Does Refactoring of Test Smells Induce Fixing Flaky Tests?

Palomba, Fabio; Zaidman, Andy

DOI
10.1109/ICSME.2017.12

Publication date
2017

Document Version
Peer reviewed version

Published in
Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017

Does Refactoring of Test Smells Induce Fixing Flaky Tests?

Fabio Palomba and Andy Zaidman
Delft University of Technology, The Netherlands
f.palomba@tudelft.nl, a.e.zaidman@tudelft.nl

***Abstract*—Regression testing is a core activity that allows developers to ensure that source code changes do not introduce bugs. An important prerequisite then is that test cases are deterministic. However, this is not always the case as some tests suffer from so-called *flakiness*. Flaky tests have serious consequences, as they can hide real bugs and increase software inspection costs. Existing research has focused on understanding the root causes of test flakiness and devising techniques to automatically fix flaky tests; a key area of investigation being concurrency. In this paper, we investigate the relationship between flaky tests and three previously defined test smells, namely *Resource Optimism*, *Indirect Testing* and *Test Run War*. We have set up a study involving 19,532 JUnit test methods belonging to 18 software systems. A key result of our investigation is that 54% of tests that are flaky contain a test code smell that can cause the flakiness. Moreover, we found that refactoring the test smells not only removed the design flaws, but also fixed all 54% of flaky tests causally co-occurring with test smells.**

***Index Terms*—Test Smells; Flaky Tests; Refactoring;**

I. Introduction

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

Unfortunately, test suites are often affected by bugs that can preclude the correct testing of software systems [12], [13]. A typical bug affecting test suites is *flakiness* [6]. Flaky tests are tests that exhibit both a passing and a failing result with the same code [6], *i.e.,* unreliable test cases whose outcome is not deterministic. The relevance of the flaky test problem is well-known and highlighted by both researchers and practitioners. For instance, dozens of daily discussions are opened on the topic on social networks and blogs [14], [15], [16], [17], [6].

Some key issues that are associated with the presence of flaky tests are that: (i) they may hide real bugs and are hard to reproduce due to their non-determinism [18], (ii) they increase maintenance costs because developers may have to spend substantial time debugging failures that are not really failures, but just flaky [19]. Perhaps most importantly, from a psychological point of view flaky tests can reduce a developer's confidence in the tests, possibly leading to ignoring actual test failures [17]. Because of this, the research community has spent considerably effort on trying to understand the causes behind test flakiness [18], [20], [21], [22] and on devising automated techniques able to fix flaky tests [23], [24], [25]. However, most of this research mainly focused the attention on some specific causes possibly leading to the introduction of flaky tests, such as concurrency [26], [25], [27] or test order dependency [22] issues, thus proposing *ad-hoc* solutions that cannot be used to fix flaky tests characterized by other root causes. Indeed, according to the findings by Luo *et al.* [18] who conducted an empirical study on the motivations behind test code flakiness, the problems faced by previous research only represent a part of whole story: 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 this paper, we aim at making a further step ahead toward the comprehension of test flakiness, by investigating the role of so-called *test smells* [28], [29], [30], *i.e.,* poor design or implementation choices applied by programmers during the development of test cases. In particular, looking at the test smell catalog proposed by Van Deursen *et al.* [28], we noticed that the definitions of three particular design flaws, *i.e., Resource Optimism*, *Indirect Testing* and *Test Run War*, are closely connected to the concept of test flakiness. For instance, the *Resource Optimism* smell refers to a test case that makes optimistic assumptions about the state or the existence of external resources [28], thus possibly producing a non-deterministic behavior due to the presence/absence of the referenced external resource.

Therefore, in this study we hypothesize that *test smells can represent a useful source of information to locate flaky tests.* We also hypothesize that the *refactoring of test smells may, as indirect effect, mean the resolution of the test flakiness.* To investigate our hypotheses, our investigation is steered by the following research questions:

- **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 refactoring of test smells help in removing test flakiness?*

To verify our hypotheses and seek answers to our research questions, we have set up an empirical study which involves the source code of 19,532 JUnit test methods belonging to 18 large software systems. Key results from our study indicate that flaky tests are quite common: almost 45% of JUnit test methods that we analyzed are flaky. Moreover, we found that 61% of flaky tests are affected by one of the test smells that we considered. A further investigation revealed that 54% of flaky tests are caused by problems attributable to the characteristics of the smells, and the refactoring of such flaws removed both the design problem and test code flakiness.

