Noname manuscript No.
(will be inserted by the editor)

The Smell of Fear: On the Relation between Test Smells and Flaky Tests

Fabio Palomba · Andy Zaidman

Received: date / Accepted: date

Abstract Regression testing is the activity performed by developers to check whether new modifications have not introduced bugs. A crucial requirement to make regression testing effective is that test cases are deterministic. Unfortunately, this is not always the case as some tests might suffer from so-called *flakiness*, i.e., tests that exhibit both a passing and a failing outcome with the same code. Flaky tests are widely recognized as a serious issue, since they hide real bugs and increase software inspection costs. While previous research has focused on understanding the root causes of test flakiness and devising techniques that automatically fix them, in this paper we explore an orthogonal perspective: the relation between flaky tests and test smells, i.e., suboptimal development choices applied when developing tests. Relying on (1) an analysis of the state-of-the-art and (2) interviews with industrial developers, we first identify five flakiness-inducing test smell types, namely *Resource Optimism*, *Indirect Testing*, *Test Run War*, *Fire and Forget*, and *Conditional Test Logic*, and automate their detection. Then, we perform a large-scale empirical study on 19,532 JUnit test methods of 18 software systems, discovering that the five considered test smells causally co-occur with flaky tests in 75% of the cases. Furthermore, we evaluate the effect of refactoring, showing that it is not only able to remove design flaws, but also fixes all 75% flaky tests causally co-occurring with test smells.

Keywords Test Smells · Flaky Tests · Refactoring

Fabio Palomba
University of Zurich, Switzerland
E-mail: palomba@ifi.uzh.ch

Andy Zaidman
Delft University of Technology, The Netherlands
E-mail: a.e.zaidman@tudelft.nl

1 Introduction

Test cases form the first line of defense against the introduction of software faults, especially when testing for regression faults [1, 2, 3]. As such, with the help of testing frameworks like, for example, JUnit developers create test methods and run these periodically on their code [1, 4, 5, 6]. The entire team relies on the results from these tests to decide on whether to merge a pull request [7] or to deploy the system [8, 9, 10, 11]. When it comes to testing, developer productivity is partly dependent on both (i) the ability of the tests to find real problems with the code being changed or developed [8, 12] and (ii) the cost of diagnosing the underlying cause in a timely and reliable fashion [13].

Unfortunately, test suites are not immune to bugs: indeed, they often suffer from issues that can preclude the effective testing of software systems [14, 15]. Besides functional bugs, *test code flakiness* represents a typical bug affecting test suites [8]. Flaky tests are tests that exhibit both a passing and a failing result with the same code [8], being therefore unreliable test cases whose outcome is not deterministic. The harmfulness of flaky tests is widely known and recognized by both researchers and practitioners: as a matter of fact, dozens of daily discussions are opened on the topic on social networks and blogs [8, 16, 17, 18, 19]. More specifically, there are three key issues associated with the presence of flaky tests: (i) they may hide real bugs and are hard to reproduce due to their non-determinism [20]; (ii) they increase maintenance costs because developers may have to spend substantial time debugging failures that are not really failures, but just flaky [21], and (iii) from a psychological point of view flaky tests can reduce a developer's confidence in the tests, possibly leading to ignoring actual test failures [19].

All the aforementioned problems pushed the research community to study test code flakiness deeper. Most of the effort has been devoted to understanding the causes behind the presence of flaky tests [20, 22, 23, 24] and in the creation of automated techniques able to fix them [25, 26, 27]. Despite the notable advances produced so far, most of the previous studies in the field focused the attention on some specific causes possibly leading to the introduction of flaky tests, such as concurrency [27, 28, 29] or test order dependency [24] issues. As a consequence, they proposed *ad-hoc* solutions that cannot be used to fix flaky tests characterized by other root causes. This observation was also reflected in the findings of an empirical study on the motivations behind test code flakiness conducted by Luo et al. [20]: as highlighted by the authors, the problems faced by previous research only represent *part of whole story* and a deeper analysis of possible fixing strategies of other root causes (e.g., flakiness due to wrong usage of external resources) is still missing.

In our previous work [30], we started investigating the role of *test smells* [31, 32, 33], i.e., poor design or implementation choices applied by programmers during the development of test cases, as indicators of the presence of flaky tests. Specifically, from the catalog proposed by Van Deursen et al. [31] we identified three test smell types, i.e., *Resource Optimism*, *Indirect Testing* and *Test Run War*, whose definitions were closely connected to the concept of

test code flakiness. From our empirical investigation—conducted on 19,532 JUnit test methods of 18 large software systems—we discovered that 61% of flaky tests were affected by one of the test smells that we considered. A further investigation revealed that 54% of flaky tests were caused by problems attributable to the characteristics of the smells, and the removal of such flaws eliminated both the design problem and test code flakiness.

In this paper, we extend our previous work [30] by enlarging the empirical knowledge on the relation between test smells and flaky tests. We first systematically elicit a catalog of test smells whose characteristics recall the concept of test flakiness—we call them *flakiness-inducing test smells*—by (1) analyzing the state-of-the-art and (2) performing semi-structured interviews with 10 software testers. Secondly, we analyze the causal co-occurrence between these smells and flaky tests. Finally, we analyze whether the removal of such new test smells can be considered a valid flaky tests fixing strategy.

More specifically, we:

1. Define a catalog of flakiness-inducing test smells by means of a mixed-method approach [34] that takes into account both the research conducted in the field and the opinions of 10 software testers having more that 20 years of experience in testing of software systems. From this study, we define five flakiness-related test smells, i.e., *Resource Optimism*, *Indirect Testing*, *Test Run War*, *Fire and Forget*, and *Conditional Test Logic*, along with the corresponding removal operations;
2. Devise and/or evaluate detection strategies for the test smells in the catalog, coming up with a detection tool that practitioners can use to identify test smells;
3. Analyze to what extent the defined flakiness-inducing test smells causally co-occur with flaky tests in our dataset and evaluate how much test smell removal can be applied to fix those instances;

The results of our study confirm what we previously observed: *test smells represent a precious source of information to locate flaky tests*. Indeed, 75% of the tests in our dataset causally co-occur with one of the five test smells considered. Moreover, test smell removal represents a lightweight and powerful technique to remove both test smells and test code flakiness.

To sum up, this paper provides the following contributions:

- A catalog of flakiness-inducing test smells, that (i) practitioners can use to learn about the bad practices to avoid when implementing test cases with the aim of reducing the risk of test flakiness and (ii) researchers and tool vendors can exploit to prioritize the definition of new techniques for the automated removal of these smells;
- The definition of accurate detection strategies for two of the smells included in the catalog, i.e., *Fire and Forget* and *Conditional Test Logic*, as well as a replication of the performance evaluation for the detector of *Resource Optimism*, *Indirect Testing*, and *Test Run War*;

– An empirical study that demonstrates the relation between test smells and flaky tests, and that shows the extent to which the two phenomena are related to each other and how removal of test smells can be exploited to fix test code flakiness.
– Two new large datasets reporting (1) the presence of flaky tests in the considered systems and (2) the list of the smelliness-inducing test smells. They can be exploited by other researchers to further study the problem of test flakiness as well as how test code quality relates to flaky tests.

Structure of the paper. The remainder of this paper is organized as follows. Section 2 presents the elicitation strategies used to define a catalog of flakiness-inducing test smells as well as the new mechanisms devised for detecting them. Section 3 reports the empirical study planning and design, while Section 4 discusses the results of the study. In Section 5 we debate about the possible threats that could have influenced our results. Subsequently, in Section 6 we report the literature related to flaky tests and test smells before concluding in Section 7.

2 Toward a Catalog of Flakiness-inducing Test Smells

This section reports the methodology used to elicit a catalog of flakiness-inducing test smells and describes the high-level detection mechanisms for automating their identification.

2.1 Defining the Catalog

In the following subsections, we describe the methodology followed to define the catalog as well as the resulting set of flakiness-inducing test smells.

2.1.1 Test Smell Discovery Procedure

In a recent study, Garousi and Küçük [35] proposed a multivocal literature mapping [36] aimed at reporting all the test smells defined so far by both scientific research papers and grey literature (e.g., practitioners' blog posts, white papers, and presentation videos). As a result, they found 126 test smell types belonging to 8 macro-categories (e.g., dependency- or design-related).

Among all the test smells defined so far, our aim was to identify the subset of them whose definition might indicate some sort of test flakiness: for this reason, we started the definition of our catalog by selecting, from the smells reported in the paper by Garousi and Küçük [35], those having the desired characteristic. At the same time, since there might exist further bad design practices inducing test flakiness that are not included in the multivocal literature mapping of Garousi and Küçük [35], we complemented the state-of-the-art analysis with semi-structured interviews involving 10 software testers.

In the following, we expose the methodology adopted in each of the two steps.

State-of-the-art analysis. To identify which of the smells reported in the reference multivocal literature review [35] may have an impact on test flakiness, we systematically inspected all the smell definitions. In particular, one of the authors of this paper played the role of *inspector*; given the entire list of test smells proposed by Garousi and Küçük [35], his task was to mark each definition as *flakiness-inducing* or *non-flakiness-inducing*. Whenever needed, the inspector opened a discussion with the other author of this paper on the actual relationship between a certain test smell and the possibility that it may produce a flakiness behavior in the affected test case. Following this process, we identified five flakiness-inducing test smell types.

While the constant discussion between the inspectors mitigated some threats to the validity of the selected flakiness-inducing test smells, we also performed a sanity-check by involving two external professional software testers having 7 and 10 years of testing experience, respectively. They were contacted via e-mail by the first author of this paper, who selected them from his personal contacts. We provided the *external inspectors* with a spreadsheet containing the list of test smells coming from the work by Garousi and Küçük [35] and asked them to categorize the test smells as *flakiness-inducing* or *non-flakiness-inducing*, i.e., using the same procedure as the original classification. Once the task was completed, the professional testers sent back the spreadsheet file annotated with their categorization. The classification as done by both professional testers individually was identical to the one done in the first phase. As a consequence, we are confident of the selection procedure performed.

Semi-structured interviews. Despite the large number of test smells (126) defined so far and reported in the reference multivocal literature review [35], we could not exclude that some sub-optimal practices impacting test flakiness might not have been reported yet. For this reason, to be as cautious as possible with respect to the observations of this study, we complemented the state-of-the-art analysis with semi-structured interviews involving 10 professional software testers having more than 20 years of experience in testing of modern software systems and coming from one of our industry partners. More in detail, they work for different companies, ranging from financial activities to public administration services. Most of them (7 out of the total 10) usually develop in Java, while the remaining ones work with Python. As part of their job, they have to design test cases, so they are very experienced in how to create tests and how to automate them. At the same time, their companies have up to 50 new flaky tests per week, and they are sometimes responsible to fix that flakiness. This makes them particularly suitable for the kind of study we performed.

The interviews were conducted by both authors of this paper, took 2.30 hours in total, and were semi-structured, a form of interview often used in exploratory investigations to understand phenomena and seek new insights [37]. The focus of the interviews was to make participants discuss about the bad

practices related to the emergence of a flaky test and, thus, all the questions were centered on whether there exist peculiar sub-optimal implementation choices that may induce a flakiness behavior of a test case. For this reason, all the questions targeted the analysis of their experience as testers that deal with flaky tests, with the aim of (i) eliciting bad practices possibly resulting in the definition of new test smell types or (ii) confirming existing ones.

The answers of the interviewees were transcribed to ease the data analysis. Then, both the authors of this paper performed an open coding process on the reported bad practices and jointly discuss them to identify possible test smells. This process did not lead to the emergence of additional test smell types, however it further confirmed those emerged in the state-of-the-art analysis. Thus, the semi-structured interviews can be seen as an additional validation of the activities performed to elicit flakiness-inducing test smells.

