UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ELVYS ALVES SOARES

A Multimethod Study of Test Smells:
Cataloging, Removal, and New Types

Recife

2023

ELVYS ALVES SOARES

A Multimethod Study of Test Smells:
Cataloging, Removal, and New Types

Work presented to the Programa de Pós-graduação em Ciência da Computação of the Centro de Informática da Universidade Federal de Pernambuco, as a partial requirement to obtain the degree of Doctor in Computer Science.

Concentration Area: Software Engineering and Programming Languages

Supervisor: André Luis de Medeiros Santos

Co-supervisor: Márcio de Medeiros Ribeiro

Recife

2023

Catalogação na fonte
Bibliotecária Nataly Soares Leite Moro, CRB4-1722

S676m Soares, Elvys Alves
A multimethod study of test smells: cataloging, removal, and new types / Elvys Alves Soares – 2023.
140 f.: il., fig., tab.

Orientador: André Luis de Medeiros Santos.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2023.
Inclui referências e apêndices.

1. Engenharia de software e Linguagens de programação. 2. Teste de software. 3. Test smells. I. Santos, André Luis de Medeiros (orientador). II. Título

005.1 CDD (23. ed.) UFPE - CCEN 2023 – 130

Elvys Alves Soares

"A Multimethod Study of Test Smells: Cataloging, Removal, and New Types"

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovado em: 31/05/2023.

You prepared me for the biggest challenges life can give. This one is for you too, mom, there where you are.

ACKNOWLEDGEMENTS

Such an alone endeavor, yet so many people to thank. The results of so many years of nonstop study and abdication could only be achieved with much dedication from the great team I was lucky to be involved with, which is why do not dare to write this thesis in the first person. Here, I reinforce my deepest thanks to all the contributors while take the chance to paraphrase our late and accurate King Pele: “*Success is no accident. It is hard work, perseverance, learning, studying, sacrifice, and most of all, love of what you are doing or learning to do.*”

ABSTRACT

Test smells are symptoms in the test code that indicate possible design or implementation problems. Their presence in automated test suites, along with their harmfulness, has already been demonstrated by previous research. Although test smells have been the subject of much gray and academic literature since their proposal in 2001, many questions regarding their adherence in the industry are yet to be clarified: concerning test smells — proposed by numerous studies and gray literature — no publicly available catalog aggregates them; considering the evolution of test frameworks and programming languages, there is no correspondence between the newly proposed features and their capability of refactoring or preventing test smells; finally, considering that test automation requires a significant initial investment not always available to software projects, little is known to the possibility of test smells' existence in manual test suites, as well as how to identify and remove them. This work presents a multimethod study aimed at fulfilling these knowledge gaps in the test smells area, which comprises surveying state of the art on test smells and refactoring actions, the use of manual and automatic analyses of open-source repositories, the conduction of surveys with software testing professionals, the study of new test framework features and the proposition of test smell refactoring actions, and the submission of contributions to active and popular open-source software projects. The results present: (i) a catalog that unifies 127 primary studies and 480 distinct test smells in a previously unseen effort; (ii) the confirmation that new test framework features can refactor and prevent test smells, where we propose and evaluate new refactorings based on 7 JUnit 5 features intended for 13 test smells; (iii) the proposition of a catalog containing 8 new test smells specific to manual test suites, their identification strategies based on natural language processing, and their frequency in important government, industry and open-source systems. The findings of this work give directions for further development in various fronts of the test smells study area.

Keywords: software engineering; software testing; test smells.

RESUMO

Test smells são sintomas no código de teste que indicam possíveis problemas de design ou implementação. Sua presença em conjuntos de testes automatizados, juntamente com sua nocividade, já foi demonstrada por pesquisas anteriores. Embora a área de test smells tenha sido objeto de muita literatura acadêmica e cinzenta desde sua proposta em 2001, muitas questões sobre a adesão dos test smells na indústria ainda precisam ser esclarecidas: com relação aos tipos de test smells — propostas por vários estudos e literatura cinzenta — nenhum catálogo publicamente disponível os agrega; considerando a evolução dos frameworks de teste e linguagens de programação, não há correspondência entre os novos recursos propostos e sua capacidade de refatoração ou prevenção de test smells; por fim, considerando que a automação de testes requer um investimento inicial significativo nem sempre disponível para projetos de software, pouco se sabe sobre a possibilidade da existência de test smells em suítes de testes manuais, bem como como identificá-los e removê-los. Este trabalho apresenta um estudo multimétodo que visa preencher essas lacunas de conhecimento na área de test smells, que compreende o levantamento do estado da arte sobre test smells e ações de refatoração, o uso de análises manuais e automáticas de repositórios de código aberto, a realização de pesquisas com profissionais de teste de software, o estudo de novos recursos de estrutura de teste, a proposição de ações de refatoração test smells e o envio de contribuições para projetos de software de código aberto ativos e populares. Os resultados apresentam: (i) um catálogo inédito que unifica 127 estudos primários e 480 test smells distintos; (ii) a confirmação de que novos recursos de frameworks de teste podem refatorar e prevenir test smells, onde propomos e avaliamos novas refatorações baseadas em 7 recursos JUnit 5 destinados a 13 test smells; (iii) a proposição de um catálogo contendo 8 novos cheiros de teste específicos para suítes de teste manual, suas estratégias de identificação baseadas no processamento de linguagem natural e sua frequência em importantes sistemas governamentais, industriais e de código aberto. As conclusões deste trabalho fornecem direções para um maior desenvolvimento em várias frentes da área de estudo de test smells.

Palavras-chave: engenharia de software; teste de software; test smells.

LIST OF FIGURES

LIST OF SOURCE CODES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

GQM	Goal Question Metric
MLM	Multivocal Literature Mapping
MLR	Multivocal Literature Review
NLP	Natural Language Processing
SLR	Sistematic Literature Review
SMS	Systematic Mapping Study

CONTENTS

1 INTRODUCTION

The automation of testing tasks is a well-known activity, and the development of test scripts is frequent in the software industry (DUSTIN; RASHKA; PAUL, 1999; KARHU et al., 2009). It is a consensus that automated test suites with high internal quality facilitate maintenance activities such as code understanding and regression testing. However, the development of test code is tedious, requires a significant initial investment, and may suffer from design issues just as source code does (BAVOTA et al., 2015).

Researchers coined the term *test smells* to indicate possible design problems in the test code as an analogy to the symptoms carried by poorly designed source code (*i.e.*, *code smells*) (DEURSEN et al., 2001). Such symptoms may lead tests to present erratic behavior (i.e., test flakiness, false positives, and false negatives), thus compromising software quality due to limited defect-catching capabilities. Although the concept of test smells is not recent, a handful of previous studies have demonstrated them to be frequent in practice, both in open-source and industry projects, as well as their negative impact on code maintenance and understanding activities (BAVOTA et al., 2015; ROMPAEY et al., 2007; PALOMBA et al., 2016; PERUMA et al., 2019).

For instance, Source Code 1 presents a test code snippet extracted from the TestContainers[1] project on GitHub. One can notice the assertion method — which represents the test itself — is inside a decision structure (line 6). According to Meszaros (2007), this is the *Conditional Test* smell, where branching the execution order may cause the test to finish without executing the intended assertion, possibly generating a false positive outcome.

Source Code 1 – The Conditional Test smell

```
1 @Test
2 void first_test() {
3   if (lastContainerId == null) {
4     lastContainerId = genericContainer.getContainerId();
5   } else {
6     assertNotEquals(lastContainerId, genericContainer.getContainerId());
7   }
8 }
```

Source: research data

[1] <https://git.io/JtRpX>

1.1 KNOWLEDGE GAPS

Since the proposal of test smells, much research has been dedicated to their detection (REICHHART; GÎRBA; DUCASSE, 2007; NöDLER; NEUKIRCHEN; GRABOWSKI, 2009; BAVOTA et al., 2012; GREILER; DEURSEN; STOREY, 2013; PALOMBA et al., 2014; BAVOTA et al., 2015), prevention (DESIKAN, 2006; MEYER, 2008; S.M.K; FAROOQ, 2010; GREILER et al., 2013), and correction (DEURSEN et al., 2001; MESZAROS, 2007; PERUMA et al., 2019; GREILER; DEURSEN; STOREY, 2013) with correlates in the gray literature (i.e., technical blogs and posts). However, although all these studies and gray literature had to define test smells types before proposing their detection, prevention, and correction, no study until today was dedicated to assembling a publicly available catalog of test smells. Such a lack of centralized information represents our first knowledge gap in the test smells area.

Concerning test smell correction, test code refactorings are the most commonly applied operation. It is important to emphasize that although both test and code smells may present correlation (TUFANO et al., 2016) and even share a few refactorings (MESZAROS, 2007), refactoring source code is generally different from refactoring test code (DEURSEN et al., 2001). The proposition of test code refactorings is generally described in conceptual operations, for instance, *"Setup External Resource"* (DEURSEN et al., 2001), leaving their implementation steps to vary depending on the programming language and test automation framework version. Despite generally keeping backward compatibility, test automation frameworks offer new and improved features in every new version, and the use of such new features may represent an optimization to implement such conceptual operations. In this context, the literature demonstrates that JUnit 4 is the most used test automation framework for open-source Java projects on GitHub (ZEROUALI; MENS, 2017), also stating that *"from all considered libraries, projects are very likely (97%) to use JUnit"*. Taking the JUnit framework evolution into consideration, another research found that, although a new framework version (JUnit 5) has been available since late 2017, the migration of GitHub projects to this specific version *"is done manually and slowly,"* and that *"test developers are sometimes unaware of the features provided by testing frameworks, and thus apply alternative sub-optimal solutions"* (KIM et al., 2021a). Thus, we identify our second knowledge gap concerning how developers use new test framework features and whether these features could represent solutions to refactor existing test code to remove test smells.

But test smells are not exclusive to automated tests. In fact, analyzing manual test descriptions written in natural language from the point of view of test smells is not necessarily a new idea. It is well known that manual tests are often of poor quality and written without software engineering best practices in mind (HAUPTMANN et al., 2013). Similarly to known issues with natural language requirements, the documentation of tests in natural language often results in test cases that are incomprehensible, ambiguous, and difficult to maintain, with problems such as translation and spelling errors, inconsistent wording, use of inconsistent vocabulary, different description styles for similar test procedures, or excessive use of abbreviations (JUHNKE; NIKIC; TICHY, 2021). Hauptmann et al. (2013) were the first to study such proposition and coined the term *natural language test smells* to represent the possible design problems existing in manual software testing that may prejudice comprehension, execution and maintenance activities in manual testing. However, after Hauptmann et al. (2013) publication, we noticed a research gap of almost ten years concerning natural language test smells which did not happen in the context of smells in automatic tests (GAROUSI; KÜÇÜK, 2018; PERUMA et al., 2019; PANICHELLA et al., 2022). Such absence motivates a handful of questions concerning the existence of additional natural language test smells, their frequency, the possible problems they indicate, and since they are written in natural language, if their identification can be made with Natural Language Processing (NLP) mechanisms. These questions, therefore, represent our third knowledge gap.

1.2 RESEARCH QUESTIONS

To address our identified knowledge gaps and focusing on — but not being restrict to — open-source software repositories as information source for this thesis, we propose a series of studies aimed both at answering the presented research questions and providing directions for future research, as follows:

- Regarding known test smells:
  - **RQ$_1$: "What are the test smells found in literature?"** We conduct a Multivocal Literature Review (MLR), which is a form of Sistematic Literature Review (SLR) that includes the gray literature (e.g., blog posts, videos and white papers) in addition to the published literature (e.g., journal and conference papers) (GAROUSI; FELDERER; MÄNTYLÄ, 2019). By combining both the formal and informal literature, a MLR has the advantage to provide us with a better understanding of the state-of-

the-art as well as the state-of-practice in this topic. Moreover, from the performed MLR, we derive an openly accessible and maintainable catalog of test smells and lessons learned when proposing new test smells.

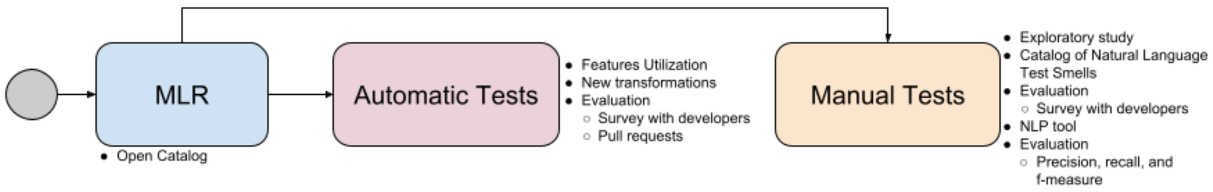
- Regarding test smell removal:
  - **RQ$_2$: "To what extent do projects use new test framework features?"** We execute an empirical study considering 485 popular Java open-source projects — which automate tests with JUnit — to identify how many of them use the latest library and how they use its features.
  - **RQ$_3$: "Can new test framework features help removing test smells?"** We carefully analyze the features introduced in JUnit 5 and identify that some might help removing test smells. In particular, we list 13 test smells, such as *Assertion Roulette*, *Test Code Duplication*, and *Conditional Test Logic*, that can be removed from test code using our transformations.
  - **RQ$_4$: "How do developers perceive our test code transformations to eliminate smells?"** We create an online survey where 212 developers from 39 countries could choose (and comment) between an original (smelly) and our transformed test code version and comment on their answers.
  - **RQ$_5$: "To what extent do open-source developers accept our transformations in their projects?"** We submit 38 Pull Requests, using our proposed transformations. Our contributions reached a 94% acceptance rate among respondents in this specific study.
- Regarding test smells in natural language tests
  - **RQ$_6$:"What already proposed natural language test smells can be observed?"** We conduct an exploratory study to analyze a statistically relevant sample of manual test descriptions of three systems from different domains. We identify the occurrence of two already proposed natural language test smells (*i.e., Conditional Test* and *Ambiguous Test*).
  - **RQ$_7$:"What new natural language test smells can be observed?"** The exploratory study enabled us to identify six new smells (*i.e., Unverified Action*, *Misplaced Precondition*, *Misplaced Verification*, *Misplaced Action*, *Eager Action*, and *Tacit Knowledge*) specific to manual tests in natural language.

- **RQ$_8$:"How frequent are these test smells?"** The exploratory study also enabled us to answer how frequent are both the already proposed and our newly proposed test smells in the analyzed systems.
- **RQ$_9$:"How software testing professionals evaluate our proposed smells?"** We conduct an empirical study using an online survey to evaluate our catalog of natural language test smells with 24 testing professionals. Our proposals had an average acceptance of 80.7% and the test professionals also contributed with additional concerns raised from poor test writing.
- **RQ$_{10}$:"How precise can the automated discovery of natural language test smells be when using NLP?"** We develop a NLP-based tool that reaches a precision of 92%, recall of 95%, and f-measure of 93.5%, indicating a suitable detection level for our proposals. Overall, the tool execution evidenced 13,169 test smell occurrences in the 2,007 tests of the analyzed systems.

1.3 STRUCTURE

This section demonstrates how the presented studies relate. The first study, presented in Chapter 3, serves as the foundation for all subsequent studies. It utilizes an MLR to organize the test smells area. Our remaining contributions are divided into two sub-areas. First, we show how our research eliminates test smells in automatic tests in Chapter 4. Second, in Chapter 5, we highlight our contributions to the discovery of natural language test smells in manual testing. To better understand the flow of our work, please refer to Figure 1.

Figure 1 – Thesis fluxogram



Source: research data

2 MOTIVATING EXAMPLES

This chapter presents examples that, together with the knowledge gaps and research questions, motivate the remainder of this thesis. We divide this chapter in three sections. The current knowledge fragmentation in the test smells area inspires a central reference point better detailed in Section 2.1. Also, the current test smell refactoring operations performed by project maintainers do not benefit from the capabilities provided by new framework features, as shown in Section 2.2. Finally, manual test descriptions written in natural language present more indications of problems than anticipated by current research, as shown in Section 2.3.

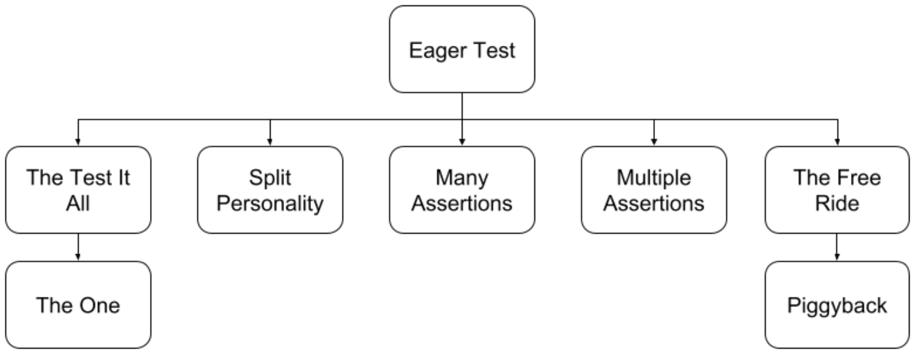
2.1 INFORMATION

To characterize the current state of fragmentation of test smell definitions and its implications, we will further detail the case of the *Eager Test* smell. Defined by Deursen et al. (2001) in the seminal study that initiated the test smells area as *"When a test method checks several methods of the object to be tested, it is hard to read and understand, and therefore more difficult to use as documentation,"* the *Eager Test* may appear under different names in formal and informal literature.

The study by Garousi and Küçük (2018) states that the *Eager Test* (DEURSEN et al., 2001) may appear in literature named as *The Test It All* (KUMMER, 2015) and *Split Personality* (KOSKELA, 2015), while Kummer (2015) adds *Many Assertions* (SCRUGGS, 2009), *Multiple Assertions* (SCHMETZER, 2005), and *The Free Ride* (A, 2014) to the list. However, other literature items affirm that these alternative names, in turn, also have their own alternative names: *The Test It All* may also be called *The One* (KUMMER, 2015) and *The Free Ride* may also be called *Piggyback* (GAROUSI; KÜÇÜK, 2018). Figure 2 presents the — possibly not final — *Eager Test* name tree. Even being one of the first proposed test smells, the *Eager Test* is not free from having many alternative — and inventive — names that make, at least, learning and researching on the topic less efficient.

Moreover, almost 50 formal and informal studies refer to the *Eager Test* directly, and at least a dozen other studies present details of the *Eager Test* with an alternative name. Since information when presenting a test smell is not standardized, someone interested in further details like causes and effects, frequency of occurrence, and code examples, would need help

Figure 2 – The *Eager Test* name tree



Source: research data

selecting sources for study. In such a case, an organized and centralized source of information that could give guidance on the available literature would improve efficiency in learning and research on this test smell. Chapter 3 presents our proposal to address this issue.

2.2 REFACTORINGS

Turning our attention to test smell refactorings, Source Code 2 presents our motivating example, committed in 04/2020 to GitHub's Jenkins[1] project, which had not migrated to JUnit 5 at the time. This example shows a unique test method with three distinct sections: lines 3–6, 8–11, and 13–16. Each section tests the same sequence of `getRemote()` and `getParent()` methods from the `FilePath` class, varying only the parameter values after its instantiation. The main fragility of this test design is related to the presence of code duplication, which negatively impacts maintainability.

Source Code 2 – Motivating example presenting the Duplicate Assert

```
 1  @Test
 2  public void getParent() {
 3    FilePath fp = new FilePath(null, "/abc/def");
 4    assertEquals("/abc", (fp = fp.getParent()).getRemote());
 5    assertEquals("/", (fp = fp.getParent()).getRemote());
 6    assertNull(fp.getParent());
 7
 8    fp = new FilePath(null, "abc/def\\ghi");
 9    assertEquals("abc/def", (fp = fp.getParent()).getRemote());
10    assertEquals("abc", (fp = fp.getParent()).getRemote());
11    assertNull(fp.getParent());
12
13    fp = new FilePath(null, "C:\\abc\\def");
14    assertEquals("C:\\abc", (fp = fp.getParent()).getRemote());
15    assertEquals("C:\\", (fp = fp.getParent()).getRemote());
16    assertNull(fp.getParent());
17  }
```

Source: research data

[1] <https://git.io/JuTCN>

Peruma et al. (2019) classify it as the Duplicate Assert test smell, defined as *"when a test method tests for the same condition multiple times within the same test method."* The proposed refactoring states that new test methods should be created when the test method needs to test the same condition multiple times with different values (PERUMA et al., 2019). The refactored version is presented, as proposed, in Source Code 3, where the original `getParent()` method is split into three different ones.

Source Code 3 – Refactoring the Duplicate Assert and causing Test Code Duplication

```java
@Test
public void getParent_relativePath() {
  FilePath fp = new FilePath(null, "/abc/def");
  assertEquals("/abc", (fp = fp.getParent()).getRemote());
  assertEquals("/", (fp = fp.getParent()).getRemote());
  assertNull(fp.getParent());
}

@Test
public void getParent_mixedPath() {
  fp = new FilePath(null, "abc/def\\ghi");
  assertEquals("abc/def", (fp = fp.getParent()).getRemote());
  assertEquals("abc", (fp = fp.getParent()).getRemote());
  assertNull(fp.getParent());
}

@Test
public void getParent_absolutePath() {
  fp = new FilePath(null, "C:\\abc\\def");
  assertEquals("C:\\abc", (fp = fp.getParent()).getRemote());
  assertEquals("C:\\", (fp = fp.getParent()).getRemote());
  assertNull(fp.getParent());
}
```

Source: research data

Splitting the original code improves the execution completeness, hence all tests will be executed. However, the code duplication is now found across different test methods and still negatively impacts the code maintainability. The result presents the Test Code Duplication test smell, defined by Deursen et al. (2001) as when the test code presents duplicated steps, either in the same test class or across test classes. Deursen et al. (2001) recommend applying the *Extract Method* (FOWLER, 2018) refactoring, and Meszaros (2007) gives further details on the creation of helper verification methods. The result of the *Extract Method* is shown in Source Code 4, which now centralizes the duplicated code in a parameterized helper method called by the test methods. The solution achieved by the *Extract Method*, in this case, solves the execution completeness and avoids code duplication. However, the test of many different conditions would generate a verbose test class that calls the helper method with different values.

Source Code 4 – Refactoring the Test Code Duplication

```java
@Test
public void getParent_relativePath() {
  getParent("/abc/def", "/abc", "/");
}

@Test
public void getParent_mixedPath() {
  getParent("abc/def\\ghi", "abc/def", "abc");
}

@Test
public void getParent_absolutePath() {
  getParent("C:\\abc\\def", "C:\\abc", "C:\\");
}

public void getParent(String path, String parent, String subParent) {
  fp = new FilePath(null, path);
  assertEquals(parent, (fp = fp.getParent()).getRemote());
  assertEquals(subParent, (fp = fp.getParent()).getRemote());
  assertNull(fp.getParent());
}
```

Source: research data

Available since JUnit 5.0.0 (09/2017), the `ParameterizedTest` annotation allows developers to test different values without the need to duplicate test steps, create helper methods, or a verbose test class. It is also possible to define a variety of argument sources through specific annotations to test for collections, as an example, without prejudice to execution completeness or maintainability. Source Code 5 presents the motivating example rewritten with JUnit 5 Parameterized Tests. Beyond maintaining advantages of Source Code 4, the Parameterized Tests feature brings additional benefits: less code is necessary, further improving its maintainability and extensibility.

Source Code 5 – Removing code duplication with JUnit 5 Parameterized Tests

```java
@ParameterizedTest
@CsvSource({
    "/abc/def,      /abc,       /",
    "abc/def\\ghi,  abc/def,    abc",
    "C:\\abc\\def,  C:\\abc,    C:\\"
})
void getParent(String path, String parent, String subParent) {
  FilePath fp = new FilePath(null, path);
  assertEquals(parent, (fp = fp.getParent()).getRemote());
  assertEquals(subParent, (fp = fp.getParent()).getRemote());
  assertNull(fp.getParent());
}
```

Source: research data

The possibility and benefits of using test parameterization show the importance of assessing (i) which new features or constructions are in use by developers, (ii) how these features could be used towards removing test smells, and (iii) whether developers find these propositions practical to their test automation activities. We address these questions in Chapter 4.

2.3 SMELLS IN MANUAL TESTS

In some cases, tight time and budget requirements make the initial investment for test automation unfeasible and lead to the choice of manual tests in the project. However, since they are usually written in natural language, these tests can also present several problems. As an example, Table 1 shows an excerpt from a Ubuntu Operational System (OS) manual test[2] that presents quality problems.

Table 1 – Steps of An Ubuntu OS test having conditions phrased in natural language

No	Action	Verification
1	Plug a USB device in and attempt to use it	The device is correctly recognized The software normally used with the device functions normally The device behaves as expected The USB device works in every port You are able to disconnect and re-connect the USB device correctly without errors
2	*If the device is a USB 3.0 storage device and you have a USB 3.0 port, transfer a large file between the two*	*The transfer is above USB 2.0 speed*
3	Repeat for each USB device you have	

Source: research data

In the test, the second action step presents two conditions, *"USB 3.0 storage device"* and *"USB 3.0 port,"* that must be met prior to performing the action *"transfer a large file"*. At this point, the test case contains a branch of the test flow formulated in natural language, which can lead to two problems: first, since different hardware can present different USB ports, it is unclear whether the condition for the branch is satisfied in the same way in each test run. This step description can lead to nondeterministic test runs and unmatched test results. Second, because of the condition expressed in natural language, it is not obvious for the tester how to report the test outcome should the condition not be met. This lack of clarity can lead to test cases that are difficult to understand or even misinterpreted, which negatively affects the test case executability and maintainability. We address this and other quality problems in manual test descriptions from the point of view of test smells in Chapter 5.

[2] [Online]. *"testcases\hardware\1476_USB Ports"* test, available: <https://git.launchpad.net/ubuntu-manual-tests>

3 FIRST THINGS FIRST: CATALOGING TEST SMELLS

Our first study addresses the concerns originated from the increasing popularity of test smells, followed by and the ever-rising proposition of new types both in formal and informal literature. As detailed in Section 2.1, the first concern is related to the many names a test smell may be given.

As a second concern, despite the large amount of academic and gray literature on test smells, only some studies surveyed such sources of information. Any literature may become unavailable. However, as gray literature is more prone to obsolescence (*e.g.,* discontinued websites or archived posts), this particular source of information is more endangered to be lost without being cataloged. More than 20% of the gray literature referenced in the most extensive study (GAROUSI; KÜÇÜK, 2018) is already unavailable.

The third concern is the lack of standardization on information across academic and gray literature. While some sources may present definition, code example, refactoring proposal, causes, effects, and other information, it is not a general practice to have all these description items when proposing a new test smell. Hence, the selection of good sources of information to study and research on test smells may be a challenging task which can be made simpler with some guidance on the available literature.

The lack of centralized information about known test smells may negatively impact software testing practices since no common ground has been established for learning, development, and research. Therefore, this chapter presents a systematic identification and characterization of test smells found in academic and gray literature, centralizing the collected data in a data set and a catalog, both openly accessible. For this purpose, we conduct a MLR (OGAWA; MALEN, 1991; PATTON, 1991), which is *"a form of SLR that includes the gray literature (*e.g., *blog posts, videos, and white papers), in addition to the published literature (e.g., journal and conference papers)"* (GAROUSI; FELDERER; MÄNTYLÄ, 2019) of test smells.

3.1 A MULTIVOCAL LITERATURE REVIEW

This section details the execution of our MLR. After the defined settings (Section 3.1.1) for the vocabulary used in the study, addressed research questions, search databases, keywords, inclusion, exclusion and quality criteria, we proceed with the selection of primary studies (Section 3.1.2). Then, details of the data extraction (Section 3.1.3), classification (Section 3.1.4), and correlation (Section 3.1.5) processes are presented together with their results. Finally, the threats to the validity of this study are presented (Section 3.1.7). For reproducibility and findings verification, the results are publicly available in a data set (SOARES et al., 2022b).

3.1.1 Protocol

As the intent of this study is to analyze the literature aiming to identify and characterize test code design and implementation issues with respect to their smell aspects, it is necessary to present the definitions concerning the used vocabulary, research question, databases, search string, selection and quality criteria. Concerning the vocabulary used in this study, we address formal and informal literature as follows:

- Peer-reviewed academic studies (*i.e.,* published journal papers and conference proceedings) as *formal studies*;
- Scholarly but non-peer-reviewed content (*i.e.,* bachelor's and master's theses and doctoral dissertations) and gray literature items (*i.e.,* books, pre-prints, e-prints, technical reports, lectures, data sets, videos, blogs, and web pages) as *informal studies*; and
- Formal and informal studies as *primary studies*.

In particular, this study aims to answer our first research question: **RQ$_1$: "What are the test smells found in literature?"** This question provides a basis for identifying and characterizing the test smells and building the catalog. The combination of formal and informal literature gives the MLR the advantage of providing a better understanding of the state-of-the-art and state-of-practice in this topic (GAROUSI; FELDERER; MÄNTYLÄ, 2019).

Concerning the selected databases, while traditional SLR focus on formal studies, MLR include informal items of information. Informal studies are relevant for a comprehension of the state-of-the-practice. This study used academic databases, namely ACM Digital Library, IEEEXplore, and Google Scholar for formal literature. For informal studies, Google and Bing search engines were used.

In the planning of the MLR, a preliminary search process was performed before finalizing the search scope and keywords. The main goal was to test and evaluate distinct search strings to find a set of relevant literature items that could be used as a sanity check when conducting the actual search for the studies. This activity used keywords such as "test smell" and "anti-pattern", understanding test smells as *"a type of anti-pattern"* (GAROUSI; KÜÇÜK, 2018). The initial results from the formal and informal databases were carefully analyzed by looking for keywords and synonyms that could be useful. The searched terms "test smell" and "anti-pattern" were most frequently employed in primary studies and played an essential role in the searches. In addition, "poor quality" was observed as a frequently associated term.

Since the aim was to identify and characterize as many test smells as possible, a string focusing on the population (test code) and the intervention (test smells, anti-patterns, and symptoms) was built. Thus, the following search string was used: **"test code" AND ("test* smell*" OR anti-pattern* OR "poor quality")**. Note that the symbol "*" matches lexically related terms such as plurals. The exact string was used in formal and informal databases, and no filter was applied to the content. Thus, the search considered the whole text of the primary studies.

The inclusion criteria establish the properties a primary study must accomplish to be selected, and the exclusion criteria define its ineligibility after selection. Equation 3.1 presents the selection of primary studies considering the statements I_1 to I_4 as *inclusion criteria*. As *exclusion criteria*, primary studies not fully written in English were removed.

$$I_1 \wedge (I_2 \vee I_3 \vee I_4) \tag{3.1}$$

- I_1: Full-text accessible;
- I_2: It discusses poor test quality aspects;
- I_3: It identifies one or more test smells;
- I_4: It defines or provides a reference in the literature that defines one or more test smells.

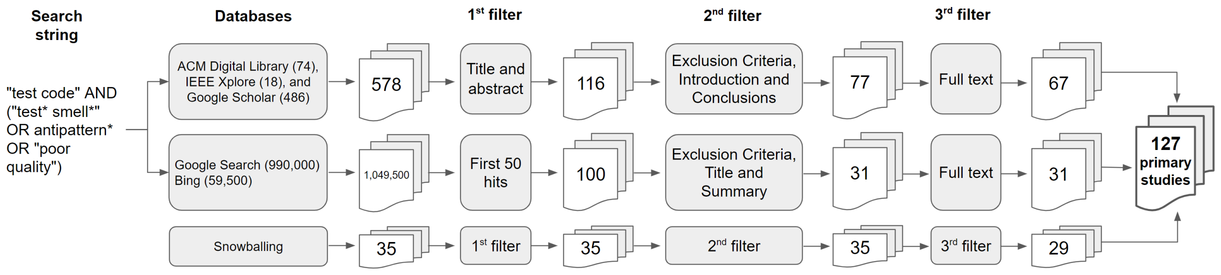
Apart from the selection criteria and in a non-excluding step, a quality assessment during the selection of primary studies was also performed. This assessment had a dual purpose: (i) measuring the completeness of each description given by a primary study, and (ii) gathering indications of where to find fine-grained information and provide literature guidance in the intended catalog. The quality criteria were formulated as *yes* or *no* items to ease the analysis. For every test smell in every primary study, the quality criteria indicated the presence of the following:

- Q_1: Example of the contextualized test smell;
- Q_2: Discussion of causes or effects;
- Q_3: Frequency of occurrence.

3.1.2 Selection of Primary Studies

Figure 3 presents the stages of the primary study selection process. The exact search string was used as input and distinguished between a search on formal and informal databases. Performed on April 14, 2022, the search retrieved 578 studies in the formal databases and over one million web sources in the informal ones. The studies and web sources were submitted to a filtering process until final selection. In the filtering process, the studies and web sources were categorized into three categories of inclusion, namely, include, exclude, and uncertain.

Figure 3 – Selection of primary studies


Search string
Databases
1st filter
2nd filter
3rd filter
"test code" AND ("test* smell*" OR antipattern* OR "poor quality")
ACM Digital Library (74), IEEE Xplore (18), and Google Scholar (486)
578
Title and abstract
116
Exclusion Criteria, Introduction and Conclusions
77
Full text
67
Google Search (990,000) Bing (59,500)
1,049,500
First 50 hits
100
Exclusion Criteria, Title and Summary
31
Full text
31
127 primary studies
Snowballing
35
1st filter
35
2nd filter
35
3rd filter
29

Source: research data

The selection consisted of three filters in the formal databases. In the first filter, paper's title and abstract were read against the inclusion and exclusion criteria. From the total of 578 studies, 116 were included in the first filter. Continuing the second filter by reading the introduction and conclusion of the resulting studies, 77 studies were included. In the third filter, the full content of the studies were read against the inclusion criteria, and 10 studies were excluded. The remaining 67 studies had the quality criteria assessed.

The amount of information retrieved by the search engines in the informal databases (1,049,500 results) was far beyond our analysis capabilities. Following the guidelines of previous studies that also used the MLR approach (KULESOVS, 2015; TOM; AURUM; VIDGEN, 2013), the first 50 hits of each search engine were filtered to restrict the search space, resulting in 100 web sources. It is worth mentioning that these search engines have ranking algorithms based on relevance, and as we observed, the most relevant results usually appear in the first hits (KULESOVS, 2015; TOM; AURUM; VIDGEN, 2013). The exclusion criteria were applied in the second filter, identifying duplicates retrieved in the search engines and web sources not entirely written in English and selecting 31 informal studies. The third filter made no changes to the selected sources list.