Structure of the paper. The remainder of this paper is organized as follows. Section II reports the empirical study planning and design, while Section III discusses the results of the study. In Section IV we debate about the possible threats that could have influenced our results. Subsequently, in Section V we report the literature related to flaky tests and test smells before concluding in Section VI.

II. 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 refactoring operations can be successfully applied to fix flaky tests by removing the design flaws affecting the test code. The *perspective* is of both researchers and practitioners, who are interested in understanding the benefits of refactoring for improving the effectiveness of test suites.

The first research question is 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.* [18], who performed a similar analysis inspecting 201 commits that likely fix flaky tests. With **RQ**$_2$ our goal is to perform a fine-grained investigation into the relationship between test smells and flaky tests. Finally, **RQ**$_3$ investigates the ability of refactoring in fixing test code flakiness by removing test smells.

A. Context Selection

The *context* of the study was composed of (i) subject systems, and (ii) test smells.

As for the former, Table I 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 different communities.

As for the test smells, we focused our attention on 3 design flaws coming from the catalog proposed by Van Deursen *et al.* [28], *i.e., Resource Optimism*, *Indirect Testing*, and *Test Run War*. While most of the test smells defined in literature relate to pure maintainability issues (*e.g.,* the *Assertion Roulette* represents a method having too many non-documented assertions [28]), we decided to focus on these test smells because they are somehow related to test flakiness. Indeed, by looking at the definitions reported below, we observed that all of them may cause a non-deterministic behavior due to a wrong usage of external resources, or to a random fail of objects that should be exercised by other test classes. More specifically, the definitions of the smells are the following:

Resource Optimism. test code that makes optimistic assumptions about the state or the existence of external resources: it may cause non-deterministic behavior in the test outcome [28]. To remove the smell, Van Deursen *et al.* suggest the use of *Setup External Resource* refactoring, *i.e.,* explicitly allocating the external resources before testing, being sure to release them when the test ends [28].

Indirect Testing. test methods affected by this smell test different classes with respect to the production class corresponding to the test class [28]. To remove this smell, Van Deursen *et al.* firstly suggest the application of an *Extract Method* refactoring [31] able to isolate the part of the method that actually tests different objects, and then (if needed) the application of a *Move Method* refactoring [31] to move the extracted code towards the appropriate class.

Test Run War. this smell arises when a test method allocates resources that are also used by other test methods [28], causing a possible resource interference making the result of a test non-deterministic. The refactoring operation associated with the smell is the *Make Resource Unique* refactoring, which consists of creating unique identifiers for all resources that are allocated by a test case [28].

B. Data Extraction

Once we had cloned the source code of the subject systems from the corresponding GitHub repositories (using the `git clone` command), we run two tools in order to (i) detect the test smells, and (ii) identify the tests having a non-deterministic outcome.

As for the test smells, we relied on the detection tool proposed by Bavota *et al.* [32], which has been employed in several previous works in the area [32], [33], [34]. Unlike other existing detection tools (*e.g.,* [35] or [36]), this tool can identify all the test smells considered in this study. Basically, the tool relies on the detection rules described in Table II. The identification of *Resource Optimism* follows the guideline defined by Van Deursen *et al.* [28], checking whether a test method contained in a JUnit class uses an external resource and does not check its status before using it, respectively. 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. Finally, for *Test Run War* the tool evaluates whether a test method allocates resources that are referenced by other test methods in the same JUnit class.

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

TABLE I: Characteristics of the Systems involved in the Study.

System	Description	Classes	Methods	KLOCs	Test Methods
Apache Ant 1.8.3	Java library and 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**

TABLE II: Rules used for the Detection of Test Smells.