2.1.2 Resulting Catalog

In the following paragraphs, we describe the five test smell types that came out of the test smell discovery phase.

Resource Optimism. The first test smell comes from the catalog by Van Deursen et al. [31], and was explicitly defined by the authors as a smell that may cause flakiness.

Definition. Test code that makes optimistic assumptions about the state or the existence of external resources.

Motivation. The outcome of the test method may depend on the state/existence of the external resource, meaning that it can exhibit a passing/failing behavior based on the ability to properly acquire the external resource.

Detection Mechanism. JUnit test methods using an external resource without allocating it explicitly in the `setUp` method [31].

Removal. To remove the smell, Van Deursen et al. recommended the use of *Setup External Resource* strategy, i.e., explicitly allocating the external resources before testing (in the `setUp` method), being sure to release them when the test method ends (in the `tearDown` method) [31].

Indirect Testing. The second test smell comes from the catalog by Van Deursen et al. [31], and is related to the way a test method interacts with the production code.

Definition. Test methods affected by this smell test different classes with respect to the production class corresponding to the test class [31].

Motivation. Indirect Testing methods might not properly set the environment needed to test production methods belonging to different classes:

as a consequence, their outcome may depend on the order of execution of test methods, i.e., the outcome may change if the environment is or is not set before calling the smelly test.

Detection Mechanism. JUnit test methods invoking, besides methods of the corresponding production class, methods of other classes in the production code [31].

Removal. To remove this smell, Van Deursen et al. firstly suggest the application of an *Extract Method* refactoring [38] able to isolate the part of the method that actually tests different objects.

Test Run War. This smell belongs to the catalog by Van Deursen et al. [31] and revolves around the allocation of test resources.

Definition. This smell arises when a test method allocates resources that (i) are also used by other test methods or (ii) interact between them in an unstable manner (within the same test method) [31].

Motivation. The allocation of resources to multiple tests might cause possible resource interferences making the outcome of a test non-deterministic. Similarly, a unstable interaction between objects in a test might lead to intermittent failures.

Detection Mechanism. JUnit test methods that allocate resources that are also used by other test methods (e.g., temporary files) or that contains objects interacting with each other in a suspicious manner in the same thread [31].

Removal. The original operation associated with the smell is the *Make Resource Unique*, which consists of creating unique identifiers for all resources that are allocated by a test case [31]. At the same time, the *Decouple Objects* [35] can be adopted to make the objects used by the test independent.

Fire and Forget. This smell was identified by Garousi and Küçük [35] analyzing the grey literature on test smells.

Definition. A test that is at risk of exiting prematurely because it does not properly wait for the results of external calls.

Motivation. The missing control of the runtime environment might naturally lead to intermittent failures due to the wrong management of the resources or to unexpected events.

Detection Mechanism. JUnit test methods that do not properly wait for the ending of the operations performed in a thread.

Removal. The operation associated with this smell is called *Add Await Condition* [35], and concerns the addition of an await condition that allows the execution of the test to be suspended until the return of the called method.

Conditional Test Logic. The final test smell, also known as *Indented Test Code*, belongs to the catalog by Meszaros [32].

Definition. Conditional Test Logic occurs when a test does more than the required, by using control structures (`if-else` statements) that make its outcome dependent on the execution path [32].

Motivation. Depending on the actual path executed, the test environment may or may not be properly set, possibly causing issues due to test order dependency [32].

Detection Mechanism. JUnit test methods using `if-else` constructs.

Removal. The removal of this smell consists of the application of an *Extract Method* refactoring [38] that isolates the different portions of code that can be executed by the test.

The detection mechanisms have been automated in order to perform a large-scale identification of the flakiness-inducing test smells. The specific technical rules implemented and their evaluation are reported in the following section.

3 Empirical Study Definition and Design

The *goal* of the study is to understand whether specific test smell types can represent the underlying cause of test flakiness, with the *purpose* of evaluating to what extent test smell removal operations can be successfully applied to fix flaky tests by removing the test smells affecting the test code. The *perspective* is that of both researchers and practitioners, who are interested in understanding the benefits of test smell removal for improving the effectiveness of test suites.

The specific research questions investigated in this study are listed in the following:

- **RQ$_1$:** *What are the causes of test flakiness?*
- **RQ$_2$:** *To what extent can flaky tests be explained by the presence of test smells?*
- **RQ$_3$:** *To what extent does removal of test smells help in removing test flakiness?*

The first research question was intended to be a preliminary analysis aimed at understanding what are the causes leading tests to be flaky. This investigation can be considered as a large-scale replication of the study proposed by Luo et al. [20], who performed a similar analysis inspecting 201 commits that likely fix flaky tests. The rationale behind this research question is twofold: on the one hand, as our dataset is much larger than the one used by Luo et al. [20], it is worth understanding whether their findings hold in our case; at the same time, it allows us to identify the roots of the flaky tests considered in the study, so that we can relate them to the symptoms behind test smells. With $\mathbf{RQ}_2$ our goal was to perform a fine-grained investigation into the relationship between test smells and flaky tests. In this way, we could understand whether test smells can be actually induce test flakiness. Finally, $\mathbf{RQ}_3$ investigated the ability of test smell removal strategies in fixing test code flakiness by removing test smells. In this case, we aimed at assessing the extent to which such strategies can be applied as flakiness fixing mechanisms. Our replication package is available at [39].

Table 1: Characteristics of the Systems involved in the Study.

System	Description	Classes	Methods	KLOCs	Test Methods
Apache Ant 1.8.3	Command-line tool to build systems	813	8,540	204	3,097
Apache Cassandra 1.1	Scalable DB Management System	586	5,730	111	586
Apache Derby 10.9	Relational DB Management System	1,929	28,119	734	426
Apache Hive 0.9	Data Warehouse Software Facilities Provider	1,115	9,572	204	58
Apache Ivy 2.1.0x	Flexible Dependency Manager	349	3,775	58	793
Apache Hbase 0.94	Distributed DB System	699	8,148	271	604
Apache Karaf 2.3	Standalone Software Container	470	2,678	56	199
Apache Lucene 3.6	Search Engine	2,246	17,021	466	3,895
Apache Nutch 1.4	Web-search Software built on Lucene	259	1,937	51	389
Apache Pig 0.8	Large Dataset Query Maker	922	7,619	184	449
Apache Qpid 0.18	AMQP-based Messaging Tool	922	9,777	193	786
Apache Struts 3.0	MVC Framework	1,002	7,506	152	1,751
Apache Wicket 1.4.20	Java Serverside Web Framework	825	6,900	179	1,553
Elastic Search 0.19	RESTful Search Engine	2,265	17,095	316	397
Hibernate 4	Java Persistence Manager	154	2,387	47	132
JHotDraw 7.6	Java GUI Framework for Technical Graphics	679	6,687	135	516
JFreeChart 1.0.14	Java Chart Library	775	8,746	231	3,842
HSQLDB 2.2.8	HyperSQL Database Engine	444	8,808	260	59
Overall		**16,454**	**161,045**	**3,852**	**19,532**

3.1 Context Selection

The *context* of the study was composed of (i) subject systems, and (ii) test smells. As for the former, Table 1 reports the characteristics of the software projects involved in the study. Their selection was driven by two main factors: firstly, since we had to run test smell and flaky test detection tools, we limited the analysis to open-source projects; secondly, we analyzed software systems actually having test classes and having different size and scope. Thus, we randomly selected 13 projects belonging to the Apache Software Foundation from the list available on GitHub[1], as well as other 5 projects belonging to

[1] Available here: `https://github.com/apache`

different communities. As for the test smells, we focused on the five flakiness-inducing design issues in the catalog proposed in Section 2, i.e., *Resource Optimism*, *Indirect Testing*, *Test Run War*, *Fire and Forget*, and *Conditional Test Logic*.

Table 2: Rules Used for the Detection of Test Smells.

Test Smell	Rule
Resource Optimism	JUnit methods using an external resource (i.e., they use a variable of type `File`) without checking its status (e.g., the methods `exists` or `isReadable` are not called).
Indirect Testing	JUnit methods invoking, besides methods of the corresponding production class, methods of other classes in the production code.
Test Run War	JUnit methods that allocate resources that are also used by other test methods (e.g., temporary files)
Fire and Forget	JUnit tests that do not properly wait for the return of a call.
Conditional Test Logic	JUnit test methods using `if-else` constructs.

3.2 Detecting Test Smells

Once we had cloned the source code of the subject systems from the corresponding GitHub repositories (using the `git clone` command), we first performed the automatic detection of the flakiness-inducing test smells and evaluated the accuracy of such tools.

Test Smell Detection Procedure. Among all the detection tools proposed in literature [40, 41, 42], we relied on the one devised by Bavota et al. [40] for the detection of *Resource Optimism*, *Indirect Testing*, and *Test Run War* instances. This tool implements a heuristic-based approach that analyzes code metrics. It has been previously employed in several works in the area showing good performance [40, 43, 44]. More specifically:

- The identification of *Resource Optimism* follows the guidelines defined by Van Deursen et al. [31], checking whether a test method contained in a JUnit class uses an external resource and does not check its status before using it, respectively. From a technical point of view, the detector firstly checks if a test instantiates a variable of type `File`: if so, it marks the test as smelly in case (i) the existence of the variable is not preventively checked (i.e., the method `exists` is not called); or (ii) the status is not controlled (i.e., any of methods `isReadable`, `isWritable`, and `isExecutable` is executed).
- In the case of *Indirect Testing*, the tool takes into account the method calls performed by a certain test method, in order to understand whether it exercises classes different from the production class it is related to. To identify the class corresponding to a test, the detector employs a traceability approach based on naming convention, i.e., it identifies the class under

test by removing the string *'Test'* from the name of the JUnit test class, similarly to Zaidman et al. in [45].

- For *Test Run War* the tool evaluates whether (i) a test method allocates resources that are referenced by other test methods in the same JUnit class or (ii) a test method itself contains objects that interact with each other in the same thread in a non-desirable manner. This means that if a test instantiates a Java `File`, the detector extracts its path and checks whether other test methods in the same class use it. If so, the test method is marked as smelly. At the same time, the tool relies on the Infer toolkit[2] to perform thread-safety analysis [46] and identify possible violations leading to concurrency issues. If the tool detects anomalous behaviors, the method is marked as smelly.

As for the remaining two test smells in the catalog analyzed, i.e., *Fire and Forget* and *Conditional Test Logic*, we noticed a lack of automated solutions enabling their detection and, thus, we built our own identification mechanisms:

- In the case of *Fire and Forget*, the detector checks if a test performs an external call followed by a `Thread.sleep` instruction. In this case, the detector marks the test as smelly, since its waiting mechanism tightly depends on the parameter passed to the function (i.e., the specified number of milliseconds that a test should wait).
- Finally, the identification of *Conditional Test Logic* instances is based on the definition of the smell: if a test uses an `if-else` statement, then it is considered smelly.

For the sake of comprehensibility, a summary of the detection rules adopted is reported in Table 2.

Test Smell Detection Performance. The proposed empirical study builds upon the accuracy of the detection mechanisms we adopted: for instance, if the identification approach used for *Fire and Forget* has a very low F-Measure, then the overall results of the study might be seriously threatened. At the same time, even though the rules adopted for some smells have already been evaluated in previous work showing high performance [40], it is important to understand whether such performance holds in our own context: indeed, as recently pointed out [47], software engineering tools can sometimes be sensitive to the dataset, meaning that it is not ensured that the performance of a tool is the same when running it in a different context. Thus, having well-performing tools is a key aspect to consider. To account for it, we performed a preliminary analysis of the performance of the test smell detection rules.

