrence (t_i and t_j, respectively). The third and most relevant column proposes the recommended correction strategy for each co-occurrence.

The categorization of test smell co-occurrences can directly influence the decision of whether or not to apply a joint refactoring. In particular, co-occurrences classified as statistical should be analyzed on a case-by-case basis, since their relevance strongly depends on the context in which they occur. For this reason, we propose refactorings only for co-occurrences classified as structural and vicious. We also emphasize that the refactoring suggestions presented here are not universal and may not be applicable to all cases.

By following the best practice where each test method should only check a single method of the production class, preferably using a single assertion per method, it is likely that the test class will not present test smells such as *Assertion Roulette*, *Duplicate Assert*, *Eager Test*, *Lazy Test*, *Unknown Test*, and *Empty Test*. This type of practice not only serves to refactor but also prevents new test smells from being introduced into the method and test class.

In addition, we recommend avoiding dependencies on external resources, such as database, network, or file access, in order to minimize smells such as *Mystery Guest*, *Resource Optimism*, *Verbose Test*, *Sleepy Test*, *Magic Number Test*, and *Exception Handling*. Finally, it is important to ensure that all test methods are properly implemented and functional, in order to avoid smells such as *Ignored Test*, *Unknown Test*, and *Empty Test*.

To illustrate the suggested refactorings, we manually applied some refactorings to real examples extracted from projects available in *Artifact Availability* section.

6 Threats Validity

Internal Validity. Since this study focuses on the co-occurrence of test smells, other software analyses were not proven. The dataset selection was restricted to the requirements of the imposed detection tool, JNose Test, which imposes restrictions related to

Table 5: Test Smells co-occurrences in Test methods.

	AR	CTL	CI	DA	EgT	ET	EH	GF	IT	LT	MNT	MG	PS	RA	RO	SE	ST	UT	VT
AR	-	0.12	0.00	0.14	0.49	0.00	0.13	0.00	0.02	0.00	0.31	0.03	0.00	0.01	0.03	0.08	0.01	0.00	0.16
CTL	0.58	-	0.00	0.15	0.52	0.00	0.23	0.00	0.05	0.00	0.33	0.06	0.02	0.01	0.05	0.06	0.04	0.11	0.36
CI	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DA	0.84	0.18	0.00	-	0.56	0.00	0.24	0.00	0.03	0.00	0.46	0.04	0.00	0.01	0.04	0.08	0.02	0.01	0.44
EgT	0.75	0.16	0.00	0.14	-	0.00	0.14	0.00	0.02	0.00	0.28	0.03	0.01	0.01	0.03	0.08	0.02	0.06	0.17
ET	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
EH	0.65	0.24	0.00	0.20	0.46	0.00	-	0.00	0.02	0.00	0.29	0.07	0.01	0.02	0.07	0.09	0.04	0.04	0.32
GF	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.04	0.00	0.00	0.04	0.00	0.00	0.00	0.00
IT	0.64	0.39	0.00	0.16	0.45	0.00	0.18	0.00	-	0.00	0.25	0.05	0.04	0.00	0.04	0.04	0.01	0.36	0.27
LT	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MNT	0.89	0.19	0.00	0.21	0.52	0.00	0.16	0.00	0.02	0.00	-	0.04	0.01	0.01	0.03	0.06	0.02	0.00	0.27
MG	0.68	0.26	0.00	0.15	0.45	0.00	0.30	0.04	0.03	0.00	0.28	-	0.00	0.01	0.83	0.04	0.03	0.07	0.28
PS	0.72	0.51	0.00	0.12	0.56	0.00	0.27	0.00	0.16	0.00	0.36	0.03	-	0.06	0.03	0.08	0.06	0.13	0.53
RA	0.81	0.12	0.00	0.16	0.57	0.00	0.31	0.00	0.00	0.00	0.22	0.03	0.03	-	0.03	0.02	0.00	0.00	0.18
RO	0.65	0.27	0.00	0.15	0.46	0.00	0.32	0.04	0.03	0.00	0.27	0.98	0.01	0.01	-	0.04	0.03	0.08	0.29
SE	0.81	0.11	0.00	0.13	0.52	0.00	0.17	0.00	0.01	0.00	0.19	0.02	0.01	0.00	0.02	-	0.00	0.01	0.14
ST	0.71	0.53	0.00	0.24	0.72	0.00	0.53	0.00	0.02	0.00	0.47	0.09	0.03	0.00	0.08	0.03	-	0.10	0.53
UT	0.03	0.12	0.00	0.01	0.19	0.00	0.04	0.00	0.05	0.00	0.01	0.02	0.00	0.00	0.02	0.00	0.01	-	0.07
VT	0.79	0.37	0.00	0.36	0.54	0.00	0.31	0.00	0.04	0.00	0.48	0.07	0.02	0.01	0.06	0.07	0.04	0.07	-