Test Smell	Rule
Resource Optimism	JUnit methods using an external resource without checking its status.
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)

Note that the authors of the detector reported a precision close to 88% and a recall of 100%. In the context of this paper, we re-evaluated the precision of the detector[2] on a statistically significant sample composed of 366 test smell instances identified in the subject software projects. Such a set represents a 95% statistically significant stratified sample with a 5% confidence interval of the 7,812 total smell instances detected by the tool (*i.e.,* 40% of the test methods analyzed are smelly). The validation has been conducted manually by the authors of this paper, who checked the source code of each test method in order to confirm/refuse the presence of a design problem. As a result, the precision of the approach on our dataset is 86%: thus, the performance is in line with that of previous work and we can claim the accuracy of the information provided by the tool to be sufficiently high.

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 a software system which was not used in our experiment, *i.e.,* APACHE XERCES. 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, even setting α with higher values, the number of non-deterministic tests does not change, suggesting that ten is the right number of runs to discover flaky tests. For this reason, in the context of the study we set $\alpha = 10$ to identify flaky tests. Using this strategy, we identified 8,829 flaky tests (*i.e.,* 45% of the test methods analyzed are flaky). Note that while we cannot ensure that this identification strategy covers all the possible flaky tests, the results of the calibration allow us to be confident about its accuracy.

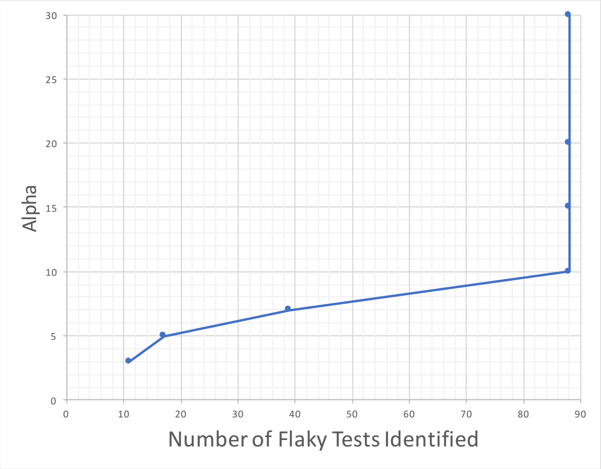

Fig. 1: Calibration of the parameter α used to identify flaky tests.

C. Data Analysis

To answer **RQ**$_1$, we manually investigated each flaky test identified in order to understand the root cause behind its

[2] Note that the recall could not be evaluated because of the lack of a comprehensive oracle of test smells for the projects considered.

TABLE III: Taxonomy of the root causes of flaky tests [18].

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.

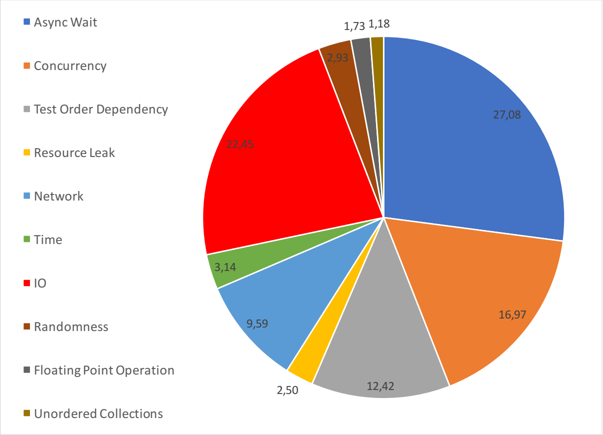

Fig. 2: Distribution of flaky tests across different categories.

flakiness. These causes have been classified by relying on the taxonomy proposed by Luo *et al.* [18], who identified ten common causes of test flakiness. Table III 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 approximatively 200 man/hours. In Section III we report the distribution of the flaky tests across the various categories belonging to the taxonomy.

As for **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*. 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). We call these test smell instances *flakiness-inducing test smells*. In Section III 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 **RQ**$_3$ we manually analyzed the source code of the test methods involved in a design problem and performed refactoring operations according to the guidelines provided by Van Deursen *et al.* [28]. This task has been performed by relying on (i) the definitions of refactoring and (ii) the examples provided by Van Deursen *et al.* [28]. It is worth remarking that the refactoring of test smells consists of program transformations that only involve the test code and that do not impact the external behavior of such test. The output of this phase consists of the source code where the test smells have been removed. Once we have refactored the source code, we have repeated the flaky test identification, in order to evaluate whether the refactoring operations had an effect on the test flakiness. In Section III we report descriptive statistics of the number of times refactoring operations successfully fix a flaky test instance by removing a test smell. In addition, we also provide qualitative examples that show why refactoring may be a powerful method to remove test code flakiness.