To achieve this goal, we needed to build an oracle reporting the actual test smell instances in the considered dataset. However, as a manual detection of all the test smells affecting the 18 software systems in our dataset would have been prohibitively expensive, we created a smaller sample, that we call *evaluation dataset*. We randomly picked 3 projects from our initial dataset, i.e.,

[2] https://fbinfer.com

Apache Ant, Apache Lucene, and JHotDraw, thus taking into account a total of 7,508 test methods. Then, we started the manual detection of the flakiness-inducing test smells such tests contained. However, we evaluated the detection rules *after* having defined them, and thus we might have been biased when constructing the manually-validated test smell set, leading to mark as smelly those instances presenting the characteristics used by the detectors (i.e., we might have been affected by the so-called *Observer-expectancy effect* [48]). To avoid any form of bias and perform a fair evaluation, we then recruited two external professional developers having more than 8 years of experience in the development of software systems, actively working on the definition of test cases in their own companies, and with a high experience in software design and bad practices.

Specifically, the external inspectors were provided with the (i) original definition of the test smells, (ii) the test code of the three considered projects, and (iii) five spreadsheets, one for each test smell to analyze, each of them containing the list of all the test methods belonging to the systems under analysis, and that were used to classify instances of the five flakiness-inducing test smells; they performed the task independently and were allowed to inspect the production code too, since it might be useful to properly assess the existence of a test smell (e.g., in the case of *Indirect Testing*, they needed the production code to understand whether a test case actually calls methods in different methods than the production one). The task was to assign a truth value in the set $\{true, false\}$ to each of the 7,508 test test methods present in the spreadsheets: the inspector assigned the value `true` when a code component was affected by a certain test smell, `false` otherwise. Once the inspectors had completed this task, the produced oracles were compared, and the inspectors discussed the differences, i.e., test smell instances present in the oracle produced by one inspector, but not in the oracle produced by the other. All the test methods positively classified by both the inspectors were considered as actual smells. As for the other instances, the inspectors opened a discussion in order to resolve the disagreement and jointly took a decision. At the end of this process, the oracle comprised 569 *Resource Optimism*, 472 *Fire and Forget*, 389 *Indirect Testing*, 319 *Test Run War*, and 142 *Condition Test Logic* instances.

To measure the level of agreement between the two inspectors, we computed the Jaccard similarity coefficient [49], i.e., number of test smell instances identified by both the inspectors over the union of all the instances identified by them. The overall agreement between the two inspectors before the discussion was 87%. Of the remaining 13% of the cases, they reached an agreement during the discussion. Once we had built the manually-validated set of flakiness-inducing test smells, we ran the test smell detection rules over the same set of test methods and compared their output with the oracle. To measure the performance of the detection mechanisms we computed three well-known metrics: precision, recall, and F-Measure [50].

The results of the empirical assessment are reported in Table 3. As it is possible to see, the detection mechanisms adopted have high accuracy for all

Table 3: Test Smell Detection Performance - P=Precision; R=Recall; F-M=F-Measure.

Project	Resource Optimism			Indirect Testing			Test Run War			Fire and Forget			Conditional Test Logic		
	P	R	F-M	P	R	F-M	P	R	F-M	P	R	F-M	P	R	F-M
Ant	100%	100%	100%	93%	90%	91%	84%	91%	87%	100%	100%	100%	100%	100%	100%
Lucene	100%	100%	100%	86%	82%	84%	88%	88%	88%	100%	100%	100%	100%	100%	100%
JHotDraw	100%	100%	100%	87%	92%	89%	83%	84%	83%	100%	100%	100%	100%	100%	100%
Overall	**100%**	**100%**	**100%**	**89%**	**88%**	**89%**	**86%**	**88%**	**87%**	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**

the flakiness-inducing test smells (the F-Measure is never lower than 87%). On the one hand, this confirms previous findings on the reliability of the detection approach proposed by Bavota et al. [40] when employed for the identification of *Resource Optimism*, *Indirect Testing*, and *Test Run War* instances; on the other hand, this result highlights that the devised rules for *Fire and Forget* and *Obscure Test* detection are highly effective and allow to perform an empirical study that can properly draw conclusions on the relation between test smells and flaky tests.

While the detection mechanisms are generally effective, it is worth noting the presence of some false positives. In the case of *Indirect Testing*, these are generally due to errors in the identification of the class under test: we identified the exercised class by relying on a textual-based heuristic that simply removes the string "Test" from the name of the JUnit class. However, sometimes it happens that developers do not use such convention, invalidating our retrieval mechanism and, therefore, not allowing the detection rule to work properly. This indicates that more research on traceability might be beneficial also in the context of test smell detection. On the other hand, false positive *Test Run War* instances can be found in cases where test methods properly use shared resources. Indeed, the fact that more methods use the same resource does not directly imply concurrency problems: if the methods are always executed independently, then they would not lead to the emergence of any problem. A detection rule solely based on static analysis cannot properly consider the context where the test is executed, meaning that more sophisticated dynamic techniques could represent a valid alternative.

To generalize the achieved detection results over the entire dataset used in the empirical study, we asked the two external inspectors to analyze a further sample of 365 test smell instances identified by the detectors in the 15 subject software projects *excluded* from the initial evaluation. Such a set represents a 95% statistically significant stratified sample with a 5% confidence interval of the 9,185 total smell instances detected by the tools over the systems that were not considered in the initial evaluation. In this case, the inspectors could only produce data useful to compute the precision of the detection strategies (the recall could not be evaluated because of the lack of a comprehensive oracle of test smells for the projects considered). As a result of this step, the overall precision was 94%: thus, the performance was in line with that achieved in the initial evaluation. All in all, we can claim that the accuracy of the information provided by the detection mechanisms were sufficiently high to properly perform our study.

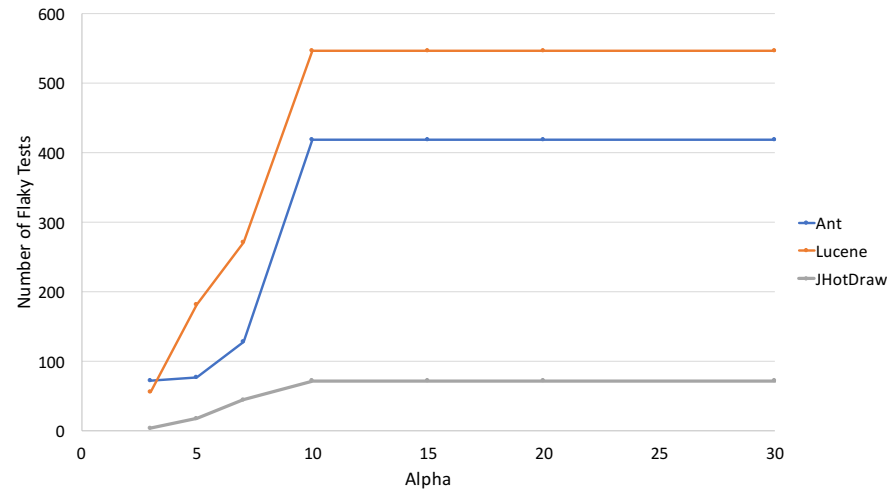

Fig. 1: Calibration of the parameter α used to identify flaky tests.

3.3 Detecting Flaky Tests

Once we had concluded the smell detection phase, we identified flaky tests by running the JUnit classes present in each subject system multiple times, checking the outcome of the tests over the different executions. Specifically, each test class was run α times: if the output of a test method was different in at least one of the α runs, then a flaky test was identified. Since the related literature did not provide clues on the suitable number of runs needed to accurately identify a flaky test, we empirically calibrated such a parameter on the evaluation dataset previously adopted for the assessment of the test smell detection rules, i.e., Apache Ant, Apache Lucene, and JHotDraw. In particular, we experimented with different values for the parameter, i.e., $\alpha = 3, 5, 7, 10, 15, 20, 30$, evaluating how the number of flaky tests found changed based on the number of runs. The results of the calibration are reported in Figure 1. As it is possible to observe, a number of runs lower than 10 does not allow the identification of many flaky tests, while $\alpha = 10$ represents the minimum number of runs ensuring the identification of the maximum number of flaky tests contained in the system: indeed, setting α to higher values does not increase the number of non-deterministic tests, suggesting that ten is the right number of runs to discover flaky tests. This statement is also confirmed by the fact that $\alpha = 10$ was the best solution for all three systems that we experimented with. For this reason, in the context of the study we ran each test case 10 times in order to identify flaky tests. From a technical perspective, to run the tests we used the developer-provided build scripts, so that we could exercise the source code with the exact environment set up by the developers of the considered projects. To do so, we repeatedly run the `mvn verify` command. Using this strategy, we identified 8,829 flaky tests (i.e., 45%

of the test methods analyzed are flaky). It is worth noting that we cannot ensure that this identification strategy covers all the possible flaky tests, however the results of the calibration allow us to be confident about its accuracy.

Table 4: Taxonomy of the root causes of flaky tests [20].

Category	Description
Async Wait	A test method making an asynchronous call and that does not wait for the result of the call.
Concurrency	Different threads interact in a non-desirable manner.
Test Order Dependency	The test outcome depends on the order of execution of the tests.
Resource Leak	The test method does not properly acquire or release one or more of its resources.
Network	Test execution depends on the network performance.
Time	The test method relies on the system time.
IO	The test method does not properly manage external resources.
Randomness	The test method uses random number.
Floating Point Operation	The test method performs floating-point operations.
Unordered Collections	Test outcome depends on the order of collections.

3.4 Data Analysis

Once we had collected data about the presence of test smells and flaky tests over the entire dataset considered, we answered $\mathbf{RQ}_1$ by manually investigating each flaky test identified in order to understand the root cause behind its flakiness. These causes have been classified by relying on the taxonomy proposed by Luo et al. [20], who identified ten common causes of test flakiness. Table 4 reports, for each common cause, a brief description. Specifically, the manual classification has been performed analyzing the (i) source code of the flaky tests, and (ii) the JUnit log reporting the exceptions thrown when running the tests. The task consisted of mapping each flaky test onto a common cause, and required approximately 200 man/hours. In Section 4 we report the distribution of the flaky tests across the various categories belonging to the taxonomy.

As for $\mathbf{RQ}_2$, we firstly determined which of the previously categorized flaky tests were also affected by one of the test smells considered: this allowed us to measure to what extent the two phenomena occur together. However, to deeper understand the relationship between flaky tests and test smells a simple analysis of the co-occurrences is not enough (e.g., a test affected by a *Resource Optimism* may be flaky because it performs a floating point operation). For this reason, we set up a new manual analysis process with the aim of measuring in how many cases the presence of a test smell is actually related to the test flakiness. In particular, the task consisted of the identification of the test smell instances related to the root causes of test code flakiness previously classified: this means that if the cause of test flakiness could be directly mapped on the characteristics of the smell, then the co-occurrence was considered as *causal*, i.e., we started from flaky tests and evaluated if the reason of flakiness was

due to the presence of a test smell. For example, we marked a co-occurrence between a flaky test and a *Resource Optimism* instance causal if the flakiness of the test case was due to issues involving the management of external resources (e.g., missing check of the status of the resource). In Section 4 we report descriptive statistics of the number of times a given test smell is related to each category of the flakiness motivations.

Finally, to answer $\mathbf{RQ}_3$ we manually analyzed the source code of the test methods involved in a design problem and performed removal operations according to the guidelines provided by Van Deursen et al. [31], Meszaros [32], and Garousi and Küçük [35], depending on the type of smell considered. It is worth highlighting that the authors of this paper have a programming and testing experience of 10 and 20 years, respectively; moreover, they use to teach testing practices in their universities. This task was performed by relying on (i) the definitions of the removal operations and (ii) the examples provided in the original catalogs. More specifically:

- In the case of *Resource Optimism*, we applied a *Setup External Resource* by creating the content of a file directly in the `setUp` method. As suggested by Van Deursen et al. [31], the ideal usage of external resources would be that of building the content of a file directly when setting up the environment. Thus, we simply pull the contents of the external resource into the `setUp`. This was done by creating a `String` object containing the text of the external file, and using the method `println` of the `PrintWriter` API to actually create the file in the test directory. It is worth noting that the application of this removal operation may produce less readable source code, as the fixture may contain a large number of statements. We are aware of this, however the definition of strategies that minimize the impact of the modifications on other non-functional attributes of test code is not part of this work; in other words, we simply followed the recommendations of Van Deursen et al. [31] on how to remove *Resource Optimism* instances.

