Understanding Flaky Tests: The Developer's Perspective

Moritz Eck,[1] Fabio Palomba,[1] Marco Castelluccio,[2] Alberto Bacchelli[1]
[1]University of Zurich, Switzerland — [2]Mozilla Software Foundation, United Kingdom
eck@uzh.ch,mcastellucio@mozilla.org,palomba@ifi.uzh.ch,bacchelli@ifi.uzh.ch

ABSTRACT

Flaky tests are software tests that exhibit a seemingly random outcome (pass or fail) when run against the same, identical code. Previous work has examined fixes to flaky tests and has proposed automated solutions to locate as well as fix flaky tests—we complement it by examining the perceptions of software developers about the nature, relevance, and challenges of this phenomenon.

We asked 21 professional developers to classify 200 flaky tests they previously fixed, in terms of the nature of the flakiness, the origin of the flakiness, and the fixing effort. We complement this analysis with information about the fixing strategy. Subsequently, we conducted an online survey with 121 developers with a median industrial programming experience of five years. Our research shows that: The flakiness is due to several different causes, four of which have never been reported before, despite being the most costly to fix; flakiness is perceived as significant by the vast majority of developers, regardless of their team's size and project's domain, and it can have effects on resource allocation, scheduling, and the perceived reliability of the test suite; and the challenges developers report to face regard mostly the reproduction of the flaky behavior and the identification of the cause for the flakiness.

KEYWORDS

Flaky Tests; Empirical Studies; Mixed-Method Research.

1 INTRODUCTION

Software tests are *flaky* when they exhibit a seemingly random outcome (i.e., pass or fail) despite exercising code that has not been changed. Even though previous work has proposed techniques to automatically fix some types of flaky tests [10, 11, 14, 21, 39], our scientific knowledge on this phenomenon is still very limited, especially on all the possible causes leading to test flakiness, the prevalence of the phenomenon (e.g., does it affect only larger projects or certain domains?), how problematic developers' perceive flaky tests and the challenges they face when they handle these tests.

Luo et al. [24] presented the earliest and most significant work advancing our empirical knowledge on flaky tests. They personally inspected 201 commits that likely fixed flaky tests from 51 Apache projects. They, thus, derived a taxonomy of the most common root causes (their *nature*, henceforth) of flaky tests, as well as identified strategies to manifest and fix certain types of flakiness [24].

The work we present in this paper continues on this line of research about understanding flaky tests. Here we present *the developer's perspective*: Our goal is to investigate developers' perception on the causes and effort in fixing flaky tests, the significance of the problem, as well as what they deem to be the most important challenges. An improved comprehension of these aspects is key to the definition of solutions and research lines to better help practitioners in diagnosing and fixing test flakiness.

To achieve our goal, (1) we asked 21 professional Mozilla developers to classify 200 real-world flaky tests they had previously fixed, in terms of the nature of the flakiness, the origin of the flakiness (test or production code), and the fixing efforts. We complement this analysis with information about the fixing strategy, which we collect from the source code repository. Focusing on a single ecosystem gives us the opportunity to detect the less frequent cases that would not likely appear by panning wide. Subsequently, (2) we conducted an online survey that was answered by 121 developers (106 professionals and 15 academics). Since these developers are from different projects and backgrounds, their answers give us the opportunity to learn from a variety of cases.

The categorization by the Mozilla developers led to the discovery of four previously unreported causes of flakiness, which are also deemed as those requiring the most effort to fix. We also found that most surveyed developers (79%) consider flaky tests a moderate to serious problem, regardless of their team's size and project's domain; and that 40% deal with flaky tests at least weekly. In terms of problems, flaky tests are reported to have serious consequences on the scheduling, allocation, and reliability of the testing process. Finally, reproducing the flaky behavior and classifying its cause are perceived as the major challenges. Our main contributions are:

(1) An extended **taxonomy** of the nature of flaky tests (including their **frequency** in the Mozilla ecosystem), based on the analyses of 21 professional developers, which validates and widens the taxonomy previously defined [24];

(2) Empirical data on the **fixing strategies** and **effort** to diagnose and fix each flaky test type, which also reveals that the newly identified types of flakiness are among the most diffused and hard to fix according to developers;

(3) Empirical evidence, from the analysis of 121 survey responses, on the **relevance** of the flakiness problem in practice;

(4) A set of **challenges** that developers perceive when fixing flaky tests, reporting a list of aspects that, if solved, would make the management of test flakiness more efficient;

Moritz Eck,[1] Fabio Palomba,[1] Marco Castelluccio,[2] Alberto Bacchelli[1]

We publicly release the full dataset concerning our contributions, as well as the materials for the other analyses [12].

2 RESEARCH GOALS AND METHODOLOGICAL OVERVIEW

The *goal* of this study is to capture's the developer's perspective on flaky tests with the *purpose* of: (1) understanding the nature of test flakiness, investigating why and where flaky tests arise, as well as the fixing effort and strategies, (2) gathering the relevance of and problems caused by flaky tests in practice, and (3) collecting the challenges faced by developers when fixing flaky tests. The *perspective* is of both researchers and practitioners, to better grasp this complex argument and decrease the time developers spend handling them, thus reducing the cost of the maintenance process.