III. Analysis of the Results

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

A. *RQ*$_1$*: Causes behind test code flakiness*

As mentioned in Section II, 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 [18], [12]. Figure 2 depicts a pie chart reporting the distribution of the flaky tests belonging to each of the categories defined by Luo *et al.* [18]. 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. With respect to the findings by Luo *et al.* [18], we can observe a notable difference in the number of flaky tests due to IO reasons: indeed, in our study this number is much higher than previous work (≈ 22% vs ≈ 3%). Likely, this mismatch is 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). However, the method does not check the actual existence of such files, being subject of flakiness when the files are not present at the invoked location. 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 the one reported in previous work [18]. 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.

Listing 2: Network issue found in the Elastic Search project

```
@Test
public void testDecodeQueryString() {
  Map<String, String> params = newHashMap();

  String uri = "something?test=value";
  RestUtils.decodeQueryString(uri, uri.indexOf(’?’) + 1,
       params);
  assertThat(params.size(), equalTo(1));
  assertThat(params.get("test"), equalTo("value"));

}
```

An example of an issue caused by the use of network capabilities is the one present in ELASTIC SEARCH, a project that provides a RESTful search engine and, as such, transfers data over a network. Listing 2 shows the test method `testDecodeQueryString` belonging to the `RestUtilsTests` class, which has the goal of exercising the behavior of the production method `decodeQueryString`. To this aim, it performs a REST query with a sample Uniform Resource Identifier: the problem arises when a remote connection fails or delays, making the test case unstable.

On the other hand, categories such as RANDOMNESS, RESOURCE LEAK, FLOATING POINT OPERATOR, and UNORDERED COLLECTIONS are scarce and represent a small portion of the root causes for test code flakiness (8,34% in total).

In general, from this preliminary analysis it is important to note that most of the categories collecting the highest number of flaky tests are potentially related to the presence of design issues. Indeed, the IO issue shown in Listing 1 clearly relates to the presence of a *Resource Optimism* smell, as well as the CONCURRENCY problem that may be related to the *Test Run War* smell creating resource interferences [18], and may be avoided by adopting refactoring operations. A fine-grained analysis of the relationship between flaky tests and test smells is presented in the subsequent section.

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 (≈ 27%), IO (≈ 22%), and CONCURRENCY (≈ 17%), however reasons such as TEST ORDER DEPENDENCY (≈ 11%), and NETWORK (≈ 10%) explain a consistent part of flaky tests.

B. *RQ$_2$: The Relationship between Flaky Tests and Test Smells*

Over all the 18 systems analyzed in the study, the test smell detector identified 7,812 test methods affected by one of the smells considered, *i.e.,* almost 40% of the test cases contained a design issue. The most common smell was the *Resource Optimism*, which affected 3,017 test methods, while the *Indirect Testing* and *Test Run War* smells were detected in 2,522 and 2,273 test methods, respectively.

Table IV 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

TABLE IV: Co-occurrences between flaky tests and test smells.

Category	All Smells		Resource Optimism		Indirect Testing		Test Run War	
	Total	Causal	Total	Causal	Total	Causal	Total	Causal
Async Wait	27%	0%	26%	0%	0%	0%	1%	0%
Concurrency	70%	68%	1%	0%	1%	0%	68%	68%
Test Order Dependency	93%	80%	13%	0%	80%	80%	0%	0%
Resource Leak	1%	0%	1%	0%	0%	0%	0%	0%
Network	81%	81%	81%	81%	0%	0%	0%	0%
Time	12%	0%	0%	0%	12%	0%	0%	0%
IO	81%	81%	81%	81%	0%	0%	0%	0%
Randomness	1%	0%	0%	0%	0%	0%	1%	0%
Floating Point Operation	1%	0%	1%	0%	0%	0%	0%	0%
Unordered Collections	2%	0%	1%	0%	1%	0%	0%	0%
Overall	**61%**	**54%**	**31%**	**26%**	**11%**	**10%**	**19%**	**18%**

smell (see Section II-C). 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 frequently occur together (*i.e.,* 61% 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 54% of the cases, and (ii) have strong relationships with three top root causes of test flakiness, *i.e.,* CONCURRENCY, TEST ORDER DEPENDENCY, and IO. Consequently, this means that *refactoring may potentially remove more than half of flaky tests present in a software project*.