- As for the *Indirect Testing*, we applied the *Extract Method* refactoring [38], thus creating new test methods and moving the parts of the smelly method that actually tests different objects in such new tests.
- When removing *Test Run War* instance, we applied *Make Resource Unique* and *Decouple Objects* operations. In cases where the former action was needed, we duplicated the common resources (i.e., the files used by different tests) assigning to them unique identifiers: in this way, each test worked on independent resources. In cases where the latter refactoring was needed, we modified the source code according to the recommendations made by Infer, i.e., by synchronizing resources or adding missing checks on the execution of the threads.
- In the case of *Fire and Forget*, the *Add Await Condition* consisted in adding an explicit call to the `await` method of the `Condition` class[3], in order to

[3] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html

allow the test to demand to the Java Virtual Machine the management of the time required before executing other instructions.

- The removal operation suggested for *Conditional Test Logic* is the *Extract Method* [38]. As explained above, in this case we created a new test for each source code portion belonging to different conditions in the smelly test. Thus, as a result each test method contained the code to test a specific path.

It is important to remark that all the operations performed as well as the way we executed them exactly follow the guidelines provided in previous work [31, 32, 35]. Such operations only involve the test code, not producing side effects on the production code. With respect to the removal of other forms of design problems (e.g., code smells), fixing test smells is somehow easier because most of the operations to be performed concern basic program transformations. For example, the *Setup External Resource* operation is something that can be easily automated, as it requires the moving of the file creation to the `setUp` method; similarly, the *Make Resource Unique* is an action that requires the duplication of the resource over the different test methods and thus it is automated pretty easily. The discussion is similar also for the *Add Await Condition*: in this case, a technique should only add an `await` statement in the corpus of the test. On the other hand, the most complex removal action is the *Extract Method* refactoring, because it first consists of understanding what is the part of the test that should be moved in another method: as shown by previous research in the field (e.g., [51]), there are many different ways in which this refactoring can be performed and automatically detecting them can be very challenging. Thus, we can generally say that the research on automated solutions for the removal of test smells should focus on those operations that require the understanding of the actual portions of source code to be moved or modified.

Overall, the manual removal required approximately 350 man/hours (counting the effort spent by both the inspectors): we believe that most of this effort might be semi-automated or completely automated. Indeed, differently from refactoring of production code [52], tests are generally much simpler (e.g., limited size or number of external dependencies) and, therefore, their reorganization might limit the typical issues of automated smell removal, e.g., the automatic update of the references of methods after an *Extract Method* operation. Our future research agenda is focused on the definition of an automated solution for removing test smells.

The output of this phase consisted of the source code where the test smells have been removed. Once we have re-organized the source code, we have repeated the flaky test identification, in order to evaluate whether the removal operations had an effect on the test flakiness. In this stage, we followed the same procedure applied to identify flaky tests in $\mathbf{RQ}_1$, i.e., we repeatedly run the `mvn verify` command. It is worth noting that we re-run all the tests (not only the one identified as flaky): this was done to ensure the same environment and context before and after the test smell removal. In Section 4 we report

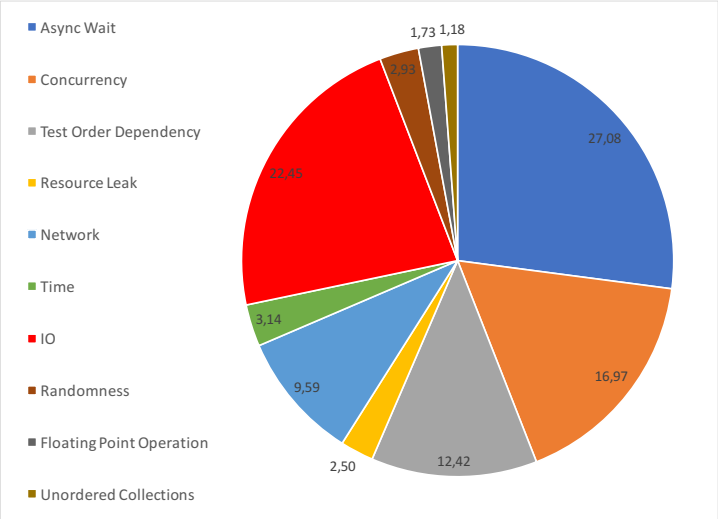

Fig. 2: Distribution of flaky tests across different categories.

descriptive statistics of the number of times removal operations successfully fix a flaky test instance by removing a test smell. In addition, we also provide qualitative examples that show why test smell removal may be a powerful method to remove test code flakiness.

4 Analysis of the Results

In this section we provide answers to the research questions previously formulated, discussing each of them independently.

4.1 $\mathbf{RQ}_1$: Causes behind test code flakiness

Overall, we found 8,829 flaky tests over the total of 19,532 JUnit test methods analyzed: thus, 45% of the tests suffer of flakiness and, therefore, in the first place we can confirm previous findings on the relevance of the phenomenon [14, 20]. Figure 2 depicts a pie chart reporting the distribution of the flaky tests belonging to each of the categories defined by Luo et al. [20]. Looking at the figure, we can observe that the most prominent causes of test code flakiness are represented by the Async Wait, IO, and Concurrency.

In the case of Async Wait, we can confirm the findings by Luo et al. [20]: this is actually the most frequent reason leading to flakiness. At the same time, it is worth noting how close the definition of this category is with that of the *Fire and Forget* test smell. As such, when a smell is concerned with missing controls over the runtime environment and, in particular, asynchronous wait issues, there seems to exist a relationship between its presence and the

appearance of flakiness. This aspect further motivates the analysis done in the context of the following research questions.

On the other hand, with respect to the findings by Luo et al. [20], we can observe a notable difference in the number of flaky tests due to IO reasons: in our study this number is much higher than in the work by Luo et al. [20] ($\approx 22\%$ vs $\approx 3\%$). This mismatch might be due to the larger dataset employed in this study.

Listing 1: IO issue found in the Apache Pig project

```
1  @Test
2  public void testPigServerStore() throws Exception {
3   String input = "input.txt";
4   String output= "output.txt";
5   String data[] = new String[] {"hello\tworld"};
6   ExecType[] modes = new ExecType[] {ExecType.MAPREDUCE, ExecType.LOCAL};
7   PigServer pig = null;
8   for (ExecType execType : modes) {
9    try {
10    if(execType == ExecType.MAPREDUCE) {
11      pig = new PigServer(ExecType.MAPREDUCE, cluster.getProperties());
12    } else {
13      Properties props = new Properties();
14      props.put(MapRedUtil.FILE_SYSTEM_NAME, "file:///");
15      pig = new PigServer(ExecType.LOCAL, props);
16    }
17
18    Util.createInputFile(pig.getPigContext(), input, data);
19    pig.registerQuery("a = load '" + input + "';");
20    pig.store("a", output);
21    pig.registerQuery("b = load '" + output + "';");
22    Iterator<Tuple> it = pig.openIterator("b");
23    Tuple t = it.next();
24    Assert.assertEquals("hello", t.get(0).toString());
25    Assert.assertEquals("world", t.get(1).toString());
26    Assert.assertEquals(false, it.hasNext());
27   }  finally {
28    Util.deleteFile(pig.getPigContext(), input);
29    Util.deleteFile(pig.getPigContext(), output);
30   }
31  }
32 }
```

An interesting example is reported in Listing 1, where the test method `testPigServerStore` of the JUnit class `TestInputOutputFileValidator` is shown. The method refers to the sample files `input.txt` and `output.txt` to test the storage capabilities of the Apache Pig server (see lines 2 and 3 in Listing 1). While the `createInputFile` function creates a temporary input file (line 18) that is then deleted after the test ends (line 28), the `output.txt` is not subject to the creation procedure. As a consequence, the test appears to be flaky in case this file is not present, i.e., if the `output.txt` is not available the test raises a `FileNotFoundException` exception. Overall, we discovered the presence of a consistent number of similar cases (1,982).

At the same time, we found that the distribution of flaky test motivations is much more scattered across the different categories than was reported in previous work [20]. Indeed, while Luo et al. found that almost 77% of flaky tests were due to only three root causes, we instead observe that on a larger dataset other categories are quite popular as well. For instance, Network

is the cause of almost 10% of the total flaky tests. Finally, categories such as Randomness, Resource Leak, Floating Point Operator, and Unordered Collections represent a small portion of the root causes for test code flakiness (8,34% in total).

Implications. In general, this preliminary analysis provides *two key implications* for both our study and the research community:

1. Most of the categories collecting the highest number of flaky tests are potentially related to the presence of test smells. For instance, the IO issue shown in Listing 1 clearly relates to the presence of a *Resource Optimism* smell; issues due to asynchronous wait might relate to a *Fire and Forget* instance, and Concurrency problems may be associated with the presence of *Test Run War* smells that create resource interferences [20], and may be avoided by adopting test smell removal operations. While a fine-grained analysis of the relationship between flaky tests and test smells is presented in the subsequent section, our results encourage the research community in providing a deeper understanding on how *source code quality attributes of test code relate to test code effectiveness*. At the same time, further studies investigating the presence of additional (anti-)patterns behind flaky tests might represent the next breakthrough step toward the comprehension and resolution of the flakiness problem.
2. According to our results, some test flakiness motivations seem much more important than others, leading to a natural prioritization of the activities to be done by the research community in order to properly face the test code flakiness problem. On the one hand, a higher attention to how developers introduce such issues might lead to the definition of methods that preventively alert developers of the possible introduction of source code potentially causing flakiness; on the other hand, techniques able to deal with the major causes of flaky tests should be devised. In the context of this paper, we try to assess the role of test smell detection and removal to locate and/or remove test case flakiness.

Summary of RQ$_1$. Almost 45% of the test methods analyzed have a non-deterministic outcome. Unlike previous work, we found that test flakiness is due to a different variety of reasons: the most prominent ones are Async Wait ($\approx$ 27%), IO ($\approx$ 22%), and Concurrency ($\approx$ 17%), however reasons such as Test Order Dependency ($\approx$ 11%), and Network ($\approx$ 10%) also explain a consistent part of flaky tests. We also noticed that some flakiness motivations might closely relate to test smells.

4.2 RQ$_2$: The Relationship between Flaky Tests and Test Smells

Over all the 18 systems analyzed in the study, 11,120 test methods were discovered to be affected by one of the smells considered, i.e., almost 56% of

Table 5: Co-occurrences between flaky tests and test smells.

Category	All		Resource Optimism		Indirect Testing	
	Total	Causal	Total	Causal	Total	Causal
Async Wait	100%	100%	26%	0%	0%	0%
Concurrency	71%	68%	1%	0%	1%	0%
Test Order Dependency	98%	84%	13%	0%	80%	80%
Resource Leak	6%	0%	1%	0%	0%	0%
Network	92%	86%	81%	81%	0%	0%
Time	14%	0%	0%	0%	12%	0%
IO	82%	81%	81%	81%	0%	0%
Randomness	2%	0%	0%	0%	0%	0%
Floating Point Operation	1%	0%	1%	0%	0%	0%
Unordered Collections	32%	0%	1%	0%	1%	0%
Overall	**81%**	**75%**	**31%**	**26%**	**11%**	**10%**

Category	Test Run War		Fire and Forget		Conditional Test Logic	
	Total	Causal	Total	Causal	Total	Causal
Async Wait	1%	0%	100%	100%	3%	0%
Concurrency	68%	68%	1%	0%	0%	0%
Test Order Dependency	0%	0%	0%	0%	5%	4%
Resource Leak	0%	0%	0%	0%	5%	0%
Network	0%	0%	11%	5%	0%	0%
Time	0%	0%	0%	0%	2%	0%
IO	0%	0%	0%	0%	1%	0%
Randomness	1%	0%	0%	0%	1%	0%
Floating Point Operation	0%	0%	0%	0%	0%	0%
Unordered Collections	0%	0%	8%	0%	3%	0%
Overall	**19%**	**18%**	**18%**	**17%**	**1%**	**1%**

the test cases was smelly. The most common smell was the *Resource Optimism*, which affected 3,017 test methods; then, the *Indirect Testing* appeared in 2,522 tests, while the *Fire and Forget* in 2,435 cases. Finally, *Test Run War* and *Conditional Test Logic* had 2,273 and 873 instances, respectively.