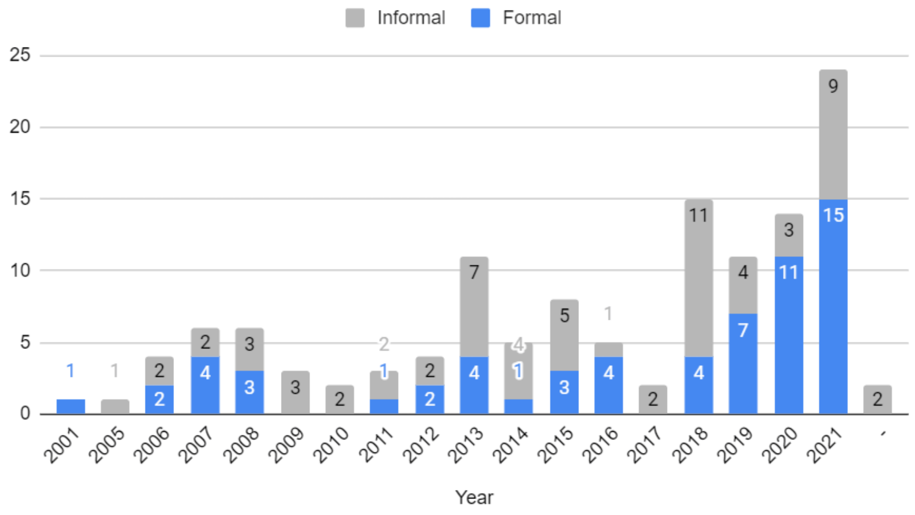
A snowballing was performed in a single iteration (WOHLIN, 2014) to find additional primary studies that the search string could not capture. Through this approach, the references of the primary studies, related work, and citations were examined when available. Using backward and forward snowballing, 5 formal studies and 30 informal studies we identified, in their respective databases, totaling 35 primary studies. After applying the same filters used in formal and informal databases, whether the study was formal or informal, one formal study and 5 informal ones were excluded in the third filter, resulting in 29 primary studies selected through snowballing.

Within the results, 4 studies that conducted systematic mappings and reviews on test smells were found. As these studies attend to our selection criteria (Section 3.1.2), we treated them as primary studies — retrieving test smell definitions and quality criteria when provided —, also submitting them to the snowballing process.

At the end of the primary studies selection process, 127 (63 formal and 64 informal) studies were selected. Figure 4 shows the distribution of the selected formal and informal primary studies. The year of publication of two informal primary studies could not be determined, so they were placed in the bottom bar of Figure 4. Starting from 2019, a growth in the number of formal production, when the total formal primary studies surpass the total informal ones, is noticeable. A complete list of the selected primary is presented in Appendix A.

Concerning the informal databases, since their search engines weight results by relevance, including clicks as relevance indicators (KULESOVS, 2015; TOM; AURUM; VIDGEN, 2013), some older results may lose relevance. Hence, they would not be displayed in the first 50 results and, consequently, not considered in our selection process. The snowballing process helped select older but relevant informal studies from secondary studies like the one by Garousi and Küçük (GAROUSI; KÜÇÜK, 2018), from which 19 informal studies issued before 2015 and unlisted in the informal search results were retrieved. This result reinforces the importance of the snowballing process.

Figure 4 – Distribution of selected primary studies per year



Source: research data

3.1.3 Data Extraction

Individual test smell data was extracted from each primary study. In the provided data set, concerning the publications, we list the title, link, type (*i.e.,* article, book, paper, webpage, thesis/dissertation, technical paper/manuscript, and lecture/presentation video), and whether the primary study uses the expressions "test smell" or "anti-pattern"; concerning the test smells in each primary study, we list the name, Also Known As (AKA) names, definition and quality criteria indications (*e.g.,* code examples, causes and effects, and frequency of occurrence). In total, 1,331 occurrences of 480 test smells from 127 selected primary studies were extracted.

We noticed the terms "test smells" and "anti-patterns" in use for the same concepts. Table 2 presents the distribution of both terminologies in the selected primary studies and the total number of primary studies considered in this study. The data extraction shows 40 test smells classified with both terminologies. For instance, the *Slow Test* is presented as anti-pattern and test smell in the catalog references (HAMMERLY, 2013) and (MESZAROS, 2007), respectively. Likewise, the *Redundant Assertion* is an anti-pattern in (SCHMETZER, 2005) and a test smell in (KUMMER, 2015). The same happens to the *Excessive Setup*, also an anti-pattern and a test smell in (KEMPF, 2016) and (KUMMER, 2015), respectively.

As stated in Chapter 1, *test smells* are analogous to *code smells* (ROMPAEY et al., 2007). Many code smell publications refer to them jointly as *smell* or *anti-pattern* because they *"reflect design and/or implementation issues in software source code that could have a negative impact on software quality"* (TAHIR et al., 2020). The same rationale is used in this work when mapping

Table 2 – The terminology in use by the selected sources

Terminology	Informal	Formal	Total
Anti-pattern	34	0	**34 (26.8%)**
Test Smell	30	63	**93 (73.2%)**
Total	**64 (50.4%)**	**63 (49.6%)**	**127 (100%)**

Source: research data

studies with both terminologies, as also performed by Garousi and Küçük (GAROUSI; KÜÇÜK, 2018) in their MLM. Nevertheless, Table 2 shows the lack of distinction to happen only in the informal studies, practically dividing them into halves — 34 studies used “anti-pattern”, whereas 30 studies used “test smell”. Meanwhile, no formal study used “anti-pattern.”

Regarding the amount of test smells described by each source, the top selected sources were the thesis by Kummer (KUMMER, 2015) with 77 test smells, the blog post by Frieze (FRIEZE, 2018) with 70, and the paper by Aljedaani *et al.* (ALJEDAANI et al., 2021) with 65 test smell definitions. Our set of selected sources has an average number of 10.5 test smells being described per source, a median of 6 test smells, and a mode of 1 test smell — having 24 sources dedicated to a single smell.

In our data set, we chose not to indicate the specific technology for which the test smell was originally intended. The reason for that choice is based upon studies dedicated to finding test smells in specific languages; such studies end up demonstrating a previously proposed test smell to also to occur in an additional programming language. Extreme examples are the test smells identified in the study by Hauptmann *et al.* (HAUPTMANN et al., 2013), intended for test smells in natural language tests, which verified — among others — the existence of the *Conditional Test* smell, proposed initially by Meszaros (MESZAROS, 2007) and exemplified in Java.

3.1.4 Data Classification

After extracting information from the selected primary studies, and using the gathered test smell descriptions, we classified the 480 smells extracted from the selected primary studies according to the classification proposed by Garousi and Küçük (GAROUSI; KÜÇÜK, 2018): (1) smells related to test execution/behavior, (2) smells related to test semantic / logic, (3) design-related test smells, (4) issues in test steps, (5) mock and stub-related smells, (6) smells in association with production code, (7) code related smells, and (8) smells related to dependencies. Their classification allowed us to classify the gathered test smells. Some test smells may be classified in more than one type. Hence, the classification types are not disjoint. For instance, the *Goto Statement* test smell described as *"When a goto statement is used"* (NEUKIRCHEN; ZEISS; GRABOWSKI, 2008) may cause *issues in test steps* (type 4) due to a (failing) manual control of the execution flow. However, its main problem is the use of a code statement that should be avoided and, as such, this test smell is better classified as *code-related* (type 7).

In this process, two independent researchers — the author and an assistant — classified the gathered test smells using the available types, always respecting the already classified test smells present in the test smell classification board proposed by Garousi and Küçük (2018). Divergences in the classification were discussed in two meetings, where 87 test smells mostly included in the *"Other test logic related"* and *"Violating coding best practices"* categories were discussed. Each test smell was allowed to belong to one type only, and the researchers decided on the most representative one. Table 3 shows the distribution of the 480 test smells per category. One can notice almost 32% of test smells having relation to code (type 7) and this percentile rises to almost 42% if we consider dependencies (type 8) and exception handling smells, indicating their close relationship to code smells. We added the test smell classification in our public data set and present the complete list of test smell names per category in Appendix C.

3.1.5 Data Correlation

As a last activity, we identified test smells by using the alternative names — *Also Known As (AKA)* — provided by the primary studies we selected. If a test smell had multiple alternative names, we replaced all occurrences of those names with the original test smell as an AKA. We also did this for alternative names that themselves had alternative names. As a result, we consolidated all alternative names into the first occurrence of the test smell, and subsequent occurrences were cross-referenced with the original test smell. Equation 3.2 formalizes this activity considering test smells S_1 to S_n and an equality given by their AKA names:

Table 3 – Test smells per category

Category	Test Smells	%
1. Test execution/behavior		
1.1 Performance	14	2.9%
1.2 Other test execution/behavior	15	3.1%
2. Test semantic/logic		
2.1 Testing many things	14	2.9%
2.2 Testing many units	4	0.8%
2.3 Other test logic related	67	14.0%
3. Design related		
3.1 Not using test patterns	15	3.1%
4. Issues in test steps		
4.1 Issues in setup	34	7.1%
4.2 Issues in assertions	54	11.3%
4.3 Issues in teardown	14	2.9%
4.4 Issues in exception handling	10	2.1%
5. Mock and stub related		
5.1 Mock and stub related	18	3.8%
6. In association with production code		
6.1 In association with production code	29	6.0%
7. Code related		
7.1 Code duplication	28	5.8%
7.2 Complex/hard to understand	41	8.5%
7.3 Violating coding best practices	84	17.5%
8. Dependencies		
8.1 Dependencies among tests	20	4.2%
8.2 External dependencies	19	4.0%
Total	**480**	**100%**

Source: research data

$$\begin{gathered}(S_1 = S_2) \wedge (S_2 = S_3) \wedge ... \wedge (S_{n-1} = S_n) \\ \Longrightarrow \\ S_1 = \{S_2, S_3, ..., S_{n-1}, S_n\} \wedge (S_2 = S_1) \wedge (S_3 = S_1) \wedge ... \wedge (S_{n-1} = S_1) \wedge (S_n = S_1)\end{gathered} \tag{3.2}$$

Following the extraction of data, a total of 52 test smells were identified as having alternative names. After applying the identification method outlined in Equation 3.2, this number increased to 82 test smells. However, we were not able to find references to every alternative name. For instance, the *Ugly Mirror* (GAROUSI; KÜÇÜK, 2018) test smell may also be called *Tautological test*, but we did not retrieve any source of information for the latter. In this sense, accounting for the alternative names of a test smell as the original one, our catalog of 480 test smell names would actually represent 447 different definitions. All in all, Table 4 provides an overview of the test smells with the most alternative names. Notably, the *Eager Test* stands out with its alternative names, suggesting that more attention should be paid to existing literature when proposing new test smells.

Table 4 – The 10 test smells with the most alternative names

Test smell	**AKA - Quantity**	**AKA - Names**
Eager Test	7	The Test It All, Split Personality, Many Assertions, Multiple Assertions, The Free Ride, Silver Bullet, Piggyback
Excessive setup	6	Large Setup Methods, Inappropriately Shared Fixture, The Mother Hen, The Stranger, The Distant Relative, The Cuckoo
Assertionless Test	3	Lying Test, The Line Hitter, No Assertions
Manual Intervention	3	Interactive Test, Manual Testing, Manual Test
Obscure Test	3	Long Test, Complex Test, Verbose Test
Anonymous Test	2	Unclear Naming, Naming Convention Violation
Conditional Test Logic	2	Indented Test Code, Guarded Test
Generous Leftovers	2	Wet Floor, Sloppy worker
Order Dependent Tests	2	Chained Tests, Chain Gang
Slow Test	2	Long Running Test, The Slow Poke

Source: research data

3.1.6 Summary

Table 5 presents the 20 most popular test smells after the data extraction, classification and correlation activities, including the occurrences of their alternative names. The total occurrences of the listed test smells (*i.e.,* 503) stand for roughly 38% of the total 1,331 occurrences in this study. It is important to notice that, from the initial list of 10 test smells proposed by van Deursen *et al.* (DEURSEN et al., 2001), only the *For Testers Only* — 22nd position with 13 occurrences — and the *Test Run War* — 28th position with 10 occurrences — do not appear in Table 5. With a total of 336 occurrences, the smells proposed by the authors (*i.e., Mystery Guest*, *Resource Optimism*, *General Fixture*, *Eager Test*, *Lazy Test*, *Assertion Roulette*, *Indirect Testing*, *For Testers Only*, *Sensitive Equality*, and *Test Code Duplication*) stand for 25% of all occurrences in formal and informal sources. The complete list of test smells and their occurrences can be found in Appendix B.

Table 5 – The 20 most frequent test smells, including AKA occurrences

No	Test Smell	Sources
1	Eager Test	60
2	Assertion Roulette	48
3	Mystery Guest	41
4	General Fixture	40
5	Sensitive Equality	34
6	Resource Optimism	29
7	Conditional Test Logic	29
8	Obscure Test	25
9	Lazy Test	22
10	Test Code Duplication	20
11	Indirect Testing	19
12	Excessive setup	18
13	Empty Test	17
14	Redundant Assertion	16
15	Magic Number Test	15
16	Unknown Test	14
17	Sleepy Test	14
18	Ignored Test	14
19	Duplicate Assert	14
20	Constructor Initialization	14

Source: research data

3.1.7 Threats to Validity

As threats to the internal validity, we use the definition of test smells, anti-patterns, or quality problems — or the indication of a study that does so — as a selection criterion for our set of primary studies. Other selection criteria, reflected in the search string in the searched bases, could generate different results. To minimize this threat, we conducted a preliminary search to verify the terms most used by the intended studies for self-description. Moreover, we used Google Scholar among the search engines for scientific bases. Although this search engine indexes scientific databases and university repositories, we analyze all its results. A search only in repositories of peer-reviewed works could bring different and narrower results from those that might be found instead. Another threat is the potential error on the test smell classification activity (Section 3.1.4). We address this second threat by performing the classification in pairs and solving divergences in specific meetings, leading us to a reasonable level of inter-rater reliability (percent agreement) of 81.9% (SKRONDAL; EVERITT, 2010).

Concerning the threats to the conclusion validity, with regards to the alternative names raised in the Data Correlation activity (Section 3.1.5), there is a possibility of misclassification of equivalent smells. We address this threat by cataloging AKA names only when indicated by a selected source.

As threats to the construct validity, Section 3.1.3 mentions that 40 test smells were presented simultaneously as anti-patterns or test smells by different sources. Although there may be a differentiation between such concepts in the literature, it was not considered in our results and, to avoid missing test smell (or anti-pattern) occurrences, we included all the obtained results. Therefore, as some of the cataloged test smells may be anti-patterns — considering the differentiation used for code smells and anti-patterns — we expect future studies to establish the division of concepts better as we already provide a data set of both concepts.

Regarding the threats to dependability, which concern the consistency and repeatability of the findings, we highlight that (i) the internal algorithms of the informal databases alter the results of the first 50 results as new results are indexed and weighted and that (ii) we carried out our search in the first quarter of 2022. Therefore, as there are differences between our study and the systematic search performed by Garousi and Küçük (GAROUSI; KÜÇÜK, 2018), differences in the results of a new execution are expected. We mitigate this threat by extensively including non-peer-reviewed — although scholar — content, which ensures future peer-reviewed publications of such content are already included in our results.

Finally, regarding the external threats to validity, analyzing 50 results per informal search engine is not enough to achieve generalization to the above 1 million results. We minimize this threat by using recommendations of known literature on the topic which detail the ranking algorithms used by search engines and inform the low probability of useful clicks after the 50^{th} result (KULESOVS, 2015; TOM; AURUM; VIDGEN, 2013).

3.2 A CATALOG FOR EVERYONE

This section details the construction of the online catalog using the data set resultant from the data extraction, classification and correlation activities (Section 3.1). We present the catalog's settings in Section 3.2.1, an overview of the catalog functionalities in Section 3.2.2, and implications for practice in Section 3.2.3.

3.2.1 Settings

The catalog seeks to provide a visualization of the data set compiled in the MLR (Section 3.1) as a quick reference guide for researchers and practitioners. We present all the 480 test smells with as much ease to use as possible and indicate the known formal and informal literature, with per source quality attributes, for additional information. Also, as we intend to extend the catalog maintenance to the software testing community, the catalog is actually an open-source project and enables any community member to submit a contribution.

We chose *Read the Docs*,[1] an *"open-sourced free software documentation hosting platform"* that generates documentation written with the Sphinx documentation generator,[2] to create the catalog. While the *Read the Docs* platform hosts the documentation, compiling it directly from the GitHub project where the source files are kept on, Sphinx enables us to write simple markdown source files that can be exported to several formats (*e.g.,* HTML, PDF, and EPUB) and are easily maintained.

Finally, the catalog is permanently available online at <https://test-smell-catalog.readthedocs.io/>.

3.2.2 Overview

Here we present an overview of the created catalog and its options. Figure 5 presents the initial catalog screen. After an introductory text explaining the catalog objectives, origin, organization and basic functionalities, test smells are presented as clickable links and grouped by category, which should be easily located by users who already know the intended test smell by its name.

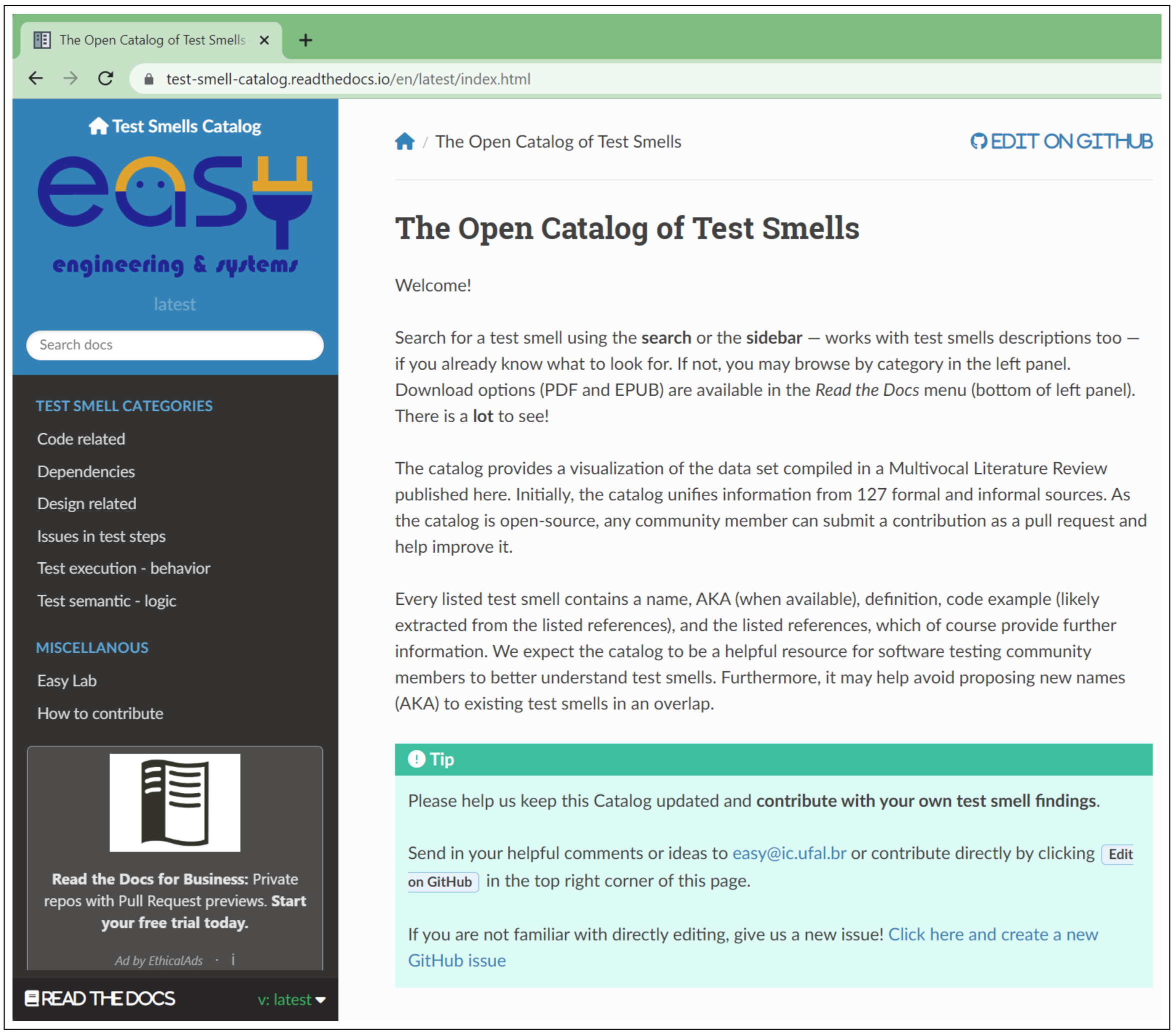
A search bar is available in the upper left panel of the main screen and works for any content in the catalog, including test smell cards (*e.g.,* name, description, AKA, code, and bibliography). Figure 6 presents an example of search results.

Below the search bar, test smell categories are presented as an expansible menu containing the subcategories which, in turn, contain the grouped test smells. Figure 7 presents the categories menu.

Below the test smell categories (Figure 7), a contribution guide page teaches contributors

[1] <https://readthedocs.org/>
[2] <https://github.com/sphinx-doc/sphinx>

Figure 5 – Initial catalog screen



The Open Catalog of Test Smells

Welcome!

Search for a test smell using the **search** or the **sidebar** — works with test smells descriptions too — if you already know what to look for. If not, you may browse by category in the left panel. Download options (PDF and EPUB) are available in the *Read the Docs* menu (bottom of left panel). There is a **lot** to see!

The catalog provides a visualization of the data set compiled in a Multivocal Literature Review published here. Initially, the catalog unifies information from 127 formal and informal sources. As the catalog is open-source, any community member can submit a contribution as a pull request and help improve it.

Every listed test smell contains a name, AKA (when available), definition, code example (likely extracted from the listed references), and the listed references, which of course provide further information. We expect the catalog to be a helpful resource for software testing community members to better understand test smells. Furthermore, it may help avoid proposing new names (AKA) to existing test smells in an overlap.

Tip

Please help us keep this Catalog updated and **contribute with your own test smell findings**.

Send in your helpful comments or ideas to easy@ic.ufal.br or contribute directly by clicking Edit on GitHub in the top right corner of this page.

If you are not familiar with directly editing, give us a new issue! Click here and create a new GitHub issue

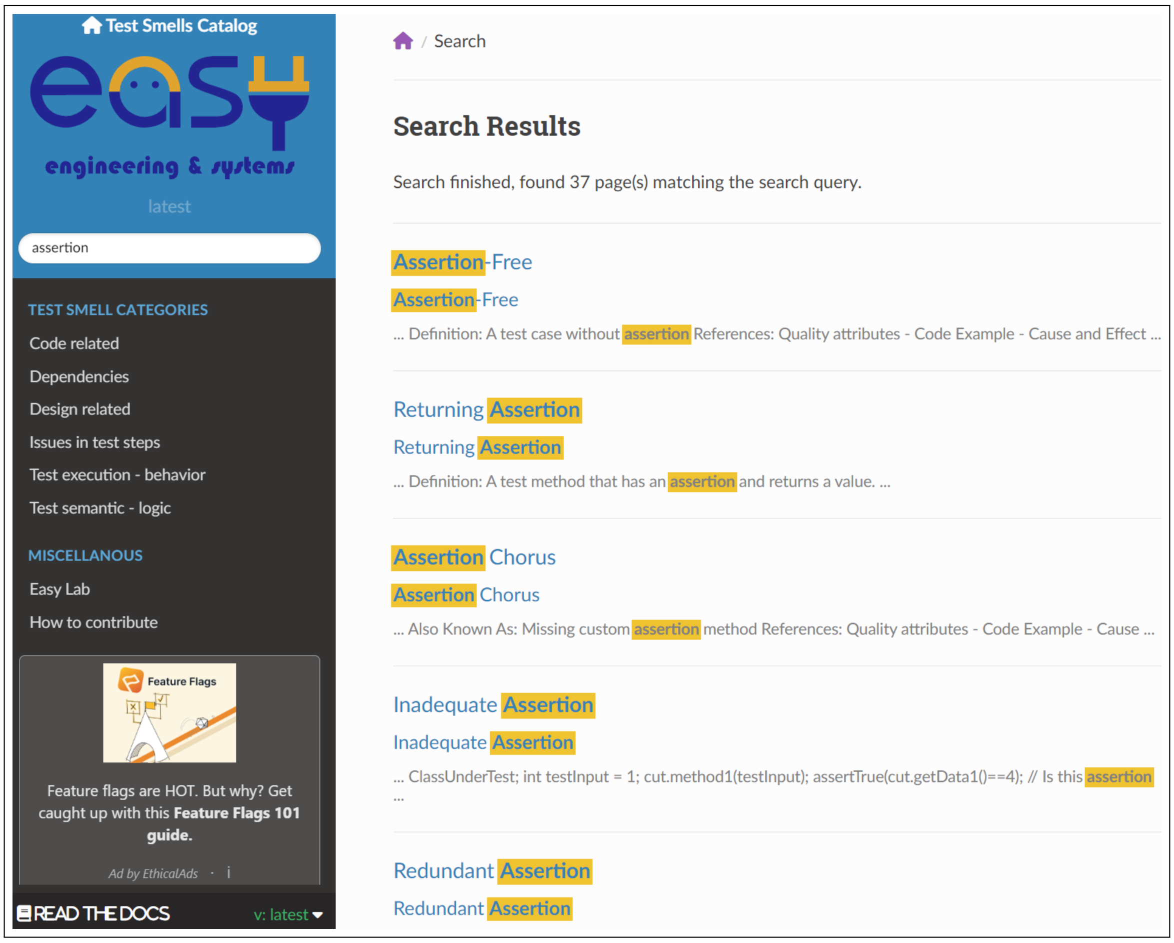
Source: research data

how to submit new test smells — using a provided template —, modify any information or report bugs in the catalog. At last, the bottom of the left panel — *Read the Docs* link on Figure 6 —, users can visualize any catalog version and download it exported to PDF and EPUB formats, as well as verify compilation details as changes are submitted to the project hosted on GitHub.

An example of a test smell card is presented in Figure 8. Every test smell card contains the following fields:

- **Navigation:** A sequence of clickable links providing current location among categories and subcategories;
- **Contribution:** A link to edit the displayed page directly on the open-source project hosted on GitHub;

Figure 6 – Search screen demonstration



Feature flags are HOT. But why? Get caught up with this **Feature Flags 101 guide.**

Search Results

Search finished, found 37 page(s) matching the search query.

Assertion-Free
Assertion-Free
... Definition: A test case without assertion References: Quality attributes - Code Example - Cause and Effect ...

Returning Assertion
Returning Assertion
... Definition: A test method that has an assertion and returns a value. ...

Assertion Chorus
Assertion Chorus
... Also Known As: Missing custom assertion method References: Quality attributes - Code Example - Cause ...

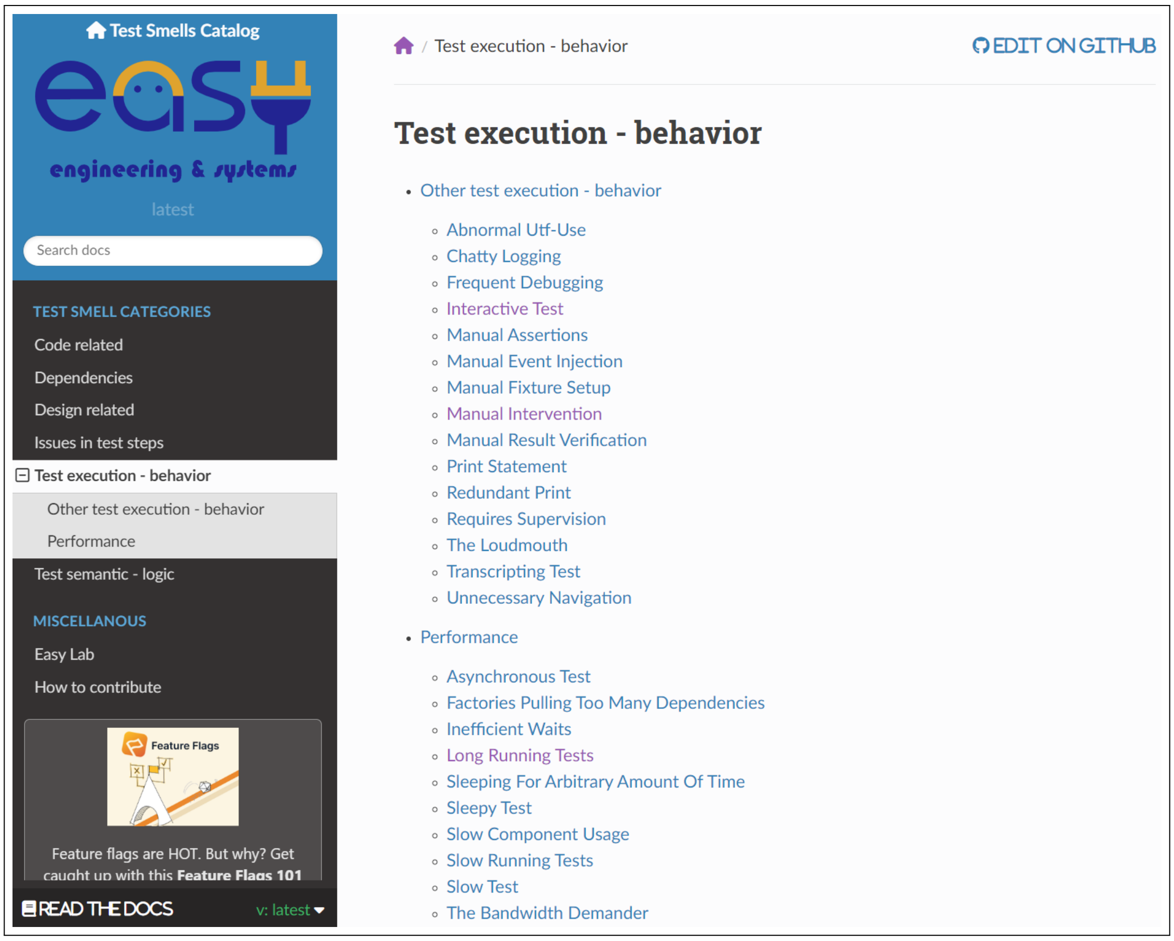
Inadequate Assertion
Inadequate Assertion
... ClassUnderTest; int testInput = 1; cut.method1(testInput); assertTrue(cut.getData1()==4); // Is this assertion
...

Redundant Assertion
Redundant Assertion

Source: research data

- **Name:** The test smell name;
- **Definition:** A description text retrieved from a listed reference;
- **Also known as:** Alternative names of the currently displayed test smell, retrieved from the references;
- **Code Block Example:** Code sample extracted from a reference, when available;
- **References:** All the identified (formal and informal) literature referring to the currently displayed test smell with badges (icons) indicating whether they present additional code examples, discussion of causes and effects, or frequency of occurrence;
- **Navigation buttons:** Meant to navigate sideways and see test smells' individual pages.

Figure 7 – Categories menu


Test Smells Catalog

easy engineering & systems

latest

Search docs

TEST SMELL CATEGORIES
- Code related
- Dependencies
- Design related
- Issues in test steps
- Test execution - behavior
  - Other test execution - behavior
  - Performance
- Test semantic - logic

MISCELLANOUS
- Easy Lab
- How to contribute

Feature flags are HOT. But why? Get caught up with this Feature Flags 101

READ THE DOCS v: latest

Test execution - behavior

EDIT ON GITHUB

Test execution - behavior

- Other test execution - behavior
  - Abnormal Utf-Use
  - Chatty Logging
  - Frequent Debugging
  - Interactive Test
  - Manual Assertions
  - Manual Event Injection
  - Manual Fixture Setup
  - Manual Intervention
  - Manual Result Verification
  - Print Statement
  - Redundant Print
  - Requires Supervision
  - The Loudmouth
  - Transcripting Test
  - Unnecessary Navigation
- Performance
  - Asynchronous Test
  - Factories Pulling Too Many Dependencies
  - Inefficient Waits
  - Long Running Tests
  - Sleeping For Arbitrary Amount Of Time
  - Sleepy Test
  - Slow Component Usage
  - Slow Running Tests
  - Slow Test
  - The Bandwidth Demander

Source: research data

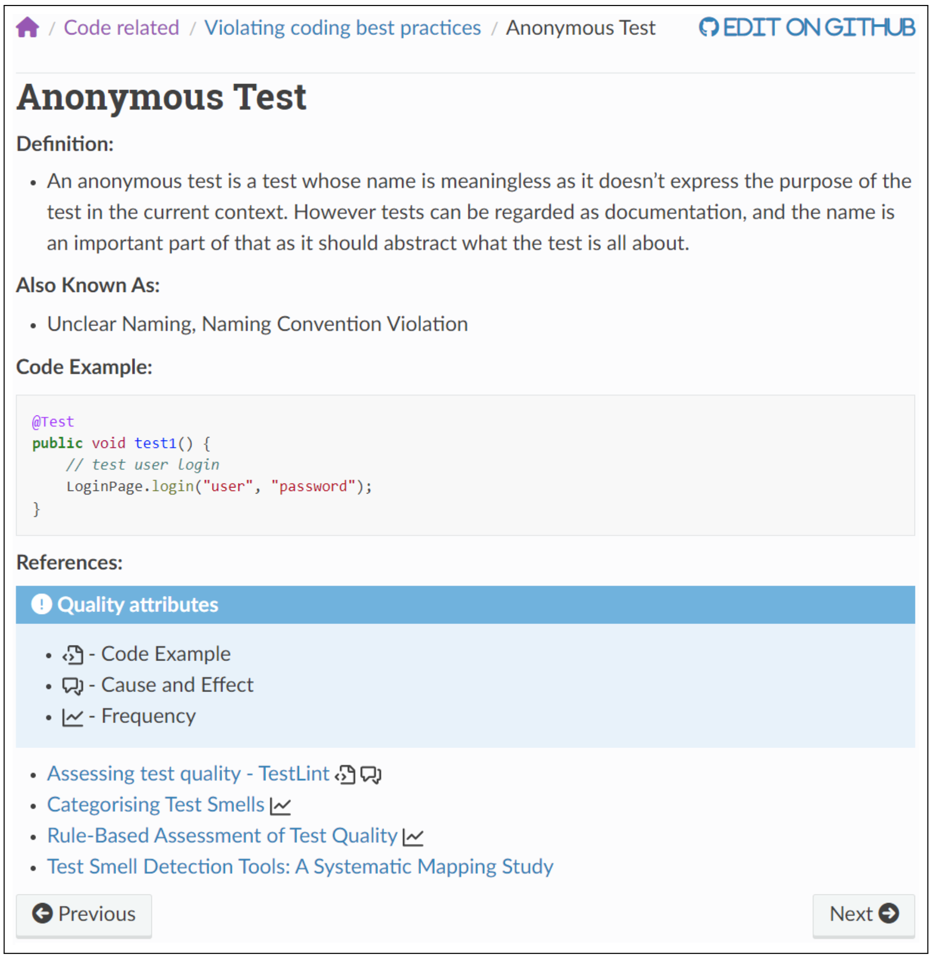
3.2.3 Implications for Practice

We now present two implications for practice of our catalog:

- **Avoiding the proposition of new test smell names for already existing test smells:** Since test smells are grouped in categories and a search bar is available, researchers and practitioners can now search in a centralized repository for existing test smells — possibly with a different name — prior to proposing a new test smell.