When we analyzed our findings more in depth per smell, we found that *Resource Optimism* is a design flaw closely related 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 3: 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 3 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-occuring 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 [37] 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.* [18] 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 found 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.

Turning the attention to the results achieved when considering the *Indirect Testing* smell, 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 4: Indirect Testing instance detected in the Hibernate project

```
1  @Test
2  public void testJarVisitor() throws Exception{
3     ...
4
5    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 4 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 4) 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.

Finally, 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 5: Test Run War instance detected in the Apache Qpid 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 }
```

The most common issue observed in this case was due to *deadlock* conditions [38], which appear when more threads wait for other threads to release the lock. For example, consider the case of the test method `AbstractAcquirableResourceTests.testSimple` of the APACHE LUCENE project shown in Listing 5. 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 5) and never turned into ready. Thus, other threads cannot continue their execution, causing a deadlock condition (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.

Summary of RQ$_2$. 61% of the flaky tests are also affected by a test smell. Interestingly, the cause of flakiness of 54% of the tests is directly attributable to the presence of the design smell. As a direct consequence, the refactoring of these smells may provide benefits in terms of flakiness removal.

C. RQ$_3$: The Role of Refactoring

In the context of **RQ$_3$** we applied refactoring operations only on the test smells causally related to flaky tests. Thus, we refactored 4,767 test smell instances (*i.e.,* the 54% of the

TABLE V: Number of Flakiness-Inducing Test Smells Before and After Refactoring Application.

Category	Any		Resource Optimism		Indirect Testing		Test Run War	
	Before	After	Before	After	Before	After	Before	After
Concurrency	1,593	0	-	-	-	-	1,593	0
Test Order Dependency	879	0	-	-	879	0	-	-
Network	684	0	684	0	-	-	-	-
IO	1,611	0	1,611	0	-	-	-	-
Overall	**4,767**	**0**	**2,295**	**0**	**879**	**0**	**1,625**	**0**

flakiness-inducing test smells found in **RQ**$_2$). Clearly, we could not refactor the remaining 46% of flaky tests because they are not affected (or not causally affected) by tests smells. Consequently, this means that in **RQ**$_3$ we aimed at removing 54% of flaky tests by means of refactoring.

As expected, the refactoring removed the design smells occurring in the affected test code: indeed, after refactoring the source code, the test smell detector was not able to identify those smells anymore. Much more interesting, the refactoring also had a strong beneficial effect on the number of flaky tests occurring in the subject systems. As shown in Table V, the number of flaky tests occurring after the application of a refactoring operation aimed at removing a *flakiness-inducing test smell* was reduced to zero: thus, **54% of the total flaky tests were removed through refactoring**. This confirmed our hypothesis, showing that *developers can remove a sizable part of flaky tests by performing simple program transformations making their test code self-contained and focused on a given production class.*