Table 5 reports the co-occurrences between flaky tests and test smells: specifically, for each flaky test motivation previously described, the relationship between such motivation and (i) any one of the test smells considered, and (ii) each smell independently, is presented. The "Total" column represents the percentage of co-occurrences found globally, while column "Causal" highlights the percentage of co-occurrences for which the test smell was actually related to the flaky test. It is worth remarking that we considered as causal only the co-occurrences for which the cause of test flakiness could be directly mapped on the characteristics of the co-occurring smell (see Section 3.4). In addition, the row "Overall" reports the results achieved by considering all the flaky tests independently. Looking at the table, in the first place we can notice that flaky tests and test smells very frequently occur together (i.e., 81% of the test methods are both flaky and smelly). More importantly, we found that the test smells (i) are causally related to flaky tests in 75% of the cases, and (ii) have strong relationships with four top root causes of test flakiness, i.e., Async Wait, Concurrency, Test Order Dependency, and IO. Consequently, this means that *test smell removal may potentially remove 75% of flaky tests present in a software project.*

Before further discussing the results achieved, it is worth reporting the number of times in which the presence of a test smell did not cause any kind of flakiness, i.e., a test smell detector might output a "false positive" recommendation with respect to the co-presence of a flakiness issue. Specifically, we identified 11,120 test smells: 75% of the 8,829 flaky tests were caused by one of the considered smells, meaning that 6,622 tests were both smelly and flaky; as a consequence, 4,498 test smell instances (40% of the total smells identified) did not cause flakiness. While the presence of a considerable number of "false positive" flakiness-inducing test smells might result in a limited generalizability of our results, we can still claim that a large number of test smells cause flakiness and thus developers should carefully take into account the possibility to remove them to avoid flakiness problems. Moreover, it is important to remark that the goal of this paper is to empirically understand the extent of the relation between smelliness and flakiness and whether test smell removal represents a viable flakiness fixing strategy. The definition of detectors specifically designed to capture flakiness-inducing instances is out of the scope of our analyses. Finally, it is also worth highlighting that, even if a certain test smell instance does not induce flakiness, developers should still take care of test code as the presence of test smells is strongly associated with additional problems for software quality. In particular, as reported in recent work [53], test smells not only make tests more change- and defect-prone, but they are also detrimental to the tests' ability to find defects in production code.

Resource Optimism. According to our findings, this test smell closely relates to flaky tests caused by the IO and NETWORK factors. The relationship with IO is mainly due to the fact that often test cases rely on external resources during their execution. Besides the case reported in Listing 1, we found several other similar cases.

Listing 2: Resource Optimism instance detected in the Apache Nutch project

```
@Test
public void testPages() throws Exception {
  pageTest(new File(testDir, "anchor.html"), "http://foo.com/", "http://
      creativecommons.org/licenses/by-nc-sa/1.0", "a", null);

  // Tika returns <a> whereas parse-html returns <rel>
  // check later
   pageTest(new File(testDir, "rel.html"), "http://foo.com/", "http://
       creativecommons.org/licenses/by-nc/2.0", "rel", null);

  // Tika returns <a> whereas parse-html returns <rdf>
  // check later
   pageTest(new File(testDir, "rdf.html"), "http://foo.com/", "http://
       creativecommons.org/licenses/by-nc/1.0", "rdf", "text");
}
```

For example, Listing 2 shows a *Resource Optimism* instance detected in the test method `testPages` of the JUNIT class `TestCCParseFilter` of the APACHE NUTCH system. In particular, the invoked method `pageTest` takes as input the directory where the `html` files are located, as well as the name of the files needed to exercise the production method. The test method is smelly because it does not check the existence of the files employed; at the same time,

this issue causes intermittent fails in the test outcome because the `testDir` folder is created only when it does not contain files having the same name. In our dataset, we found that all the *Resource Optimism* instances co-occurring with flaky tests caused by an IO issue are represented by a missing check for the existence of the file. This result is also statistically supported: indeed, we computed the Kendall's rank correlation [54] in order to measure the extent to which test cases affected by this smell are related to IO-flaky tests, finding the Kendall's $\tau = 0.69$ (note that $\tau > 0.60$ indicates a strong positive correlation between the phenomena). In the cases where the test flakiness is due to input/output issues, but these tests are not smelly, we found that the problem was due to errors in the usage of the `FileReader` class: similarly to what Luo et al. [20] have reported, often a test that would open a file and read from it, but not close it until the `FileReader` gets garbage collected.

As for the relationship between *Resource Optimism* and the Network motivation, we found quite commonly that test cases are flaky because they wait for receiving the content of a file from the network, but this reception depends on the (i) quantity of data being received, or (ii) network fails. With respect to these two phenomena, the Kendall's $\tau = 0.63$.

The *Resource Optimism* smell also sometimes co-occurs with other flaky tests characterized by different issues such as Async Wait or Test Order Dependency, however we did not find any causation behind such relationships. Thus, these co-occurrences are simply caused by the high diffuseness of the smell. Also in this case, Kendall's τ supports our conclusion, being equal to 0.13 and 0.11, respectively.

Indirect Testing. We observed some small non-significant co-occurrences with flaky tests related to Time and Concurrency issues. In these cases the test flakiness was not due to the fact that the test method covers production methods not belonging to the method under test, but to problems related to (i) asserts statements that compare the time with the `System.currentTimeMillis` Java method, thus being subject to imprecisions between the time measured in the test case and the exact milliseconds returned by the Java method, and (ii) threads that modify data structures concurrently, causing a `ConcurrentModificationException`.

On the other hand, we discovered a large and significant relationship between this smell and flaky tests related to the Test Order Dependency factor (Kendall's $\tau = 0.72$). The reason behind this strong link is that test methods exercising production methods not belonging to the method under test often do not properly set the environment needed to test these methods. As a consequence, tests run fine depending on the order of execution of the test cases that set the properties needed.

Listing 3: Indirect Testing instance detected in the Hibernate project

```
@Test
public void testJarVisitor() throws Exception{
   ...

  URL jarUrl  = new URL ("file:./target/packages/defaultpar.par");
```

```
6     JarVisitor.setupFilters();
7     JarVisitor jarVisitor =  JarVisitorFactory.getVisitor(jarUrl,
          JarVisitor.getFilters(), null);
8     assertEquals(FileZippedJarVisitor.class.getName(), jarVisitor.getClass
          ().getName());
9
10    jarUrl  = new URL ("file:./target/packages/explodedpar");
11    ExplodedJarVisitor jarVisitor2 =  JarVisitorFactory.getVisitor(jarUrl,
           ExplodedJarVisitor.getFilters(), null);
12    assertEquals(ExplodedJarVisitor.class.getName(), jarVisitor2.getClass
          ().getName());
13
14    jarUrl  = new URL ("vfszip:./target/packages/defaultpar.par");
15    FileZippedJarVisitor jarVisitor3 =  JarVisitorFactory.getVisitor(
          jarUrl, FileZippedJarVisitor.getFilters(), null);
16    assertEquals(FileZippedJarVisitor.class.getName(), jarVisitor3.
          getClass().getName());
17  }
```

For instance, Listing 3 shows a snippet of the test method `testJarVisitor` of the JUnit class `JarVisitorTest` belonging to the HIBERNATE project. The test has three assert statements to check the status of objects from either the corresponding production class `JarVisitor` or the external classes `ExplodedJarVisitor` and `FileZippedJarVisitor`. The flakiness manifests itself when the filters of the external classes (lines #10 and #14 in Listing 3) are not set by test cases executed before the `testJarVisitor` one. While the occurrence of *Indirect Testing* instances can be considered causal for 80% of the flaky tests having issues related to TEST ORDER DEPENDENCY (i.e., flakiness caused by test methods exercising objects of other classes whose correct setting depends on the execution order), in the remaining 20% of the cases the flakiness is due to explicit assumptions made by developers about the state of an object at a certain moment of the execution.

Test Run War. We discovered a strong relationship (Kendall's $\tau = 0.62$) between the *Test Run War* smell and flaky tests caused by CONCURRENCY issues. Given the definition of the smell, the result is somehow expected since test methods allocating resources used by other tests can naturally lead to concurrency problems.

Listing 4: Test Run War instance detected in the Elastic Search project

```
1   @Test
2   public void testSimple() throws Exception {
3     ExecutorService executorService = Executors.newCachedThreadPool();
4     List<Future> results = new ArrayList<Future>();
5     final CyclicBarrier barrier1 = new CyclicBarrier(cycles * 2 + 1);
6     final CyclicBarrier barrier2 = new CyclicBarrier(cycles * 2 + 1);
7
8     for (int i = 0; i < cycles; i++) {
9       results.add(executorService.submit(new Callable() {
10        @Override public Object call() throws Exception {
11          barrier1.await();
12          barrier2.await();
13          for (int j = 0; j < operationsWithinCycle; j++) {
14             if(barrier1.isReady()) {
15               barrier1.setReady(false);
16               assertThat(acquirableResource.acquire(), equalTo(true));
17             }
18          }
19          return null;
```

```
20        }
21      }));
22      results.add(executorService.submit(new Callable() {
23        @Override public Object call() throws Exception {
24           barrier1.await();
25           barrier2.await();
26           for (int j = 0; j < operationsWithinCycle; j++) {
27             acquirableResource.release();
28           }
29           return null;
30        }
31      }));
32  }
```

For example, consider the case of the test method `AbstractAcquirableResourceTests.testSimple` of the Elastic Search project shown in Listing 4. This method initializes several concurrent threads that perform simultaneous actions on the two objects called `barrier1` and `barrier2`; the problem in this case arises because the state of `barrier1` is changed in not ready (line 15 in Listing 4) and never turned into ready. Thus, other threads cannot continue their execution (see lines 11 and 24). From a statistical perspective, the strength of the relationship between this smell and flaky tests having concurrency issues obtained a Kendall's $\tau = 0.66$. As explained before, when the test has concurrency issues without being affected by smells, we found that the main reason is related to threads that simultaneously modify data structures.

Fire and Forget. The relation between this smell and the Async Wait motivation appeared to be causal in all the cases, indicating the strong relation between the presence of *Fire and Forget* instances and asynchronous wait problems. This result was somehow expected given the findings of $\mathbf{RQ}_1$, where we already observed that the smell intrinsically relates to this kind of flakiness. The Kendall's τ in this case was 0.89, thus supporting the conclusion that, based on our findings, the *Fire and Forget* smell represents an important source to locate tests making asynchronous calls and that might be flaky because they fail to wait for the result of such calls.