Table 6: Refactoring of test smells co-occurrence.

#	Test smells (t_i and t_j)	Suggested refactoring
1	RO and MG **Structural**	Replace external dependencies with mocks/fakes, and consequently it will not be necessary to check for its existence.
2	Other types and AR **Structural**	First isolate assertions for each test method. Each test method should only test its respective production method. If only the AR is removed, refactor the other type of test smell next.
3	ST and EgT **Vicious**	Refactor EgT by creating test methods for each production method. If wait operations are still required, replace external dependencies with mocks or fakes.
4	ST and VT **Vicious**	Check for external dependencies (MG) and replace them with mocks or fakes. If still verbose, split the test or extract setup code.
5	EgT and LT **Vicious**	First isolate assertions for each test method. Each test method should only test its respective production method. This will remove both test smells.
6	ST and MNT **Structural**	If the magic number is in Thread.sleep(), and sleep is removed from the method, both test smells will be refactored.
7	IT and UT **Vicious**	Remove all non-functional test methods, i.e. methods that do not have executable statements, assertions, are commented out or ignored (@Ignore in JUnit 4 or @Disabled in JUnit 5). If the method needs to be kept, rewrite it by removing the tag to be ignored and add valid executable statements for the respective method.
8	DA and EgT **Vicious**	Split the test method so that each production method has its own corresponding test method. Adopt parameterization to avoid duplicated assertions.
9	PS and EgT **Vicious**	Split the test method so that each production method has its own corresponding test method. Remove print statements.
10	VT and EgT **Vicious**	Split the test method so that each production method has its own corresponding test method.
11	ST and EH **Vicious**	Replace external dependencies with mocks or fakes, and if needed, split the method to cover success/failure scenarios.
12	ST and CTL **Vicious**	Split the test to cover each conditional branch. If wait operations are still necessary, replace external dependencies with mocks or fakes.
13	PS and VT **Vicious**	Verify if the assertions in the test method checks only the respective production method. If not, split the test into smaller methods, each testing its own production method. If applicable, replace external dependencies with mocks or fakes, and extract the setup code.
14	CTL and EgT **Vicious**	Split the test to cover each conditional branch. Check if the new methods has EgT and if applicable, split it again until the test methods only test its respective production method.
15	MNT and EgT **Structural**	Split the test method so that each production method has its own corresponding test method, then create constants to replace literals and remove magic numbers.
16	PS and CTL **Vicious**	Split the test to cover each conditional branch and remove prints.

the programming language, the framework used, and the need to use Maven. Consequently, the dataset consisted of 22 open-source projects selected for convenience, with projects of varying sizes, representing different development realities. Even smaller projects, e.g., JFNR, are subject to the presence and co-occurrence of test smells. In addition, priority was given to projects already used in previous research and in different software domains, contributing to the results' *external validation*.

Construct Validity. The detection rules applied in the analysis of the projects were exclusively those provided by the chosen detection tool JNose Test, this tool was previously validated in the literature, ensuring a certain level of reliability in the automatic detection of test smells. However, the identification strategies embedded in the tool may pose a potential threat to the construct validity of this work since the definition of smells depends on these previously rules.