Listing 6: Refactoring the Indirect Testing instance detected in the Hibernate project

```java
@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 sake of clarity, consider the case of the Indirect Testing previously presented in Listing 4. After the application of the *Extract Method* refactoring, the test code becomes as shown in Listing 6: 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 referring to the corresponding production class. Moreover, before passing the filters as parameter of the `getVisitor` method (see `getFilters` method calls), such filters are set up through an explicit call to the method `setupFilters` present in all the production classes tested. This refactoring made the test cases independent from their execution order. Therefore, their flakiness was removed after this operation.

The *Setup External Resource* and *Make Resource Unique* refactorings (needed to remove the *Resource Optimism* and *Test Run War* smells, respectively) have similar positive effects on flaky tests. Indeed, all the refactoring operations performed produced a version of test cases where not only the design problems were removed, but also the flakiness was fixed.

Summary of RQ$_3$**.** Refactoring 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 54% of flaky tests were removed thanks to refactoring operations making the test code more self-contained and focused on a specific target.

IV. Threats to 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 identify test smell instances we relied on the detector proposed by Bavota *et al.* [32], which is publicly available and provides good performance, assessed at 88% of precision and 100% of recall. In our work we double-checked the accuracy of the tool taking into account a statistical significant sample of 366 test smell instances, finding a precision of 86%. While this analysis does not ensure that all the test smell instances in the subject system have been identified (*i.e.,* we cannot ensure that the tool achieves 100% recall), we believe that the results achieved in terms of precision represent a good indication of the accuracy of the detector. Moreover, it is worth noting that this tool has been used in other works in the field [33], [34].

Another threat is related on 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 APACHE XERCES system, where we observed that ten runs are enough for discovering the maximum number of flaky tests (more details in Section II).

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 **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 [18]. When performing the task, we followed the same guidelines provided by Luo *et al.* [18]. Therefore, we are confident about the classification described in Section III.

As for **RQ**$_2$, to effectively discriminate the *flakiness-inducing test smells* we performed a manual investigation in order to go beyond the simpler co-occurrence analysis, that we did not consider totally reliable to understand the relationship between the test smells considered and the flakiness of test code. In addition, we employed the Kendall rank correlation test [37], 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 [39].

Finally, in the context of **RQ**$_3$ we manually refactored the source code because there is no tool able to perform automatic refactoring 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 refactoring 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 refactoring operations were applied.

Still in this category, another threat may be related to the effect of the refactoring applied on the effectiveness of test cases. Theoretically, refactoring is the process of changing the source code without altering its external behavior [31], and thus the effectiveness of test code before and after the refactoring should be the same. However, from a practical point of view, the application of a refactoring may lead to undesired effects [40]. To verify that the re-organization of the test code did not have negative effects on vital test code characteristics, we performed an additional analysis aimed at measuring the code coverage of the test cases before and after the application of refactoring. 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 of the test cases before and after refactored using the JACOCO toolkit[3]. 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.

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 having different scope and characteristics in order to make the results as generalizable as possible. However, replications of the study on larger datasets are desirable.

V. RELATED WORK

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

A. About Test Smells

While the research community devoted a lot of effort on understanding [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52] and detecting [53], [54], [55], [56], [57], [58], [59], [60], [61] 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 [8], while Van Deursen *et al.* [28] defined a set of 11 test smells, *i.e.,* a catalog of poor design solutions to write tests, together with refactoring operations aimed at removing them. Later on, Meszaros defined other smells affecting test code [29]. Based on these catalogs, Greiler *et al.* [35], [62] 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* [35]. Van Rompaey *et al.* [36] 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.* [32] 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.* [34] 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

[3] http://www.eclemma.org/jacoco/

nicely adopted to locate flaky tests as well as the ability of refactoring to be a good test code fixing strategy. Moreover, our paper may represent a step ahead toward a higher awareness of the importance of test smells and refactoring from the developers' point of view.

B. About Flaky Tests

As pointed out by several researchers and practitioners, flaky tests represent an important issue for regression testing [18], [25], [14], [20], [21], [22]. Memon and Cohen [20] 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 [20]. Marinescu *et al.* [63] 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.* [18] 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.* [22] focused on test suites affected by test dependency issues, by reporting methodologies to identify these tests. At the same time, Muslu *et al.* [21] found that test isolation may help in fault localization, while Bell and Kaiser [25] proposed a technique able to isolate test cases in Java applications by tracking the shared objects in memory.