Listing 5: Fire and Forget instance detected in the Apache Cassandra project

```
1   @Test
2   public void testServiceTopPartitionsNoArg() throws Exception {
3     BlockingQueue<Map<String, Map<String, CompositeData>>> q = new
          ArrayBlockingQueue<>(1);
4     ColumnFamilyStore.all();
5     Executors.newCachedThreadPool().execute(() -> {
6       try {
7         q.put(StorageService.instance.samplePartitions(1000, 100, 10,
              Lists.newArrayList("READS", "WRITES")));
8         Thread.sleep(2000);
9       } catch (Exception e) {
10        e.printStackTrace();
11      }
12    });
13
14    SystemKeyspace.persistLocalMetadata();
15    Map<String, Map<String, CompositeData>> result = q.poll(11, TimeUnit.
          SECONDS);
16
```

```
17    List<CompositeData> cd = (List<CompositeData>) (Object) Lists.
          newArrayList(((TabularDataSupport) result.get("system.local").get
          ("WRITES").get("partitions")).values());
18
19    assertEquals(1, cd.size());
20  }
```

To provide an example of such relation, Listing 5 reports a *Fire and Forget* instance detected on the Apache Cassandra system, and particularly in the test method `testServiceTopPartitionsNoArg` of the `org.apache.cassandra.tools.TopPartitionsTest` class. The test aims at exercising the partitioning functions of the system. To this aim, the test starts a thread involving an asynchronous call to the `StorageService` class (line #7 of Listing 5); however, the `sleep` is set to 2 seconds, causing a non-deterministic behavior in case the `StorageService` does not return in a timely manner. We found similar cases in all the co-occurrences between such smell and the Async Wait motivation.

At the same time, a notably smaller percentage of co-occurrences (5%) between this smell and the Network root cause was discovered. In this case, the reason was due to the presence of `http` requests whose execution time depends on the performance of the network and that were not properly managed by tests.

Conditional Test Logic. The final smell of the catalog is concerned with the presence of conditional statement(s) in a test case that may influence the portion of the production code exercised, thus producing the risk to run methods whose environment is not properly set and that fail dependent on the order of execution of test cases. Looking at the results, we can partially confirm our conjecture: indeed, while 5% of flaky tests co-occur with a *Conditional Test Logic*, we observed that only 4% of them are actually due to the presence of the test smell. This means that the *Conditional Test Logic* appears to be less harmful of other test smells considered in the study even though it can sometimes explain test flakiness.

Listing 6: Conditional Test Logic instance detected in the Apache Pig project

```
1   @Test
2   public void testStopOnFailure() throws Throwable {
3
4   ...
5
6    if (Util.isMapredExecType(cluster.getExecType())) {
7      JobGraph jobGraph = PigStats.get().getJobGraph();
8      List<JobStats> failedJobs = jobGraph.getFailedJobs();
9
10     assertEquals(2, failedJobs.size());
11
12     for (JobStats stats : failedJobs) {
13       if (stats.getAlias().equals("A,B")) {
14         // Job with alias A,B should have failed because of streaming
               error
15         assertTrue(stats.getMessage().contains(
16                  "Received Error while processing the map plan: "
17         + "'false (stdin-org.apache.pig.builtin.PigStreaming/stdout-org.
               apache.pig.builtin.PigStreaming)'"
18         + " failed with exit status: 1"));
```

```
19      } else {
20        // Job with alias A1 with sleep should be killed as a result of
              stop on failure
21        assertTrue(stats.getErrorMessage().startsWith("Failing running
              job for -stop_on_failure"));
22      }
23  }
24
25 ...
26
27 }
```

To better explain the situations where the *Conditional Test Logic* smell can induce a Test Order Dependency flakiness, let consider the case of the `testStopOnFailure` method belonging to the `org.apache.pig.test.TestGrunt` class of the Apache Pig project, which is presented in Listing 6. As shown, the test exercises two different production methods depending on the result of the `if` statement at line #13. The flakiness manifests itself where the condition is false (i.e., the block starting at line #20 is executed), since the `getErrorMessage` method succeeds only in case it is properly set by the `testInvalidParam` method, i.e., only in case the order of execution of test methods is the proper one.

As a final note, it is worth remarking that we did not observe differences when analyzing the results per project. This means that the size of systems does not seem to impact our findings.

Implications. The overall output of this research question consists of *four main implications*, discussed in the following:

1. Test smells represent an important source of information to locate tests possibly suffering of flakiness. For this reason, the development of accurate test smell detectors as well as techniques able to promptly highlight to developers the emergence of a certain design issue (e.g., just-in-time approaches [55]) should be the top-priorities for researchers working in the field of software design and testing. While some steps toward this direction have been already done so far [41, 42, 56], we believe that more research in this field is needed, in order to better support developers when analyzing the quality of test code.
2. Our findings revealed that some test smells might be considered more important than others, at least considering their harmfulness with respect to test code flakiness. This paves the way for a new generation of test smell prioritization approaches able to exploit flakiness-related information to recommend which are the most risky test smells and, thus, suggesting to developers a possible test smell removal prioritization.
3. Given the impact of test smells on test code flakiness, we believe that one major implication of our study is related to the way software testing is taught to software engineering students. Indeed, we argue and warmly invite software engineering educators to give more focus and consideration to test code quality and, more in particular, test smells through specialized courses, seminars, and/or practical exercises. We consider the educational

aspect fundamental to pose the basis to form the next generations of software engineers.

4. As the relation between test smells and flaky tests is often causally tied, this might imply that operations performed to remove test smells might have an impact on test flakiness, besides removing the smells themselves: the assessment of this aspect is the goal of the next research question.

Summary of RQ$_2$. 81% of the flaky tests are also affected by a test smell. Interestingly, the cause of flakiness of 75% of the tests is directly attributable to the presence of the design smell. As a direct consequence, the removal of these smells may provide benefits in terms of flakiness removal.

Table 6: Number of Flakiness-Inducing Test Smells Before and After Their Removal.

Category	Any		Resource Optimism		Indirect Testing	
	Before	After	Before	After	Before	After
Async Wait	2,391	0	-	-	-	-
Concurrency	1,018	0	-	-	-	-
Test Order Dependency	921	0	-	-	879	0
Network	728	0	684	0	-	-
IO	1,605	0	1,605	0	-	-
Overall	**6,663**	**0**	**2,289**	**0**	**879**	**0**

Category	Test Run War		Fire and Forget		Conditional Test Logic	
	Before	After	Before	After	Before	After
Async Wait	-	-	2,391	0	-	-
Concurrency	1,018	0	-	-	-	-
Test Order Dependency	-	-	-	-	42	0
Network	-	-	44	0	-	-
IO	-	-	-	-	-	-
Overall	**1,018**	**0**	**2,435**	**0**	**42**	**0**

4.3 RQ$_3$: The Role of Test Smell Removal

In the context of **RQ$_3$** we applied operations only on the test smells causally related to flaky tests. Thus, we removed 6,663 test smell instances. Clearly, we could not remove the remaining 25% of flaky tests because they are not affected (or not causally affected) by test smells. Consequently, this means that in **RQ$_3$** we aimed at removing 75% of flaky tests by means of the recommended removal operations.

As expected, the fixing operations removed the test smells occurring in the affected test code: indeed, after re-organizing the source code, the test smell detector was not able to identify those smells anymore, suggesting that the operations are the correct solutions to deal with the considered test smells. While this result was expected (removal operations have the goal to remove

smells), it is still important because it suggests that such operations were properly applied in our context.

Much more interesting, the test smell removal also had a strong beneficial effect on the number of flaky tests occurring in the subject systems. As shown in Table 6, the number of flaky tests occurring after the application of a removal operation aimed at removing a flakiness-inducing test smell was reduced to zero: thus, **75% of the total flaky tests were removed through the recommended removal strategies**. This confirmed our hypothesis, showing that *developers can remove a sizable part of flaky tests by performing simple program transformations that do not functionally alter the tests (i.e., they do not test more/less after removal) and that make test code self-contained, focused on a given production class, or paying attention on how external calls are acquired and released.*

Listing 7: Removal Operation for the Indirect Testing instance detected in the Hibernate project

```
@Test
public void testJarVisitor() throws Exception {
...

URL jarUrl  = new URL ("file:./target/packages/defaultpar.par");
JarVisitor.setupFilters();
JarVisitor jarVisitor =  JarVisitorFactory.getVisitor(jarUrl, JarVisitor
    .getFilters(), null);
assertEquals(JarVisitor.class.getName(), jarVisitor.getClass().getName()
    );
}

@Test
public void testExplodedJarVisitor() throws Exception {
...

URL jarUrl  = new URL ("file:./target/packages/explodedpar");
ExplodedJarVisitor.setupFilters();
ExplodedJarVisitor jarVisitor =  JarVisitorFactory.getVisitor(jarUrl,
    ExplodedJarVisitor.getFilters(), null);
assertEquals(ExplodedJarVisitor.class.getName(), jarVisitor.getClass().
    getName());
}

@Test
public void testFileZippedJarVisitor() throws Exception {
...

URL jarUrl  = new URL ("vfszip:./target/packages/defaultpar.par");
FileZippedJarVisitor.setupFilters();
FileZippedJarVisitor jarVisitor =  JarVisitorFactory.getVisitor(jarUrl,
    JarVisitor.getFilters(), null);
assertEquals(FileZippedJarVisitor.class.getName(), jarVisitor.getClass()
    .getName());
}
```

For the sake of clarity, consider the case of the Indirect Testing previously presented in Listing 3. After the application of the *Extract Method*, the test code became as shown in Listing 7: specifically, the indirection was removed by creating two new test methods, called `testExplodedJarVisitor` and `testFileZippedJarVisitor`, besides the existing `testJarVisitor` test method. Each test method is responsible to test the `getVisitor` method refer-

ring to the corresponding production class. Moreover, before passing the filters as parameter to the `getVisitor` method (see `getFilters` method calls), such filters are set up through an explicit call to the method `setupFilters`, which is present in all the production classes tested. This removal operation made the test cases independent from their execution order. Therefore, their flakiness was removed after this operation.

The other removal operations, i.e., *Setup External Resource*, *Make Resource Unique*, *Add Await Condition*, and *Extract Method* (needed to remove the *Resource Optimism*, *Test Run War*, *Fire and Forget*, and *Conditional Test Logic* smells, respectively), had similar positive effects on flaky tests. As a matter of fact, all the removal operations performed produced a version of test cases where not only the design problems were removed, but also the flakiness was fixed.

Implications. The findings discussed so far have one *major implication* for both tool vendors and researchers. Based on our results, we can claim that removal of test smells represent an important methodology that developers can use to improve the overall effectiveness of test cases. Unfortunately, up to now there are no tools that allow the automatic test smell removal [57]. Thus, the definition of accurate approaches, able to recommend to developers design solutions that remove certain test smells, represents the next challenge to pursue. This is a call for both researchers and tool vendors, in order to build effective tools assisting developers during their daily activities: to this aim, analyses aimed at understanding *when* to recommend removal operations (e.g., at commit-time vs deadline-time) might be a useful piece of additional information to exploit to make test smell removal tools effective. At the same time, techniques making developers aware of the harmfulness of test smells (e.g., test smell summarizers) can represent an important plus to accelerate technological transfer and allow test smell detectors and removal tools to be adopted in practice.

Summary of RQ_3. Test smell removal represents a vital activity not only aimed at removing design flaws, but also aimed at fixing a sizable portion of the flaky tests affecting a software system. Specifically, we found that 75% of flaky tests were removed thanks to operations making the test code more self-contained, focused on a specific target, or careful when managing external calls and resources.

5 Threats to Validity

This section discusses possible threats that might have influenced our observations as well as the way we mitigated them.

5.1 Threats to construct validity

The main threats related to the relationship between theory and observation (*construct validity*) are related to possible imprecisions in the measurement performed. In principle, to elicit a catalog of flakiness-inducing test smells we performed a two-step process: on the one hand, we manually went over the definitions of the test smells defined so far in the literature, by deeply considering the multivocal literature review proposed by Garousi and Küçük [35]; on the other hand, we also performed semi-structured interviews with 10 professional software testers having more than 20 years of experience in testing of modern software systems. The latter step was done with the aim of ensuring that no further bad practice, not yet defined in literature, might not have been considered in the context of our study.