- **Common learning point for test smells and related bibliography:** As a public catalog with the most significant amount of compiled information on test smells, students, practitioners, and researchers can learn about any test smell without having to blindly search studies and web sources for specific information. The test smell card indicates the primary studies where the test smell is discussed and which additional information can be found in each study.

Figure 8 – Test smell card example



/ Code related / Violating coding best practices / Anonymous Test
EDIT ON GITHUB

Anonymous Test

Definition:

- An anonymous test is a test whose name is meaningless as it doesn't express the purpose of the test in the current context. However tests can be regarded as documentation, and the name is an important part of that as it should abstract what the test is all about.

Also Known As:

- Unclear Naming, Naming Convention Violation

Code Example:

```
@Test
public void test1() {
    // test user login
    LoginPage.login("user", "password");
}
```

References:

Quality attributes

- - Code Example
- - Cause and Effect
- - Frequency

- Assessing test quality - TestLint
- Categorising Test Smells
- Rule-Based Assessment of Test Quality
- Test Smell Detection Tools: A Systematic Mapping Study

Source: research data

3.3 LESSONS LEARNED

As ending such an extensive study to organize hundreds of test smells without a few observations is inevitable, we present our contributions:

Be as thorough as possible: It is important to provide thorough information to ensure a clear understanding of the test smell you want to explain. Unfortunately, out of the 480 test smells analyzed in this study, only 46% (*i.e.*, 221) have at least a description and a code example. This lack of information can hinder comprehension and lead to the proposition of new alternative names. Based on our experience, complete information should include definitions, causes and effects, code examples, strategies for removal, and frequency of occurrence. These topics provide a complete understanding of the problem and the necessary action.

Descriptive names are better than metaphorical ones: It is ineffective to obey someone to read the description of your test smell to get to understand the metaphor in its name.

Also, bad metaphors may incentive alternative names. For example, *Anal Probe* (KUMMER, 2015), *X-Ray Specs* (GAROUSI; KÜÇÜK, 2018), *I wrote it like this* (FRIEZE, 2018), *You Do Weird Things to Get at the Code Under Test* (WILLIAMS; DIETRICH, 2017) are all related to access modifiers on attributes and methods.

Use language-agnostic descriptions: The *Assertion Roulette* (DEURSEN et al., 2001) exists in Java (BAVOTA et al., 2012; PERUMA et al., 2020; SANTANA et al., 2020), Scala (BLESER; NUCCI; ROOVER, 2019a; BLESER; NUCCI; ROOVER, 2019b), JavaScript (JORGE; MACHADO; ANDRADE, 2021), Python (WANG et al., 2021), and C++ (BREUGELMANS; ROMPAEY, 2008). Likewise, the *Conditional Test* (MESZAROS, 2007) exists in manual tests (HAUPTMANN et al., 2013). If the original works were restricted to a programming language, other test smells with different names for the same problems could emerge in later studies. As done in the work by van Deursen *et al.* (DEURSEN et al., 2001), we reinforce the importance of using high-level descriptions, similar to design patterns (GAMMA et al., 1994), to address test smells. By doing so, these descriptions can be applied to multiple contexts and technologies, making them more useful and easier to reuse.

3.4 RESEARCH DIRECTIONS

Built upon our findings, research directions for cataloging test smells would involve:

- The data set containing the results of data extraction, classification, and correlation activities can be a basis for developing automated information extraction mechanisms to find new relationships between the cataloged test smells. The proposed catalog provides a helpful resource for the software testing community members to better understand test smells. More than a single point of consultation for definitions, examples, and related studies, the catalog indicates relationships between test smells (AKA).
- Work to organize the available information on test smells is still necessary to provide a basis for upcoming research. In this sense, as future work, the catalog can be extended by adding the existing test smell refactorings as transformations (SOARES et al., 2023), as we detail in the next chapter, to enable new automatic detection and refactoring tools.
- Having a corpus of test smell names and definitions enables further research on their similarities, therefore automatically identifying additional AKA and grouping test smells

by subject-related keywords.

- Such corpus of test smell names, definitions and literature can also be used to pinpoint the smells for which there is empirical evidence that their existence can impact readability/maintenance/test quality, therefore providing a ranking for the most harmful cataloged test smells.

4 LATEST TEST FRAMEWORK FEATURES REMOVE SMELLS

Now that we have an initial organization of the test smells area, with definitions and references to formal and informal primary studies, and the additional information they provide, we can continue the contributions of this thesis. In particular, we focus on the currently available implementations of refactoring operations versus the possibilities given by new versions of test libraries. Because of its popularity and wide availability of open-software projects on public repositories like the GitHub, we use the JUnit test framework as subject for the investigations in this chapter, in which we present a mixed-study investigation consisting of three parts.

First (Section 4.1), we execute an empirical study considering 485 popular Java open-source projects — which automate tests with JUnit — to identify how many of them use the JUnit 5 library and how they use its features. Second (Section 4.2), we carefully analyze the features introduced in JUnit 5 and identify that some might help remove test smells. After our analysis, we propose code transformations based on the studied new features. Third, to evaluate our proposals (Section 4.3), we create two studies aimed at answering our next research questions: an online survey with 212 developers and the submission of 38 pull requests to popular open-source projects. We end this chapter by providing research directions for advancing smell removal from test code based on new test framework features (Section 4.4).

4.1 SOFTWARE PROJECTS ARE NOT UP-TO-DATE

In this study, considering the Goal Question Metric (GQM) aproach (BASILI; CALDIERA; ROMBACH, 1994), our goal is to analyze GitHub repositories of Java projects concerning the imports of classes related to JUnit 5 from the viewpoint of test developers in the context of the JUnit 5 framework adoption. In particular, we answer **RQ$_2$: “To what extent do projects use new test framework features?”** To do so, we automatically analyze test classes from popular Java projects that use the JUnit 5 library and gather information about which of such projects use JUnit 5 features. Hence, we can evaluate if features capable of preventing or removing test smells are well adopted.

4.1.1 Settings

As there is no publicly available and maintained list of open-source projects that use the JUnit 5 library, we relied on GitHub search to retrieve an initial list of Java projects. We filtered the list by project popularity, considering that GitHub retains over 4.3 million Java projects.[1] As the popularity of projects in GitHub is measured in terms of the stars each one receives, when configuring the search string to Java projects to have more than 3,000 stars (*language:Java stars:>3000*), in January 19th 2021, we retrieved a total of 767 projects.

We developed a tool to perform the analysis. After downloading the source code of each project, always considering the latest production release, our tool detects the project test classes and retrieves information from the imports section. Import statements from JUnit 4 and below versions can be differentiated from JUnit 5. While the latter version can be identified through matches for the *"org.junit.jupiter"* string, JUnit 4 and below versions have their imports matching the *"org.junit"* string. With the help of the JUnit 5 User Guide (BECHTOLD et al., 2020) and Javadoc, we related the imported classes to a feature they implement. For instance, `org.junit.jupiter.params.provider.CsvSource` is a class of the Parameterized Tests feature. Then we grouped the found classes per feature. As a final step to assemble the results, we removed information about classes from features already available in previous JUnit versions, e.g., `Test`, `AssertEquals`, and `Fail`, and the ones that merely replaced a previous class with a more meaningful name while keeping the same behavior: `BeforeAll` (replaces `BeforeClass`), `AfterAll` (replaces `AfterClass`), `BeforeEach` (replaces `Before`), `AfterEach` (replaces `After`), `Disabled` (replaces `Ignore`), and `Tag` (replaces `Category`).

4.1.2 Results

We downloaded (cloned) each project, which meant over 1.1 million files and 98GB of data, and submitted them to our tool. From the initial list of 767 projects, we found that 485 (i.e., 63.2%) use JUnit for test automation. The distribution of the JUnit library versions among projects is 408 (i.e., 84.1%) for JUnit 4 and below versions, and 77 (i.e., 15.9%) projects using JUnit 5. Concerning the domain of the latter projects, we had 37 applications, 8 example projects, 22 frameworks, and 10 libraries. As our tool performs the analysis only considering the imports section of each test class, we may have a threat to validity, to be

[1] From GitHub search, using *language:Java* as parameter

discussed in Section 4.1.4.

Table 6 presents the total numbers concerning projects using JUnit 5. Here, the total Lines of Code (LOC) was calculated using the Browser Extension GLOC plugin,[2] and the total number of test files (classes) and unit tests (methods) were obtained using the TestFileDetector[3] (PERUMA et al., 2019) tool. Our tool detected 5,717 import occurrences of 65 JUnit 5 classes related to 17 newly introduced features, according to the JUnit 5 User Guide (BECHTOLD et al., 2020).

Table 6 – Total study numbers

Projects	77
LoC	49,075,700
Test files (classes)	48,709
Tests (methods)	229,577
JUnit 5 imports (occurrences)	5,717
JUnit 5 features in use	17

Source: research data

Concerning other test libraries coexisting with JUnit 5, we found that 72 projects did present them. Mockito was the most frequent test library alongside JUnit 5 in 59 projects, followed by Hamcrest (53), AssertJ (44), TestNG (5) and Serenity (1). The average test libraries per project — including JUnit 5 — was 3.1.

Table 7 presents the distribution, considering the 77 analyzed projects jointly, of the 5,717 import occurrences per JUnit 5 feature and their percentage concerning the total imports. The features we use in this study towards test code transformations to remove test smells (Chapter 4.2) are also highlighted in Table 7. Finally, Table 8 presents the usage of the new JUnit 5 features per project. Despite the presence of the JUnit 5 library, 14 projects did not use any new features.

Table 8 – Distribution of new JUnit 5 features per project

Project (new JUnit 5 features in use)		
Java and Spring Tutorials (15)	MyBatis-Plus (4)	LMAX Disruptor (1)
Neo4j (12)	Zipkin (4)	MinecraftForge (1)
Quarkus (12)	Apache SkyWalking (3)	MyBatis Generator (1)
Spring Framework (11)	Apache ZooKeeper (3)	Netty (1)
GoCD (10)	Cucumber Common Components (3)	Oracle OpenGrok (1)
Apache Camel (9)	Micronaut (3)	Sentinel (1)

[2] Available at <https://github.com/artem-solovev/gloc>
[3] Supplementary tool for the Test Smell Detector (tsDetect) tool (PERUMA et al., 2020)

Apache Dubbo (8)	OkHttp (3)	Soul (1)
Eclipse Jetty (8)	Vespa (3)	Spring PetClinic (1)
Java Design Patterns (8)	Checkstyle (2)	spring-cloud-alibaba (1)
Spring Boot (8)	Debezium (2)	spring-cloud-netflix (1)
Aeron (7)	JADX (2)	spring-data-examples (1)
CAS (7)	Mockito (2)	Alibaba Arthas (0)
Graylog (7)	Mybatis-3 (2)	Apache Hadoop (0)
JanusGraph (7)	Seata (2)	Apache Ignite (0)
Apache JMeter (6)	Simplify (2)	AWS Code Examples Repository (0)
Dropwizard (6)	Spring Boot Admin (2)	Distributed Transaction Framework - LCN (0)
Reactor Core (6)	spring-boot-demo (2)	FlexibleAdapter (0)
Testcontainers (6)	Activiti (1)	Internet Architect (0)
Bisq (5)	Apache Beam (1)	Karate (0)
Cryptomator (5)	Apache Hive (1)	Mindustry (0)
Flowable (5)	Apache Kafka (1)	MyBatis Spring-Boot-Starter (0)
JavaParser (5)	Apache Shiro (1)	Onemall (0)
Jenkins (5)	Data Transfer Project (1)	RabbitMQ Tutorials (0)
Jodd (5)	Halo (1)	springboot-guide (0)
OWASP ZAP (5)	Jsoup (1)	XXL-JOB (0)
Lettuce (4)	Keycloak (1)	

Source: research data

4.1.3 Discussion

Our study verified that 77 out of 485 projects (i.e., 15.9%) use the JUnit 5 library. Therefore, the high percentage of projects using previous versions of JUnit might represent sub-optimal test case implementations, which could pave the way to the existence of test smells. As explained in Section 4.1.1, the occurrences detailed in Table 7 represent imports related to new (introduced) JUnit 5 features. Even considering every test class to have a single import statement, it would represent a maximum of 11.7% of the total 48,709 test classes using new features. One possible explanation for such low use is in the study performed by Kim et al. (2021a), where the authors claim that test developers are sometimes unaware of the features provided by test frameworks.

Table 7 shows that Custom Extensions, Exception Testing, and Parameterized Tests are the most popular JUnit 5 features with more than 70% of the features utilization. The Custom Extensions feature is the only form to extend the behavior of test classes and methods — formerly provided by Runners and Rules — and the Exception Testing feature is the only way

Table 7 – Distribution of occurrences per new JUnit 5 feature

No	Feature	Occurrences	%
1	Custom Extensions	2,336	40.9
2	**Exception Testing**	**1,159**	**20.3**
3	**Parameterized Tests**	**676**	**11.8**
4	**Parallel Execution**	**263**	**4.6**
5	Nested Tests	233	4.1
6	Test Execution Order	232	4.1
7	**Temporary Directory**	**185**	**3.2**
8	**Conditional Test Execution**	**170**	**3.0**
9	Display Names	141	2.5
10	Dynamic Tests	122	2.1
11	Dependency Injection	79	1.4
12	Migration Support	44	0.8
13	Timeout Assertions	37	0.6
14	**Repeated Tests**	**23**	**0.4**
15	**Grouped Assertions**	**12**	**0.2**
16	String List Assertions	3	0.0
17	Iterable Assertions	2	0.0
		Total: 5,717	**100%**

Source: research data

to verify exceptions — formerly provided by Rules and the *expected* parameter of the `@Test` annotation. Although the Grouped Assertions feature represents only 0.2% of the usages, it is capable of removing the Assertion Roulette, which is the most frequent test smell in Java projects (PERUMA et al., 2019). In this scenario, the popularization of the JUnit 5 features and the introduction of test smell removal strategies that consider such features — presented in Section 4.2 — are important goals towards improving test code quality.

4.1.4 Threats to Validity

As internal threats, selecting projects by popularity to analyze the use of current libraries for test automation may not be a good choice criterion. The selected projects may not represent real software projects (i.e., sample repositories) or have no automated testing at all. However, because they are popular in the GitHub community, we believe these projects have their complete development cycle, including testing automation. Moreover, our search did not consider projects' activity frequency (last commit), which is as important as the popularity metric. Adding this parameter to filter projects active since 01-01-2020 in a new search performed in 10^{th} January 2022[4] resulted in 768 projects, from which 645 (84%) are present in our original search. The 123 (16%) projects that could not be found in the new search may have been discontinued during 2021 or have their last activity before 01-01-2020.

Also, analyzing the project's dependency/build file would answer which JUnit framework version is used, even when projects use more than one version (e.g., subprojects). However, as we needed more fine-grained information about the utilization of features, we would need to analyze the test classes and their import statements in any case. In this regard, there is a threat to the correctness of our tool when analyzing imports. To minimize this threat, we conducted a preliminary experiment. We selected two popular open-source projects: the Checkstyle project[5], which uses JUnit 5 and from which we manually analyzed the results of 100 test classes; and the Apache Cassandra project[6], which uses a previous version of the JUnit framework and we executed the tool in the whole project (750 test classes). The results indicate our tool to correctly identify the JUnit 5 import in the analyzed test classes from the first project, and no JUnit 5 imports were detected in the second project. The list of the Checkstyle project's validated test classes and the log from Apache Cassandra's analysis are on the companion website.

As external threats, in this study, we did not randomly select what projects to analyze in the GitHub repository — we retrieved the projects list from the search engine with a popularity bias. In this context, as we did not follow primary sampling rules as defined by II, Kotrlik and Higgins (2001), we cannot make generalizations about the usage of the JUnit 5 library in projects hosted in the entire GitHub base.

4.2 TRANSFORMING SMELLY TEST CODE

Here, we present our proposals to transform test code and remove test smells based on the new features introduced in our subject test automation framework: JUnit. For that purpose, before presenting a list of test smells and our proposed transformations, we present a brief background on JUnit 5 features.

[4] using (stars:=>3000 pushed:=>2020-01-01 language:Java) in GitHub search
[5] <https://github.com/checkstyle/checkstyle>
[6] <https://github.com/apache/cassandra>

4.2.1 Background on New JUnit5 Features

This section briefly introduces some of the JUnit 5 features, stable and experimental,[7] which we consider potentially beneficial to remove test smells. We created this feature list by carefully studying the JUnit 5 User Guide (BECHTOLD et al., 2020) and comparing feature implementations with formal literature about test smell refactorings (DEURSEN et al., 2001; MESZAROS, 2007; PERUMA et al., 2019).

As a general characteristic, the execution of a test method with many individual assertions stops after the first failed one, which defines the *failed* test method outcome. The *Grouped Assertions* feature — available since JUnit 5.0.0 (09/2017) — explicitly allows all assertions to be executed independently, and a report containing all assertions and their individual results is generated at the end of the test method execution. Source Code 6 presents this feature.

Source Code 6 – JUnit 5 Grouped Assertions feature

```
@Test
public void test() {
  MyClass c = new MyClass();
  assertAll(
    () -> assert(c.getPropertyA()),
    () -> assert(c.getPropertyB()),
    () -> assert(c.getPropertyC()),
    ...
    () -> assert(c.getPropertyN())
  );
}
```

Source: research data

Providing annotations describing conditions to systems, runtime and environment variables, system properties, and even encapsulating custom condition verifications in a boolean method, the *Execution Condition API* — since JUnit 5.0.0 (09/2017) — allows entire test containers, or single test methods, to be executed only after a condition verification. Source Code 7 presents the usage of execution conditions, having a custom condition encapsulated in a boolean method.

Source Code 7 – JUnit 5 Execution Condition feature

```
@Test
@EnableIf("isCondition")
public void test() {
  MyClass c = new MyClass();
  assert(c.getProperty());
}

public boolean isCondition() {
  return ...;
}
```

[7] An EXPERIMENTAL feature status means a new feature that can be promoted to MAINTAINED, STABLE, or be discontinued according to community feedback(BECHTOLD et al., 2020).

Source: research data

The @RepeatedTest annotation — JUnit 5.0.0 (09/2017) — enables a test to be repeated a specified number of times, removing method repetition structures from within the test method body. Each repetition considers the execution of pre and post-condition methods when defined. Source Code 8 presents the usage of the repeated tests feature.

Source Code 8 – JUnit 5 Repeated Tests feature

```java
@RepeatedTest(2)
public void test() {
  MyClass c = new MyClass();
  c.performTask();
  assert(c.isAnyStatus());
}
```

Source: research data

An entire self-destroying temporary directory can be defined with the @TempDir annotation, which has been experimentally available since JUnit 5.4 (02/2019). Unlike current JVM implementation for temporary files — they are deleted after JVM stops — the JUnit 5 implementation deletes the temporary directory after every test method (or container) execution, even in parallel scenarios. This feature is handy for preventing leftovers from previous unsuccessful executions, specially shared file resources. Also, by delegating the cleanup steps to the framework, the test maintainability is improved. Source Code 9 presents its usage.

Source Code 9 – JUnit 5 Temporary Directory feature

```java
@Test
public void test(@TempDir Path tempDir) {
  File myFile = tempDir.resolve("file.txt");
  myFile.createNewFile();
  MyClass c = new MyClass();
  assert(c.performTask(myFile));
}
```

Source: research data

The assertThrows method — JUnit 5.0.0 (09/2017) — enables test methods to verify exceptions more clearly and straightforwardly compared to previous exception verification JUnit features. It dismisses the @expected annotation, rule objects, or Try/Catch blocks, improving test code clarity and maintainability. Source Code 10 presents an example.

Source Code 10 – JUnit 5 Exception Verification feature

```java
@Test
public void test() {
  MyClass c = new MyClass();
  Exception e = assertThrows(ExpectedException.class, () -> c.throwExpectedException());
  assertEquals("Error Message", e.getMessage());
}
```

Source: research data

The Resource Lock experimental feature — available since version 5.3.0 (09/2018) — provides synchronized access to shared resources when test containers are in parallel execution mode. The shared resource and the access mode required by each test are informed in the `@ResourceLock` annotation, ensuring reliable test execution with no two tests that write to the same resource being executed concurrently. An example of this feature is provided by Source Code 11.

Source Code 11 – JUnit 5 Resource Lock feature

```
@Test
@ResourceLock(value = SYSTEM_PROPERTIES, mode = READ_WRITE)
public void test() {
  System.setProperty(...);
  assert(...);
}
```

Source: research data

As presented in our motivating example (see Chapter 2.2), the parameterization of test methods — JUnit 5.0.0 (09/2017) — allows developers to test different values using a single test method. Source Code 12 presents the Parameterized Tests feature. In this example, the task parameter is instantiated to every value in the `strings` property of the `@ValueSource` annotation. Optionally, it is possible to print the parameter values in use by adding them to the name property of the `@ParameterizedTest` annotation(BECHTOLD et al., 2020), which helps tracing failed test executions.

Source Code 12 – JUnit 5 Parameterized Tests feature

```
@ParameterizedTest
@ValueSource(strings = {"a","b","c", ... ,"n"})
void test(String task) {
  assert(MyClass.performTask(task));
}
```

Source: research data

4.2.2 Transformations

Before proceeding, we present definitions and justify the terminology we adopted throughout this work regarding test smell correction. As mentioned (Chapter 1), test code refactorings are the most commonly applied operation for test smell correction. Differently from code refactoring, which focuses on observable behavior (FOWLER, 2018), Deursen et al. (2001) defines test refactoring as *"changes (transformations) of test code that: (1) do not add or remove test cases, and (2) make test code better understandable/readable and/or maintainable."*

Considering (1), in JUnit, a test is implemented through an assertion and a test method with many assertions naturally performs many tests. As the propositions we present in this section do not add or remove assertions, they do not disrespect the definition of test code refactoring. Regarding (2), although Deursen et al. (2001) definition focuses the test code improvements on readability and maintainability, the literature shows that other improvements are also desirable: Fowler (2011) explains that indeterminism (flakiness) in tests is a bad characteristic which makes them *"useless for their purpose"* and Luo et al. (2014) shows developers' effort on eliminating indeterminism to increase tests reliability. Removing indeterminism, for instance, is one of the benefits of adopting our proposals. Even with some support from the literature (GUERRA; FERNANDES, 2007; MURPHY-HILL; BLACK, 2008), as our proposals do not strictly adhere to Deursen et al. (2001) definition, we will present them as test code transformations.

We now present a list of test smells and, for every test smell, we provide definition, the already existing refactoring(s), a template of the test smell transformation, an example of smelly code found in the analyzed projects, other applicable test smells (if any), and the improvements of the proposed approach.

4.2.2.1 Assertion Roulette

The Assertion Roulette is defined by Deursen et al. (2001) as a collection of unexplained assertions in a single test method that, in the event of a test failure, challenge tracing which exact assertion had a problem.

Existing Refactorings: (i) The *"Add Assertion Explanation"* (DEURSEN et al., 2001) operation helps to identify the failed assertion but still lacks the execution result of the remaining assertions, and (ii) The creation of *"Single-Condition Tests"* (MESZAROS, 2007) could generate test code duplication across test methods, concerning test fixtures, to less observant developers.

Template: We propose Transformation 1 to the Assertion Roulette test smell. The left-hand side (LHS) contains a `public` (or default access modifier) test T, no return type, optionally parameterized (parameters omitted for simplicity), annotated with `@Test` or any test annotation variation. The test contains any optional set of statements $stmt$ followed by a set of n sequential assertions A, finalized by a set of optional statements $stmt'$. The right-hand side (RHS) removes the test smell using the JUnit 5 Grouped Assertions.

Transformation 1 – Removing the Assertion Roulette

```
@Test
void $T$() {
  $stmt$
  $A_1$
  ...
  $A_n$
  $stmt'$
}
```

$\longrightarrow$

```
@Test
void $T$() {
  $stmt$
  assertAll(
    () -> $A_1$
    ...
    () -> $A_n$
  );
  $stmt'$
}
```

Source: research data

Example: Figure 9a presents an example of the Assertion Roulette extracted from the Spring Framework[8] project. In this example, $stmt$ and $stmt'$ are empty. Moreover, we have four assertions A_1–A_4 related to lines 3–8, respectively. Figure 9b presents the original code rewritten according to Transformation 1.

Figure 9 – Example of the Assertion Roulette

```java
1  @Test
2  void aggregationOptionsShouldSetOptionsAccordingly() {
3    assertThat(aggregationOptions.isAllowDiskUse()).isTrue();
4    assertThat(aggregationOptions.isExplain()).isTrue();
5    assertThat(aggregationOptions.getCursor()).contains(new Document("batchSize", 1));
6    assertThat(aggregationOptions.getHint()).contains(dummyHint);
7  }
```

(a) Original

```java
1  @Test
2  void aggregationOptionsShouldSetOptionsAccordingly() {
3    assertAll(
4      () -> assertThat(aggregationOptions.isAllowDiskUse()).isTrue(),
5      () -> assertThat(aggregationOptions.isExplain()).isTrue(),
6      () -> assertThat(aggregationOptions.getCursor().get()).isEqualTo(new Document("
           batchSize", 1)),
7      () -> assertThat(aggregationOptions.getHint()).contains(dummyHint)
8    );
9  }
```

(b) Refactored

Source: research data

Improvements: Before the transformation, a test containing the Assertion Roulette test smell provides difficulties tracing the failed assertion. After the transformation, all assertions in a test method will be executed, and their individual results (passed/failed) will be reported together, eliminating the test smell.

[8] <https://git.io/JLczX>

4.2.2.2 Conditional Test Logic - Decision Structures

The Conditional Test Logic is defined as a test containing code that may or may not be executed due to branching logic (decision and repetition structures) (MESZAROS, 2007). There is a false impression that no further verification of a passed test containing branching logic is necessary. However, the branching logic may cause the test to finish without executing the intended assertion(s), which is a skipped test. In such a case, the *"passed"* test outcome is unreliable.

Existing Refactorings: The *"Add guarded assertion"* (MESZAROS, 2007) operation fails the test if there are no conditions to its execution, producing a false *"failed"* test outcome. This false outcome prevents Continuous Integration and Continuous Delivery (CI/CD) tools from successfully finishing their tasks.

Transformation 2 – Removing the Conditional Test Logic with Conditional Test Execution

```
@Test
void T() {
  if (C) {
    stmt
  }
}
```

$\longrightarrow$

```
boolean isC() {
  return C;
}

@Test
@EnableIf("isC")
void T() {
  stmt
}
```

Provided
1) Variables in C do not appear in $stmt$;

Source: research data

Template: We propose Transformation 2 to the Conditional Test Logic test smell formed by a decision structure. The LHS contains a `public` (or default access modifier), no return type, unparameterized test T, annotated with `@Test`, or any test annotation variation. The test method contains condition a C and a decision structure enclosing a set of statements $stmt$. Following the notation proposed by Borba et al. (BORBA; SAMPAIO; CORNÉLIO, 2003), variables in C must not appear in $stmt$. The RHS template removes the test smell using JUnit 5 Conditional Test Execution, enabling the condition to be verified in an external method, assigned via the `@EnabledIf` annotation.

Example: Figure 10a presents an example of the Conditional Test Logic test smell observed in the Dropwizard[9] project. The example has condition C found in line 3 and an enclosed set

[9] <https://git.io/JtsMX>

of statements $stmt$ containing an assertion (lines 4–6). According to Transformation 2, the example is transformed to the code presented in Figure 10b, which removes the test smell.

Figure 10 – Example of the Conditional Test Logic

```
1  @Test
2  void returnsASetOfErrorsForAnObject() throws Exception {
3    if ("en".equals(Locale.getDefault().getLanguage())) {
4      ...
5      assertThat(errors).containsOnly("outOfRange must be between 10 and 30 MINUTES");
6    }
7  }
```

(a) Original

```
1  @Test
2  @EnableIf("isEnglishLocale")
3  void returnsASetOfErrorsForAnObject() throws Exception {
4    ...
5    assertThat(errors).containsOnly("outOfRange must be between 10 and 30 MINUTES");
6  }
7
8  public boolean isEnglishLocale() {
9    return "en".equals(Locale.getDefault().getLanguage());
10 }
```

(b) Refactored

Source: research data

Improvements: Before the transformation, test methods with the Conditional Test Logic smell can produce false *"passed"* results when the intended assertion is skipped. After the transformation, skipped tests are correctly identified, which increases the reliability of test outcomes and the separation of the conditional from the test improves the test code maintainability.

4.2.2.3 Conditional Test Logic - Repetition Structures

Some cases of the Conditional Test Logic test smell happen through repetition structures. A failed assertion inside a repetition structure will exit the test method execution, but it could make sense to keep testing the remaining values, for example, elements of a collection.

Existing Refactorings: Meszaros (MESZAROS, 2007) suggested encapsulating this test logic in a Test Utility Method with an Intent-Revealing Name, which would still miss the possibility of independent test method executions to collection elements.

Template: We propose Transformation 3 to the Conditional Test Logic test smell when the test steps are within a repetition structure. The LHS contains a `public` (or default access modifier), no return type, unparameterized test T, annotated with `@Test`, containing a set of statements $stmt$ enclosed in a repetition structure (here presented as a `for` loop). According to the provided condition — shown below the LHS in Transformation 3 — the value n must not appear in $stmt$ (BORBA; SAMPAIO; CORNÉLIO, 2003). The RHS removes the test smell using the JUnit 5 Repeated Tests feature.

Transformation 3 – Removing the Conditional Test Logic with Repeated Tests

```
@Test                          @RepeatedTest(n)
void T() {                     void T() {
  for (n) {         ⟶            stmt
    stmt                       }
  }
}
```

Provided
1) n does not appear in $stmt$

Source: research data

Example: Figure 11 presents an example extracted from the Aeron[10] project. Listing 11a shows a repetition structure enclosing a set of statements in lines 3–7 and repeating the whole test 100.000 times ($n = 100000$), which are transformed — according to Transformation 3 — to the code found in Figure 11b.

Figure 11 – Example of the Conditional Logic transformed using Repeated Tests

```
1 @Test
2 public void shouldNotExceedTmaxBackoff() {
3   for (int i = 0; i < 100000; i++) {
4     double delay = generator.generateNewOptimalDelay();
5     assertThat(delay, lessThanOrEqualTo((double) MAX_BACKOFF));
6   }
7 }
```

(a) Original

```
@RepeatedTest(100000)
void shouldNotExceedTmaxBackoff() {
  double delay = generator.generateNewOptimalDelay();
  assertThat(delay, lessThanOrEqualTo((double) MAX_BACKOFF));
}
```

(b) Refactored

Source: research data

Improvements: Before the transformation, test methods with the Conditional Test Logic smell implemented through a repetition structure do not execute all the intended assertions (fail-first design). After the transformation, all assertions will be independently executed.

[10] <https://git.io/J3Uo0>

4.2.2.4 Test Code Duplication

Already defined in the motivating example (Chapter 2.2), the Test Code Duplication smell presents duplicated steps, either in the same test class or across test classes.

Existing Refactorings: When the duplicated code is in the same test class, it is common to apply the *"Extract Method"* (FOWLER, 2018) refactoring.

Template: We propose Transformation 4 to remove the Test Code Duplication test smell. The LHS contains a `public` (or default access modifier), no return type, unparameterized test T, annotated with `@Test`. The test method contains n groups of repeated statements $stmt$. The RHS removes the test smell using JUnit 5 Repeated Tests.

Transformation 4 – Removing the Test Code Duplication

```
@Test
void T() {
  // Repetition 1
  stmt
  ...
  // Repetition n
  stmt
}
```

$\longrightarrow$

```
@RepeatedTest(n)
void T() {
  stmt
}
```

Source: research data

Example: Figure 12a presents an example extracted from the Dropwizard[11] project. There are 2 sets ($n = 2$) of statements $stmt$ (lines 3–6 and 8–11). Figure 12b presents the example transformed according to Transformation 4.

[11] <https://git.io/JtZmL>

Figure 12 – Example of the Test Code Duplication rewritten with Repeated Tests

```
@Test
public void multipleTestingOfSameClass() {
  assertThat(ConstraintViolations.format(validator.validate(new CorrectExample()))).
      isEmpty();
  assertThat(TestLoggerFactory.getAllLoggingEvents()).isEmpty();

  assertThat(ConstraintViolations.format(validator.validate(new CorrectExample()))).
      isEmpty();
  assertThat(TestLoggerFactory.getAllLoggingEvents()).isEmpty();
}
```

(a) Original

```
@RepeatedTest(2)
void multipleTestingOfSameClass() {
  assertThat(ConstraintViolations.format(validator.validate(new CorrectExample()))).
      isEmpty();
  assertThat(TestLoggerFactory.getAllLoggingEvents()).isEmpty();
}
```

(b) Refactored

Source: research data

Improvements: Before the transformation, a test containing the Test Code Duplication smell favors the introduction of errors due to the presence of duplicated code and the incomplete execution of all assertions (fail-first). After the transformation, all assertions will be independently executed, and the elimination of the duplicated test code improves its maintainability.

4.2.2.5 Mystery Guest

The Mystery Guest test smell is defined by van Deursen *et al.* (DEURSEN et al., 2001) as to occur when a test is not self-contained due to the use of external resources. This use introduces hidden dependencies to the state, consistency, and availability of the external resources. The authors also state that the chances for this test smell increase as more tests use the same resource.

Existing Refactorings: The *"Setup External Resource"* (DEURSEN et al., 2001) removes hidden dependencies when the external resource is needed but does not prevent it from being accessed in a multi-threaded context.

Template: We propose Transformation 5 to the Mystery Guest test smell when the mystery guest is an external file. The LHS contains a `public` (or default access modifier) test T, no return type, unparameterized, annotated with `@Test`, or any test annotation variation. The test contains a set of optional statements $stmt$, commands for file creation, which use a list

Transformation 5 – Removing the Mystery Guest

```
@Test
void T() {
  stmt
  File.createTempFile(params)
  stmt′
}
```

$\longrightarrow$

```
@Test
void T(@TempDir File D) {
  stmt
  D.createTempFile(params)
  stmt′
}
```

Source: research data

of parameters $params$ for temporary files creation, and another set of optional statements $stmt'$. The RHS side removes the test smell using the JUnit 5 Temporary Directory feature. Note that the transformation annotates a temporary directory D and considers its use in the external file instantiation steps. Other analogous template declarations would consider new `File(`D`,`$name$`)` and `File.createNewFile()` to instantiate the external resource, which we omitted to avoid text from becoming fatiguing.