Another well-studied root cause of flaky test is concurrency. In particular, Farchi *et al.* [26] 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.* [64] reported instead a comprehensive study into the characteristics of concurrency bugs, by providing hints about their manifestation and fixing strategies. Still, Jin *et al.* [27] 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 refactoring operations aimed at making test code self-contained.

Finally, some studies focused on test code bugs. For instance, Daniel *et al.* [23] 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.* [24], 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.

VI. Conclusion

In this paper we conducted an empirical study aimed at understanding whether refactoring of test smells induces the fixing of flaky tests. To this aim, we have first performed a preliminary 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 three test smells of which the definitions suggest a relation with test code flakiness, namely *i.e., Resource Optimism*, *Indirect Testing* and *Test Run War*. Specifically, we measured how frequently flaky tests and test smells co-occur and to what extent the design flaws causally relate to the root-causes leading tests to be flaky. Finally, to assess the role of refactoring we manually removed the test smells causally related to flaky tests. Afterwards, we re-ran the flaky test identification to understand whether such removal also fixed the test code flakiness.

Our investigation provided the following notable findings:

- Flaky tests are quite diffused in real 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 61% of flaky tests are affected by one of the three test smells considered. More importantly, we found that the cause of 54% of the flaky tests can be attributed to the characteristics of the co-occurring smell.
- All the flaky tests causally related to test smells can be fixed by applying refactoring operations. As a consequence, we conclude that *refactoring is an effective flaky test fixing strategy able to fix more than half of the tests having non-deterministic outcomes*.

We believe that our findings provide a strong motivation for practitioners to adopting test code quality checkers while developing test cases [65]. At the same time, the results represent a call to the arms for researchers to define effective automated tools able to locate test design flaws and refactor test code to improve the effectiveness of test suites.

Our future agenda is focused on the design of accurate test smell detectors and refactoring approaches. We also plan to extend our investigation, further investigating the interplay between (i) test code quality, (ii) test code refactoring and (iii) mutation analysis [66].

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)*, M. Ali Babar, M. Vierimaa, and M. Oivo, Eds. Springer Berlin Heidelberg, 2010, pp. 3–16.

[3] 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.

[4] 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.

[5] 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.

[6] 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

[7] 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.

[8] Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] M. Fowler, "Eradicating non-determinism in tests." [Online]. Available: https://martinfowler.com/articles/nonDeterminism.html

[15] C. Developers, "Flakiness dashboard howto." [Online]. Available: http://www.chromium.org/developers/testing/flakiness-dashboard

[16] G. Developers, "No more flaky tests on the go team." [Online]. Available: https://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team

[17] E. Melski, "6 tips for writing robust, maintainable unit tests." [Online]. Available: https://blog.melski.net/tag/unit-tests/

[18] 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.

[19] F. J. Lacoste, "Killing the gatekeeper: Introducing a continuous integration system," in *2009 Agile Conference*, Aug 2009, pp. 387–392.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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)*, D. Giannakopoulou and D. Méry, Eds. Springer Berlin Heidelberg, 2012, pp. 455–470.

[25] 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.

[26] 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.–.

[27] 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.

[28] 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.

[29] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison Wesley, 2007.

[30] 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.

[31] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] M. Kendall, "Rank Correlation Methods," *Charles Griffin & Company Limited*, 1948.

[38] J. W. Havender, "Avoiding deadlock in multitasking systems," *IBM Syst. J.*, vol. 7, no. 2, pp. 74–84, Jun. 1968. [Online]. Available: http://dx.doi.org/10.1147/sj.72.0074

[39] 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

[40] 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.

[41] 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.

[42] 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.

[43] 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)*, p. To appear, 2017.

[44] 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.

[45] 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)*. New York, NY, USA: ACM, 2007, pp. 31–34.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] ——, “Do code smells reflect important maintainability aspects?” in *International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 306–315.

[52] 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.

[53] 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)*. Hong Kong, China: IEEE, 2009, pp. 305–314.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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*, R. Capilla, R. Ferenc, and J. C. Dueas, Eds. IEEE Computer Society Press, March 2010.

[59] 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.

[60] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, “A textual-based technique for smell detection,” in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[61] N. Tsantalis and A. Chatzigeorgiou, “Identification of move method refactoring opportunities,” *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[62] 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.

[63] 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)*, ser. ISSTA 2014. ACM, 2014, pp. 93–104.

[64] 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.

[65] 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.

[66] T. A. Budd, “Mutation analysis of program test data,” Ph.D. dissertation, New Haven, CT, USA, 1980, aAI8025191.