To identify test smell instances we relied on detection mechanisms that involved the use of (i) the tool proposed by Bavota et al. [40], which is publicly available and of which the performance was assessed at 88% of precision and 100% of recall in detecting *Resource Optimism*, *Indirect Testing*, and *Test Run War* instances and (ii) our own detector for the identification of *Fire and Forget* and *Conditional Test Logic* smells. To be sure of the actual suitability of such tools in our context, we performed a preliminary empirical study aimed at assessing the detection performance in our context: this process relied on a manual identification of test smell instances belonging to 3 projects of our dataset. The oracle construction involved two external professional developers having more than 7 years of programming experience, that actively work on the definition of test cases in their own companies, and having extensive experience in software design and bad practices. Moreover, we also generalized the detection accuracy by asking the external inspectors to establish the actual smelliness of a statistically significant sample of test smell candidates output by the exploited detectors and belonging to the systems of our dataset that were not considered in the initial evaluation. It is worth remarking that we relied on external inspectors with the aim of avoiding biases due to *Observer-expectancy effect* [48]. Such an extensive analysis validated that the detection approach was pretty effective, which entails that the vast majority of instances of the types of tests smells that were considered are part of our study.

Another threat is related to how we identified flaky tests: specifically, we run the test cases of a certain application ten times, and marked all the tests showing a different behavior in one of the ten runs as flaky. However, the choice of the number of runs was not random, but driven by the calibration carried out on three of the systems employed in our study, where we observed that ten runs are enough for discovering the maximum number of flaky tests (more details in Section 3).

5.2 Threats to conclusion validity

Threats to *conclusion validity* are related to the relationship between treatment and outcome. Most of the work done to answer our research questions was conducted by means of manual analysis, and we are aware of possible imprecisions made in this stage. However, it is important to note that a manual analysis was needed to perform our investigation. For example, in $\mathbf{RQ}_1$ we manually classified the motivations behind flaky tests because of the lack of automated tools able to perform this task automatically; moreover, it is worth noting that also the authors of the taxonomy used manually labeled flaky tests to understand their root causes [20]. When performing the task, we followed the same guidelines provided by Luo et al. [20]. Therefore, we are confident about the classification described in Section 4.

As for $\mathbf{RQ}_2$, it may be possible that the majority of the test smells appearing in tests do not cause any kind of flakiness, thus potentially decreasing the scope of our work and the conclusions we draw. In our dataset, we identified 11,120 test smells over the total 19,532 tests, and discovered that 75% of the 8,829 flaky tests were caused by one of the considered test smells. This means that 6,622 tests were flaky and smelly at the same time. As a consequence, 4,498 test smell instances did not cause flakiness. Thus, 40% of the smells are not harmful from a flakiness perspective.

On the one hand, from this analysis we notice the presence of a large number of "false positives", i.e., test smells that do not cause flakiness problems. On the other hand, however, we argue that this does not necessarily negatively influence the conclusions we made. Indeed, we still believe that a large number of test smells cause flakiness and thus developers should carefully take into account the possibility to remove them to avoid flakiness problems. At the same time, we also argue that, even if a certain test smell instance does not induce flakiness, developers should still re-organize test code as the presence of test smells is strongly associated with additional problems for software quality. In particular, as reported in recent work [53], test smells not only make tests more change- and defect-prone, but are also detrimental for the ability of tests to find defects in production code. Therefore, we conclude that test smell removal represents a powerful methodology helping with both flakiness- and quality-related issues.

Still in $\mathbf{RQ}_2$, to effectively discriminate the role of each flakiness-inducing test smell on the actual flakiness of a test method we performed a manual investigation in order to go beyond the simpler co-occurrence analysis, that we did not consider reliable (co-occurrence does not imply causality) to understand the relationship between the test smells considered and the flakiness of test code. To ensure the validity of our conclusions, we have also performed an additional validation of our data. In this regard, we involved the two external professional testers that built the oracle of test smells for the assessment of the detection approach and asked them to validate the lists of flaky tests and flakiness-inducing test smells. Similarly to the creation of an oracle of test smells, the inspectors were provided with (i) the source code of the subject

systems, (ii) the list of flaky tests identified, (iii) the list of smelliness-inducing flaky tests. It is important to remark that both the inspectors have experience with test flakiness and thus could provide valuable feedback. Besides the provided data, they could run the flaky test identification mechanism as well as the test smell detector if needed. The task they performed was to assign the value `true` to the items of the two lists that were considered as trustworthy, `false` otherwise. The reports we received from the inspectors fully supported our data analysis process. For this reason, we are even more confident of the validity of our analysis. In addition, we employed the Kendall rank correlation test [54], to statistically evaluate the strength of the relationship between the two phenomena taken into account. It is worth remarking that we selected this statistical test because it is generally more accurate than other correlation tests [58].

Finally, in the context of $\mathbf{RQ}_3$ we manually re-organized the source code because there is no tool able to perform automatic removal of test smells. While a manual re-organization of the source code may lead to a certain degree of subjectivity, we verified our ability in the application of removal strategies by re-running the test smell detector on the re-organized version of the subject systems: as a result, the detector did not detect the smells anymore. For this reason, we are confident about the way such operations were applied. Still in this category, another threat may be related to the effect of the test smell removal operation applied on the effectiveness of test cases. Theoretically, test smell removal is the process of changing the source code without altering its external behavior [38], and thus the effectiveness of test code before and after the re-organization should be the same. However, from a practical point of view, the application of such operations may lead to undesired effects [59]. To verify that the re-organization of the test code did not have negative effects on vital test code characteristics, we performed an initial additional analysis: we measured the code coverage of the test cases before and after the application of test smell removal operations. In particular, we considered the test cases of the two larger software systems in our dataset, i.e., Apache Lucene and Elastic Search, and measured the coverage before and after the code re-organization using the JaCoCo toolkit[4]. We observed that the level of branch coverage of both versions is exactly the same (on average, it reaches 64%): thus, while maintaining the same coverage we were able to fix more than half of the flaky tests by removing test smells.

5.3 Threats to external validity

Finally, as for threats to *external validity*, the main discussion point regards the generalizability of the results. We conducted our study taking into account 18 systems; we are aware that most of them come from a single ecosystem (the Apache Software Foundation). While this is still a limitation of our

[4] `http://www.eclemma.org/jacoco/`

study, it is important to note that the considered projects have different scope and characteristics, that allow us to claim that the results of the study hold when considering a wide range of system types. Nevertheless, replications of the study on larger and different datasets are desirable. At the same time, replications in the context of industrial settings might reveal different results due to, for instance, the different strategies adopted by developers when dealing with flaky tests and/or test smells. Our future research agenda includes the replication of the study in industry.

6 Related Work

In this section we provide an overview on the research conducted in the recent years on both test smells and flaky tests.

6.1 Test Smells

While the research community devoted a lot of effort on understanding [60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76] and detecting [47, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89] design flaws occurring in production code, smells affecting test code have only been partially explored.

Beck was the first to highlight the importance of well-designed test code [10], while Van Deursen et al. [31] defined a set of 11 test smells, i.e., a catalog of poor design solutions to write tests, together with operations aimed at removing them. Later on, Meszaros defined other smells affecting test code [32]. Based on these catalogs, Greiler et al. [41, 90] showed that test smells affecting test fixtures frequently occur in industrial contexts, and for this reason they presented TestHound, a tool able to identify fixture-related test smells such as *General Fixture* or *Vague Header Setup* [41]. Van Rompaey et al. [42] devised a heuristic code metric-based technique able to identify instances of two test smells, i.e., *General Fixture* and *Eager Test*, but the results of an empirical study showed the low accuracy of the approach.

As for empirical studies conducted on test smells, Bavota et al. [40] performed a study where they evaluated (i) the diffusion of test smells in 18 software projects, and (ii) their effects on software maintenance. The results showed that 82% of JUnit classes are affected by at least one test smell, and that their presence has a strong negative impact on the maintainability of the affected classes. Finally, Tufano et al. [44] conducted an empirical study aimed at measuring the perceived importance of test smells and their lifespan during the software life cycle. The main findings indicate that test smells are usually introduced during the first commit involving the affected test classes, and in 80% of the cases they are never removed, essentially because of poor awareness of developers.

Clearly, the study reported in this paper strongly differs from the existing literature since we showed how test smells can be nicely adopted to locate

flaky tests as well as the ability of test smell removal operations to be good test code fixing strategies. Moreover, our paper may represent a step ahead toward a higher awareness of the importance of test smells and their removal from the developers' point of view.

6.2 Flaky Tests

As pointed out by several researchers and practitioners, flaky tests represent an important issue for regression testing [20, 27, 16, 22, 23, 24]. Memon and Cohen [22] described a set of negative effects that flaky tests may create during regression testing, finding that their presence can even lead to missing deadlines due to the fact that certain features cannot be tested sufficiently [22]. Marinescu et al. [91] analyzed the evolution of test suite coverage, reporting that the presence of flaky tests produces an intermittent variation of the branch coverage.

Other researchers tried to understand the reasons behind test code flakiness. Luo et al. [20] manually analyzed the source code of tests involved in 201 commits that likely fixed flaky tests, defining a taxonomy of ten common root-causes. Moreover, they also provide hints on how developers usually fix flaky tests. As a result, they found that the top three common causes of flakiness are related to asynchronous wait, concurrency, and test order dependency issues. In $\mathbf{RQ}_1$, we partially replicated the study by Luo et al. in order to classify the root-causes behind the flaky tests in our dataset, discovering that on larger datasets other root-causes, such as network and input/output problems, are quite frequent as well.

Besides Luo *et al.*, also other researchers investigated the motivations behind flaky tests as well as devised strategies for their automatic identification. For instance, Zhang et al. [24] focused on test suites affected by test dependency issues, by reporting methodologies to identify these tests. At the same time, Muslu et al. [23] found that test isolation may help in fault localization, while Bell and Kaiser [27] proposed a technique able to isolate test cases in Java applications by tracking the shared objects in memory. Bell et al. [92] proposed DeFlaker, an automated technique that identifies flaky tests by running a mix between static and dynamic analyses: with respect to this paper, our work is complementary since we aim at studying how much wrong design or implementation choices applied by programmers during the development of test cases have an influence on test code flakiness.

Another well-studied root cause of flaky test is concurrency. In particular, Farchi et al. [28] identified a set of common erroneous patterns in concurrent code and suggested the usage of static analysis tools as a possible way to automatically detect them. Lu et al. [93] reported a comprehensive study into the characteristics of concurrency bugs, by providing hints about their manifestation and fixing strategies. Still, Jin et al. [29] devised a technique for automatically fixing concurrency bugs by analyzing the single-variable atomicity violations.

With respect to these studies, our work can be considered as a further step ahead toward the resolution of the flaky test problem: indeed, we demonstrated how most of the flaky tests can be fixed by applying operations aimed at making test code self-contained.

Finally, some studies focused on test code bugs. For instance, Daniel et al. [25] proposed an automated approach for fixing broken tests that perform changes in test code related to literal values or addition of assertions. Another alternative was proposed by Yang et al. [26], who devised the use of Alloy specifications to repair tests.

It is important to note that these fixing strategies refer to tests that fail deterministically, and cannot be employed for fixing flaky tests. Conversely, our solution can be easily applied in this context.

7 Conclusions and Future Work

In this paper, we first adopted a systematic mixed-method analysis to identify, from the whole set of test smells defined in literature, those of which the characteristics might determine some sort of test flakiness, which arises when tests do not have a deterministic outcome. The output of (1) a state-of-the-art analysis and (2) semi-structured interviews enabled us to define a catalog of five flakiness-inducing test smells, i.e., *Resource Optimism*, *Indirect Testing*, *Test Run War*, *Fire and Forget*, and *Conditional Test Logic*, originally coming from three different sources such as the works by Van Deursen et al. [31], Meszaros [32], and Garousi and Küçük [35].