Example: Figure 13a presents an example extracted from Oracle's OpenGrok[12] project. Lines 4 and 5 represent the set of statements $stmt$ that precede the external file instantiation step declared in line 6, and the remaining steps represent the additional set of statements $stmt'$. Listing 13b shows the temporary directory created in line 2 and the temporary file instantiation considering the temporary directory in line 6.

Figure 13 – Example of Mystery Guest

```
1  @Test
2  public void emptyingATryBlockWithTwoHandlers() throws IOException {
3    manipulator = OptimizerTester.getGraphManipulator(CLASS_NAME, "tryBlockWithTwoCatches()"
         );
4    File out = File.createTempFile("test", "simplify");
5    classManager.getDexBuilder().writeTo(new FileDataStore(out));
6    out.delete();
7  }
```

(a) Original

```
1  @Test
2  public void emptyingATryBlockWithTwoHandlers(@TempDir File tempDir) throws IOException {
3    manipulator = OptimizerTester.getGraphManipulator(CLASS_NAME, "tryBlockWithTwoCatches()"
         );
4    File out = tempDir.createTempFile("test", "simplify");
5    classManager.getDexBuilder().writeTo(new FileDataStore(out));
6  }
```

(b) Refactored

Source: research data

Other applicable test smells: The Temporary Directory feature can also refactor the Resource Optimism, the Resource Leakage, and the Interacting Test Suites test smells (MESZAROS,

[12] <https://git.io/JtZec>

2007). The Resource Optimism test smell happens when test methods do not verify the existence, or the state, of external resources like files and databases. This optimistic absence of external resource verifications may cause non-deterministic behavior to a test outcome (DEURSEN et al., 2001). A Resource Leakage happens whenever a unit test fails to acquire or release one or more of its resources properly (MESZAROS, 2007). The Coupling Between Test Methods is present when test methods are not isolated from each other, sharing maintenance activities (BUGAYENKO, 2015).

Improvements: Before applying Transformation 5, tests that manipulate external file resources and present the Mystery Guest smell provide weak memory management since the resources are cleaned only when the JVM stops. the transformation corrects this JVM behavior by using test framework features, which improves the test code maintainability.

4.2.2.6 Exception Handling

Defined by Meszaros (MESZAROS, 2007) as the Expected Exception Test, this test smell occurs when language-based structures lead test execution to error-handling code that manually handles the final test outcome. Peruma *et al.* (PERUMA et al., 2019) define it as the Exception Handling test smell.

Existing Refactorings: Both Meszaros (MESZAROS, 2007) and Peruma *et al.* (PERUMA et al., 2019) suggest using framework-specific features to handle exception testing. JUnit 4 provides two features to that end: the `@ExpectedException` annotation and the Rule check. Previous approaches (SOARES et al., 2020; KIM et al., 2021a) show developers find the first feature unsafe — it is impossible to track which test step raises the exception — and unaware of the second feature.

Template: We propose Transformation 6 to the Exception Handling test smell. The LHS contains a `public` (or default access modifier), no return type, optionally parameterized, annotated with `@Test` (or any test annotation variation) test T. The test contains an optional set of statements $stmt$ followed by a `try/catch` block containing a set of statements $stmt'$ and a manual call to the `fail()` method in the `try` block, and optional exception verification steps evs in the `catch` block, followed by optional statements $stmt''$. According to the provided condition (illustrated below the LHS in Transformation 6), a statement $stmt'_i$, from $stmt'$, raises the expected exception E. The RHS eliminates the test smell using the JUnit 5 Exception Handling feature.

Transformation 6 – Removing the Exception Handling

```
@Test
void T() {
  stmt
  try {
    stmt'
    fail();
  } catch (E) {
    evs
  }
  stmt''
}
```

$\longrightarrow$

```
@Test
void T() {
  stmt
  stmt'-stmt'_i
  assertThrows(E, () -> stmt'_i);
  evs
  stmt''
}
```

Provided
$stmt'_i \in stmt'$ and raises the exception E

Source: research data

Example: Figure 14 presents an example of the Exception Handling test smell extracted from the JMeter[13] project. In Figure 14a, line 3 represents the statements $stmt$, line 5 is both the optional set of statements $stmt'$ and the exception raising step $stmt'_i$, and the expected exception E is presented in line 7 with no optional exception verification (evs) or further optional $stmt''$ steps.

Figure 14 – Example of Exception Handling

```
1 @Test
2 public void test4() throws Exception {
3   ...
4   try {
5     xb.closeTag("abcd");
6     fail();
7   } catch (IllegalArgumentException e) {
8   }
9 }
```

(a) Original

```
@Test
void test4() throws Exception {
  ...
  assertThrows(IllegalArgumentException.class, () -> xb.closeTag("abcd"));
}
```

(b) Refactored

Source: research data

Improvements: Before the transformation, tests that present the Exception Handling smell contain an error-handling code that manually defines the test outcome, which favors the introduction of errors. This fragility is eliminated after applying Transformation 6, which improves the test code maintainability.

[13] <https://git.io/J3TiK>

4.2.2.7 Test Run War

According to van Deursen *et al.* (DEURSEN et al., 2001), this test smell arises when multiple test execution threads cause resource interference between each other. Initially exemplified with non-unique temporary files, the test run war can also be generalized to using and allocating the system's resources.

Existing Refactorings: "Make Resource Unique" (DEURSEN et al., 2001). However, this operation is not possible when dealing with system properties. The cases where the shared resources are system files or external resources can be analyzed and transformed according to the Temporary Directory feature description, presented in Section 4.2.2.5.

Template: We propose Transformation 7 to the Test Run War test smell when the shared resource is a system property. The LHS contains a class C annotated to execute its tests concurrently through the `@Execution(CONCURRENT)` annotation. The test class also contains a `public` (or default access modifier) test T, no return type, optionally parameterized — parameters omitted for simplicity — and annotated with `@Test` or any other test annotation variation, having any set of statements $stmt$ with at least one use of `System.setProperty()`. The RHS removes the test smell using the JUnit 5 Resource Lock feature.

Transformation 7 – Removing the Test Run War

```
@Execution(CONCURRENT)
Class C {
  ...
  @Test
  void T() {
    stmt
  }
  ...
}
```

$\longrightarrow$

```
@Execution(CONCURRENT)
Class C {
  ...
  @Test
  @ResourceLock(value=SYS_PROPS, mode=READ_WRITE)
  void T() {
    stmt
  }
  ...
}
```

Provided
$stmt$ contains at least one `System.setProperty(...)`

Source: research data

Example: Figure 15a presents a partial example — it lacks the definition of parallel execution in the test class declaration — of the Test Run War test smell found in the Cryptomator[14] project. Line 3 represents the instruction that sets a system property value, and the remaining method contents are the set of statements $stmt$. Figure 15b declares the test to modify the system properties in line 2, which prevents its parallel execution with other tests that also modify the system properties.

[14] <https://git.io/JLWby>

Figure 15 – Example of Test Run War

```java
@Test
public void testEmptyList() {
  System.setProperty("test.path.property", "");
  List<Path> result = env.getPaths("test.path.property").collect(Collectors.toList());
  MatcherAssert.assertThat(result, Matchers.hasSize(0));
}
```

(a) Original

```java
@Test
@ResourceLock(value=SYSTEM_PROPERTIES, mode=READ_WRITE)
void testEmptyList() {
  System.setProperty("test.path.property", "");
  List<Path> result = env.getPaths("test.path.property").collect(Collectors.toList());
  MatcherAssert.assertThat(result, Matchers.hasSize(0));
}
```

(b) Refactored

Source: research data

Improvements: Before applying Transformation 7, tests with the Test Run War smell present flaky behavior due to concurrent modification and access to shared resources. the transformation eliminates the non-determinism (flakiness) without changing the original test method content.

4.2.2.8 Duplicate Assert

The Duplicate Assert test smell is defined as *"when a test method tests for the same condition multiple times within the same test method"* (PERUMA et al., 2019). This test smell is detailed in our motivating example in Chapter 2.2.

Existing Refactoring: *"Create new test methods with different values"* (PERUMA et al., 2019), also detailed in Chapter 2.2.

Template: We propose Transformation 8 to the Duplicate Assert test smell. The LHS contains a `public` — or default access modifier — test T, no return type, unparameterized, annotated with `@Test`, containing m sets of statements $stmt_1$ to $stmt_n$ that make use of parameterizable values lists $P_1..W_1$ to $P_m..W_m$. The RHS uses the JUnit 5 Parameterized Tests, where the parameterizable values lists are declared in a specific annotation, and the test method is executed m independent times.

Example: Our motivating example in Source Code 2 presents an example of the Duplicate Assert test smell. There are 3 sets ($m = 3$) of 4 statements $stmt$ ($n = 4$): lines 3–6, 8–12, and 14–18. The parameterizable values list is composed of three values: P_1 is a String to represent

Transformation 8 – Removing the Duplicate Assert

```
@Test
void $T$() {
   $stmt_1(P_1)$
      ...
   $stmt_n(W_1)$
   ...
   $stmt_1(P_m)$
      ...
   $stmt_n(W_m)$
}
```

$\longrightarrow$

```
@ParameterizedTest
@CsvSource({
     "$P_1$, ... , $W_1$",
     ...
     "$P_m$, ... , $W_m$"
})
void $T$($P$, ... , $W$) {
   $stmt_1(P)$
      ...
   $stmt_n(W)$
}
```

Provided
$\exists i \in [1..n] \mid stmt_i$ contains an assertion

Source: research data

the file Path (lines 3, 8, and 14), a parameter Q_1 is a String to the parent directory (lines 4, 9, and 15), and W_1 is a String to the sub-parent directory (lines 5, 11, and 17). Source Code 5 presents the transformed test method, according to Transformation 8.

Other applicable test smells: The Parameterized Tests feature can also refactor the *Test Code Duplication* and the *Lazy Test* smells. The latter happens when several test methods use the same fixtures and call the same production method with variations only in the passed parameters (DEURSEN et al., 2001).

Improvements: Before applying the transformation, tests having the Duplicate Assert smell present code duplication, which facilitate the introduction of errors. Such fragilities are corrected after applying the Transformation 8, which removes the code duplication, thus improving the test code maintainability, and executes all assertions.

4.2.2.9 Discussion

In the previous section, we presented a non-exhaustive list of transformations based on JUnit 5 features which helps removing 13 different test smells and brings several improvements to both test cases and test code. We summarize the test smells that can benefit from our transformations and the introduced JUnit 5 features in Table 9.

We divide the improvements brought by our transformations into test case and test code. The test case improvements are related to test execution and results (run all assertions and remove test flakiness); the test code improvements promote non-functional benefits (improving readability and/or maintainability). Table 10 summarizes the improvements promoted by our transformations.

Table 9 – Test framework features and their applicable test smell refactorings

Feature	**Test Smell**
Grouped Assertions	Assertion Roulette
Parameterized Tests	Duplicate Assert, Test Code Duplication, Lazy Test
Conditional Test Execution	Conditional Test Logic
Temporary Directory*	Mystery Guest, Resource Optimism, Resource Leak, Interacting Test Suites, Coupling between test methods
Repeated Tests	Test Code Duplication, Conditional Test Logic
Exception API	Exception Handling
Resource Lock*	Test Run War

* Experimental Features

Source: research data

Table 10 – Improvements of the proposed transformations

	Improvements		
	Test Case		**Test Code**
	Run All Assertions	**Remove Test Flakiness**	**Improve Code Maintainability**
T1	✓		
T2			✓
T3	✓		
T4	✓		✓
T5			✓
T6			✓
T7		✓	
T8	✓		✓

Source: research data

4.3 EVALUATION

This section presents our evaluation studies: an online survey with developers and the submission of contributions (Pull Requests) to popular open-source projects on GitHub. The development of both studies aims to answer the research questions **RQ$_4$: "How do developers perceive our test code transformations to eliminate smells?"** and **RQ$_5$: "To what extent do open-source developers accept our transformations in their projects?"**

4.3.1 Online Survey

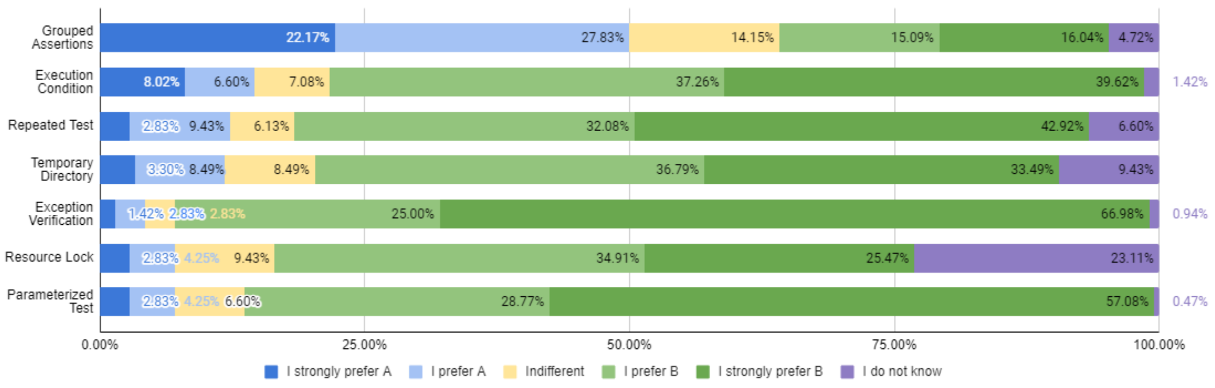
This study planned to assess developers' opinions about the transformations we proposed in Chapter 4.2.2 through an online survey. By choosing between two code snippets, one initially found in a current open-source project and the other transformed according to our proposals, along with the developers' comments on their answers, we would be able to validate whether the respondents were aware of the benefits achieved by the transformations.

We assembled the survey with 11 questions: four concerning demographics and experience and seven questions corresponding to the code samples they would evaluate. The code samples were the original and transformed ones presented in Chapter 4.2 and the motivating example from Chapter 2.2 (Source Codes 2 and 5).

Considering that side A was the original test code and side B the transformed one, we presented developers with the following answering options (unique): *"I strongly prefer A"*, *"I prefer A"*, *"Indifferent"*, *"I strongly prefer B"*, *"I prefer B"*, *"I do not know"*. Also, every question had comments field. We published the survey invitation on open developers forums, professional networks, and developer mailing groups.

We performed the survey in December 2021, achieving 212 responses. Concerning the demographics, we had participants working in 39 countries, where 91% defined their primary work area as the industry (over academia). Also, 66% of the respondents declared to have 10+ years of experience with software testing — and about 85% if we consider 5+ years. The average of their daily time spent on testing activities is 40%. Figure 16 details the obtained results concerning the developers' preferences about the code samples.

Figure 16 – Online survey results



Source: research data

Both numeric and comment analyses are fundamental to interpret the obtained results. The numeric results indicate whether developers prefer the original or the transformed test code version, while the comment analysis answers if they are doing so because they correctly recognize the proposition to fix the present test smell through a JUnit 5 feature.

The transformation based on the Grouped Assertions feature (Figure 9) divided developers' opinions. While 69% of the respondents stated to prefer side A, to be indifferent or not knowing how to answer the question, their comments' analysis revealed that 26% did not know the `assertAll()` method and its behavior. Other 45% affirmed the use of lambdas to make the test code more verbose, thus hard to maintain. Such affirmatives can be found in developers' comments like *"Haven't used assertAll, so I don't know the differences, I suspect that all the assertions are run with it even if one assertion fails, I will check this to add to my repertoire, so I prefer B but mark it as I do not know."* and *"B is way too complex for no good reason. So many lambdas seems like a lot cognitive overhead."* Oppositely, the developers who preferred the transformed version agreed to the benefits of reporting the results of the individual assertions together. These advantages can be found in comments like *"Reports all failures at once, reducing time to feedback."* and *"With the first implementation, one error can hide 3 others."* Despite dividing developers, as 71% of the respondents who did not prefer the transformed code version did so because they dislike the feature syntax or do not know its behavior, their answers do not invalidate the benefits ratified by the developers who chose the transformed test code.

The Execution Condition feature (Figure 10) was the choice of about 77% of the developers. Code reuse, separation of concerns, elimination of false positives, and readability were positive comments. Such advantages can be found in comments like *"Separation of concerns: the 'if'-statement itself is not part of the test."* and *"Much more readable, implementing other test depending on the same property doesn't require code duplication"*. A frequent disadvantage concerned the syntax of the execution condition feature, which includes declaring the condition method name as a string parameter to an annotation, thus difficulting method renaming operations. This disadvantage is found in comments such as the following ones: *"The annotation binds a method using its name in a string, adding another source of error."* and *"A makes the intent clear. I prefer language syntax over annotation magic."*

Our third proposed transformation, which used the Repeated Test feature (Figure 12), was also the choice of the majority of the respondents. Such respondents highlighted benefits like ease of debugging, code clarity, and maintenance in comments like *"Declarative is usually more readable. Decouples data and logic."* and *"Easier to adjust and avoids index errors."* There were concerns about computational cost, single-threaded execution, and verbose output among the respondents who did not prefer the transformed version. Such concerns are present in the following comments: *"In B the test-start, test-stop, record-results code may push so much stuff out of CPU caches etc it may perform no better than 100 isolated runs would perform."* and *"Option B is more declarative, but I don't believe it is likely useful or informative to report*

100 repetitions of the test into the test results."

The developers that agreed to Transformation 5 — Temporary Directory feature (Figure 13) — highlighted the advantages of cleaner code, separating concerns, and ensuring environment clean up after every test execution independently of its result. Such information is found in comments like *"Avoids non-test boilerplate, ensures that the temporary file is cleaned up directly after the test."* and *"Making and cleaning up temporary files can be annoying and tricky to get right, especially across different platforms. A delegates the work to JUnit, which I prefer."*. The comments of the respondents who did not explicitly prefer the transformed code evidenced unfamiliarity with the feature use, as seen in *"A seems to be trying to clean after itself so that the test is repeatable."* and *"It's not clear what this tests, though test A gets a bonus point for cleaning itself up and preventing the test from failing on a subsequent run."*.

The transformation based on the Exception Verification (Figure 14) feature was our most popular proposal, reaching about 92% acceptance between respondents. Conciseness, better readability, and using the framework features are positive comments. We present some of these comments as follows: *"B is a clearer and more concise, it makes use of the tools that JUnit already provides."* and *"A does not really test for the exception to be thrown. If the exception is part of the contract it should be properly asserted."*. The use of lambdas is also a source of criticism between the non-adherents, as seen in *"Sometimes lambda turns the code more confuse."* and *"It's all down to how easy it is to read the test. A is quite explicit that we are expecting the Exception - B takes a bit more reading to understand what's happening."*

The transformation based on the Resource Lock (Figure 15) contained the feature most unknown to developers. About 40% of the respondents did not prefer the transformed code, and their comments' analysis revealed that 60% of them were unfamiliar with the feature. Other concerns regard impact on runtime, as seen on *"B feels safer, but it could also be a performance bottleneck in the test codebase, since it seems to be obtaining an exclusive lock on the Java system properties. I would default to B but be wary of the impact on test runtime."*. Positive feedback received from about 60% of the respondents highlighted the benefit of safety in parallel executions and synchronized access to shared resources, as in *"Avoids flaky tests due to concurrent system property changes; clearly indicates that shared/static state is modified."* and *"For tests running in parallel, synchronization of access to a shared resource is required."*.

The developers also agreed to the transformation of our motivating example, which uses JUnit 5 Parameterized Tests (Source Code 5). Separation of concerns, avoidance of code duplication, ease of maintenance, and improved readability are among the benefits stated by the respondents, as in *"Simpler, less repetition, test cases more easily identified, easier to maintain, clearer intent."* and *"B is DRYer and B runs all the test cases even if one of them fails. It's probably easier to tell which case has failed (A only gives the line number in addition to actual/expect result)."* Respondents who did not opt for the transformed code showed unfamiliarity with the feature in their comments, also highlighting difficulties in its syntax, as in *"B is tidy but relies on annotations that I've never used. The less "magic", the more clear my test code, the better."* and *"Setting breakpoints on a certain data row in B is cumbersome. It's not clear how the CSV is parsed, how are spaces and commas handled, or need to be escaped. I don't see on a single view what happens."*.

Summary: The online survey shows that developers mostly prefer our transformation proposals to eliminate undesired test behavior and benefit the test code. Their comments show several additional benefits such as separation of concerns, ease of maintenance, avoidance of code duplication, improved readability, and the elimination of false positives. Among the highlighted disadvantages, most developers claim the verbose use of lambdas and the insecurity of string-based parameters in annotations —— posed by Java syntax and therefore some JUnit 5 features —— challenging to adopt some transformations. Additionally, many developers are still unfamiliar with some JUnit 5 features. As stated in their comments, the lack of knowledge on the new features led them to prefer the non-transformed code side, abdicating the advantages brought by the transformations.

4.3.2 Pull Requests

The overall planning of this study is to validate the transformations proposed in Chapter 4.2. We will use the submission of Pull Requests as our second validation strategy. The necessary steps to accomplish this planning are (i) to retrieve a list of GitHub projects that use the JUnit 5 library, (ii) find test smells in such projects, (iii) refactor the identified test smells according to our transformations presented in Chapter 4.2, (iv) submit Pull Requests, and (v) collect the developers' opinions.

To submit the Pull Requests, we first need to select projects hosted on GitHub. Here, we consider the same 77 projects that used the JUnit 5 library at the time. Afterward, we need to find test smells in such projects to refactor them and submit the Pull Requests. To find the test smells, we rely on an existing tool named tsDetect (PERUMA et al., 2019), which has been used in several previous works (SPADINI et al., 2020; KIM, 2020; PANICHELLA et al., 2020; KIM; CHEN; YANG, 2021; SOARES et al., 2020).

After executing the tsDetect tool against the 77 projects, we have the potential number of opportunities to apply our transformations. Notice, however, that the tsDetect might raise cases where the code does not match the LHS of our proposals. Therefore, the number of transformation opportunities we present in Table 11 represents an upper bound, noted as a threat to validity presented in Chapter 4.3.3.

Table 11 – Transformation opportunities

Transformation	Test Smells
Grouped Assertions	19,399 (Assertion Roulette)
Exception API	23,899 (Exception Handling)
Parameterized Tests	6,766 (Duplicate Assert)
	6,222 (Lazy Tests)
Repeated Tests	6,766 (Duplicate Assert)
	7,434 (Conditional Test Logic)
Conditional Test Execution	7,434 (Conditional Test Logic)
Temporary Directory	1,645 (Mystery Guest)
	1,687 (Resource Optimism)

Source: research data

Our candidate selection criterion used the guidance of tsDetect logs. We manually searched for test smells that fit our generic templates (LHS) in the test classes where tsDetect indicated the presence of a test smell. We present our strategies to select cases from the tsDetect logs as follows:

1) Grouped Assertions: We searched for test classes indicated as containing the Assertion Roulette test smell.

2) Exception API: We searched for test methods containing `try/catch` blocks containing assertions in the `catch` parts, or methods to specifically pass or fail the test method, indicated as Exception Catching Throwing in tsDetect.

3) Parameterized Tests: Methods suitable for such transformation can be found indicated in tsDetect as Duplicate Asserts and Lazy Tests.

4) Repeated Tests: Suitable methods normally contain repetition structures involving all test method steps, named as Conditional Execution is tsDetect, or contain exclusively several copies of the same specific assertion, which are named as Duplicate Asserts.

5) Conditional Test Execution: Indicated as Conditional Test Logic in tsDetect. We selected the ones that conditioned all test method steps execution to a specific value or range.

6) Temporary Directory: Test classes indicated by tsDetect as to have the Mystery Guest or the Resource Optimism test smells.

7) Resource Lock: We analyzed all 15 test classes our crawler engine indicated to use the `@Execution` annotation in the downloaded projects. These test classes would also have to posses test methods writing to system's common resources. We were not successful in this search.

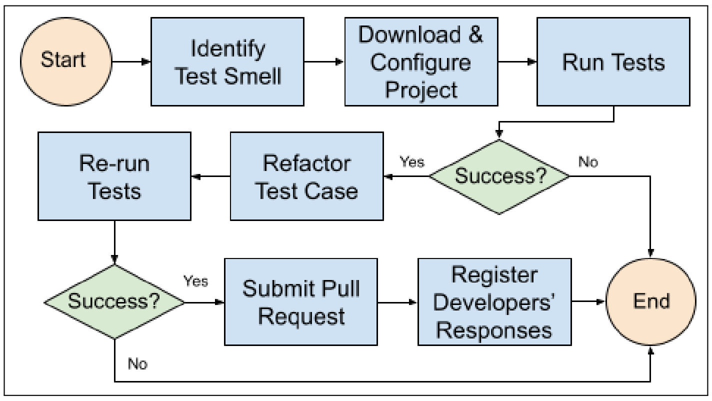
Once a test smell was identified, we followed each project's contribution guidelines to download the project and its dependencies and executed the automated test suites before making any changes. If this step were successful, we would perform the transformation according to the LHS and RHS of the transformation to be applied. We noticed that when another test automation framework is used in conjunction with JUnit, behavioral changes incompatible with our proposed transformations may happen. For this reason, after applying the transformation, we again execute the automated test suites as a final round check to assure that the test code change would not cause a build failure. Lastly, we formalized the contribution in a Pull Request.

We decided to submit only one contribution per project to minimize bias on Pull Request acceptance from the same developer/maintenance team. We transformed only one test smell in a single class in this contribution, even if other test smells were identified. As contributing to open-source project repositories demands non-trivial and resource-consuming tasks (e.g., project-code deployment and the study of contribution guidelines per project), and considering our available resources, we limited each of our proposed transformations in Table 12 to a maximum of 7 Pull Requests.

In addition to sending the Pull Request, we also submitted a comment justifying the transformation that, unless stated differently by specific projects submission rules, followed the pattern of (i) presenting the problem definition, (ii) the description of the transformation (proposed solution), and (iii) the before and after (code snippets) changes.

Finally, when necessary, we discussed with the developers to better explain our submission before accepting or rejecting the submitted Pull Request. Figure 17 summarizes the steps we performed for every Pull Request submission. In the figure, the are two unsuccessful paths: (i) when tests could not be run after project deploy and (ii) when a test execution result was not successful after applying the transformation. We were not able to correctly deploy 10 of the 77 projects identified using JUnit 5 due to problems in environment configuration and missing external dependencies. Despite foreseen in Figure 17, no performed transformation failed a previously passing test method.

Figure 17 – Steps for Pull Request Submission



Source: research data

We submitted a total of 38 Pull Requests. Unfortunately, our established submission rules (one unique submission per project) provided us with only three examples that could be transformed with Temporary Directory. Table 12 presents the distribution of Pull Requests per transformation.

Table 12 – Pull Requests per Transformation

Transformation	**Submitted**	**Accepted**	**Integrated**	**Rejected**	**Submission Error**
Grouped Assertions	7	6	5	0	0
Exception API	7	7	3	0	0
Parameterized Tests	7	5	4	0	1
Repeated Tests	7	5	5	0	1
Conditional Test Execution	7	6	5	1	0
Temporary Directory	3	2	2	1	0
Total	**38**	**31**	**24**	**2**	**2**

Source: research data

Some observations are necessary regarding the summation of those numbers. First, we consider a Pull Request **accepted** when developers manifest concordance with the transformation, not always merging the code due to contribution rules. The latter cases happened when a contribution, instead of transforming all opportunities throughout the project, solely resolved one of them (as defined by our submission rules). Second, an **integrated** Pull Request is an acceptance merged to the main repository code. Third, a **rejected** Pull Request happens when developers disagree with the proposed transformation; and a **submission error** happens when contributions are rejected due to inadequacy to submission guidelines.

Following every open-source project's contribution guidelines and our transformations, we managed to submit 38 Pull Requests. Here, we observed 2 submission errors and 3 submissions had no response. From the 33 responded submissions and considering the 31 accepted and the 2 rejected ones, we achieved 94% of positive evaluations with little to no further discussions. Many accepted contributions were merged (74% of accepted pull requests) to the projects' actual test code, which leads us to conclude that developers agree that our propositions help to produce code without the presence of specific test smells.

We applied our transformations to test methods of different complexity and size. For example, the pull request submitted to the Eclipse Jetty[15] project kept the same test functionality substituting 10 LOC by 5 in a more straightforward code when applying Transformation 6 (JUnit 5 Exception Handling feature). The pull request submitted to the Oracle Opengrok[16] project substituted 28 LOC by 22 and improved the detection of skipped tests with Transformation 2 (Conditional Test Execution).

Table 13 presents a summary of the pull requests we submitted in this study. It also summarizes the developers' comments. Considering the submissions per transformation action, we achieved acceptance rates (among respondents) of 100% for Grouped Assertions, Exception API, Parameterized Tests, and Repeated Tests, 86% acceptance rate for Conditional Test Execution, and 67% for Temporary Directory.

Table 13 – Submitted Pull Requests

No	Project Name	Transformation	Result	Comments
1	JanusGraph	Grouped Assertions	Integrated	Dev1: Assertion code is lengthier. Dev2: Grouping is benefitial Dev3: Didn't know this feature
2	Quarkus	Grouped Assertions	Integrated	Thanks for your contribution
3	Halo	Grouped Assertions	Integrated	Thanks for your contribution
4	Spring PetClinic	Grouped Assertions	Accepted	Thanks for your contribution
5	Spring-Boot-Starter	Grouped Assertions	Open	
6	Spring Boot Admin	Grouped Assertions	Integrated	Thanks for your contribution
7	Cryptomator	Grouped Assertions	Integrated	Mostly a matter of taste, but thanks for your contribution
8	Apache JMeter	Exception API	Accepted	Thanks for your contribution
9	OkHttp	Exception API	Accepted	The use of this API promotes boilerplate reduction
10	LMAX Disruptor	Exception API	Accepted	Thanks for your contribution
11	Seata	Exception API	Integrated	Looks good to me
12	Sentinel	Exception API	Integrated	Looks good to me

15 <https://git.io/J3q6P>
16 <https://git.io/JsAiw>

13	Eclipse Jetty	Exception API	Integrated	Looks good to me
14	Lettuce	Exception API	Accepted	N/A
15	jsoup	Parameterized Tests	Open	
16	Jenkins	Parameterized Tests	Integrated	The feature makes it easier to read the test results
17	CAS	Parameterized Tests	Submission Error	
18	Checkstyle	Parameterized Tests	Accepted	Thanks for your contribution
19	Jodd	Parameterized Tests	Integrated	N/A
20	Mindustry	Parameterized Tests	Integrated	Thanks for your contribution
21	OWASP ZAP	Parameterized Tests	Integrated	Thanks for your contribution
22	Spring Framework	Repeated Tests	Integrated	Thanks for your contribution
23	MyBatis-Plus	Repeated Tests	Integrated	Thanks for your contribution
24	JavaParser	Repeated Tests	Submission Error	
25	Reactor Core	Repeated Tests	Integrated	Good way to repeat when the iteration # doesn't really matter
26	Aeron	Repeated Tests	Integrated	N/A
27	MyBatis Generator	Repeated Tests	Integrated	Thanks. Nice improvement.
28	Dropwizard	Repeated Tests	Open	
29	Oracle OpenGrok	Conditional Test Execution	Integrated	Thanks for your contribution
30	Apache Camel	Conditional Test Execution	Integrated	N/A
31	Apache Dubbo	Conditional Test Execution	Integrated	Looks good to me
32	Java Design Patterns	Conditional Test Execution	Integrated	Looks good to me
33	Apache Kafka	Conditional Test Execution	Accepted	Thanks for your contribution
34	Flowable	Conditional Test Execution	Rejected	Broader tests need if conditions
35	Mybatis-3	Conditional Test Execution	Integrated	Thanks for your contribution
36	Java & Spring Tutorial	Temporary Directory	Rejected	As a tutorial, we prefer to focus only on the File API
37	Simplify	Temporary Directory	Integrated	Thanks for your contribution
38	Zookeeper	Temporary Directory	Integrated	It simplifies test steps

Source: research data

We now discuss our results. First, regarding developers' comments about the Pull Requests, it is vital to notice that most of the merged and accepted ones ended with a simple *"Thanks for your contribution"* message or no message at all, which we registered as *"N/A"*.

One developer of the Reactor Core[17] project commented the following about repeated tests: *"that's a good idea, since the iteration # doesn't really matter even in the case one of the iterations fails"*, corroborating our proposition. As mentioned in literature (KIM et al., 2021a), we could also observe the developer's unawareness of new features, as in the JanusGraph[18]

[17] <https://git.io/JsoWs>
[18] <https://git.io/J3qKi>

project, where one of the project maintainers made the following statement: *"Didn't know this feature"*.

We had two rejected pull requests. In the Flowable[19] project, one developer complained about the use of parameterized tests: *"It is not good approach. It is very compact and very technological but VERY complicated and cryptic. We need easy to read tests and make them as much as possible to be focused on human ability to read them easily"*; the project they were maintaining was migrating to specification through behavior-driven ideas, and maybe that is why the developer quoted the importance of human readability. The second rejection happened in the Java & Spring Tutorials[20] project, where we transformed the test code using the Temporary Directory feature and obtained the developer comment: *"That would be the better way to test this in a real-world scenario. In this case, the code is meant to support a tutorial on working with files in Java, so we prefer to keep it simple and focus only on the File API."*.

We had two submission errors. The first one, to the CAS[21] project, happened due to a lack of observance to project rules for Pull Requests, which was a constant risk and generated a bad submission. The second submission error, submitted to the JavaParser[22] project, was caused due to a lack of understanding of the test behavior, which did not fit our proposed transformation template (LHS) and was not supposed to be transformed.

The acceptance rates per transformation show the potential of our proposals. For instance, all responded Grouped Assertions were accepted, and almost all Conditional Test Execution (6 accepted and 1 rejected) did too. We achieved only 67% for Temporary Directory. However, it is essential to note that we submitted only 3 Pull Requests considering such transformation.

Summary: The 94% acceptance rate of our contributions demonstrates our transformations' potential when considering developers' opinions. Thus, $\mathbf{RQ}_{4.2}$ is positively answered.

4.3.3 Threats to Validity

As an internal threat to validity, the opportunities presented in Table 11 may be overestimated to our transformations. Although the tsDetect (PERUMA et al., 2019) tool indicates test smells, the test code may not match the LHS of our proposed transformations. Concerning

[19] <https://git.io/JsaYt>
[20] <https://git.io/JsMlF>
[21] <https://git.io/J3q6F>
[22] <https://git.io/JsABf>

the submitted pull requests per test smell type, these may not be enough to generalize results per studied test smell.