Conclusion Validity. In this study, we expanded the analysis by including a more significant number of test smell types and investigating correlations at both the class and method levels, broadening the investigation scope compared to previous studies.

7 Conclusions and Future Work

Refactoring multiple test smells can be challenging since strategies suggested in the literature, if misapplied, can, for example, introduce new test smells unduly.

For a software engineer to be able to correct multiple test smells in a project, it is necessary first to know the basic refactoring strategies proposed by Fowler [10], such as Extract, Move, and Rename. In addition, the professional needs to be familiar with the refactorings of individual instances of test smells proposed in the literature.

In this context, we suggested 16 refactoring strategies for 30 co-occurrence test smells. We selected a set of 22 open-source systems, identified test smells with the help of a tool, hanked the co-occurrences of test smells, and proposed refactoring strategies to eliminate them.

The results obtained can help developers improve the quality of their tests, ensuring better quality test code. However, we aim not to have these refactoring techniques applied manually since this is a costly task for the project. Our future work is to evaluate and implement these strategies in a tool. In addition, we will identify clusters of test smells, i.e., co-occurrence of three or more types of test smells.

ARTIFACT AVAILABILITY

The data package is publicly available at https://github.com/arieslab/santana-sast2025-artifact.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, Oldenburg, Germany, 181–190.

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

[3] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? an empirical study. *Empirical Software Engineering* 20 (2015), 1052–1094.

[4] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2015. How (much) do developers test?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2* (Florence, Italy) *(ICSE '15)*. IEEE Press, Florence, Italy, 559–562.

[5] Felipe Diniz Dallilo, Marcio Eduardo Delamaro, and Simone Senger Souza. 2024. A methodology to support the execution of proficiency tests for software quality assessment. In *Proceedings of the XXIII Brazilian Symposium on Software Quality (SBQS '24)*. ACM, New York, NY, USA, 49–59.

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

[7] Vinicius H. S. Durelli, Rafael S. Durelli, Simone S. Borges, Andre T. Endo, Marcelo M. Eler, Diego R. C. Dias, and Marcelo P. Guimarães. 2019. Machine Learning Applied to Software Testing: A Systematic Mapping Study. *IEEE Transactions on Reliability* 68, 3 (2019), 1189–1212.

[8] Mehdi Esnaashari and Amir Hossein Damia. 2021. Automation of software test data generation using genetic algorithm and reinforcement learning. *Expert Systems with Applications* 183 (2021), 115446.

[9] Francesca Arcelli Fontana and Stefano Spinelli. 2011. Impact of refactoring on quality code evaluation. In *Proceedings of the 4th Workshop on Refactoring Tools* (Waikiki, Honolulu, HI, USA) *(WRT '11)*. ACM, New York, NY, USA, 37–40.

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

[11] Vahid Garousi, Michael Felderer, Marco Kuhrmann, Kadir Herkiloğlu, and Sigrid Eldh. 2020. Exploring the industry's challenges in software testing: An empirical study. *Journal of Software: Evolution and Process* 32, 8 (2020), e2251.

[12] Vahid Garousi, Baris Kucuk, and Michael Felderer. 2019. What We Know About Smells in Software Test Code. *IEEE Software* 36, 3 (2019), 61–73.

[13] Giovanni Grano, Cristian De Iaco, Fabio Palomba, and Harald C. Gall. 2020. Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Adelaide, SA, Australia, 336–347.

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

[15] Manju Khari and Prabhat Kumar. 2019. An extensive evaluation of search-based software testing: a review. *Soft Computing* 23, 6 (2019), 1933–1946.

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

[17] Gerard Meszaros. 2007. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, Upper Saddle River, NJ.

[18] Soukaina Najihi, Sakina Elhadi, Rachida Ait Abdelouahid, and Abdelaziz Marzak. 2022. Software Testing from an Agile and Traditional view. *Procedia Computer Science* 203 (2022), 775–782.