With this catalog as a basis, we then conducted a large-scale investigation aimed at providing empirical evidence of the relation between flakiness-inducing test smells and actual flakiness of the test cases. Moreover, we aimed at understanding whether removal of test smells induces the fixing of flaky tests. In this empirical study, we have first performed an analysis aimed at measuring the magnitude of the flaky test phenomenon by (i) measuring the extent to which tests are flaky, and (ii) identifying the root-causes leading tests to be non-deterministic. Subsequently, we have looked deeper into the relationship between flaky tests and flakiness-inducing test smells. Specifically, we measured how frequently flaky tests and test smells co-occur and to what extent such smells causally relate to the root-causes leading tests to be flaky. Finally, to assess the role of test smell removal operations, we manually removed the test smells causally related to flaky tests. Afterwards, we re-ran the flaky test identification mechanisms to understand whether this removal also fixed the test code flakiness.

Our investigation provided the following notable findings:

- Flaky tests are quite diffused in open-source software systems, as we found that almost 45% of the JUnit test methods analyzed have a non-deterministic outcome. While the most prominent causes of flakiness are asynchronous waits, input-output issues, and concurrency problems, other

motivations such as test order dependency and network issues are also quite popular.
- Almost 81% of flaky tests are affected by one of the five flakiness-inducing test smells contained in our catalog. More importantly, we found that the cause of 75% of the flaky tests can be directly attributed to the characteristics of the co-occurring smell.
- All the flaky tests causally related to test smells can be fixed by applying test smell removal operations. As a consequence, we conclude that *test smell removal is an effective flaky test fixing strategy able to fix most of the tests having non-deterministic outcomes.*

We believe that our findings provide a strong motivation for practitioners to adopt test code quality checkers while developing test cases [94]. The results also represent a call to arms for researchers to define effective automated tools able to locate test smells and re-organize test code to improve the effectiveness of test suites. Finally, it is worth remarking the importance of our results for teaching software testing: indeed, we believe that educators should more carefully teach the effects of test smells on test flakiness, and how these test smells can be avoided.

Our future agenda focuses on the design of accurate test smell detectors and removal approaches. Furthermore, we plan to replicate our empirical study in an industrial setting, to assess the extent to which different development standards have an effect on the findings reported in this study. To fully guarantee replicability, we also plan to build a platform aimed at reporting and visualizing, for each refactoring operation performed in the context of $\mathbf{RQ}_3$, the specific action that was performed and the rationale behind it. Finally, we also plan to extend our investigation, further investigating the interplay between (i) test code quality, (ii) test code re-organization and (iii) test reliability as measured by mutation analysis [95] and/or code coverage [96].

Acknowledgements

We would like to thank the 10 developers that participated in the interviews, and the 2 external inspectors that helped us categorize the test smells. We thank the anonymous reviewers, whose comments and feedback significantly improved this paper. This work was partially sponsored by the EU Horizon 2020 ICT-10-2016-RIA "STAMP" project (No. 731529), the NWO "TestRoots" project (No. 639.022.314), and the SNF "Data-Driven Code Review" project (No. PP00P2_170529).

References

1. L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012, pp. 33:1–33:11.

2. E. Engström and P. Runeson, "A qualitative survey of regression testing practices," in *Proceedings of the International Conference on Product-Focused Software Process Improvement (PROFES)*. Springer Berlin Heidelberg, 2010, pp. 3–16.
3. F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 130–141.
4. M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their IDEs," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 179–190.
5. M. Beller, G. Gousios, and A. Zaidman, "How (much) do developers test?" in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 559–562.
6. M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the ide: Patterns, beliefs, and behavior," *IEEE Transactions on Software Engineering (TSE)*, to Appear.
7. G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 358–368.
8. J. Micco, "Flaky tests at Google and how we mitigate them," 2016, last visited, March 24th, 2017. [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html
9. M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*. ACM, 2017, pp. 356–367.
10. Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
11. M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 447–450.
12. Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 214–224.
13. A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2017, pp. 654–664.
14. A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 101–110.

15. D. Hao, L. Zhang, H. Zhong, H. Mei, and J. Sun, "Eliminating harmful redundancy for testing-based fault localization using test suite reduction: an experimental study," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2005, pp. 683–686.
16. M. Fowler, "Eradicating non-determinism in tests." [Online]. Available: https://martinfowler.com/articles/nonDeterminism.html
17. C. Developers, "Flakiness dashboard howto." [Online]. Available: http://www.chromium.org/developers/testing/flakiness-dashboard
18. G. Developers, "No more flaky tests on the go team." [Online]. Available: https://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team
19. E. Melski, "6 tips for writing robust, maintainable unit tests." [Online]. Available: https://blog.melski.net/tag/unit-tests/
20. Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 643–653.
21. F. J. Lacoste, "Killing the gatekeeper: Introducing a continuous integration system," in *2009 Agile Conference*, Aug 2009, pp. 387–392.
22. A. M. Memon and M. B. Cohen, "Automated testing of gui applications: Models, tools, and controlling flakiness," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1479–1480.
23. K. Muşlu, B. Soran, and J. Wuttke, "Finding bugs by isolating unit tests," in *Proceedings of the SIGSOFT Symposium on Foundations of Software Engineering and the European Conference on Software Engineering (ESEC/FSE)*. ACM, 2011, pp. 496–499.
24. S. Zhang, D. Jalali, J. Wuttke, K. Muslu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2014, pp. 385–396.
25. B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "Reassert: Suggesting repairs for broken unit tests," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2009, pp. 433–444.
26. G. Yang, S. Khurshid, and M. Kim, "Specification-based test repair using a lightweight formal method," in *Proceedings of the International Symposium on Formal Methods (FM)*, 2012, pp. 455–470.
27. J. Bell and G. Kaiser, "Unit test virtualization with VMVM," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 550–561.
28. E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *Proceedings International Parallel and Distributed Processing Symposium*, April 2003, pp. 7 pp.–.
29. G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM,

2011, pp. 389–400.
30. F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on.* IEEE, 2017, pp. 1–12.
31. A. van Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, 2001, pp. 92–95.
32. G. Meszaros, *xUnit Test Patterns: Refactoring Test Code.* Addison Wesley, 2007.
33. L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, "On the interplay between software testing and evolution and its effect on program comprehension," in *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer, 2008, pp. 173–202.
34. N. Mackenzie and S. Knipe, "Research dilemmas: Paradigms, methods and methodology," *Issues in educational research*, vol. 16, no. 2, pp. 193–205, 2006.
35. V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, 2018.
36. V. Garousi, M. Felderer, and M. V. Mäntylä, "The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering.* ACM, 2016, p. 26.
37. R. S. Weiss, *Learning from strangers: The art and method of qualitative interview studies.* Simon and Schuster, 1995.
38. M. Fowler, *Refactoring: improving the design of existing code.* Addison-Wesley, 1999.
39. F. Palomba and A. Zaidman, "The smell of fear: On the relation between test smells and flaky tests - online appendix," 2018. [Online]. Available: https://tinyurl.com/ycnmnd6w
40. G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? An empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.
41. M. Greiler, A. van Deursen, and M. A. Storey, "Automated detection of test fixture strategies and smells," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST).* IEEE, 2013, pp. 322–331.
42. B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, Dec 2007.
43. F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in *Proceedings of the International Workshop on Search-*

Based Software Testing (SBST). ACM, 2016, pp. 5–14.
44. M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 4–15.
45. A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011. [Online]. Available: https://doi.org/10.1007/s10664-010-9143-7
46. S. Kleiman, D. Shah, and B. Smaalders, *Programming with threads*. Sun Soft Press Mountain View, 1996.
47. D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 612–621.
48. D. L. Sackett, "Bias in analytic research," in *The Case-Control Study Consensus and Controversy*. Elsevier, 1979, pp. 51–63.
49. P. Jaccard, "Étude comparative de la distribution florale dans une portion des alpes et des jura," vol. 37, pp. 547–579, 1901.
50. R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.
51. N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.
52. T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.
53. D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE*, 2018.
54. M. Kendall, "Rank Correlation Methods," *Charles Griffin & Company Limited*, 1948.
55. Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.
56. F. Palomba, A. Zaidman, and A. Lucia, "Automatic test smell detection using information retrieval techniques," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE*, 2018.
57. J. Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and software Technology*, vol. 58, pp. 231–249, 2015.
58. C. Croux and C. Dehon, "Influence functions of the spearman and kendall correlation measures," *Statistical Methods & Applications*, vol. 19,

no. 4, pp. 497–515, 2010. [Online]. Available: http://dx.doi.org/10.1007/s10260-010-0142-z

59. G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? An empirical study," in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2012, pp. 104–113.
60. M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2011, pp. 181–190.
61. R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in *Proceedings of the International Workshop on Refactoring Tools*. ACM, 2011, pp. 33–36.
62. M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *Transactions on Software Engineering (TSE)*, vol. 43, no. 11, pp. 1063–1088, 2017.
63. F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.
64. A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," in *Proceedings of the international workshop on Principles of software evolution (IWPSE)*. ACM, 2007, pp. 31–34.
65. F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *In Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 101–110.
66. F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *IEEE Transactions on Software Engineering*, 2017.
67. F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Information and Software Technology*, vol. 99, pp. 1–10, 2018.
68. ——, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, pp. 1–34, 2017.
69. G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.
70. F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *Program*

Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on. IEEE, 2017, pp. 176–185.
71. R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Proceedings of the European Conference on Software Maintenance and ReEngineering (CSMR)*. IEEE, 2012, pp. 411–416.
72. D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2004, pp. 223–232.
73. D. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *Software Engineering, IEEE Transactions on*, vol. 39, no. 8, pp. 1144–1156, Aug 2013.
74. A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 682–691.
75. ——, "Do code smells reflect important maintainability aspects?" in *International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 306–315.
76. A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter, "Inter-smell relations in industrial and open source systems: A replication and comparative analysis," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 121–130.
77. F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A Bayesian approach for the detection of code and design smells," in *Proceedings of the 9th International Conference on Quality Software (QSIC)*. IEEE, 2009, pp. 305–314.
78. M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.
79. R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2004, pp. 350–359.
80. N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.
81. M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proceedings of the International Software Metrics Symposium (METRICS)*. IEEE, 2005.
82. R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on B-Splines," in *Proceedings of the* 14^{th} *Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, March 2010, pp. 248–251.

83. F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *Software Engineering, IEEE Transactions on*, vol. 41, no. 5, pp. 462–489, May 2015.
84. F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10.
85. F. Palomba, D. A. Tamburri, F. Arcelli Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, "Beyond technical aspects: How do community smells influence the intensity of code smells?" *IEEE Transactions on Software Engineering*, 2019.
86. N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.
87. F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," *IEEE Transactions on Software Engineering*, 2017.
88. G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman, "Developer-related factors in change prediction: An empirical assessment," in *Proceedings of the International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 186–195.
89. G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman, "Enhancing change prediction models using developer-related factors," *Journal of Systems and Software*, vol. 143, no. 9, pp. 14–28, 2018.
90. M. Greiler, A. Zaidman, A. van Deursen, and M.-A. Storey, "Strategies for avoiding text fixture smells during software evolution," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 387–396.
91. P. Marinescu, P. Hosek, and C. Cadar, "Covrig: A framework for the analysis of code, test, and coverage evolution in real software," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2014, pp. 93–104.
92. J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 433–444.
93. S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2008, pp. 329–339.
94. D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Trans. Software Eng.*, vol. 40, no. 11, pp. 1100–1125, 2014.
95. T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, New Haven, CT, USA, 1980, aAI8025191.

96. Y. K. Malaiya, M. N. Li, J. M. Bieman, and R. Karcich, "Software reliability growth with test coverage," *IEEE Transactions on Reliability*, vol. 51, no. 4, pp. 420–426, 2002.