As an external threat to validity, and considering that our contributions (pull requests), although made to popular projects, do not achieve statistical confidence (II; KOTRLIK; HIGGINS, 2001), our results may not remain consistent in an attempt to generalize beyond the selected project list. Also, we gather answers from software developers of open-source projects only, and this bias may influence the generalization of results to other audiences. We address this threat by considering their experience and geographic distribution worldwide, whose variety may bring better representation of different views to our results.

4.4 RESEARCH DIRECTIONS

Built upon our findings, research directions for test smell removal would involve:

- The number of transformation opportunities shown in Table 11 indicates that manual refactoring is inefficient and suggests an automatic mechanism to be most necessary. In this sense, an effort to extend our contributions both in creating more transformations or implementing them in automatic tools, as already performed by Paula and Bonifácio (2022), seems reasonable.
- The formalism we used to express the transformations in this work can be used, and extended, to express current and further test transformations and refactorings, as already performed by Martins, Costa and Machado (2023).

5 MANUAL TESTS ALSO SMELL

Despite the format differences, bad choices when implementing automatic tests or describing a manual test using natural language may pose similar threats to the testing activity, as exemplified in our motivating example from Section 2.3 and generating the questions from the third knowledge gap presented in Chapter 1.

In this chapter, we present the studies dedicated to the analysis of manual test descriptions through test smells. We first conduct an exploratory study (Section 5.1) to analyze a statistically relevant sample of manual test descriptions of three systems from different domains: (i) the Ubuntu Operational System (OS), which is open-source; (ii) the Brazilian Electronic Voting Machine, in an institutional partnership between the Federal University of Pernambuco (UFPE) and the Superior Electoral Court (TSE); and (iii) a large smartphone manufacturer's UI — name omitted due to non-disclosure of proprietary information agreement —, also in partnership. In this first study, we intend to answer the following research questions:

- **RQ$_6$:"What already proposed natural language test smells can be observed?"**,
- **RQ$_7$:"What new natural language test smells can be observed?"**, and
- **RQ$_8$:"How frequent are these test smells?"**

In Section 5.2, we present our catalog of natural language test smells. In sequence, we conduct an empirical study (Section 5.3) using an online survey to evaluate our catalog. We recruited 24 testing professionals and presented them with our definitions and examples, asking for their agreement level to our propositions. In this study, we answer the following research question:

- **RQ$_9$:"How software testing professionals evaluate our proposed smells?"**

We also contribute to developing an NLP-based tool to identify our catalog's natural language test smells automatically in Section 5.4. Our tool implements our defined rules using spaCy,[1] a *"free, open-source library for industrial-strength Natural Language Processing (NLP) in Python,"*, and its capabilities concerning syntactic analysis (i.e., elements of the sentences and their properties) like verification verbs and declarative sentences, which are

[1] [Online]. Available: <https://spacy.io/>

present in multiple languages and whose implementation can be mostly reused — as we do to Portuguese, used in the tests of the Brazilian electronic voting machine. To evaluate our tool, we conduct one last empirical study presented in Section 5.5 to answer the following research question:

- **RQ$_{10}$:"How precise can the automated discovery of natural language test smells be when using NLP?"**

The survey dataset, tool logs, and tool validation records — the last two documents for Ubuntu OS tests — can be downloaded at our online repository (SOARES et al., 2022a).

5.1 EXPLORING MANUAL TESTS

This section describes how we analyzed natural language test descriptions to prospect test smells. Also, we give further detail on the selected systems, a sample set of tests, and the distribution (frequency) of our findings. In particular, this exploratory study answers **RQ**$_6$, **RQ**$_7$, and **RQ**$_8$.

5.1.1 Settings

This exploratory study aims to prospect a set of manual tests from different systems and gather the identified occurrences of test smells. To increase the representativeness of our results, we selected manual tests written in natural language from important systems of three distinct domains: open-source, government, and industry. Considering the limits imposed by the agreements for non-disclosure of confidential information, we detail the obtained tests as follows:

- **Ubuntu OS:** As open-source software (LTD., 2023), the Ubuntu OS manual tests are available in a public repository.[2] In the repository, test descriptions are written in English (natural language) and XML format, with standardized tags for test suites, test cases, and action and verification steps. In total, 305 test files containing 973 tests are available.
- **Brazilian Electronic Voting Machine (BEVM):** An open-source web-based test management and test execution system manages the manual test descriptions of the

[2] [Online]. Available: <https://git.launchpad.net/ubuntu-manual-tests>

BEVM. In the ecosystem, test descriptions are in Portuguese. In total, we had access to 133 tests exported to HTML format.

- **Large Smartphone Manufacturer (LSM):** The manual test descriptions of this industry partner are managed by a proprietary issue-tracking product that allows bug tracking and agile project management. Manual test descriptions for this system are in English. In total, 898 test descriptions were made available for our analysis and exported to spreadsheet format.

Three independent researchers — the author and two assistants — manually analyzed a randomly selected subset of test descriptions to perform the exploratory study. Using their know-how on test smells for automatic and manual tests, the researchers quoted every questionable description and indicated the possible smell, discussing results in follow-up meetings. It is important to emphasize that access to BEVM and LSM tests was controlled and accessed by cleared researchers only. As to the analysis procedure, all researchers involved in this activity started with the Ubuntu manual tests to achieve standardization of actions, continuing the analysis in the remaining systems according to their access grants.

Concerning the already proposed smells for tests written in natural language, from the existing list of seven test smells (HAUPTMANN et al., 2013), five are identified using metrics from an automatic analysis (JONES, 1972): *Badly Structured Test Suite*, *Inconsistent Wording*, *Hard-Coded values*, *Long Test Steps*, and *Test Clones*. As we intended to manually read test descriptions and take notes of the identified problems, using any tool to generate such metrics was out of scope.

Finally, to make our manual analysis effort feasible and guarantee the best generalization possible, we used Cochran's Sample Size Formula (II; KOTRLIK; HIGGINS, 2001) to calculate the sample needed to obtain an 80% confidence level with a 5% margin of error for each system individually. Table 14 presents the analyzed sample test set per system.

5.1.2 Results

We found similarities in all systems regarding the structure of their manual tests. Although Ubuntu's team does not use a specific test managing tool, they describe their tests as the other two systems, which use open-source and proprietary software for such activities. Figure 18

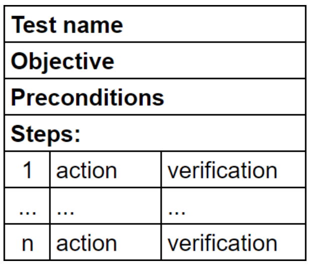
Table 14 – Analyzed sample set of tests per system.

System	Manual tests	Sample size
Ubuntu OS	973	141
BEVM	136	75
LSM	898	139
Total	**2,007**	**355**

Source: research data

presents a test visualization. Table 15 details the test section's writing, regarding the sentence types, with examples from the Ubuntu OS tests.

Figure 18 – Common test design found in the exploratory study



Test name		
Objective		
Preconditions		
Steps:		
1	action	verification
...	...	...
n	action	verification

Source: research data

Table 15 – Common test structure found in the exploratory study.

Section	Sentence type	Example
Objective	Declarative	*This test checks that Audio project menu Works*
Preconditions	Declarative	*VMWare Player version $\geq$ 4.0 is required*
	Imperative	*Ensure that your system has no Internet access before proceeding*
Action	Imperative	*Click the 'Restart now' button*
Verification	Declarative	*An 'Installation Complete' dialog appears*
	Imperative	*Verify the system upgraded correctly*

Source: research data

The exploratory study identified eight test smells, briefly defined in Table 16 and further detailed in Section 5.2. From this list, two smells (*i.e.*, *Ambiguous Test* and *Conditional Test*) are proposals from the literature on natural language test smells (HAUPTMANN et al., 2013), and the remaining ones are contributions from our study. Also, we manually accounted for 447 occurrences of the identified test smells, and Figure 19 presents their distribution per system.

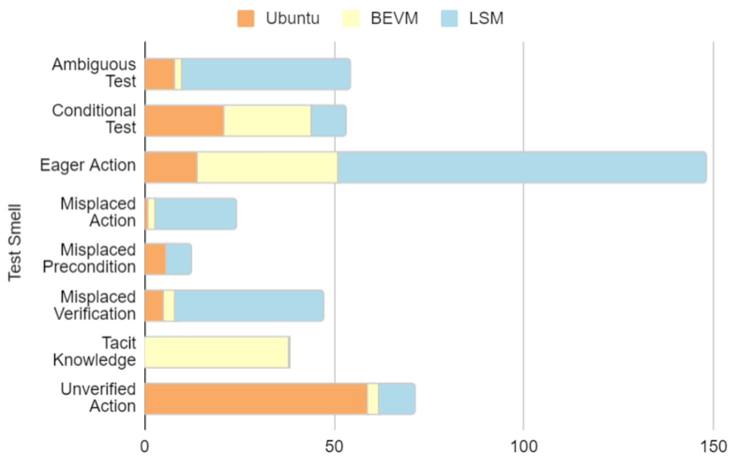
As, in the exploratory study, BEVM and LSM tests were analyzed by one author only, we present the inter-rater reliability concerning the Ubuntu OS tests only, which was the only

Table 16 – Cataloged test smells

Test Smell	Brief definition
Ambiguous Test	Test steps leaving room for interpretation
Conditional Test	Conditional logic phrased in natural language
Eager Action	Single action steps that group multiple actions
Misplaced Action	Action steps written as verification steps
Misplaced Precondition	Preconditions as action steps
Misplaced Verification	Verification steps written as action steps
Tacit Knowledge	Unexplained terms and abbreviations
Unverified Action	Action steps without corresponding verifications

Source: research data

Figure 19 – Distribution of identified test smells per system



Source: research data

system analyzed by all authors. From the 447 occurrences found in this study, 114 belonged to Ubuntu OS tests. From these occurrences, authors initially rated 91 classifications in agreement, which lead us to 79.8% of initial inter-rated reliability (percent agreement (SKRONDAL; EVERITT, 2010)) in the exploratory study. As stated, the divergences were solved in two meetings, reaching a consensus in the classifications.

5.1.3 Discussion

The structural test pattern found in the analyzed systems (Section 5.1.2) enabled us to propose the test smells presented in Section 5.2. Moreover, the distribution of such smells demonstrates the analysis of natural language test descriptions from the point of view of test smells to present promising results. The manual analysis offers some insights whose reality is precisely shown in Section 5.4. These insights, for now, indicate that:

- Most observed test smells are common to all analyzed systems (*e.g., Eager Action*);
- There are test smells unique to a single system (*e.g., Tacit Knowledge*);
- Each system has its own test smell trend (*e.g.,* Ubuntu tests suffer more from *Unverified Action*).

Answering **RQ**$_5$, we could observe two already proposed natural language test smells in the analyzed systems. In addition, six new test smells are observed, which answers **RQ**$_6$. Answering **RQ**$_7$, the test smells are frequent throughout the analyzed systems. In particular, Eager Action and Tacit Knowledge tend to be the most and the least frequent ones.

5.1.4 Threats to Validity

Concerning the conclusion threats, our identified test smells relate to the common test structure in all three analyzed systems. However, it is important to notice that BEVM and LSM tests are managed by well-adopted software solutions throughout the industry, leading us to understand the found pattern as generally widespread, possibly minimizing this threat.

In the internal threats, as the accuracy of the exploratory study (*i.e.,* 80%) is not ideal for generalizations in the analyzed systems (II; KOTRLIK; HIGGINS, 2001), the distribution of test smells presented in Figure 19 may not be precise. We minimized this problem by modeling and validating a Natural Language Processing (NLP)-based tool, further detailed in Section 5.4, to provide the exact distribution of the presented test smells.

As external threats, analyzing a few software systems may not be enough to identify relevant or well-spread test smells. We minimize this probability by using systems representative of different domains and spoken languages and finding test smells common to such systems.

5.2 A CATALOG OF NATURAL LANGUAGE TEST SMELLS

We now present our catalog, the main product of our exploratory study. We show the identified test smells in terms of their names, definition, problem, and identification rules for their detection with examples — when applicable — from the analyzed Ubuntu OS test descriptions.

5.2.1 Ambiguous Test

Definition: Originally proposed by Hauptmann *et al.* (HAUPTMANN et al., 2013), this smell indicates an *"under-specified test that leaves room for interpretation"* in integration tests.

Problem: It negatively impacts test comprehension and execution, since the aim needs to be clarified and multiple test executions are not comparable (HAUPTMANN et al., 2013).

Identification: The original detection rule was the occurrence of any word from a fixed list of "vague words."[3] As Hauptmann *et al.*'s (HAUPTMANN et al., 2013) keyword list originated from occurrences in their analyzed test suites and we found a slightly different list in our exploratory study (*e.g.,* some, other, and any), we noticed such keywords to be common in their semantics (syntactic analysis). We propose a more general set of detection rules which consider keyword semantics instead of a fixed list, and examples, in Table 17.

Table 17 – Ambiguous Test Identification

Rule	Example
Verb + indefinite determiner	***Open any*** *application and suspend machine*
Indefinite pronouns	*At "Write changes to disks", verify that* ***everything*** *is right and select YES*
Comparative adjectives	*Is the performance* ***similar*** *or* ***better*** *with no graphical display issues?*
Superlative adjectives	*The root filesystem uses* ***most*** *of the SD card.*
Adverbs of manner	*Does fast user switching work* ***quickly****?*
Comparative adverbs	*Does everything function* ***better*** *than the stable version?*

Source: research data

5.2.2 Conditional Test

Definition: Tests containing conditional logic phrased in natural language.

Problem: The Conditional Test turns tests very complex and difficult to maintain, negatively impacting test comprehension and correctness since it is hard to understand the intention, and complex tests are more likely to have errors (HAUPTMANN et al., 2013).

Identification: Originally, Hauptmann *et al.* (HAUPTMANN et al., 2013) proposed a fixed list of words for its detection.[4] As the list is non-exhaustive concerning subordinating conjunctions,

[3] similar, better, similarly, worse, having in mind, take into account, take into consideration, clear, easy, strong, good, bad, efficient, useful, significant, adequate, fast, recent, far, close

[4] if, whether, depending, when, in case

we propose any subordinating conjunction, as in Table 18, to identify this smell as a more robust detection rule.

Table 18 – Conditional Test Identification

Rule	**Example**
Subordinating conjunctions	***If*** *you have a USB drive, plug it in.*

Source: research data

5.2.3 Eager Action

Definition: Single action steps that group multiple actions.

Problem: This test smell may hide implementation problems when any action lacks verification, negatively affecting test effectiveness.

Identification: Imperative verbs represent actions. Example in Table 19.

Table 19 – Eager Action Identification

Rule	**Example**
Multiple imperative verbs	***Change*** *some sound settings or other settings (night mode, call history, SMS, etc.) and* ***display*** *them on the phone,* ***download*** *some applications, etc.*

Source: research data

5.2.4 Misplaced Action

Definition: Indicative of a structurally malformed test, the Misplaced Action smell arises when action steps are written as results.

Problem: It negatively impacts test maintainability, since the test structure is not consistent.

Identification: Imperative verbs, excluding verification verbs,[5] present in verification steps. Example in Table 20. A test except containing the *Misplaced Action* test smell is shown in Table 21.

[5] Verification verbs identified in use: check, verify, observe, recheck

Table 20 – Misplaced Action Identification

Rule	Example
Imperatives, excluding verification verbs, as verification steps	***Give*** *a name to the directory and* ***add*** *files to it as you* ***did*** *in the previous step*

Source: research data

Table 21 – Test excerpt containing the *Misplaced Action* test smell

N	Action	Verification
1	Press the "Add" button (+)	A new window asking for files appears
2	Click on "Create Directory"	**Give a name to the directory and add files to it as you did in the previous step**
3	Select several files	The files are added to the project. All information about the files is displayed: Contents, Size, Local Path. The size bar will contains information now.

Source: research data

5.2.5 Misplaced Precondition

Definition: Also an indicative of structurally malformed tests, here, preconditions are written as action steps.

Problem: Difficulties in test correctness, since the incorrect placement of preconditions may influence the tester to report test failure should a precondition be unattended.

Identification: When the first action step declares the SUT state. The common format of SUT state is a *noun (subject)* followed by an *auxiliary verb*, followed by a *past participle* verb or adjective in the same sentence (Table 22). A test except containing the *Misplaced Precondition* test smell is shown in Table 23.

Table 22 – Misplaced Precondition Identification

Rule	Example
Subject followed by an auxiliary verb followed by another verb on the past participle	*The* ***monitor is*** *not* ***connected****, and the* ***PC is*** *not* ***paired***

Source: research data

Table 23 – Test excerpt containing the *Misplaced Precondition* test smell

No	Action	Verification
1	**Make sure auto-hide is enabled.**	
2	Move the mouse on the edge and continue pushing to the left (as if you want to put the mouse offscreen) until the launcher reveal	The launcher should appear
3	Put the mouse outside of the launcher area	The launcher should disappear again after around a second

Source: research data

5.2.6 Misplaced Verification

Definition: Another indicative of structurally malformed tests, this smell arises when verification steps written as action steps.

Problem: It negatively impacts test maintainability, since the test structure is not consistent.

Identification: Sentences containing verification verbs written as or along with action steps. Example in Table 24. A test except containing the *Misplaced Verification* test smell is shown in Table 25.

Table 24 – Misplaced Verification Identification

Rule	Example
Verification in or as an action step	*Close flip and **check** app continuity*

Source: research data

Table 25 – Test excerpt containing the *Misplaced Verification* test smell

No	Action	Verification
1	Open the dash and launch rhythmbox by pressing the super key, and then entering 'rhythmbox'	rhythmbox should launch
2	Plug a music player device containing MP3 files into your system and **check whether rhythmbox imports the music correctly.**	A new entry Devices appears
3	...	...

Source: research data

5.2.7 Tacit Knowledge

Definition: This test smell is related to the use of unexplained terms and abbreviations presuming the tester's familiarity to domain-specific definitions.

Problem: It negatively impacts test comprehension and execution.

Identification: Abbreviations and domain-specific terms not explained in the test description or external reference document (*i.e.,* glossary). A hypothetical example, since we are not authorized to disclose BEVM tests, is in Table 26.

Table 26 – Tacit Knowledge Identification

Rule	**Example**
Unexplained terms and abbreviations	*Check for reported **residual votes***

Source: research data

5.2.8 Unverified Action

Definition: Action steps that miss corresponding verification steps.

Problem: Absent verification steps negatively affect test execution and correctness since there is no instruction on how the system should behave, leaving room for the testers' interpretation.

Identification: Action steps with no corresponding verification steps. A test except containing the *Unverified Action* test smell is shown in Table 27.

Table 27 – Test excerpt containing the *Unverified Action* test smell

No	Action	Verification
1	Click on the Plugins tab and Click "Add"'	The Plugins window opens
2	Choose a plugin Synth and Click "OK"	The plugin synth shows in the Plugins tab and a window opens with the plugin in it
3	**Right Click on the plugin synth and choose Activate**	
4	In the plugin window choose a patch name and Click "Send Test Note"	A sound plays through the speakers
5	**Click on the Track tab of the track window**	
6	**In the MIDI/Instrument Panel choose the Plugin as an instrument**	
7	**Click "OK" in the Track window**	
8	Click on the "Play" icon	The midi track plays using the chosen sound of the plugin instrument

Source: research data

5.3 CATALOG EVALUATION

In this section, we present the online survey performed to evaluate our proposals. This activity, in particular, answers **RQ$_9$:"How software testing professionals evaluate our proposed smells?"**

5.3.1 Planning

This study planned to assess the opinions of software testing professionals (*e.g.*, engineers, analysts, and managers) about the manual test smells we proposed in Section 5.2 through an online survey. By stating their agreement with our definitions and examples and commenting on their answers, the software testing professionals would validate whether our proposals represented valid test smells in theory and practice.

5.3.2 Settings

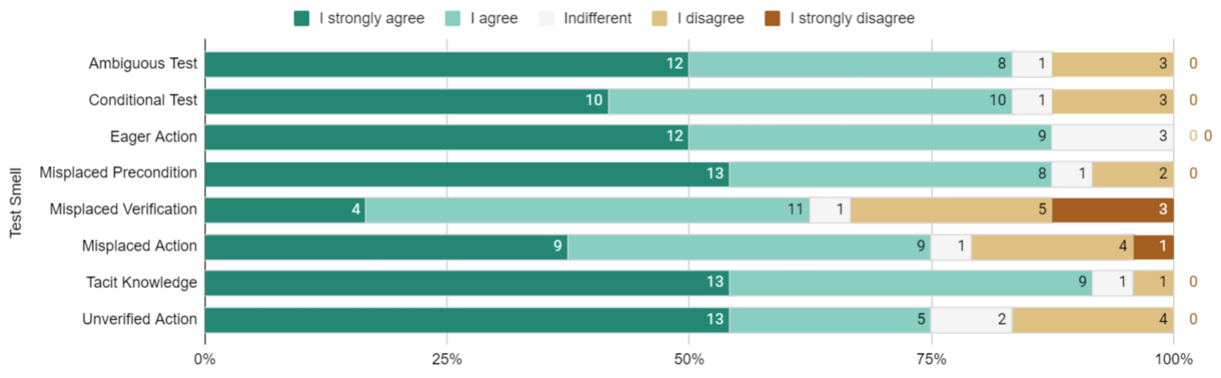
We assembled an online survey with questions corresponding to the given definition and example, same as presented in Section 5.2. Respondents were presented with the following answering options (unique): *"I strongly agree"*, *"I agree"*, *"Indifferent"*, *"I disagree"*, and *"I strongly disagree"*. Also, every question had an optional comment field.

We recruited participants through individual email invitations, using emails from our industry partner. The invitations were sent to participants of manual test teams, quality assurance professionals, and test managers, none of whom had compensation or obligation to respond to the survey.

5.3.3 Results

We performed the survey in March 2023, achieving 24 responses. Concerning the demographics, we had participants from Brazilian teams, where 83.3% defined their primary work area as the industry — over academia — and their average declared experience with software testing was 4.3 years. Figure 20 details the results concerning the participants' opinions on our proposals. Considering the extent of agreement between respondents (*i.e.,* Kendall's Coefficient of Concordance (LEGENDRE, 2005)), the obtained results achieve an inter-rater reliability of 80.73%.

Figure 20 – Survey results



Source: research data

5.3.4 Discussion

Regarding the proposed test smells (Section 5.2), the opinion of experienced test professionals active in the market (industry partner) served as validation that obtained a high acceptance rate (Figure 20). We present the details in the following paragraphs.

Already present in the literature, the *Ambiguous Test* smell (Section 5.2.1) definition was ratified by 83% of the respondents. Among the agreeing comments, the ambiguity may indeed cause tests to be poorly performed depending on the tester experience, as in *"My experience* *can improve the test coverage, however for a beginner tester is not be clear the ways to test an interruption, and this can induce he/she to repeat the same procedure/routine or try few different ways to suspend the app."* Among the testers that disagree with the test smell definition, the variance allowed by non-deterministic terms is beneficial to test different scenarios, as seen in *"I would say that exploratory test cases use a similar approach and it has been working"*.

Known in automatic and manual testing, the *Conditional Test* (Section 5.2.2) smell definition and example had the acceptance of 83% of the respondents. Concerns about the conditionals being able to improve the test coverage on features not always available arise in both sides, as in the agreeing opinion *"The only part I would not agree is if it is related to a feature that the product may not actually have implemented, for example, NFC."*, and the disagreeing opinion *"The test writer attempted to cover more possible verifications. If a step or accessory can't be verified all the test is not blocked, and the test becomes applicable to different kinds of product"*.

As the first proposal of our work, the *Eager Action* (Section 5.2.3) test smell definition was ratified by 87.5% of respondents, with no disagreeing opinion. Among the comments, difficulties in the test execution and concerns about the verifications can be found in *"it seems rather confusing and not pointing to any settings overall, it is covering multiple scenarios"* and in *"There isn't a guarantee that the tester checked all configurations available"*.

Ratified by 75% of the respondents, the *Misplaced Action* (Section 5.2.4) test smell had supporters that manifested concerns about test structure, as in *"Verification steps should be in the end of test cases. Preconditions at the beginning, and actions in the middle."* Testers that do not agree with the given definition manifest no concern to test structure, since they comprehend the test objective, as in *"If the action keeps the step valid as a single one, it makes sense to be written"*.

The concerns described by the *Misplaced Precondition* (Section 5.2.5) test smell definition were accepted by 87.5% of the respondents. Unfortunately, as that was not a mandatory task in this survey, the respondents provided no comments to this test smell description.

The *Misplaced Verification* (Section 5.2.6) test smell was our least accepted proposal, even though counting with 62.5% agreeing opinions. Testers claim the test clarity to benefit from the separation into action and verification steps, as in *"I agree because I think that is more clear and organized for the test have it in separated (verification) steps"*. On the opposite hand, testers that did not agree also claimed maintainability benefits of keeping action and verification steps written together, as in *"these actions help to avoid too many steps in a script and reduces the effort in test maintenance"*.

Our most accepted proposal, the *Tacit Knowledge* test smell (Section 5.2.7) definition had the support of 91.6% respondents. The excessive use of abbreviations and unexplained domain-specific terms is indeed a concern to agreeing respondents, as in *"In my experience, I have faced many new testers and interns having problems knowing abbreviations in test cases"*. Disagreeing opinions call attention to test maintainability, as in *"I would say it is a case by case scenario where it could be bad either way. I could have overly long texts due to unnecessary repetition that could be solved by Basic Glossary before the TCs (test cases). Or a inverse scenario where the tester is not provided with edge information to that test."*

Our last proposal, the *Unverified Action* test smell (Section 5.2.8) had the approval of 75% respondents. No agreeing respondent gave further details on their answer. Disagreeing respondents manifested concern about the verification steps to every action, as in *"Not every action, in a sequence of actions, generates a relevant result to be verified."* and *"In some situations the expected result is too obvious and can be dispensed. I believe that this helps to not tire the reader."*

The online survey shows that software testing professionals mostly agree with our proposals. In addition to positively answering $\mathbf{RQ}_9$, the provided comments show additional concerns, such as test reproducibility, length, maintainability, and coverage, all originated from doubts raised by poor test writing.

5.3.5 Threats to Validity

Concerning the internal results, some respondents made the same claim for better organization when a test has action and verification steps written together or separated, for instance, both representing agreeing and disagreeing opinions. However, the wide acceptance of our proposals votes in favor of our interpretation of the possible prejudices, minimizing the threat.

As external threats, we used responses from software testing professionals who work for our industrial partner, and this bias may influence the generalization of results to other audiences. We minimize this probability through the respondents' experience, of about 4.3 years of experience on average (Section 5.3.3), and whose answers tend to be similar to experienced professionals who test software in other domains. However, even with an average inter-rated reliability for agreeing opinions of 80.73%, the amount of obtained responses (*i.e.,* 24) advises generalizations as formally impossible.

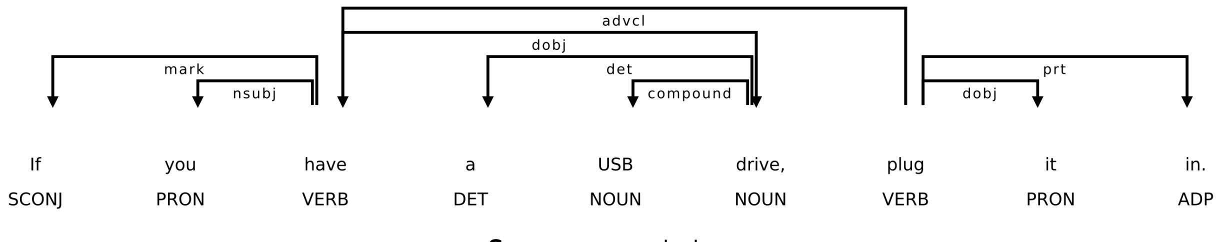
5.4 DETECTING SMELLS IN MANUAL TESTS WITH NLP

We present the development of an NLP-based tool, which we call Manual Test Sensei, to detect the natural language test smells we described in Section 5.2. This effort shows how implementing our rules for natural language test smells identification is feasible using the current state of the NLP technology.

We use Python and spaCy (HONNIBAL; MONTANI, 2023), a commercial open-source software released under the MIT license (SALTZER, 2020), to implement the NLP tool containing our rules for discovering natural language test smells. SpaCy features convolutional neural network models for part-of-speech (POS) tagging (MARCUS; SANTORINI; MARCINKIEWICZ, 1993), dependency parsing(NIVRE et al., 2016), text categorization, and named entity recognition (NER) (HONNIBAL; MONTANI, 2017). Figure 21 shows a visualization of the dependency parsing — arrows above the sentence — and the POS tagging — labels beneath each sentence element — for the Conditional Test example of Section 5.2.2.

The motivation for choosing this combination of programming language and NLP library were (i) using market tools focused on results and performance to analyze industrial-scale software and (ii) the availability of language models beyond English since BEVM tests are in Portuguese.

Figure 21 – spaCy's visualizer module example

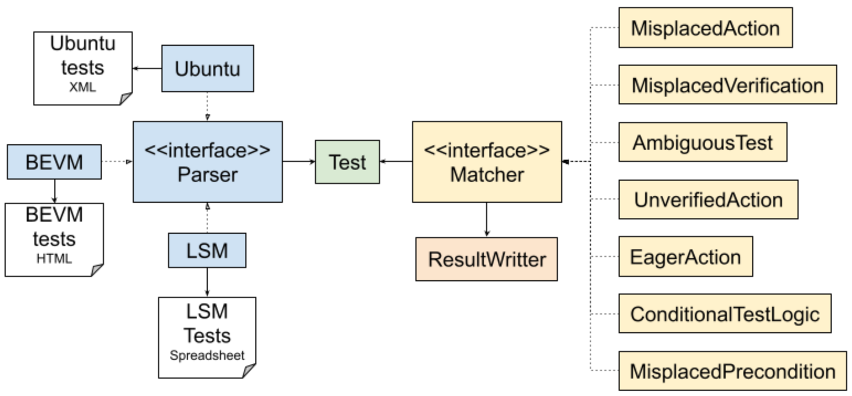


Source: research data

The chosen strategy enabled us to implement most of our identification proposals. However, identifying the *Tacit Knowledge* (Section 5.2.7) requires a more comprehensive solution. To perform it, one would consider (i) external documentation (*e.g.,* glossaries and execution manuals) — non-existent in Ubuntu and not provided in the BEVM and LSM — and (ii) a list of standard terms used in manual software testing and considered tacit in every manual testing scenario, where every outsider term would characterize the *Tacit Knowledge* test smell if not clarified. To the best of our knowledge, the proposition of such a list is yet to be performed and requires a formal study.

Also, we had to consider the different test file formats according to each analyzed project: XML for the Ubuntu OS, HTML for the BEVM tests, and spreadsheet for LSM tests. To that end, specific parsers were created for each system's test file format. Figure 22 presents a simplified UML class diagram of the Manual Test Sensei tool, where the parsers — responsible for transforming a test file into several test objects — and the test smell matchers are shown. Finally, the tool produces a CSV file as output containing the test file name, the identified test smell, the specific words or sentence span that characterize the test smell, and the analyzed (action or verification) step.

Figure 22 – Simplified UML class diagram of the developed NLP tool



Source: research data

The tool source code is available in an online repository at <https://github.com/easy-software-ufal/manual-test-sensei>.

5.5 TOOL EVALUATION

Once the proposition and development of the Manual Test Sensei tool — implementing our natural language test smell identification rules — proved possible using current NLP technology (Section 5.4), in this last study, we present the tool results and validation, therefore demonstrating how precise is the tool performance. This activity, in particular, answers **RQ$_{10}$:"How precise can the automated discovery of natural language test smells be when using NLP?"**

5.5.1 Planning

This study planned to execute the Manual Test Sensei tool against the entire test set of the three analyzed systems and validate the results. Therefore, we could verify whether the distribution found in the exploratory study (Section 5.1) is maintained in the Manual Test Sensei execution results, as well as the accuracy — in terms of precision, recall, and f-measure metrics (RIJSBERGEN, 1974; POWERS, 2020) — of such results.

5.5.2 Settings

Although we executed our tool against the entire test set of the three systems, manually validating the tool's output of 13,169 smells would be infeasible. Therefore, we randomly selected 101 tests — so that the division results in whole numbers — distributed in proportion to the number of tests available in every analyzed system.

For every selected test, an author would first analyze it manually and indicate the found test smells, then verify the tool results for that test, and finally indicate the results that were correct or true positives (TP), incorrect or false positives (FP), and the missed or false negatives (FN) test smells. Table 28 presents the distribution of the randomly selected tests per system:

Table 28 – Distribution of selected tests in the validation sample

System	Total tests	Sample size
Ubuntu OS	973	49
BEVM	136	7
LSM	898	45
Total	**2,007**	**101**

Source: research data

5.5.3 Results

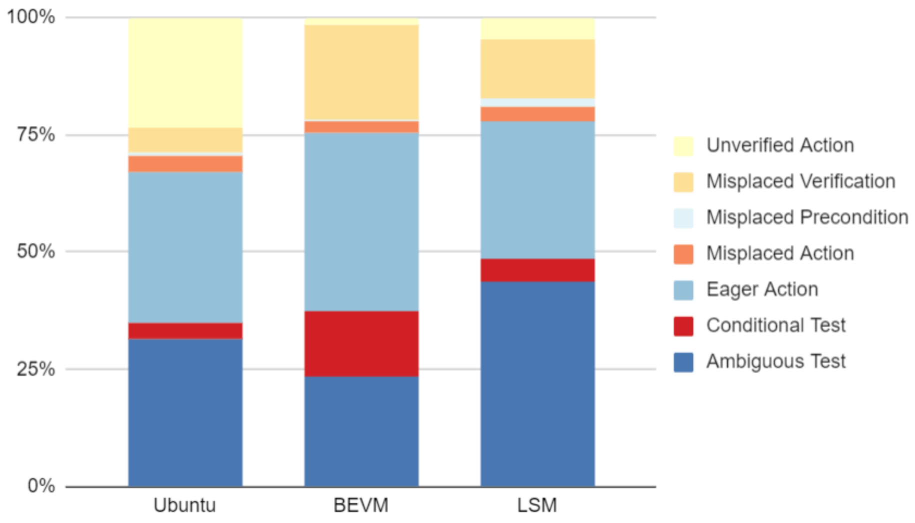
A total of 2,007 test descriptions were analyzed by the Manual Test Sensei tool. The tool indicated 13,169 test smells, with an average of 6.5 test smells per analyzed test, noticeably higher than the 1.2 test smells found in the exploratory study (Section 5.1). Considering the analyzed systems individually, we obtained an average of 8.5 test smells per Ubuntu OS test, 5.8 test smells per BEVM test, and 4.5 test smells per LSM test. Table 29 presents the results per test smell and system. Finally, a distribution of the found test smells per analyzed system is presented in Figure 23.