[19] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 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*. ACM, IEEE, Austin, TX, USA, 5–14.

[20] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. 2022. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering* 27, 7 (2022), 170.

[21] Anthony Peruma, Khalid Saeed Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2019. On the distribution of test smells in open source android applications: An exploratory study. In *29th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., USA, 193–202.

[22] Anthony Shehan Ayam Peruma. 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.

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

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

[25] Ashok Sivaji, Rosnisa Abdul Razak, Nur Faezah Mohamad, Nurshakirin Sazali, Afiqah Musa, Norzam Mohd Bajuri, Aslinda Md Hashim, Mohd Solehuddin Abdullah, Nur Diyana Joha, Nadia Ellyani Azis, Anjana Devi N Kuppusamy, Azlan Deniel, Ngip Khean Chuan, and Torkil Clemmensen. 2020. Software Testing Automation: A Comparative Study on Productivity Rate of Open Source Automated Software Testing Tools For Smart Manufacturing. In *2020 IEEE Conference on Open Systems (ICOS)*. IEEE, Kota Kinabalu, Malaysia, 7–12.

[26] Elvys Soares, Marcio Ribeiro, Guilherme Amaral, Rohit Gheyi, Leo Fernandes, Alessandro Garcia, Baldoino Fonseca, and Andre Santos. 2020. Refactoring Test Smells: A Perspective from Open-Source Developers. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing* (Natal, Brazil) *(SAST 20)*. ACM, New York, NY, USA, 50–59.

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

[28] Elvys Soares, Marcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and Andre Medeiros Santos. 2022. Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date. *IEEE Transactions on Software Engineering* 49, 3 (2022), 1–1.

[29] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the Relation of Test Smells to Software Code Quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Madrid, Spain, 1–12.

[30] Daniel Staegemann, Matthias Volk, Maneendra Perera, Christian Haertel, Matthias Pohl, Christian Daase, and Klaus Turowski. 2022. A literature review on the challenges of applying test-driven development in software engineering. *Complex Systems Informatics and Modeling Quarterly* 31 (2022), 18–28.

[31] Konstantinos Stroggylos and Diomidis Spinellis. 2007. Refactoring–Does It Improve Software Quality?. In *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*. IEEE, Minneapolis, MN, USA, 10–10.

[32] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. ACM, New York, NY, USA, 4–15.

[33] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. 2015. An empirical study of bugs in test code. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Bremen, Germany, 101–110.

[34] Pedro Henrique Dias Valle, Ricardo Ferreira Vilela, Guilherme Guerino, and Williamson Silva. 2023. Soft and Hard Skills of Software Testing Professionals: A Comprehensive Survey. In *Proceedings of the XXII Brazilian Symposium on Software Quality* (Brasília, Brazil) *(SBQS '23)*. ACM, New York, NY, USA, 90–99.

[35] Brent van Bladel and Serge Demeyer. 2021. A comparative study of test code clones and production code clones. *Journal of Systems and Software* 176 (2021), 110940.

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

[37] Tássio Virgínio, Luana Martins, Railana Santana, Adriana Cruz, Larissa Rocha, Heitor Costa, and Ivan Machado. 2021. On the test smells detection: an empirical study on the jnose test accuracy. *Journal of Software Engineering Research and Development* 9 (2021), 8–1.

[38] Tássio Virgínio, Railana Santana, Luana Almeida Martins, Larissa Rocha Soares, Heitor Costa, and Ivan Machado. 2019. On the Influence of Test Smells on Test Coverage. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering* (Salvador, Brazil) *(SBES 2019)*. ACM, New York, NY, USA, 467–471.

[39] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936.

[40] Yanming Yang, Xing Hu, Xin Xia, and Xiaohu Yang. 2024. The Lost World: Characterizing and Detecting Undiscovered Test Smells. *ACM Trans. Softw. Eng. Methodol.* 33, 3, Article 59 (March 2024), 32 pages.