Table 29 – Total NLP results

Test Smell	Ubuntu	BEVM	LSM	Total
Ambiguous Test	2,627	185	1,776	**4,588**
Conditional Test	277	110	193	**580**
Eager Action	2,664	299	1,191	**4,154**
Misplaced Action	318	19	124	**461**
Misplaced Precondition	45	3	74	**122**
Misplaced Verification	428	161	513	**1,102**
Unverified Action	1,967	11	184	**2,162**
Total	**8,326**	**788**	**4,055**	**13,169**

Source: research data

Figure 23 – Distribution of test smells per system



Source: research data

Three authors performed the verification as defined in Section 5.5.2. Table 30 presents the detailed validation totals per system and the precision, recall, and f-measure metrics achieved by the tool in this validation activity.

Table 30 – Detailed NLP tool validation and metrics

System	TP	FP	FN	Precision	Recall	F-measure
Ubuntu OS	384	43	18	0.9	0.96	92.64
BEVM	25	0	0	1	1	1
LSM	213	13	12	0.94	0.95	94.46
Total	**622**	**56**	**30**	**0.92**	**0.95**	**93.53**

Source: research data

5.5.4 Discussion

The impressive number of 13,169 test smells found in 2,007 is only justifiable due to the nature of the analyzed tests. As mentioned, these are integration and system tests, which are long ones by nature. In our analysis, test descriptions having 30 or more steps were common. Also, our tool accounted for each test smell occurrence in a test, regardless of previous detections of the same smell in previous steps of the same test.

The high expressiveness of the adopted technology, either in the identification of dependency relationships (e.g., subject + auxiliary verb + participle verb) or in the identification of the Part of Speech (POS) (e.g., indefinite pronouns), enabled us to implement most of the detection rules as defined in Section 5.2. Only one identification rule could not be implemented entirely, which was the Conditional Test, identified through subordinating conjunctions (SCONJ) at the beginning of a dependent clause in a sentence. As spaCy does not natively support splitting sentences into clauses, which varies from language to language, identifying SCONJ in a dependent clause in the middle of a sentence results in many identification problems by the pre-trained models. This problem resulted in 8 false negatives identified in the validation activity, representing approximately 27% of the test smells not identified by the tool.

We analyzed tests written in different languages (*i.e.,* English for Ubuntu OS and LSM, and Portuguese for BEVM). As our identification rules were specified in terms of syntactical (POS tagging) and morphological (dependency parsing) analyses, changing the analyzed language is a task of as low effort as informing spaCy which language model (they dispose pre-trained models for more than 20 languages) should be used. We simplified this task by informing the tool, in its command line, which language (English or Portuguese) should be used for test analysis. Further execution details are presented in the tool documentation.

We encountered various formatting, spelling, and character encoding conversion issues in the test descriptions. Using numbered and unnumbered lists, parentheses, and the lack of correct punctuation impaired the NLP engine classification in some cases reported as false positives and false negatives. For example, the implemented mechanism was not able to identify a subordinate clause in the sentence *"(If on a 'laptop') Is plugged to a power source,"* nor in *"Type in your user name and press Enter (you can accept the default if you wish),"* and could not differentiate the link label in the sentence *"Click the Choose Payment Method link,"* which lacked quotes, and was erroneously classified as multiple actions.

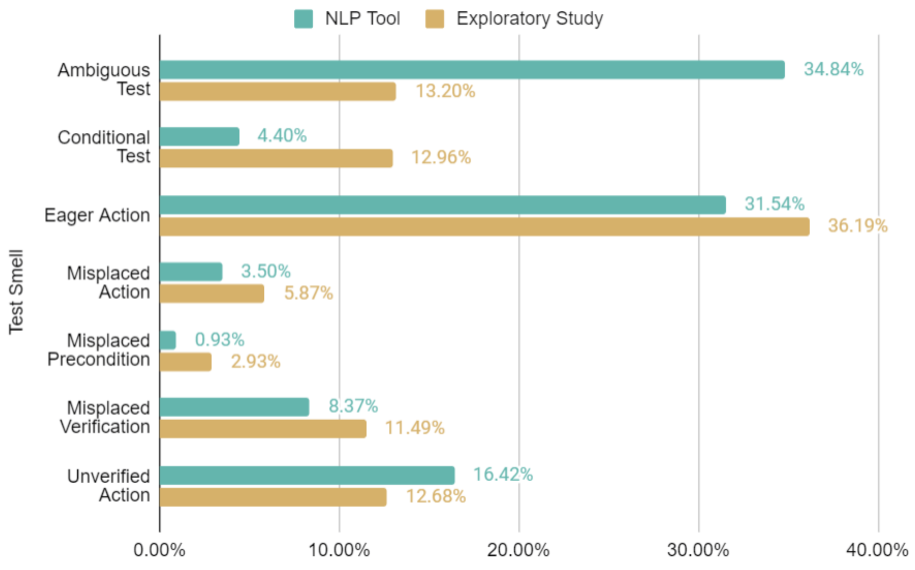
However, even with the implementation challenges and some test malformations mentioned, the result obtained in the metrics of precision, recall, and f-measure for the tool can still be considered expressive. The results remain promising even when using a trained model for a different idiom and executing the same rules — except for the list of verification verbs used in the *Misplaced Verification* detection, which needed a partner in Portuguese for BEVM tests — as seen in the metrics presented by Table 30.

According to Table 29, the most frequent test smells detected were the *Ambiguous Test* (*i.e.,* 34.8%) and *Eager Action* (*i.e.,* 31.5%). An interesting distribution noticed is that, from the 4,588 occurrences of the *Ambiguous Test*, we accounted for 2,225 (*i.e.,* 48.5%) occurring in action steps and 2,363 (*i.e.,* 51.5%) occurrences in verification steps, meaning that ambiguous tests have an almost equal probability of presenting testers with difficulties in *"what to perform in the test"* and *"what to verify as a result."* However, being less frequent may not mean less harm to the testing activity. It is important to remember that a *Misplaced Precondition* can induce the tester to declare the test failed if the precondition is not met and the test cannot be executed (SOARES et al., 2020; SOARES et al., 2023).

Comparing the distribution of test smells found in the exploratory study (Section 5.1) and the one found by the NLP tool (Section 5.5.3), shown in Figure 24, we noticed that some test smells had a different percentage result between the two activities, which was the case of the *Ambiguous Test* and the *Conditional Test*. This expressive difference was due to the more precise identification of the tool in cases of undefined determinants, which may escape the most attentive — or not sufficiently trained in the exploratory study — eyes. Still, the precision difference in the exploratory study (Section 5.1) and the tool validation (Section 5.5.3), necessary for this study to be feasible, influenced the found deviation.

Furthermore, we noticed that test smells not found in the exploratory study for specific systems, such as the *Misplaced Precondition* for the BEVM tests, are now among the results of the NLP tool (Table 29), even with few occurrences (i.e., 3). This result is also expected and

Figure 24 – Comparison between the exploratory study and the NLP tool results



Source: research data

included in the exploratory study's 5% margin of error (Section 5.1.1). Finally, the proportional distribution of test smells per system shown in Figure 23 shows that although the tests of the analyzed systems suffer from the test smells found, they do so in different proportions.

The results obtained in the tool validation show that our detection rules are effective in identifying the considered test smells. In particular, we achieved a f-measure of 93.53%, which answers **RQ**$_{10}$.

5.5.5 Threats to Validity

As internal threats, the tool's results may contain errors. We manually analyzed 101 tests to minimize this threat, which meant more than 700 results, according to Table 30. This amount of results was enough to guarantee statistical validity (II; KOTRLIK; HIGGINS, 2001) for the 13,169 results generated by the tool.

As external threats, despite the promising results, the generalization of the results obtained for other systems is impossible with the sample of three systems. We minimize this threat by choosing highly expressive systems from different domains to analyze. Nevertheless, an exploratory study would confirm whether our results indicate some degree of probability to the analysis of other systems.

5.6 RESEARCH DIRECTIONS

Built upon our findings, research directions for natural language test smells would involve:

- Enabling the implementation of the *Tacit Knowledge* test smell by performing a formal study to define common terms in software testing terminology that may be considered tacit in any manual execution of software tests;
- Executing the manual test sensei tool analysis in other candidate systems whose test management is performed using the same tools as BEVM and LSM tests to verify the generalization of our results; and
- Aggregating tests — and test file formats — from uncovered systems in the results.

6 RELATED WORK

6.1 TEST SMELLS

The study performed by Bavota et al. (BAVOTA et al., 2015) demonstrated test smell distribution in software systems and whether their presence is harmful. As part of the investigation, they performed a controlled experiment involving 61 participants among students (freshers, bachelors, and masters) and industrial developers, which were asked to perform maintenance activities on smelly and refactored test code of two software systems. The study demonstrated the negative impact of test smells in program comprehension during maintenance activities. Our study uses both open-source developers and projects to survey on test smells perceptions.

Tufano et al. (2016) investigation aimed at analyzing when test smells occur in source code, what their survivability is, and whether their presence is associated with the presence of design problems in production code (code smells). They collected the developers' perception of test smells in a study with 19 developers from Apache and Eclipse ecosystems. They demonstrated that (i) test smells have a long life cycle in software systems, and (ii) there are correlations between test and code smells. Our study extends the surveyed public in order to improve the accuracy of developers' perception.

A recent list of test smells descriptions can be found in Aljedaani et al. (2021) work, which provides a detailed study of test smell detection tools designed to work with Java, Scala, Smalltalk, and C++ test suites. The tools are described in terms of their characteristics, type of smells, target language, and availability. As findings, they could demonstrate that most tools overlap in detecting specific smell types, such as General Fixture (DEURSEN et al., 2001), and that four techniques used by the cataloged tools to detect test smells. Their test smells list presents the definition of 66 ones detected by the analyzed tools. As a characteristic of their study, they limit the test smells information only to their definition and only cataloged the test smells detected by the corresponding tools.

6.2 SYSTEMATIC REVIEWS

Garousi and Küçük (GAROUSI; KÜÇÜK, 2018) performed the most comprehensive survey so far as a Multivocal Literature Mapping (MLM) — a study that aims to gather quality attributes from previous research papers and gray literature on a topic (OGAWA; MALEN, 1991)

— to classify the body of knowledge on formal and gray literature about test smells. They collected quality attributes from the selected sources like demographics, literature type, if new test smells are proposed, contribution facet, paper type, research questions, approach type, if the paper presents correction techniques, language, and System Under Test (SUT) specificities, making the compiled information publicly available. The authors also proposed a public test smell classification board for 196 test smells extracted from 166 sources (120 gray literature and 46 formal). Their study *"lays the foundation for a follow-up, in-depth review study."* We propose this study by renewing the search for academic and gray literature — including the studies and web references published in the last five years since their mapping — raising the number of mapped test smells to 480. Going beyond literature mappings, in our review, we extract test smell data from the retrieved sources and publicize our dataset along with a catalog of 480 test smells as an open-source project intended for the software testing community use and maintenance.

Intended for test smells in System User Interactive Tests (SUITs), Rwemalika *et al.* (RWEMALIKA et al., 2021) performed an MLR on 38 references and proposed a catalog of 35 SUIT-specific test smells. The authors also proposed an automated approach for detecting diffusion and refactoring for 16 cataloged test smells and demonstrated their findings in industrial and open-source projects. Despite raising 79 test smell sources in their searches, the authors used a minor group due to the scope of their research. In our study, as we do not limit the scope of test smells, we include their proposals and references that meet our inclusion criteria by defining or referencing studies that define test smells (Chapter 3).

An SLR is also present in the study performed by Wang *et al.* (WANG et al., 2021), who proposed a Python-specific test smell detection tool. The authors presented a *"small-scale systematic mapping study on test smells to curate a list of test smells discussed in the literature."* Their study listed 33 test smells encountered in Java, Scala, and Android systems, extracted from 29 peer-reviewed studies published between 2006 and 2020, that exclusively mentioned the keyword "test smells." As our study has a broader search string — including the keyword "test smell" (Section 3.1) —, we include their references and smells in our MLR.

Aljedaani *et al.* (ALJEDAANI et al., 2021) performed a Systematic Mapping Study (SMS) to identify, categorize, and analyze literature that proposed test smell detection tools. They mapped 47 studies and, although not directly intended for test smells, their work listed 66 test smells — detected by the analyzed tools — by their names, descriptions and references. Our study also considers their references that, despite being intended for test smell detection tools, meet our inclusion criteria.

6.3 TEST SMELL REMOVAL

Concerning the discovery of test smell refactoring operations, Peruma et al. (PERUMA et al., 2020) used a mining tool to detect such refactorings from an existing dataset of unit test files and smells in 250 open-source Android Apps. Among the results, they could demonstrate that the types of refactorings applied to test and non-test files in Android apps are different and that there exist test smells and refactoring operations that co-occur frequently. Also, Lambiase et al. (LAMBIASE et al., 2020) presented the DARTS (Detection And Refactoring of Test Smells) tool, which detects instances of three test smell types (General Fixture, Eager Test, and Lack of Cohesion of Test Methods) at the commit-level and enables their automated refactoring using the refactoring techniques defined by Meszaros (MESZAROS, 2007).

Peruma et al. (2020) used a mining tool to detect refactoring operations from an existing dataset of unit test files and smells in 250 open-source Android Apps. Among the results, they could demonstrate that the types of refactorings applied to test and non-test files in Android apps are different and that there exist test smells and refactoring operations that co-occur frequently. More importantly, and following the results of our study, they could verify that developers apply refactorings for reasons other than the correction of smells, making the fixing of smells merely a byproduct.

In the study performed by Aljedaani et al. (2021), three test smell detection tools were capable of refactoring the identified test smells. The first one is an Intellij plug-in proposed by Lambiase et al. (2020) and is called DARTS (Detection and Refactoring of Test Smells). Their work focuses on the Eager Test, General Fixture, and Lack of Cohesion of Test Methods test smells, which are not in the scope of our work. The second is the RTj framework proposed by Martinez et al. (2020), where the authors propose the concept, identification, and refactoring of Rotten Green Test Cases — passing tests with at least one unexecuted assertion — which are also out of the scope of our work. Nevertheless, their refactoring actions are limited to substituting an intentionally failing assertion to a call to the `fail` method and adding a `TODO` comment to the problematic code element. The third tool is an open-source and IDE integrated tool proposed by Santana et al. (2020), called RAIDE, that works with the Assertion Roulette and Duplicated Assert test smells in Java projects. The refactoring actions they implement are the *"Add Assertion Explanation"* (DEURSEN et al., 2001) to the Assertion Roulette test smell and the separation of duplicated assertions in several test methods (PERUMA et al., 2019).

6.4 DEVELOPERS PERCEPTIONS

The developer's awareness and perception to test smells are also some studies' aims. Bleser, Nucci and Roover (2019a) assessed the diffusion and developer's perception of test smells in SCALA projects. Their results show the low diffusion of test smells across SCALA test classes and that many developers cannot correctly identify most of the smells, even though they perceive a design issue. A work proposed by Peruma et al. (2019) investigated test smells in open-source Android applications. When surveying developers, they confirmed the author's proposed smells as bad programming practices in unit test files for most proposed cases and systems. In a similar line of work, Junior et al. (2020) investigated the causes of test smell introduction by developers. Their results indicate that experienced professionals introduce test smells during their daily programming tasks, even when using standardized practices from their companies, not only for their assumptions. Another study, performed by Spadini et al. (2020), aimed at investigating the severity rating for four test smells and their perceived impact on test suite maintainability by the developers. The authors found that developers consider current detection rules for specific test smells too strict and verified that their newly defined severity thresholds align with the participants' perception of how test smells impact the maintainability of a test suite. Our work uses the developers' opinions to validate our propositions, which presumes their understanding of test smells effects and test code refactorings.

Palomba et al. (PALOMBA et al., 2014) performed a survey on the developer's perception of bad code smells. They showed production code snippets from three software systems to original and outside (students and industry) developers, asking them to indicate if they found any potential design problems and what nature and severity level they would classify the possible problems. As a result, they could divide the investigated code smells into categories related to code complexity and coding good-practices. Our study offers a similar perception by open-source developers, this time concerning test smells.

A work in progress performed by Schvarcbacher et al. (SCHVARCBACHER et al., 2019) aimed to assess the perception of developers on test smells in their codebase and which ones they would consider being more important. They present the developers' reactions to various instances of test smells pointed out by their detection tool, based on the TSDetect (PERUMA et al., 2019), in integration with the Better Code Hub (BCH).[1] Their results show that developers are only willing to remove a small portion of the found test smells, but did not present them with refactored alternatives to measure their acceptance.

Investigating the extent of developers acceptance to test smells existence and refactoring operations, our previous research (SOARES et al., 2020) consisted of a mixed-method study with two parts: (i) a survey with 73 experienced open-source developers to assess their preference and motivation to choose between 10 different smelly test code samples, found in 272 open-source projects, and their refactored versions, and the submission of 50 pull requests to assess developers' acceptance of the proposed refactorings. As results, most surveyed developers preferred the refactored proposal for 78% of the investigated test smells, and the pull requests had an average acceptance of 75% among respondents. In this previous study, it is important to quote that we validated refactoring operations already presented in the literature of the Test Smells area (DEURSEN et al., 2001; MESZAROS, 2007) and always considered the test library version used by each project when submitting a contribution.

A study performed by Spadini et al. (SPADINI et al., 2020) aimed at investigating the severity rating for four test smells and their perceived impact on test suite maintainability by the developers. They collected test smells from 1,500 open-source projects and used in-company developers' perceptions to calibrate the established severity thresholds for test smells. As results, they found that current detection rules for certain test smells are considered too strict by the developers and verified that their newly defined severity thresholds are in line with the participants' perception of how test smells impact the maintainability of a test suite.

6.5 EVOLUTION OF TEST FRAMEWORKS

The evolution of test frameworks is the focus of the empirical study performed by Kim et al. (2021a), which investigated the evolution of testing practices after the introduction of annotations in Java 5. The authors found that test annotation changes are more frequent than rename and type change refactorings, bringing empirical evidence on the evolution and maintenance of test annotations. Our work focuses on the evolution and proposal of new framework features presented as annotations, new test methods, or new API in JUnit 5.

Aiming to pave the way for recommendation tools that allow project developers to choose the most appropriate library for their needs and inform better alternatives, the empirical study

[1] <https://bettercodehub.com/>

performed by Zerouali and Mens (2017) analyzed the usage of eight testing-related libraries in 4,532 open-source Java projects hosted on GitHub. Among their findings, they found that some libraries are considerably more popular than their competitors, while some libraries become more popular over time. From the analyzed test libraries, they found that JUnit was by far the most popular (97%), and that the version 4 of this library was the most popular one.

The book authored by Garcia (2017) proposes a presentation of the available framework features, showcasing usage examples. At the time of the book issue, no experimental features like Temporary Directory and Resource Lock were already available. The book finishes presenting a description list of 27 test anti-patterns, but with no correlation between the newly introduced features and when to prevent such anti-patterns.

6.6 NATURAL LANGUAGE TEST SMELLS

Hauptmann *et al.* (HAUPTMANN et al., 2013) presented possible problems in manual test descriptions performed in natural language from the point of view of test smells. Together with coining the term *Natural Language Test Smells*, the authors propose a set of seven smells: *Hard-Coded Values*, *Long Test Steps*, *Conditional Tests*, *Badly Structured Test Suites*, *Test Clones*, *Ambiguous Tests*, and *Inconsistent Wording*. Also, the authors present identification strategies for their proposals that rely on keyword lists and complimentary metrics (*i.e., number of words*) and the frequency of the proposed test smells in nine industrial test suites. In our work, we extend the current catalog by adding six new test smells, their discovery strategies and frequency, and providing updates for the discovery of two of Hauptmann *et al.*'s list, which we base on broader definitions focused on morphological and syntactical language analysis, thus exploring the capabilities of current Natural Language Processing mechanisms.

Rajkovic and Enoiu presented a tool called NALABS to detect bad smells in natural language requirements and test specifications (RAJKOVIC; ENOIU, 2022). Similarly to Hauptmann *et al.* (HAUPTMANN et al., 2013), the proposed tool uses keyword lists to measure vagueness, referenceability, optionality, subjectivity, and weakness metrics. They also used Automated Readability Index (ARI) to measure readability and the number of words and conjunctions to measure test complexity. Again, our work differentiates from Rajkovic and Enoiu's work because we use current NLP mechanisms to identify words using morphological and syntactical language analysis.

Transferring the concept of code smells to requirements engineering, Femmer *et al.* (FEM-

MER et al., 2017) introduced a lightweight static requirements analysis approach that allows for quick checks when requirements are written down in natural language. In another work, Femmer *et al.* (FEMMER et al., 2014) derived a set of smells from the natural language criteria of the ISO/IEC/IEEE 29148 standard, showing that lightweight smell analysis can uncover many practically relevant requirements defects. Like our work, they also use tool support to analyze text in natural language descriptions.

Previous works presented test smells in test code. Some of these smells are related to ours, although we focused on natural language test smells. Meszaros *et al.* (MESZAROS, 2007) and Peruma *et al.* (PERUMA et al., 2019) studied test smells in test code, such as *Conditional Test* and *Conditional Test Logic*, which are related to Hauptmann *et al.* (HAUPTMANN et al., 2013) natural language test smell. Aljedaani *et al.* (ALJEDAANI et al., 2021) also listed the *Assertionless Test* smell, defined by the absence of assertions, which is similar to our idea of natural language tests having no verification steps (*Unverified Action*).

7 CONCLUDING REMARKS

Test smells are a subject of interest in both industry and academia, which motivates the existence of much literature on the subject. Despite the literature, there is still much to be learned, researched and practiced on test smells. Considering this thesis, a broader question we can use to unify our contributions is "how we advance current knowledge on test smells?" whose possible short answer can be "organize the area, then dive both into the evolution of test frameworks and manual testing." For that consolidated answer, we perform a series of contributions made with a mixed-method approach.

Our first and foremost contribution is the massive effort to provide organized and centralized information to leverage common ground for further, significant, and extensive study on an area that suffers from fragmented information. In this sense, providing a basis for which test smells exist and how to obtain more profound knowledge is a most needed starting point for any development in the area, and this demand had yet to be addressed.

Secondly, of course programming languages and their byproducts (i.e., frameworks and applications) are constantly evolving due to the continuous increase in customer and developer needs. Therefore, it makes sense to consider this evolution when studying strategies to remove the test smells presence and it became a natural result of our study, which is this work's second and equally significant contribution. The successful validation of our propositions with the developers' community members was an encouraging task once it enhanced the probability of spreading our contributions in practical means.

At last, we venture with the sense of contributing to developing an important area within software testing (manual testing) that was apparently dormant on being explored by test smells. The results of using NLP in discovering test smells in manual tests reinforce previous researches on the topic and encourage the development of new research fronts.

We go further and not only answer our broad question with our contributions, but also provide research directions for the areas we boldly dove in this thesis. Now, finally, we believe to have completed our aim to build on existing research and explore essential aspects of test smells in greater depth.

7.1 REVIEW OF CONTRIBUTIONS

Here, we pinpoint our contributions in this thesis:

- We conduct an MLR on 127 selected formal and informal studies to systematically identify and characterize test smells (Section 3.1);
- We create a data set with 1,331 occurrences of 480 test smells extracted from the selected references (Section 3.1.3);
- We create a publicly available and maintainable catalog with 480 test smells containing their names, definitions, AKA, code examples — when available — and related bibliography (Section 3.2);
- We analyze 485 popular Java open-source projects to evaluate the extent of their usage of the JUnit 5 library (Section 4.1.3);
- We propose test code transformations based on 7 JUnit 5 features to remove test smells Section 4.2);
- We survey 212 developers for their opinions and preferences about our proposed transformations, raising community feedback on JUnit 5 features (Section 4.3.1);
- We submit 38 Pull Requests using our transformations to the test suite of popular GitHub Java projects, reaching a 94% acceptance rate; (Section 4.3.2).
- We conduct an exploratory study for natural language test smells on systems of different domains: open-source, government, and industry (Section 5.1);
- We present a catalog of natural language test smells, with six new contributions from our study, along with detection rules that use syntactic and morphological language analysis, representing a novel approach enabled by current NLP technology (Section 5.2);
- We evaluate our catalog with 24 in-company test engineers (Section 5.3);
- We introduce a NLP-based tool to identify the proposed test smells (Section 5.4);
- We evaluate our tool by analyzing a sample of its results concerning the before-mentioned systems (Section 5.5).

7.2 LIMITATIONS

1. Implementation of data collection method. We acknowledge that our data collection experience may be limited, making the process fragile. We focus on collecting data from works written in English that include the research questions' terms. However, we are unsure about the scope of data collection in works written in other languages or if this topic is discussed in those languages. Nonetheless, the probability of this being an issue is low because academic production related to computing is predominantly in English.

2. Sample size. We utilized various samples throughout our studies, which may not always be statistically significant enough to generalize our findings. For instance, in our manual tests, we based our results on the systems we searched, and due to the high number of tests analyzed, we did not use sampling with statistical relevance to generalizations in our exploratory study. Additionally, even though each system analyzed has its importance within its operating context, it is impossible to generalize our findings to other systems.

3. Lack of previous studies in the research area. Literature review findings serve as the basis for researchers to achieve their research objectives. However, in the case of test smells, there is limited prior research, resulting in a weak foundation to identify the scope of this work. This may lead to a focus on less important research topics related to test smells, despite making significant contributions.

4. Scope of discussions. The scope and depth of discussions in this work can be compromised on many levels compared to the works of experienced scholars. For instance, discussions on test code transformations were based on features currently available in the latest JUnit library, and some of these features are experimental. There is no warranty that such features will continue in the following distributions, therefore invalidating our propositions and whether they exist — with some sort of equality — in other less established test automation frameworks.

REFERENCES

A, J. R. *Testing Anti-patterns: How to Fail With 100% Test Coverage*. 2008. Available at: <https://jasonrudolph.com/blog/testing-anti-patterns-how-to-fail-with-100-test-coverage/>. Accessed on: 2022-04-21.

A, M. A. *TDD Antipatterns: The Free Ride*. 2014. Available at: <https://semaphoreci.com/blog/2014/06/24/tdd-antipatterns-the-free-ride.html>. Accessed on: 2022-04-21.

AL2O3CR. *Hacker News on: Software Testing Anti-patterns*. 2018. Available at: <https://news.ycombinator.com/item?id=16895784>. Accessed on: 2022-04-21.

ALJEDAANI, W.; PERUMA, A.; ALJOHANI, A.; ALOTAIBI, M.; MKAOUER, M. W.; OUNI, A.; NEWMAN, C. D.; GHALLAB, A.; LUDI, S. Test smell detection tools: A systematic mapping study. In: *25th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. [S.l.: s.n.], 2021. (EASE), p. 170–180.

ARANEGA, V.; DELPLANQUE, J.; MARTINEZ, M.; BLACK, A. P.; DUCASSE, S.; ETIEN, A.; FUHRMAN, C.; POLITO, G. Rotten green tests in java, pharo and python. *Empir. Softw. Eng.*, v. 26, n. 6, p. 130, 2021.

ARCHER, M. *How test automation with Selenium can fail*. 2010. Available at: <https://mattarcherblog.wordpress.com/2010/11/29/how-test-automation-with-selenium-or-watir-can-fail/>. Accessed on: 2022-04-21.

ATHANASIOU, D.; NUGROHO, A.; VISSER, J.; ZAIDMAN, A. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, v. 40, n. 11, p. 1100–1125, 2014.

B, J. R. *Testing anti-patterns: The ugly mirror*. 2008. Available at: <https://jasonrudolph.com/blog/2008/07/30/testing-anti-patterns-the-ugly-mirror/>. Accessed on: 2022-06-17.

B, M. A. *TDD Antipatterns: Local Hero*. 2014. Available at: <https://semaphoreci.com/blog/2014/07/10/tdd-antipatterns-local-hero.html>. Accessed on: 2022-06-17.

BARRAK, A.; EGHAN, E. E.; ADAMS, B.; KHOMH, F. Why do builds fail?—a conceptual replication study. *J. Syst. Softw.*, v. 177, p. 110939, 2021.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The goal question metric approach. *Encyclopedia of Software Engineering*, v. 1, p. 528–532, 1994.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; LUCIA, A. D.; BINKLEY, D. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: *2012 28th IEEE international conference on software maintenance (ICSM)*. [S.l.: s.n.], 2012. (ICSM), p. 56–65.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; LUCIA, A. D.; BINKLEY, D. Are test smells really harmful? an empirical study. *Empir. Softw. Eng.*, v. 20, n. 4, p. 1052–1094, 2015.

BECHTOLD, S.; BRANNEN, S.; LINK, J.; MERDES, M.; PHILIPP, M.; RANCOURT, J. de; STEIN, C. *JUnit 5 User Guide*. 2020. Accessed on Feb 2021. Available at: <https://junit.org/junit5/docs/current/user-guide/>.

BISANZ, M. *Pattern-based smell detection in TTCN-3 test suites*. Master's Thesis (Master's Thesis) — Institute for Informatics, Universität Göttingen, Germany, Dec 2006.

BLESER, J. D.; NUCCI, D. D.; ROOVER, C. D. Assessing diffusion and perception of test smells in Scala projects. In: *IEEE/ACM 16th International Conference on Mining Software Repositories*. [S.l.: s.n.], 2019. (MSR), p. 457–467.

BLESER, J. D.; NUCCI, D. D.; ROOVER, C. D. SoCRATES: Scala radar for test smells. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*. [S.l.: s.n.], 2019. (Scala), p. 22–26.

BORBA, P.; SAMPAIO, A.; CORNÉLIO, M. A refinement algebra for object-oriented programming. In: *European Conference on Object-oriented Programming*. [S.l.: s.n.], 2003. (ECOOP), p. 457–482.

BRANDES, R. *A workbook repository of example test smells and what to do about them*. 2021. Available at: <https://github.com/testdouble/test-smells>. Accessed on: 2022-04-21.

BREUGELMANS, M.; ROMPAEY, B. V. Testq: Exploring structural and maintenance characteristics of unit test suites. In: *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. [S.l.: s.n.], 2008. (WASDeTT), p. 1–16.

BUFFARDI, K.; AGUIRRE-AYALA, J. Unit test smells and accuracy of software engineering student test suites. In: *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education*. [S.l.: s.n.], 2021. (ITiCSE), p. 234–240.

BUGAYENKO, Y. *A Few Thoughts on Unit Test Scaffolding*. 2015. Available at: <https://www.yegor256.com/2015/05/25/unit-test-scaffolding.html>. Accessed on: 2022-04-21.

BUGAYENKO, Y. *Unit Testing Anti-Patterns, Full List*. 2018. Available at: <https://www.yegor256.com/2018/12/11/unit-testing-anti-patterns.html>. Accessed on: 2022-04-21.

BURNS, B. *Anti-Patterns In Unit Testing*. 2021. Available at: <https://completedeveloperpodcast.com/anti-patterns-in-unit-testing/>. Accessed on: 2022-06-17.

BUWALDA, H. *Test Design for Automation: Anti-Patterns*. 2015. Available at: <https://www.techwell.com/techwell-insights/2015/09/test-design-automation-anti-patterns>. Accessed on: 2022-06-17.

CAMARA, B.; SILVA, M.; ENDO, A.; VERGILIO, S. On the use of test smells for prediction of flaky tests. In: *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing*. [S.l.: s.n.], 2021. (SAST), p. 46–54.

CAMPOS, D.; ROCHA, L.; MACHADO, I. Developers perception on the severity of test smells: an empirical study. ArXiv:2107.13902. 2021.

CAR, J. *Test-Driven Development: TDD Anti-Patterns*. 2009. Available at: <https://bryanwilhite.github.io/the-funky-knowledge-base/entry/kb2076072213/>. Accessed on: 2022-04-21.

CHAMBERLAIN, N. *How to Compare Object Instances in your Unit Tests Quickly and Easily*. 2017. Available at: <https://buildplease.com/pages/testing-deep-equalilty/>. Accessed on: 2022-04-21.

CHEN, W.-K.; WANG, J.-C. Bad smells and refactoring methods for GUI test scripts. In: *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. [S.l.: s.n.], 2012. (SNPD), p. 289–294.

CROAK, D. *Mystery Guest*. 2009. Available at: <https://thoughtbot.com/blog/mystery-gue st>. Accessed on: 2022-06-17.

DESIKAN, S. *Software testing: principles and practice*. [S.l.]: Pearson Education India, 2006.

DEURSEN, A. van; MOONEN, L.; BERGH, A. van D.; KOK, G. Refactoring test code. In: *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. [S.l.: s.n.], 2001. (XP), p. 92–95.

DIFALCO, R. *Subclass To Test Anti Pattern*. 2011. Available at: <https://wiki.c2.com/?S ubclassToTestAntiPattern>. Accessed on: 2022-04-21.

DUSTIN, E.; RASHKA, J.; PAUL, J. *Automated software testing: introduction, management, and performance*. [S.l.]: Addison-Wesley Professional, 1999.

ENGLAND, T. *Cucumber anti-patterns (part one)*. 2016. Available at: <https: //cucumber.io/blog/bdd/cucumber-antipatterns-part-one/>. Accessed on: 2022-06-17.

FEMMER, H.; FERNáNDEZ, D. M.; JUERGENS, E.; KLOSE, M.; ZIMMER, I.; ZIMMER, J. Rapid requirements checks with requirements smells: Two case studies. In: *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. [S.l.: s.n.], 2014. (RCoSE 2014), p. 10–19.

FEMMER, H.; FERNÁNDEZ, D. M.; WAGNER, S.; EDER, S. Rapid quality assurance with requirements smells. *Journal of Systems and Software*, Elsevier, v. 123, p. 190–213, 2017.

FERNANDES, D.; MACHADO, I.; MACIEL, R. Handling test smells in python: Results from a mixed-method study. In: *Proceedings of the XXXV Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2021. (SBES), p. 84–89.

FERRIS, J. *Let's Not*. 2012. Available at: <https://thoughtbot.com/blog/lets-not>. Accessed on: 2022-04-21.

FOWLER, M. *Eradicating non-determinism in tests*. 2011. Accessed on Feb 2022. Available at: <https://martinfowler.com/articles/nonDeterminism.html>.

FOWLER, M. *Refactoring: improving the design of existing code, 2nd edition*. [S.l.]: Addison-Wesley Professional, 2018.

FRIEZE, A. *Test Smells - The Coding Craftsman*. 2018. Available at: <https: //codingcraftsman.wordpress.com/2018/09/27/test-smells/>. Accessed on: 2022-04-21.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley Professional, 1994.

GARCIA, B. *Mastering Software Testing with JUnit 5: Comprehensive guide to develop high quality Java applications*. [S.l.]: Packt Publishing Ltd, 2017.

GAROUSI, V.; FELDERER, M.; MÄNTYLÄ, M. V. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, Elsevier, v. 106, p. 101–121, 2019.

GAROUSI, V.; KÜÇÜK, B. Smells in software test code: A survey of knowledge in industry and academia. *J. Syst. Softw.*, v. 138, p. 52–81, 2018.

GAROUSI, V.; KUCUK, B.; FELDERER, M. What we know about smells in software test code. *IEEE Software*, v. 36, n. 3, p. 61–73, 2018.

GAWINECKI, M. *Anti-patterns in test automation*. 2019. Available at: <https://www.codementor.io/@mgawinecki/anti-patterns-in-test-automation-101c6vm5jz>. Accessed on: 2022-06-17.

GISHU. *Unit testing Anti-patterns catalogue*. 2014. Available at: <https://stackoverflow.com/questions/333682/unit-testing-anti-patterns-catalogue>. Accessed on: 2022-06-17.

GONZALEZ, D.; RATH, M.; MIRAKHORLI, M. Did you remember to test your tokens? In: *Proceedings of the 17th International Conference on Mining Software Repositories*. [S.l.: s.n.], 2020. (MSR), p. 232–242.

GRANO, G.; IACO, C. D.; PALOMBA, F.; GALL, H. C. Pizza versus pinsa: On the perception and measurability of unit test code quality. In: *2020 IEEE international conference on software maintenance and evolution (ICSME)*. [S.l.: s.n.], 2020. (ICSME), p. 336–347.

GRANO, G.; PALOMBA, F.; NUCCI, D. D.; LUCIA, A. D.; GALL, H. C. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *J. Syst. Softw.*, v. 156, p. 312–327, 2019.

GREILER, M.; DEURSEN, A. V.; STOREY, M.-A. Automated detection of test fixture strategies and smells. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. [S.l.: s.n.], 2013. (ICST), p. 322–331.

GREILER, M.; ZAIDMAN, A.; DEURSEN, A. V.; STOREY, M.-A. Strategies for avoiding text fixture smells during software evolution. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2013. (MSR), p. 387–396.

GUERRA, E. M.; FERNANDES, C. T. Refactoring test code safely. In: *International Conference on Software Engineering Advances*. [S.l.: s.n.], 2007. (ICSEA), p. 44–44.

HALLEUX, P. de; TILLMANN, N. *Parameterized test patterns for effective testing with Pex*. 2010. Available at: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=052fc4a27b6e928a4a65f57aee3e4fb9081410f5>. Accessed on: 2022-04-21.

HAMMERLY, A. *Going on a Testing Anti-Pattern Safari*. 2013. Available at: <https://www.youtube.com/watch?v=VBgySRk0VKY>. Accessed on: 2022-04-21.

HAUPTMANN, B.; EDER, S.; JUNKER, M.; JUERGENS, E.; WOINKE, V. Generating refactoring proposals to remove clones from automated system tests. In: *2015 IEEE 23rd International Conference on Program Comprehension*. [S.l.: s.n.], 2015. (ICPC), p. 115–124.

HAUPTMANN, B.; JUNKER, M.; EDER, S.; HEINEMANN, L.; VAAS, R.; BRAUN, P. Hunting for smells in natural language tests. In: *2013 35th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2013. (ICSE), p. 1217–1220.

HEDAYATI, A.; EBRAHIMZADEH, M.; SORI, A. A. Investigating into automated test patterns in erratic tests by considering complex objects. *Int. J. Inf. Technol. Comput. Sci.*, v. 7, n. 3, p. 54–59, 2015.

HONNIBAL, M.; MONTANI, I. spacy 2: Natural language understanding with bloom embeddings, convolutional neural networks and incremental parsing. *To appear*, v. 7, n. 1, p. 411–420, 2017.

HONNIBAL, M.; MONTANI, I. *spaCy – Industrial-strength Natural Language Processing in Python*. 2023. Available at: <https://spacy.io/>. Accessed on: 2023-05-02.

II, J. E. B.; KOTRLIK, J. W.; HIGGINS, C. C. Organizational research: Determining appropriate sample size in survey research appropriate sample size in survey research. *Information Technology, Learning, and Performance Journal*, v. 19, n. 1, p. 43–50, 2001.

JONES, A. *7 ways to tidy up your test code*. 2019. Available at: <https://techbeacon.com/app-dev-testing/7-ways-tidy-your-test-code>. Accessed on: 2022-06-17.

JONES, K. S. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, MCB UP Ltd, v. 28, n. 1, p. 11–21, 1972.

JORGE, D.; MACHADO, P.; ANDRADE, W. Investigating test smells in javascript test code. In: *6th Brazilian Symposium on Systematic and Automated Software Testing*. [S.l.: s.n.], 2021. (SAST), p. 36–45.

JUHNKE, K.; NIKIC, A.; TICHY, M. Clustering natural language test case instructions as input for deriving automotive testing dsls. *The Journal of Object Technology*, v. 20, n. 3, p. 1–14, 2021.

JUNIOR, N. S.; ROCHA, L.; MARTINS, L. A.; MACHADO, I. *A survey on test practitioners' awareness of test smells*. 2020. Accessed on: arXiv:2003.05613.

KACZANOWSKI, T. *Bad Tests, Good Tests*. [S.l.]: Tomasz Kaczanowski, 2013.

KAPELONIS, K. *Software Testing Anti-patterns*. 2018. Available at: <http://blog.codepipes.com/testing/software-testing-antipatterns.html>. Accessed on: 2022-04-21.

KARHU, K.; REPO, T.; TAIPALE, O.; SMOLANDER, K. Empirical observations on software testing automation. In: *Second International Conference on Software Testing Verification and Validation*. [S.l.: s.n.], 2009. (ICST), p. 201–209.

KEMPF, W. E. *Anti-Patterns - Digital Tapestry*. 2016. Available at: <https://digitaltapestry.net/testify/manual/AntiPatterns.html>. Accessed on: 2022-04-21.

KHALILI, M. *Maintainable Automated UI Tests*. 2013. Available at: <https://code.tutsplus.com/articles/maintainable-automated-ui-tests--net-35089>. Accessed on: 2022-04-21.

KHALILI, M. *Tips to Avoid Brittle UI Tests*. 2013. Available at: <https://code.tutsplus.com/tutorials/tips-to-avoid-brittle-ui-tests--net-35188>. Accessed on: 2022-04-21.

KHORIKOV, V. *Code pollution*. 2018. Available at: <https://enterprisecraftsmanship.com/posts/code-pollution/>. Accessed on: 2022-04-21.

KIM, D. An empirical study on the evolution of test smell. In: *IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings*. [S.l.: s.n.], 2020. (ICSE-Companion), p. 149–151.

KIM, D. J.; CHEN, T.-H. P.; YANG, J. The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empir. Softw. Eng.*, v. 26, n. 5, p. 100, 2021.

KIM, D. J.; TSANTALIS, N.; CHEN, T.-H. P.; YANG, J. Studying test annotation maintenance in the wild. In: *IEEE/ACM 43rd International Conference on Software Engineering*. [S.l.: s.n.], 2021. (ICSE), p. 62–73.

KIM, D. J.; YANG, B.; YANG, J.; CHEN, T.-H. P. How disabled tests manifest in test maintainability challenges? In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2021. (ESEC/FSE), p. 1045–1055.

KING, T. *Test::Class Hierarchy Is an Antipattern*. 2018. Available at: <https://medium.com/cultured-perl/test-class-hierarchy-is-an-antipattern-391c6ef1e491>. Accessed on: 2022-04-21.

KIRKBRIDE, J. *Testing Anti-Patterns*. 2014. Available at: <https://medium.com/@jameskbride/testing-anti-patterns-b5ffc1612b8b>. Accessed on: 2022-06-17.

KOSKELA, L. *Developer Test Anti-Patterns*. 2015. Available at: <https://www.youtube.com/watch?v=3Fa69eQ6XgM>. Accessed on: 2022-06-17.

KULESOVS, I. ios applications testing. In: *Proceedings of the International Scientific and Practical Conference*. [S.l.: s.n.], 2015. (ISPC, v. 3), p. 138–150.

KUMMER, M. *Categorising Test Smells*. Master's Thesis (Master's Thesis) — University of Bern, Switzerland, Mar 2015.

KUNDRA, G. *Enhancing developers' awareness on test suites' quality with test smell summaries*. Master's Thesis (Master's Thesis) — Lappeenranta University of Technology, Finland, Nov 2018.

LAMBIASE, S.; CUPITO, A.; PECORELLI, F.; LUCIA, A. D.; PALOMBA, F. Just-in-time test smell detection and refactoring: The DARTS project. In: *Proceedings of the 28th international conference on program comprehension*. [S.l.: s.n.], 2020. (ICPC), p. 441–445.

LANUBILE, F.; MALLARDO, T. Inspecting automated test code: A preliminary study. In: *Agile Processes in Software Engineering and Extreme Programming: 8th International Conference*. [S.l.: s.n.], 2007. (XP), p. 115–122.

LEGENDRE, P. Species associations: the kendall coefficient of concordance revisited. *Journal of agricultural, biological, and environmental statistics*, Springer, v. 10, p. 226–245, 2005.

LEWIS, E. *How to write good tests*. 2019. Available at: <https://github.com/mockito/mockito/wiki/How-to-write-good-tests>. Accessed on: 2022-06-17.

LTD., C. *Ubuntu Operational System*. 2023. Available at: <https://ubuntu.com/download>. Accessed on: 2023-05-02.

LUO, Q.; HARIRI, F.; ELOUSSI, L.; MARINOV, D. An empirical analysis of flaky tests. In: *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2014. (FSE), p. 643–653.

MARABESI, M. *TDD Anti-Patterns*. 2021. Available at: <https://marabesi.com/tdd/2021/08/28/tdd-anti-patterns.html>. Accessed on: 2022-04-21.

MARABESI, M. *TDD anti patterns - Chapter 1*. 2021. Available at: <https://www.codurance.com/publications/tdd-anti-patterns-chapter-1>. Accessed on: 2022-06-17.

MARCUS, M. P.; SANTORINI, B.; MARCINKIEWICZ, M. A. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, MIT Press, Cambridge, MA, v. 19, n. 2, p. 313–330, 1993.

MARTINEZ, M.; ETIEN, A.; DUCASSE, S.; FUHRMAN, C. RTj: A java framework for detecting and refactoring rotten green test cases. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. [S.l.: s.n.], 2020. (ICSE), p. 69—72.

MARTINS, L.; BEZERRA, C.; COSTA, H.; MACHADO, I. Smart prediction for refactorings in the software test code. In: *Proceedings of the XXXV Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2021. (SBES), p. 115–120.

MARTINS, L.; COSTA, H.; MACHADO, I. On the diffusion of test smells and their relationship with test code quality of java projects. *Journal of Software: Evolution and Process*, Wiley Online Library, p. e2532, 2023.

MATHEW, D.; FOEGEN, K. An analysis of information needs to detect test smells. In: *Full-scale Software Engineering/Current Trends in Release Engineering*. [S.l.]: RWTH Aachen University, 2016. p. 19–24.

MESZAROS, G. *xUnit test patterns: Refactoring test code*. [S.l.]: Pearson Education, 2007.

MESZAROS, G. *Obscure Test*. 2009. Available at: <http://xunitpatterns.com/Obscure%20Test.html>. Accessed on: 2022-06-17.

MEYER, B. Seven principles of software testing. *Computer*, v. 41, n. 8, p. 99–101, 2008.

MOONEN, L.; DEURSEN, A. van; ZAIDMAN, A.; BRUNTINK, M. On the interplay between software testing and evolution and its effect on program comprehension. In: *Softw. Evolution*. [S.l.]: Springer, 2008. p. 173–202.

MURPHY-HILL, E.; BLACK, A. P. Refactoring tools: Fitness for purpose. *IEEE Software*, v. 25, n. 5, p. 38–44, 2008.

NERY, E.; LIMA, M. *Test Artifacts*. 2020. Available at: <https://damorimrg.github.io/practical_testing_book/goodpractices/artifacts.html>. Accessed on: 2022-06-17.

NEUKIRCHEN, H.; BISANZ, M. Utilising code smells to detect quality problems in TTCN-3 test suites. In: *International Workshop on Formal Approaches to Software Testing*. [S.l.: s.n.], 2007. (TestCom), p. 228–243.

NEUKIRCHEN, H.; ZEISS, B.; GRABOWSKI, J. An approach to quality engineering of TTCN-3 test specifications. *Int. J. Softw. Tools Technol. Transf.*, v. 10, n. 4, p. 309–326, 2008.

NIVRE, J.; MARNEFFE, M.-C. de; GINTER, F.; GOLDBERG, Y.; HAJIČ, J.; MANNING, C. D.; MCDONALD, R.; PETROV, S.; PYYSALO, S.; SILVEIRA, N.; TSARFATY, R.; ZEMAN, D. Universal Dependencies v1: A multilingual treebank collection. In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*. [S.l.: s.n.], 2016. p. 1659–1666.

NURKIEWICZ, T. *Java: code duplication in classes and their Junit test cases*. 2012. Available at: <https://stackoverflow.com/questions/10781050/java-code-duplication-in-classes-and-their-junit-test-cases>. Accessed on: 2022-06-17.

NöDLER, J.; NEUKIRCHEN, H.; GRABOWSKI, J. A flexible framework for quality assurance of software artefacts with applications to Java, UML, and TTCN-3 test specifications. In: *Second International Conference on Software Testing Verification and Validation*. [S.l.: s.n.], 2009. (ICST), p. 101–110.

OGAWA, R. T.; MALEN, B. Towards rigor in reviews of multivocal literatures: Applying the exploratory case study method. *Rev. Educ. Res.*, v. 61, n. 3, p. 265–286, 1991.

OSHEROVE, R. *Chapter 8. The pillars of good unit tests*. 2016. Available at: <https://apprize.best/c/unit/8.html>. Accessed on: 2022-04-21.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D. Do they really smell bad? a study on developers' perception of bad code smells. In: *30th IEEE International Conference on Software Maintenance and Evolution*. [S.l.: s.n.], 2014. (ICSME), p. 101–110.

PALOMBA, F.; NUCCI, D. D.; PANICHELLA, A.; OLIVETO, R.; LUCIA, A. D. On the diffusion of test smells in automatically generated test code: An empirical study. In: *Proceedings of the 9th international workshop on search-based software testing*. [S.l.: s.n.], 2016. (SBST), p. 5–14.

PALOMBA, F.; ZAIDMAN, A.; LUCIA, A. D. Automatic test smell detection using information retrieval techniques. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2018. (ICSME), p. 311–322.

PANICHELLA, A.; PANICHELLA, S.; FRASER, G.; SAWANT, A. A.; HELLENDOORN, V. J. Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities. In: *2020 IEEE international conference on software maintenance and evolution (ICSME)*. [S.l.: s.n.], 2020. (ICSME), p. 523–533.

PANICHELLA, A.; PANICHELLA, S.; FRASER, G.; SAWANT, A. A.; HELLENDOORN, V. J. Test smells 20 years later: detectability, validity, and reliability. *Empirical Software Engineering*, Springer, v. 27, n. 7, p. 170, 2022.

PATTON, M. Q. Towards utility in reviews of multivocal literatures. *Rev. Educ. Res.*, v. 61, n. 3, p. 287–292, 1991.

PAULA, E.; BONIFáCIO, R. Testaxe: Automatically refactoring test smells using junit 5 features. In: *Anais Estendidos do XIII Congresso Brasileiro de Software: Teoria e Prática*. [S.l.: s.n.], 2022. p. 89–98.

PAVLENKO, A. Quality defects detection in unit tests. *Ukrainian Eng. Softw. J.*, v. 6, n. 2, p. 24–28, 2011.

PECORELLI, F. Test-related factors and post-release defects: An empirical study. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2019. (ESEC/FSE), p. 1235–1237.

PECORELLI, F.; PALOMBA, F.; LUCIA, A. D. The relation of test-related factors to software quality: A case study on apache systems. *Empir. Softw. Eng.*, v. 26, n. 2, p. 18, 2021.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. On the distribution of test smells in open source android applications: An exploratory study. In: *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. [S.l.: s.n.], 2019. (CASCON), p. 193–202.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. TsDetect: An open source test smells detection tool. In: *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. [S.l.: s.n.], 2020. (ESEC/FSE), p. 1650–1654.

PERUMA, A.; MKAOUER, M. W.; ALMALKI, K.; NEWMAN, C. D.; OUNI, A.; PALOMBA, F. *Software Unit Test Smells*. 2019. Available at: <https://testsmells.org/>. Accessed on: 2022-04-21.

PERUMA, A.; NEWMAN, C. D. On the distribution of "simple stupid bugs" in unit test files: An exploratory study. In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2021. (MSR), p. 525–529.

PERUMA, A.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. An exploratory study on the refactoring of unit test files in android applications. In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. [S.l.: s.n.], 2020. (ICSEW), p. 350–357.

PERUMA, A. S. A. *What the smell? an empirical investigation on the distribution and severity of test smells in open source android applications*. Master's Thesis (Master's Thesis) — Rochester Institute of Technology, USA, Apr 2018.

PONTILLO, V.; PALOMBA, F.; FERRUCCI, F. Toward static test flakiness prediction: A feasibility study. In: *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*. [S.l.: s.n.], 2021. (MaLTESQuE), p. 19–24.

POWERS, D. M. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*, 2020.

QUSEF, A.; ELISH, M. O.; BINKLEY, D. An exploratory study of the relationship between software test smells and fault-proneness. *IEEE Access*, v. 7, p. 139526–139536, 2019.

RAJKOVIC, K.; ENOIU, E. P. *NALABS: Detecting Bad Smells in Natural Language Requirements and Test Specifications*. 2022. Available at: <http://www.es.mdu.se/publications/6382->. Accessed on: 2023-05-02.

REICHART, S. *Assessing test quality - TestLint*. Phd Thesis (PhD Thesis) — Universitat Bern, Switzerland, Dec 2007.

REICHHART, S.; GÎRBA, T.; DUCASSE, S. Rule-based assessment of test quality. *J. Object Technol.*, v. 6, n. 9, p. 231–251, 2007.

RIJSBERGEN, C. J. V. Foundation of evaluation. *Journal of documentation*, MCB UP Ltd, v. 30, n. 4, p. 365–373, 1974.

RODRIGUES, M. *TDD Anti-patterns: The Free Ride / Piggyback*. 2018. Available at: <https://matheus.ro/2018/04/30/tdd-antipatterns-the-free-ride-piggyback/>. Accessed on: 2022-06-17.

ROMPAEY, B. V.; BOIS, B. D.; DEMEYER, S. Characterizing the relative significance of a test smell. In: *2006 22nd IEEE International Conference on Software Maintenance*. [S.l.: s.n.], 2006. (ICSME), p. 391–400.

ROMPAEY, B. V.; BOIS, B. D.; DEMEYER, S.; RIEGER, M. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, v. 33, n. 12, p. 800–817, 2007.

RWEMALIKA, R. *On the Maintenance of System User Interactive Tests*. Phd Thesis (PhD Thesis) — University of Luxembourg, Luxembourg, Sep 2021.

RWEMALIKA, R.; HABCHI, S.; PAPADAKIS, M.; TRAON, Y. L.; BRASSEUR, M. *Smells in System User Interactive Tests*. 2021. Accessed on: arXiv:2111.02317.

SALTZER, J. H. The origin of the "mit license". *IEEE Annals of the History of Computing*, IEEE, v. 42, n. 4, p. 94–98, 2020.

SANTANA, R.; MARTINS, L.; ROCHA, L.; VIRGÍNIO, T.; CRUZ, A.; COSTA, H.; MACHADO, I. RAIDE: A tool for assertion roulette and duplicate assert identification and refactoring. In: *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2020. (SBES), p. 374–379.

SCHMENGLER, F. *Test Smell: Hard Coded Values*. 2018. Available at: <https://www.integer-net.com/test-smell-hard-coded-values/>. Accessed on: 2022-06-17.

SCHMETZER, J. *JUnit Anti-patterns*. 2005. Available at: <https://exubero.com/junit/anti-patterns/>. Accessed on: 2022-04-21.

SCHVARCBACHER, M.; SPADINI, D.; BRUNTINK, M.; OPRESCU, A. Investigating developer perception on test smells using better code hub-work in progress. In: *Seminar Series on Advanced Techniques & Tools for Software Evolution (SATTOSE 2019)*. [S.l.: s.n.], 2019. (CEUR Workshop Proceedings), p. 1–6.

SCOTT, A. *Five automated acceptance test anti-patterns*. 2015. Available at: <https://alisterbscott.com/2015/01/20/five-automated-acceptance-test-anti-patterns/>. Accessed on: 2022-04-21.

SCRUGGS, J. *Smells of Testing (signs your tests are bad)*. 2009. Available at: <https://jakescruggs.blogspot.com/2009/04/smells-of-testing-signs-your-tests-are.html>. Accessed on: 2022-04-21.

SILVA, L. P. d.; VILAIN, P. Lccss: A similarity metric for identifying similar test code. In: *Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse*. [S.l.: s.n.], 2020. (SBCARS), p. 91–100.

SILVA, T. A. *Rails Testing Antipatterns: Fixtures and Factories*. 2021. Available at: <https://semaphoreci.com/blog/2014/01/14/rails-testing-antipatterns-fixtures-and-factories.html>. Accessed on: 2022-04-21.

SKRONDAL, B. E. A.; EVERITT, A. *The Cambridge Dictionary of Statistics*. [S.l.]: Cambridge University Press Cambridge, 2010.

S.M.K, Q.; FAROOQ, S. U. Software testing – goals, principles, and limitations. *International Journal of Computer Applications*, v. 6, n. 9, p. 7–10, 2010.

SOARES, E.; ARANDA, M.; OLIVEIRA, N.; RIBEIRO, M.; SOUZA, E.; MACHADO, I.; SANTOS, A.; FONSECA, B.; BONIFáCIO, R. *Do They Smell the Same? Cataloging Natural Language Test Smells in Open-Source, Government, and Industry Systems - Files*. 2022. Available at: <https://figshare.com/s/5661004111cbdcb785d3>.

SOARES, E.; COSTA, J. A. da; TERCEIRO, M.; ROMãO, D.; RIBEIRO, M.; GHEYI, R.; FERRARI, F.; SANTOS, A.; MACHADO, I.; BONIFáCIO, R.; FONSECA, B. *A Multivocal Literature Review of Test Smells - Replication Package*. 2022. Available at: <https://doi.org/10.6084/m9.figshare.22806092>.

SOARES, E.; RIBEIRO, M.; AMARAL, G.; GHEYI, R.; FERNANDES, L.; GARCIA, A.; FONSECA, B.; SANTOS, A. Refactoring test smells: A perspective from open-source developers. In: *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*. [S.l.: s.n.], 2020. (SAST), p. 50–59.

SOARES, E.; RIBEIRO, M.; GHEYI, R.; AMARAL, G.; SANTOS, A. Refactoring test smells with JUnit 5: Why should developers keep up-to-date? *IEEE Transactions on Software Engineering*, v. 49, n. 3, p. 1152–1170, 2023.

SOARES, L. *Anti-patterns of automated testing*. 2020. Available at: <https://medium.com/swlh/anti-patterns-of-automated-software-testing-b396283a4cb6>. Accessed on: 2022-04-21.

SPADINI, D.; PALOMBA, F.; ZAIDMAN, A.; BRUNTINK, M.; BACCHELLI, A. On the relation of test smells to software code quality. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2018. (ICSME), p. 1–12.

SPADINI, D.; SCHVARCBACHER, M.; OPRESCU, A.-M.; BRUNTINK, M.; BACCHELLI, A. Investigating severity thresholds for test smells. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. [S.l.: s.n.], 2020. (MSR), p. 311–321.

SPITZER, D. *Is duplicated code more tolerable in unit tests?* 2008. Available at: <https://stackoverflow.com/questions/129693/is-duplicated-code-more-tolerable-in-unit-tests>. Accessed on: 2022-04-21.

SWETT, J. *Test smell: Obscure Test*. 2018. Available at: <https://www.codewithjason.com/test-smell-obscure-test/>. Accessed on: 2022-06-17.

TAHIR, A.; COUNSELL, S.; MACDONELL, S. G. An empirical study into the relationship between class features and test smells. In: *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. [S.l.: s.n.], 2016. (APSEC), p. 137–144.

TAHIR, A.; DIETRICH, J.; COUNSELL, S.; LICORISH, S.; YAMASHITA, A. A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. *Information Software Technology*, v. 125, p. 106333, 2020.

TANIGUCHI, M.; MATSUMOTO, S.; KUSUMOTO, S. JTDog: a gradle plugin for dynamic test smell detection. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2021. (ASE), p. 1271–1275.

TOM, E.; AURUM, A.; VIDGEN, R. An exploration of technical debt. *Journal of Systems and Software*, v. 86, n. 6, p. 1498–1516, 2013.

TOOMEY, C.; FERRIS, J. *Testing Antipatterns*. 2022. Available at: <https://thoughtbot.com/upcase/videos/testing-antipatterns>. Accessed on: 2022-06-17.

TUFANO, M.; PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D.; POSHYVANYK, D. An empirical investigation into the nature of test smells. In: *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. [S.l.: s.n.], 2016. (ASE), p. 4–15.

TUFANO, M.; PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D.; POSHYVANYK, D. *Towards Automated Tools for Detecting Test Smells: An Empirical Investigation into the Nature of Test Smells*. 2019. Accessed on: 2022-06-17.

VIRGÍNIO, T.; SANTANA, R.; MARTINS, L. A.; SOARES, L. R.; COSTA, H.; MACHADO, I. On the influence of test smells on test coverage. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2019. (SBES), p. 467–471.

WANG, T.; GOLUBEV, Y.; SMIRNOV, O.; LI, J.; BRYKSIN, T.; AHMED, I. Pynose: A test smell detector for python. In: *2021 36th IEEE/ACM international conference on automated software engineering (ASE)*. [S.l.: s.n.], 2021. (ASE), p. 593–605.

WIGENT, Z. *Test Naming Failures. An Exploratory Study of Bad Naming Practices in Test Code*. Master's Thesis (Master's Thesis) — Rochester Institute of Technology, USA, Dec 2021.

WILLIAMS, R.; DIETRICH, E. *Unit Testing Smells: What Are Your Tests Telling You?* 2017. Available at: <https://dzone.com/articles/unit-testing-smells-what-are-your-tests-telling-yo>. Accessed on: 2022-04-21.

WOHLIN, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. [S.l.: s.n.], 2014. (EASE), p. 1–10.

WOOD, J. *ABAP Assertion Anti-Patterns*. 2013. Available at: <https://blogs.sap.com/2013/02/14/abap-assertion-anti-patterns/>. Accessed on: 2022-04-21.

XIE, T.; ZHAO, J.; MARINOV, D.; NOTKIN, D. Detecting redundant unit tests for aspectj programs. In: *2006 17th International Symposium on Software Reliability Engineering*. [S.l.: s.n.], 2006. (ISSRE), p. 179–190.

ZEISS, B. *A refactoring tool for TTCN-3*. Master's Thesis (Master's Thesis) — Institute for Informatics, Universität Göttingen, Germany, Mar 2006.

ZEROUALI, A.; MENS, T. Analyzing the evolution of testing library usage in open source Java projects. In: *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*. [S.l.: s.n.], 2017. (SANER), p. 417–421.

ZHI, J.; GAROUSI, V. On adequacy of assertions in automated test suites: An empirical investigation. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. [S.l.: s.n.], 2013. (ICSTW), p. 382–391.

APPENDIX A – SELECTED PRIMARY SOURCES

Table 31 – Selected primary sources

Author	Title
Deursen et al. (2001)	Refactoring Test Code
Schmetzer (2005)	JUnit Anti-patterns
Zeiß (2006)	A Refactoring Tool for TTCN-3
Rompaey, Bois and Demeyer (2006)	Characterizing the Relative Significance of a Test Smell
Xie et al. (2006)	Detecting redundant unit tests for AspectJ programs
Bisanz (2006)	Pattern-based Smell Detection in TTCN-3 Test Suites
Reichart (2007)	Assessing test quality - TestLint
Lanubile and Mallardo (2007)	Inspecting Automated Test Code: A Preliminary Study
Rompaey et al. (2007)	On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test
Reichhart, Gîrba and Ducasse (2007)	Rule-based Assessment of Test Quality
Neukirchen and Bisanz (2007)	Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites
Meszaros (2007)	xUnit test patterns: Refactoring test code
Neukirchen, Zeiss and Grabowski (2008)	An approach to quality engineering of TTCN-3 test specifications
Spitzer (2008)	Is duplicated code more tolerable in unit tests?
Moonen et al. (2008)	On the interplay between software testing and evolution and its effect on program comprehension
A (2008)	Testing anti-patterns: How to fail with 100% test coverage
B (2008)	Testing anti-patterns: The ugly mirror
Breugelmans and Rompaey (2008)	TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites
Croak (2009)	Mystery Guest
Scruggs (2009)	Smells of Testing (signs your tests are bad)
Car (2009)	Test-Driven Development: TDD Anti-Patterns
Archer (2010)	How test automation with selenium can fail
Halleux and Tillmann (2010)	Parameterized Test Patterns For Effective Testing with Pex
Meszaros (2009)	Obscure Test
Pavlenko (2011)	Quality defects detection in unit tests
DiFalco (2011)	Subclass To Test Anti Pattern
Bavota et al. (2012)	An empirical analysis of the distribution of unit test smells and their impact on software maintenance
Chen and Wang (2012)	Bad smells and refactoring methods for GUI test script
Nurkiewicz (2012)	Java: code duplication in classes and their junit test cases
Ferris (2012)	Let’s not
Hammerly (2013)	A testing anti-pattern safari
Wood (2013)	Abap assertion anti-patterns
Greiler, Deursen and Storey (2013)	Automated Detection of Test Fixture Strategies and Smells
Kaczanowski (2013)	Bad tests, good tests
Osherove (2016)	Chapter 8. The pillars of good unit tests
England (2016)	Cucumber anti-patterns (part one)
Hauptmann et al. (2013)	Hunting for smells in natural language tests
Khalili (2013a)	Maintainable automated ui tests
Zhi and Garousi (2013)	On adequacy of assertions in automated test suites: an empirical investigation
Greiler et al. (2013)	Strategies for avoiding text fixture smells during software evolution
Khalili (2013b)	Tips to avoid brittle ui tests
A (2014)	Tdd antipatterns: Local hero
B (2014)	Tdd antipatterns: The free ride
Athanasiou et al. (2014)	Test code quality and its relation to issue handling performance
Kirkbride (2014)	Testing anti-patterns
Gishu (2014)	Unit testing Anti-patterns catalogue
Bugayenko (2015)	A few thoughts on unit test scaffolding

Bavota et al. (2015)	Are test smells really harmful? An empirical study
Kummer (2015)	Categorising Test Smells
Koskela (2015)	Developer test anti-patterns by lasse koskela
SCott (2015)	Five automated acceptance test anti-patterns
Hauptmann et al. (2015)	Generating refactoring proposals to remove clones from automated system tests
Hedayati, Ebrahimzadeh and Sori (2015)	Investigating into Automated Test Patterns in Erratic Tests by Considering Complex Objects
Buwalda (2015)	Test design for automation: Anti-patterns
Mathew and Foegen (2016)	An analysis of information needs to detect test smells
Tufano et al. (2016)	An empirical investigation into the nature of test smells
Tahir, Counsell and MacDonell (2016)	An Empirical Study into the Relationship Between Class Features and Test Smells
Kempf (2016)	Anti-Patterns - Digital Tapestry
Palomba et al. (2016)	On the diffusion of test smells in automatically generated test code: an empirical study
Chamberlain (2017)	How to Compare Object Instances in your Unit Tests Quickly and Easily
Williams and Dietrich (2017)	Unit Testing Smells: What Are Your Tests Telling You?
Palomba, Zaidman and Lucia (2018)	Automatic Test Smell Detection Using Information Retrieval Techniques
Khorikov (2018)	Code pollution
Kundra (2018)	Enhancing developers' awareness on test suites' quality with test smell summaries
al2o3cr (2018)	Hacker News on: Software Testing Anti-patterns
Spadini et al. (2018)	On the Relation of Test Smells to Software Code Quality
Garousi and Küçük (2018)	Smells in software test code: A survey of knowledge in industry and academia
Kapelonis (2018)	Software Testing Anti-patterns
Rodrigues (2018)	TDD Anti-patterns: The Free Ride / Piggyback
Schmengler (2018)	Test Smell: Hard Coded Values
Swett (2018)	Test smell: Obscure Test
Frieze (2018)	Test Smells - The Coding Craftsman
King (2018)	Test::Class Hierarchy Is an Antipattern
Bugayenko (2018)	Unit Testing Anti-Patterns, Full List
Peruma (2018)	What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications
Garousi, Kucuk and Felderer (2018)	What We Know About Smells in Software Test Code
Qusef, Elish and Binkley (2019)	An exploratory study of the relationship between software test smells and fault-proneness
Gawinecki (2019)	Anti-patterns in test automation
Bleser, Nucci and Roover (2019a)	Assessing diffusion and perception of test smells in scala projects
Lewis (2019)	How to write good tests
Peruma et al. (2019)	On the distribution of test smells in open source Android applications: an exploratory study
Virgínio et al. (2019)	On the influence of Test Smells on Test Coverage
Grano et al. (2019)	Scented since the beginning: On the diffuseness of test smells in automatically generated test code
Bleser, Nucci and Roover (2019b)	SoCRATES: Scala radar for test smells
Peruma et al. (2019)	Software Unit Test Smells
Pecorelli (2019)	Test-related factors and post-release defects: an empirical study
Jones (2019)	Writing good gherkin
Junior et al. (2020)	A survey on test practitioners' awareness of test smells
Peruma et al. (2020)	An Exploratory Study on the Refactoring of Unit Test Files in Android Applications
Soares (2020)	Anti-patterns of automated testing
Gonzalez, Rath and Mirakhorli (2020)	Did You Remember To Test Your Tokens?
Spadini et al. (2020)	Investigating Severity Thresholds for Test Smells
Lambiase et al. (2020)	Just-In-Time Test Smell Detection and Refactoring: The DARTS Project
Silva and Vilain (2020)	LCCSS: A Similarity Metric for Identifying Similar Test Code
Grano et al. (2020)	Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality

Santana et al. (2020)	RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring
Soares et al. (2020)	Refactoring Test Smells: A Perspective from Open-Source Developers
Panichella et al. (2020)	Revisiting Test Smells in Automatically Generated Tests: Limitations, Pitfalls, and Opportunities
Martinez et al. (2020)	RTj: a Java framework for detecting and refactoring rotten green test cases
Nery and Lima (2020)	Test Artifacts — The Practical Testing Book
Peruma et al. (2020)	tsDetect: an open source test smells detection tool
Burns (2021)	A workbook repository of example test smells and what to do about them
Campos, Rocha and Machado (2021)	Anti-Patterns In Unit Testing
Fernandes, Machado and Maciel (2021)	Developers perception on the severity of test smells: an empirical study
Kim et al. (2021b)	Handling Test Smells in Python: Results from a Mixed-Method Study
Jorge, Machado and Andrade (2021)	How disabled tests manifest in test maintainability challenges?
Taniguchi, Matsumoto and Kusumoto (2021)	Investigating Test Smells in JavaScript Test Code
Peruma and Newman (2021)	JTDog: a Gradle Plugin for Dynamic Test Smell Detection
Rwemalika (2021)	On the Distribution of "Simple Stupid Bugs" in Unit Test Files: An Exploratory Study
Camara et al. (2021)	On the Maintenance of System User Interactive Tests
Wang et al. (2021)	On the use of test smells for prediction of flaky tests
Silva (2021)	PyNose: A Test Smell Detector For Python
Aranega et al. (2021)	Rails Testing Antipatterns: Fixtures and Factories
Martins et al. (2021)	Rotten green tests in Java, Pharo and Python
Rwemalika et al. (2021)	Smart prediction for refactorings in the software test code
Marabesi (2021b)	Smells in System User Interactive Tests
Brandes (2021)	TDD anti patterns - Chapter 1
Marabesi (2021a)	TDD anti-patterns - the liar, excessive setup, the giant, slow poke
Wigent (2021)	Test Naming Failures. An Exploratory Study of Bad Naming Practices in Test Code
Aljedaani et al. (2021)	Test Smell Detection Tools: A Systematic Mapping Study
Pecorelli, Palomba and Lucia (2021)	The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems
Kim, Chen and Yang (2021)	The secret life of test smells-an empirical study on test smell evolution and maintenance
Pontillo, Palomba and Ferrucci (2021)	Toward static test flakiness prediction: a feasibility study
Buffardi and Aguirre-Ayala (2021)	Unit Test Smells and Accuracy of Software Engineering Student Test Suites
Barrak et al. (2021)	Why do builds fail?—A conceptual replication study
Toomey and Ferris (2022)	Rails Testing Antipatterns
Tufano et al. (2019)	Towards Automated Tools for Detecting Test Smells: An Empirical Investigation into the Nature of Test Smells

Source: Research data

APPENDIX B – CATALOGED TEST SMELLS

Table 32 – Cataloged test smells

Name	Sources	AKA	Example	C/E	Frequency
7 Layer Testing	1		✓	✓	
Abnormal UTF-Use	2				✓
Absence of why	1				
Accidental test framework	1				
Activation Asymmetry	3		✓	✓	✓
Ambiguous Tests	1			✓	✓
Anal Probe	3				✓
Anonymous Test	4	Unclear Naming, Naming Convention Violation	✓	✓	✓
Anonymous test case	1				
Army of Clones	2			✓	✓
ASSERT 1 = 2	1			✓	
Assert the world	1				
Asserting Pre-condition and Invariants	1		✓	✓	
Assertion Chorus	1	Missing custom assertion method			
Assertion diversion	1		✓		
Assertion Roulette	48		✓	✓	✓
Assertion-free	1				
Assertionless Test	6	Lying Test, The Line Hitter, No Assertions	✓	✓	✓
Assertions should be Merciless	1		✓	✓	
Asynchronous Code	1			✓	
Asynchronous Test	1		✓	✓	
Autogeneration	1		✓	✓	
Bad Comment Rate	3		✓	✓	
Bad Documentation Comment	3		✓	✓	
Bad Naming	5		✓	✓	✓
Badly Structured Test Suite	1			✓	✓
Badly Used Fixture	1	General Fixture	✓	✓	
Behavior Sensitivity	2			✓	✓
Blethery prefixes	1		✓		
Blinkered assertions	1				
Boilerplate hell	1				
Branch To Assumption Anti-Pattern	1		✓	✓	
Brittle Assertion	1				
Brittle Test	2	Fragile Test	✓		
Brittle UI Tests	1		✓		
Broad Assertion	1		✓	✓	
Bumbling assertions	1		✓		
Bury The Lede	1		✓	✓	
Calculating expected results on the fly	1		✓	✓	
Catching Unexpected Exceptions	2		✓	✓	✓
Celery data	1				
Chafing	1		✓	✓	
Chain Gang	3	Order Dependent Tests			
Changing implementation to make tests possible	1			✓	
Chatty logging	1		✓		
Circumstantial evidence	1				
Code Pollution	1		✓	✓	

Code Run Only by Tests	1				
Commented Code in the Test	2	Under-the-carpet Failing Assertion			✓
Commented Test	1		✓	✓	
Comments	1				
Comments Only Test	2				✓
Complex Assertions	1		✓	✓	
Complex Conditional	3		✓	✓	
Complex Teardown	1		✓	✓	
Complicated set up scenarios within the tests themselves	1		✓		
Conditional assertions	4		✓	✓	✓
Conditional Logic	1				✓
Conditional Logic Test	1				
Conditional Test Logic	24	Indented Test Code, Guarded Test	✓	✓	✓
Conditional Tests	1			✓	✓
Conditional Verification Logic	1		✓	✓	
Conditionals in tests	1				
Conspiracy of silence	1		✓		
Constant Actual Parameter Value	3		✓	✓	✓
Constrained test order	1		✓	✓	
Constructor Initialization	14		✓	✓	✓
Contaminated Test Subject	1		✓	✓	
Context Logic in Production Code	1		✓	✓	
Context Sensitivity	2			✓	✓
Context-Dependent Rotten Green Assertion Test	2		✓	✓	✓
Contortionist testing	1				
Control Logic	2				✓
Counting on spies	1				
coupling between test methods	1		✓	✓	
Curdled Test Fixtures	1		✓		
Cut-and-Paste Code Reuse	1			✓	
Data Sensitivity	2			✓	✓
Data-Ja Vu	1				
Dead Field	3			✓	✓
Default Test	10		✓	✓	✓
Dependent test	3			✓	
Directly executing JavaScript	1		✓		
Disabled Test	1				
Disorder	3		✓	✓	
Doppelgänger	2				
Duplicate Alt Branches	3		✓	✓	✓
Duplicate Assert	14		✓	✓	✓
Duplicate code	1			✓	
Duplicate Component Definition	3		✓	✓	
Duplicate In-Line Templates	3		✓	✓	
Duplicate Local Variable/Constant/-Timer	3		✓	✓	
Duplicate Statements	3		✓	✓	
Duplicate Template Fields	3		✓	✓	
Duplicate test code	1				
Duplicated Actions	1				
Duplicated code	5		✓	✓	
Duplicated Code in Conditional	3		✓	✓	

Eager Test	49	The Test It All, Split Personality, Many Assertions, Multiple Assertions, The Free Ride, Silver Bullet, Piggyback	✓	✓	✓
Early Returning Test	2				✓
Easy Tests	1	The Dodger			✓
Embedding implementation detail in your features/scenarios	1		✓		
Empty Method Category	2				✓
Empty Shared-Fixture	2				✓
Empty Test	17		✓	✓	✓
Empty Test-Method Category	1				✓
Equality Pollution	2		✓	✓	
Equality Sledgehammer Assertion	1				
Erratic test	2			✓	✓
Erratic Tests	1		✓		
Everything is a property	1		✓		
Evolve or . . .	1		✓	✓	
Exception Catch/Throw	1				✓
Exception Catching Throwing	1				
Exception Handling	11		✓	✓	✓
Excessive Inline Setup	2				✓
Excessive Mocking	2			✓	
Excessive setup	7	Large Setup Methods, Inappropriately Shared Fixture, The Mother Hen, The Stranger, The Distant Relative, The Cuckoo	✓		✓
Expected Exceptions and Verification	1		✓	✓	
Expecting Exceptions Anywhere	1		✓	✓	
External Data	1	Mystery Guest			
External Dependencies	2			✓	✓
External shared-state corruption	1		✓	✓	
Factories depending on database records	1		✓	✓	
Factories pulling too many dependencies	1		✓	✓	
Factories that contain unnecessary data	1				
Factories with random data instead of sequences	1		✓	✓	
Fantasy Tests	2		✓	✓	
Feature Envy	1				
Fire And Forget	3	Plate-Spinning	✓	✓	✓
Flaky locator	1		✓	✓	
Flaky test	2			✓	
Flexible Test	1		✓	✓	
For Testers Only	13			✓	✓
Fragile Fixture	2			✓	✓
Fragile Test	7	Brittle Test	✓	✓	✓
Frequent Debugging	4	Manual Debugging		✓	✓
Fully Rotten Green Test	2		✓	✓	✓
Fully-Parameterized Template	3		✓	✓	✓
Fuzzy assertions	1		✓		
General Fixture	39	Badly Used Fixture	✓	✓	✓
Generative	1		✓	✓	
Generous Leftovers	4	Wet Floor, Sloppy worker			✓
Get really clever and use random numbers in your tests	1				
Goto Statement	3		✓	✓	
Ground zero	1		✓		

Guarded Test	3	Conditional Test Logic	✓	✓	✓
Half a helper method	1		✓		
Happenstance testing	1				
Happy Path	6		✓	✓	✓
Hard-Coded Test Data	2		✓	✓	
Hard-Coded Values	2		✓	✓	✓
Hard-to-Test Code	5			✓	
hard-to-write test	1			✓	
Hardcoded environment	2			✓	✓
Hardcoded environment configuration	1				
Hardcoded test data	1	Magic Strings			
Having flaky or slow tests	1			✓	
Herp Derp	1		✓		
Hidden Complexity	1			✓	
Hidden Dependency	6				✓
Hidden Integration Test	1		✓	✓	
Hidden Meaning	1		✓		
Hidden test call	1		✓	✓	
Hidden Test Data	2			✓	✓
Hooks everywhere	1	Testing Causes an Abstraction Virus			
I wrote it like this	1				
I'll believe it when I see some flashing GUIs	2				
Identity Dodgems	1				
Idle PTC	3		✓	✓	
Ignored Test	14		✓	✓	✓
Improper clean up after tests have been run	1				
Improper Test Method Location	1			✓	
Inadequate Assertion	1		✓	✓	✓
Inappropriate Assertions	1	Wrong Assert			✓
Inappropriately Shared Fixture	2	Excessive Setup			✓
Incidental coverage	1		✓	✓	
Incidental details	1				
Inconsistent Hierarchy	1				
Inconsistent Wording	1			✓	✓
Indecent exposure	1			✓	
Indecisive	1		✓	✓	
Indented Test	2				
Indirect Testing	19		✓	✓	✓
Inefficient waits	1			✓	
Insufficient Grouping	3		✓	✓	
Integration test, masquerading as unit test	1				
Interacting Test Suites	2		✓	✓	✓
Interacting Tests	3			✓	✓
Interactive Test	1	Manual Intervention		✓	
Interface Sensitivity	2			✓	✓
Intermittent Test Failures	1			✓	
Invalid test data	1				
Invasion Of Privacy	2		✓	✓	
Invisible Assertions	1		✓	✓	
Irrelevant Information	2		✓	✓	
Is Mockito Working Fine?	1		✓	✓	
Is There Anybody There?	1		✓		
Isolated PTC	2		✓	✓	
Issues in Exception Handling	1	The Secret Catcher		✓	

It looks right to me	1		✓		
It was like that when I got here	1				
Lack of Cohesion of Test Cases	1			✓	✓
Lack of Cohesion of Test Methods	5			✓	✓
Lack of Encapsulation	3			✓	✓
Lack of Macro Events	1				
Large Fixture	1				✓
Large Macro Component	1				
Large Module	1	Large Class			
Large Test File	1				
Lazy Test	22		✓	✓	✓
Likely ineffective Object-Comparison	2				✓
Line hitter	2	Assertionless Test			
Literal Pollution	1		✓	✓	
Litter Bugs	1	Test pollution	✓	✓	
Lonely Test	2			✓	✓
Long class	1			✓	
Long Function	1	Long Method			
Long Macro Event	1	Long Keyword			
Long method	1	Long Function		✓	
Long Parameter List	5		✓	✓	
Long Running Tests	1	Slow Test			
Long Statement Block	3		✓	✓	
Long Test	5	Obscure Test	✓	✓	✓
Long Test Steps	3			✓	✓
Long/complex/verbose/obscure test	1				
Lost Test	1			✓	
Magic Number	2		✓		
Magic Number Test	15		✓	✓	✓
Magic Values	3		✓	✓	✓
Making a mockery of design	1				
Manual Assertions	1				
Manual Event Injection	2			✓	✓
Manual Fixture Setup	2			✓	✓
Manual Intervention	3	Interactive Test, Manual Testing, Manual Test		✓	✓
Manual Result Verification	2			✓	✓
Many Assertions	1	Eager Test			
Martini Assertion	1				
Max Instance Variables	2				✓
Messy Test	1		✓	✓	
Middle Man	3			✓	✓
Missed fail rotten green test	2		✓	✓	✓
Missed skip rotten green test	1		✓	✓	
Missing Assertion Message	2		✓	✓	✓
Missing Assertions	4		✓	✓	✓
Missing Log	3		✓	✓	
Missing parameterised test	1		✓		
Missing test data factory	1		✓		
Missing Variable Definition	3	UR data flow anomaly	✓	✓	
Missing Verdict	3		✓	✓	
Mistaken identity	1				
Mixed Selectors	3			✓	✓
Mixing Production and Test Code	1				
Mock everything	1				
Mock Happy	3	The Mockery			✓
Mock madness	1				
Mock'em All!	1		✓	✓	

Mockers Without Borders	1		✓	✓	
Mocking a Mocking Framework	1				
Mocking what you don't own	2				
Mockito any() vs. isA()	1		✓	✓	
Multiple Assertions	1	Eager Test	✓	✓	
Multiple points of failure	1			✓	
Multiple Test Conditions	1		✓	✓	
Multiple tests testing the same or similar things	1				
Mystery Guest	40	External Data	✓	✓	✓
Name-clashing Import	1		✓	✓	
Narcissistic	2			✓	✓
Nested Conditional	3		✓	✓	
Neverfail Test	1			✓	
No Assertions	3	Assertionless Test	✓	✓	✓
No clear structure within the test	1			✓	
No structure when creating test cases	1				
No traces left	1				
Noisy Logging	2			✓	✓
Noisy setup	1		✓	✓	
Nondeterministic Test	2		✓	✓	✓
Not Idempotent	2	Interacting Test With High Dependency,			✓
Not using page-objects	1		✓		
Obscure In-Line Setup	4			✓	✓
Obscure Test	13	Long Test, Complex Test, Verbose Test	✓	✓	✓
On the Fly	2			✓	✓
Only Easy Tests	1			✓	
Only Happy Path Tests	1			✓	
Optimizing DRY	1			✓	
Order Dependent Tests	3	Chained Tests, Chain Gang	✓	✓	✓
Over exertion assertion	1		✓		
Over refactoring of tests	1		✓		
Over-Checking	3			✓	✓
Over-eager Helper	1				
Over-specific Runs On	3		✓	✓	
Overcommented Test	4		✓	✓	✓
Overly Complex Tests	1			✓	
Overly Dry Tests	2				✓
Overly elaborate test code	1				
Overmocking	1				
Overreferencing	3		✓	✓	✓
Oversharing on setup	1		✓		
Overspecification	1		✓	✓	
Overspecified Software	2	Overcoupled Test		✓	✓
Overspecified Tests	1		✓	✓	
Overuse of abstractions	1	it's too DRY			
Paranoid	1		✓	✓	
Parsed Data	1		✓	✓	
Piggybacking on existing tests	1				
Plate Spinning	1		✓		
Premature Assertions	1		✓	✓	
Primitive Assertion	1		✓	✓	
Print Statement	2				✓
Production Logic in Test	1		✓	✓	
Proper Organization	2				✓
Quixotic	1		✓	✓	

Redundant Assertion	16		✓	✓	✓
Redundant Print	12		✓	✓	✓
Refused Bequest	2				✓
Reinventing the Wheel	1			✓	
Remote Control Mocking	1		✓		
Require External Resources	1		✓	✓	
Requires Supervision	1				
Resource Leakage	2			✓	✓
Resource Optimism	29		✓	✓	✓
Returning Assertion	2				✓
Rewriting private methods as public	1				
Rotten Green Test	3		✓	✓	✓
Second Class Citizens	3	Test Code Duplication			
Second guess the calculation	1		✓		
Self Important Test Data	1		✓	✓	
Self-Test	1		✓	✓	
Sensitive Equality	33	The Butterfly	✓	✓	✓
Sensitive Locators	2			✓	✓
Share the world	1				
Shared-state corruption	1		✓	✓	
Short Template	3		✓	✓	✓
Shotgun Surgery	1				
Singular Component Variable/Constant/Timer Reference	3		✓	✓	✓
Singular Template Reference	3		✓	✓	✓
Skip Rotten Green Test	1		✓	✓	✓
Skip-Epidemic	1				✓
Sleeping for arbitrary amount of time	1		✓		
Sleepy Test	14		✓	✓	✓
Sloppy Worker	1	Generous Leftovers			✓
Slow Component Usage	1			✓	
Slow Running Tests	1			✓	
Slow Test	6	Long Running Test, The Slow Poke	✓	✓	✓
Sneaky Checking	3			✓	✓
So the Automation Tool Wrote This Crap	1				
Split Logic	1		✓	✓	
Split Personality	1	Eager Test	✓	✓	
Stinky Synchronization	2			✓	✓
Stinky synchronization syndrome	1				
Stop in Function	3		✓	✓	
Subclass To Test	1			✓	
Success Against All Odds	3	The Liar			✓
Superfluous Setup Data	1		✓		
Surreal	1		✓	✓	
Taking environment state for granted	1				
Tangential	1		✓	✓	
Teardown Only Test	2				✓
Test body is somewhere else	1		✓		
Test By Number	1		✓	✓	
Test cases are concerned with more than one unit of code	1				
Test Clones	2			✓	✓
Test Code Duplication	17	Second Class Citizen	✓	✓	✓
Test Dependency in Production	1			✓	
Test Envy	2				✓
Test Hook	1		✓	✓	

Test Hooks	1		✓		
Test Logic in Production Code	3			✓	
Test Maverick	4			✓	✓
Test Pollution	1	Litter Bugs			
Test Redundancy	1				
Test Run War	10		✓	✓	✓
Test setup is somewhere else	1		✓		
Test tautology	1		✓		
Test-Class Name	2				✓
Test-Method Category Name	2				✓
Test-per-Method	1				
Test::class Hierarchy	1		✓		
Testing business rules through UI	1				
Testing for a specific bug	1				
Testing internal implementation	1			✓	
Testing the Authentication Framework	1			✓	
Testing the Framework	1				
Testing The Internal Monologue	1		✓		
Tests Are Difficult to Write	1			✓	
Tests cluttered with business logic	1				
Tests depend on something outside of the test suite	1				
Tests require too much intimate knowledge of the code to run	1				
Tests that Can't Fail	1				
Testy testy test test	1				
The Bandwidth Demander	1				
The Butterfly	1	Sensitive Equality			
The Conjoined Twins	2				
The Cuckoo	4	Excessive Setup			✓
The Dead Tree	2	Process Compliance Backdoor	✓		✓
The Dodger	3	Easy Tests			
The Enumerator	4	Test With No Name			✓
The Environmental Vandal	1	The Local Hero			
The First and Last Rites	1	Oops I Forgot the Setup	✓		
The Flickering Test	2				
The Forty Foot Pole Test	3				
The Free Ride	7	Eager Test	✓	✓	
The giant	8		✓		✓
The Greedy Catcher	4				✓
The Inhuman Centipede	1				
The Inspector	5				✓
The Leaky Cauldron	1				
The liar	7	Evergreen Tests, Success Against All Odds	✓		✓
The Local Hero	7	Wait and See, The Environmental Vandal			✓
The Loudmouth	4	Transcripting Test			✓
The Mockery	4	Mock Happy, Mock-Overkill			
The Mother Hen	2	Excessive Setup			✓
The Nitpicker	3				
The One	3	Eager Test			✓
The Operating System Evangelist	3				✓
The painful clean-up	1				
The Parasite	1				
The Peeping Tom	3	The Uninvited Guests			✓
The Secret Catcher	5	The Silent Catcher, Issues in Exception Handling	✓		✓

The Sequencer	4				✓
The Silent Catcher	3	The Secret Catcher	✓		
The Sleeper	3	Mount Vesuvius			✓
The slow poke	6	Slow Test	✓		
The Soloist	1				
The Stepford Fields	1		✓		
The Stranger	3	The Cuckoo			✓
The telltale heart	1				
The Temporal Tip Toe	1				
The Test It All	1	Eager Test			
The Test With No Name	3	The Enumerator			✓
The Turing Test	1				
The ugly mirror	5	Tautological tests	✓	✓	
There is too much setup to run the test cases	1				
Time Bomb Data	1				
Time Bombs	2		✓	✓	
Time Sensitive Test	1		✓	✓	
Too Many Tests	1			✓	
Transcripting Test	3	The Loudmouth	✓	✓	✓
Treating test code as a second class citizen	2			✓	
Two for the price of one	1		✓		
Unclassifi ed Method Category	2				✓
Under-the-carpet Assertion	1				✓
Under-the-carpet Failing Assertion	4	Commented Code in the Test	✓	✓	✓
Underspecification	1		✓	✓	
Unknown Test	14		✓	✓	✓
Unnecessary Navigation	1		✓	✓	
Unreachable Default	3		✓	✓	
Unrepeatable Test	2		✓	✓	✓
Unrestricted Imports	3		✓	✓	✓
Unsuitable Naming	1				
Untestable Test Code	1			✓	
Unused Definition	3		✓	✓	✓
Unused Imports	3		✓	✓	✓
Unused Inputs	1				
Unused Parameter	3		✓	✓	✓
Unused Shared-Fixture Variables	2				✓
Unused Variable Definition	3	DU data flow anomaly	✓	✓	✓
Unusual Test Order	2				✓
Unworldly test data	1				
Use Smart Values	1		✓	✓	
Using Assertions as a Substitute for all Class-Based Exceptions	1				
Using Assertions as a Substitute for all Defensive Programming Techniques	1				
Using complicated x-path or CSS selectors	1		✓		
Using fixtures	1		✓	✓	
Using the word "and" when describing a test	1				
Using the Wrong Assert	1		✓	✓	
Vague Header Setup	3			✓	✓
Verbless and Noun-full	1				
Verbose Test	7	Obscure Test		✓	✓
Very similar test cases	1			✓	

Wait and See	2	The Local Hero			
Wasted Variable Definition	3	DD data flow anomaly	✓	✓	
Web-Browsing Test	1				✓
Well, My Setup Works	1				
Wet Floor	3	Generous leftovers			
What are we Testing?	1				
Wheel of fortune	1				
Where Does This One Go?	1				
X-Ray Specs	2		✓	✓	
You Do Weird Things to Get at the Code Under Test	1			✓	

Source: Research data

APPENDIX C – TEST SMELL CLASSIFICATION

Table 33 – Test smell classification

Category	Sub-Category	Name
Test execution/ behavior	Performance	Sleepy Test, Slow Test, The Slow Poke, Asynchronous Test, Factories pulling too many dependencies, Long Running Tests, Slow Component Usage, Slow Running Tests, Too Many Tests, Inefficient waits, Sleeping for arbitrary amount of time, The Bandwidth Demander, The Temporal Tip Toe, Wait and See
	Other test execution/behavior	Redundant Print, Frequent Debugging, The Loudmouth, Manual Event Injection, Manual Fixture Setup, Manual Intervention, Manual Result Verification, Print Statement, Transcripting Test, Abnormal UTF-Use, Interactive Test, Manual Assertions, Requires Supervision, Unnecessary Navigation, Chatty Logging
Test semantic/logic	Testing many things	Eager Test, Assertion Roulette, The Free Ride, Many Assertions, Multiple Assertions, The One, The Giant, Missing Assertion Message, Assert the world, Piggybacking on existing tests, Split Logic, Split Personality, Test cases are concerned with more than one unit of code, The Test It All
	Testing many units	Indirect Testing, Test Envy, The Stranger, Feature Envy
	Other test logic related	Conditional Test Logic, Lazy Test, The Liar, Happy Path, The Inspector, The Sequencer, Anal Probe, Complex Conditional, Guarded Test, Indented Test, Insufficient Grouping, Invasion Of Privacy, Nested Conditional, Rotten Green Test, Success Against All Odds, The Dodger, X-Ray Specs, ASSERT 1 = 2, Asynchronous Code, Branch To Assumption Anti-Pattern, Chafing, Conditional Tests, Conditional Verification Logic, Conditionals in tests, Contaminated Test Subject, Context-Dependent Rotten Green Assertion Test, Control Logic, Easy Tests, Embedding implementation detail in your features/scenarios, Factories with random data instead of sequences, Flexible Test, Generative, Get really clever and use random numbers in your tests, Inconsistent Wording, Indecisive, Multiple Test Conditions, Only Easy Tests, Only Happy Path Tests, Paranoid, Parsed Data, Quixotic, Skip Rotten Green Test, Skip-Epidemic, Tangential, Test-per-Method, Testing internal implementation, Testing the Authentication Framework, Untestable Test Code, Using the word "and" when describing a test, You Do Weird Things to Get at the Code Under Test, Test By Number, Overprotective Tests, Testing private methods, Fully Rotten Green Test, Neverfail Test, Incidental Details, Underspecification, Conditional Assertions, Conditional Logic, Conditional Logic Test, Evolve or . . . , Ground zero, I wrote it like this, Invalid test data, Rewriting private methods as public, Sneaky Checking, Tests that Can't Fail, Use Smart Values, Wheel of fortune
Design related	Not using test patterns	Constructor Initialization, Unknown Test, Disorder, Missing Log, No clear structure within the test, Not using page-objects, Test Hierarchy, Testing business rules though UI, Autogeneration, Contortionist Testing, I'll believe it when I see some flashing GUIs, Narcissistic, No structure when creating test cases, So the Automation Tool Wrote This Crap, The Turing Test
Issues in test steps	Issues in setup	General Fixture, Excessive Setup, Obscure In-line Setup, Test Maverick, Vague Header Setup, Excessive Inline setup, Fragile Fixture, Idle PTC, Irrelevant Information, Isolated PTC, Refused Bequest, The Cuckoo, Unused Definition, Badly Used Fixture, Bury The Lede, Complicated set up scenarios within the tests themselves, Empty Shared-Fixture, Factories that contain unnecessary data, Inappropriately Shared Fixture, Max Instance Variables, Mother Hen, Noisy setup, Using fixtures, Curdled Test Fixtures, Hidden Test Data, Large Fixture, Noisy Logging, Oversharing on setup, Share the world, Superfluous Setup Data, Taking environment state for granted, Test setup is somewhere else, There is too much setup to run the test cases, Where Does This One Go?

Issues in test steps	Issues in assertions	Sensitive Equality, Redundant Assertion, Assertionless Test, Under-the-carpet Failing Assertion, Commented Code in the Test, Fantasy Tests, Inappropriate assertions, Missing Assertions, No Assertions, The Nitpicker, Assertion-free, Brittle Assertion, Calculating expected results on the fly, Complex Assertions, Early Returning Test, Inadequate Assertion, Incidental coverage, Invisible Assertions, Likely Ineffective Object-Comparison, Line hitter, Missed Fail Rotten Green Test, Premature Assertions, Returning Assertion, Using Assertions as a Substitute for all Class-Based Exceptions, Using Assertions as a Substitute for all Defensive Programming Techniques, Using the Wrong Assert, Fragile Test, Brittle Tests, Primitive Assertion, Bitwise Assertions, Assertions should be Merciless, 7 Layer Testing, Asserting Pre-condition and Invariants, Shotgun Surgery, Assertion Diversion, Blinkered Assertions, Broad Assertion, Bumbling Assertions, Celery Data, Circumstantial Evidence, Commented Test, Conspiracy of silence, Equality Sledgehammer Assertion, Fuzzy assertions, Happenstance testing, Martini Assertion, No traces left, On the Fly, Over exertion assertion, Over-Checking, Second guess the calculation, Self-Test, Testing The Internal Monologue, The Butterfly, Under-the-carpet Assertion
	Issues in teardown	Not Idempotent, Complex Teardown, Sloppy Worker, Teardown Only Test, Wet Floor, Generous Leftovers, Unrepeatable Test, Activation Asymmetry, External shared-state corruption, Shared-state corruption, Improper clean up after tests have been run, It was like that when I got here, The painful clean-up, The Soloist
	Issues in exception handling	Exception Handling, The Secret Catcher, Catching Unexpected Exceptions, The Greedy Catcher, Exception Catch/Throw, Exception Catching Throwing, Issues in Exception Handling, The Silent Catcher, Expecting Exceptions Anywhere, Expected Exceptions and Verification
Code related	Mock and stub related	Mock Happy, Excessive Mocking, The Mockery, Mockers Without Borders, Mocking a Mocking Framework, Mocking what you don't own, Subclass To Test, Testing the Framework, Mocking everything, Is Mockito Working Fine?, Mockito any() vs. isA(), The Dead Tree, Surreal, Making a mockery of design, Mock everything, Mock Madness, Mock'em All!, Overmocking, Remote Control Mocking
	In association with production code	For Testers Only, Behavior Sensitivity, Fire And Forget, Interface Sensitivity, Overspecified Software, Test Logic in Production Code, The ugly mirror, Changing implementation to make tests possible, code pollution, Code Run Only by Tests, Context Logic in Production Code, Mixing Production and Test Code, overly elaborate test code, Production Logic in Test, Test Dependency in Production, Plate Spinning, Overspecification, Equality Pollution, Test Hook, Hooks Everywhere, Overly elaborate test code, Overspecified Tests, Test tautology, Tests cluttered with business logic, Tests require too much intimate knowledge of the code to run, The telltale heart, Well, My Setup Works, Indecent exposure, Multiple points of failure
	Code duplication	Test Code Duplication, Duplicate Assert, Duplicated code, Duplicate Alt Branches, Duplicate Component Definition, Duplicate In-Line Templates, Duplicate Local Variable/Constant/Timer, Duplicate Statements, Duplicate Template Fields, Duplicated Code in Conditional, Code Duplication, Cut-and-Paste Code Reuse, Duplicate test code, Duplicated Actions, Reinventing the Wheel, Test Clones, Test Redundancy, Very similar test cases, Second Class Citizens, Army of Clones, Assertion Chorus, Data-Ja Vu, Duplicate Code, Half a helper method, Missing Parameterised Test, Missing Test Data Factory, Multiple tests testing the same or similar things, The First and Last Rites, Two for the price of one
	Complex/ Hard to understand	Magic Number Test, Obscure Test, Hard-to-Test Code, Long Test, Verbose Test, Bad Comment Rate, Long Statement Block, Magic Values, Overly Dry Tests, Hard-to-Write Test, Hidden Complexity, Large Macro Component, Large Module, Large Test File, Long Function, Long Macro Event, Long Test Steps, Long/complex/verbose/obscure test, Magic Number, Optimizing DRY, Overly Complex Tests, Self Important Test Data, Tests Are Difficult to Write, Using complicated x-path or CSS selectors, Overcommented Test, Hard-Coded Test Data, Hard-Coded Values, Comments, Absence of why, Boilerplate Hell, Hard-coded environment, Hardcoded environment configuration, Hardcoded test data, Herp Derp, It looks right to me, Over refactoring of tests, Overuse of abstractions, What are we Testing?, Long Method, Long Class

Code related	Violating coding best practices	Empty Test, Ignored Test, Default Test, Long Parameter List, Anonymous Test, Bad Naming, Dead Field, Bad Documentation Comment, Constant Actual Parameter Value, Fully-Parameterized Template, Goto Statement, Missing Variable Definition, Missing Verdict, Mixed Selectors, Nondeterministic Test, Over-specific Runs On, Overreferencing, Short Template, Singular Component Variable/Constant/Timer Reference, Singular Template Reference, Stop in Function, Time Bombs, Unreachable Default, Unrestricted Imports, Unused Imports, Unused Parameter, Unused Variable Definition, Wasted Variable Definition, Ambiguous Tests, Badly Structured Test Suite, Comments Only Test, The Conjoined Twins, Directly executing JavaScript, Empty Method Category, Empty Test-Method Category, Erratic Test, Flaky test, The Forty Foot Pole Test, Having flaky or slow tests, Hidden Test Call, Improper Test Method Location, Inconsistent Hierarchy, Intermittent Test Failures, Lack of Encapsulation, Lack of Macro Events, Literal Pollution, Lost Test, Name-clashing Import, Proper Organization, Test-Class Name, Test-Method Category Name, The Sleeper, The Test With No Name, Unclassified Method Category, Unsuitable Naming, Unused Inputs, Unused Shared-Fixture Variables, Treating test code as a second class citizen, The Enumerator, Accidental test framework, Blethery Prefixes, Disabled Test, Doppelgänger, Everything is a property, Flaky locator, Hidden Meaning, Integration test, masquerading as unit test, Is There Anybody There?, Missed Skip Rotten Green Test, Mistaken identity, Over-eager Helper, Sensitive Locators, Test body is somewhere else, Testing for a specific bug, Testy testy test test, The Flickering Test, The Stepford Fields, Time Bomb Data, Time Sensitive Test, Unworldly test data, Verbless and Noun-full
Dependencies	Dependencies among tests	Lack of Cohesion of Test Methods, Dependent Test, Interacting Test Suites, Interacting Tests, Order Dependent Tests, The Peeping Tom, Constrained test order, coupling between test methods, Litter Bugs, Test Pollution, Chain Gang, Lonely Test, Unusual Test Order, Constrained Test Order, Identity Dodgems, Test Run War, Lack of Cohesion of Test Cases, The Environmental Vandal, The Inhuman Centipede, The Leaky Cauldron, The Parasite
	External dependencies	Mystery Guest, Resource Optimism, The Local Hero, Context Sensitivity, Data Sensitivity, External Dependencies, Resource Leakage, The Operating System Evangelist, Counting on spies, External Data, Factories depending on database records, Middle Man, Stinky synchronization syndrome, Web-Browsing Test, Hidden Dependency, Hidden Integration Test, Require External Resources, Stinky Synchronization, Tests depend on something outside of the test suite

Source: Research data