2.1 Research Questions

We start our research by asking professional developers to classify (in terms of nature, origin, and fixing effort) flaky tests they previously fixed. This can be seen as a partial replication of the study by Luo et al. [24], with the key difference that we do not classify the underlying causes of flakiness by ourselves, rather the original developers do and add information on the origin and fixing effort. Moreover, we complement the developers' analysis with information on the fixing strategies. Our goal is to triangulate the taxonomy proposed by Luo et al. [24] from a different data source and with a different methodology, with the aim of validating and extending it with more data and dimensions. Hence, our first research question:

> **RQ$_1$.** *How do professional developers categorize flaky tests, in terms of nature, origin, and fixing effort? How do they fix them?*

Subsequently, we turn to a broader audience of respondents. We conduct a survey targeting software developers (from an open source and industrial development background and researchers working on software engineering) on their experience with flaky tests. We investigate the frequency of flaky tests in practice and how problematic they are from the developers' perspective. We ask:

> **RQ$_2$.** *How prominent is test flakiness and how problematic is it as perceived by developers?*

Finally, we investigate the challenges that developers perceive as most critical when they have to deal with flaky tests. We ask:

> **RQ$_3$.** *What are the main challenges that developers face when dealing with flaky tests?*

2.2 Overview of the Research Methodology

Our study features a mixed-methods approach [22] where quantitative and qualitative research are run in parallel with the goal of converging toward an empirical understanding of test code flakiness. We design and conduct a study to obtain data from two main sources: A novel annotated dataset (**DS**) of flaky tests (with 200 instances annotated by 21 professional developers) and the responses to an online survey (**OS**) (with 121 valid responses from developers).

2.3 Developers' Analysis Of Flaky Tests

In the following, we describe the methodological steps conducted to address our first research question.

Subject Flaky Tests. We first need to collect a dataset of flaky tests, which we then present to developers for their analysis. To this aim, we mine the Bugzilla issue tracker of Mozilla (i.e., a large free software organization counting more than 40,000 active contributors spread around the world and developing a variety of software, ranging from layout engines to operating systems). Mozilla has a database of flaky tests, verified and fixed by developers. According to such a database, the organization has 100 to 150 new flaky tests being detected every week: this makes it particularly suitable for our study because developers are very frequently in charge of diagnosing and fixing flaky tests, being therefore able to provide authoritative information on the nature of test flakiness, its origin, and the fixing effort.

We extract the data available after a flaky test has been reported. Thus, we gather the log files produced by the continuous integration system and the source code associated with the fixed flaky test (if available). In a first step, we identify all (869) fixed flaky tests the Mozilla's bug tracking system tagged `resolved` and `fixed` between May 2017 and April 2018. We consider this time window since the company only stores the log files for twelve months after their creation. From the set of retrieved flaky tests, we remove all instances for which no patch has been applied or other type of attachment is available (e.g., some flaky tests stop occurring after a while and are closed as fixed), as they were never fixed or were false alarms. This step filters out 295 tests; resulting in a set of 574 flaky tests, which we use in the next steps.

Recruiting Mozilla Developers. The remaining 574 actual flaky tests are taken into further consideration and sorted according to the developer who is listed as the assignee or patch creator. From this list, to ensure we select people with a concrete knowledge on flaky tests, we exclude 177 programmers who have only fixed less than five flaky tests. We end up with a list of 29 developers for a total of 304 fixed flaky tests. We contact these programmers through email and ask them to analyze the flaky tests as detailed in the following subsection.

Developers' Analysis. We create a spreadsheet for each developer that contains two columns: the first reports the link to the bug report related to each flaky test fixed by a certain developer; the second is used by the programmer to classify the nature of each flaky test in the list. The participating developers are allowed to use the taxonomy defined by Luo et al. [24] as a starting point for the classification process, as it (1) eases their task and (2) standardizes their answers—this is commonly done in both confirmatory and exploratory surveys [15]. However, they are also allowed to create additional categories if none of the existing ones fit; at the same time, they can indicate more than one type to specify the reasons behind the flakiness of a test (it may be flaky because of both a race condition and a network problem). We also ask the developers to specify the development effort they spent to fix the flaky tests, based on a Likert scale [28] between 1 (very low) to 5 (very high), as well as the origin (test or production code) of the flakiness. Once each developers completes the analysis, they send back their spreadsheets with the annotated classification. From 21

different developers, we receive a total of 248 responses coming from 21 professional Mozilla developers, with 7.5 years (median) of experience with Mozilla, 958 bug reports assigned to, and 14 flaky tests fixed (median) over the last year. Each developer provide us with 8.5 (median) answers (min = 4, max = 48). Of those responses, we exclude 48: (i) 24 turned into permanent failures, i.e., they were labeled as intermittent by a Mozilla sheriff,[1] but the fixing developer found it a permanent failure.; (ii) for the other 24, the type of flakiness was unknown even to the developers, meaning that the fixing developers worked around the flakiness rewriting the whole test and code-under-test, but they could not ascertain the cause of the flakiness. By removing these cases, we intend to rely on precise information, only. Nevertheless, the excluded cases are available to our online appendix [12], along with a more detailed report of the reasons behind their exclusion from our study.

Analysis of The Developers' Responses. We use the 200 classified flaky tests to define a taxonomy on the nature of flaky tests. In 72 cases (31%) the assigned categories do not correspond to those available in the taxonomy by Luo et al. [24] and, therefore, they are new and need to be defined. To this aim, we conduct an *iterative content analysis* [37] to assign a common name to these new categories. The process involves three software engineering researchers, all authors of this paper, (one graduate student, one research associate, and one faculty member) and consists of the two iterative sessions:

Iteration 1: The first author of this paper goes over the classifications made by developers. If the categories assigned belong to the taxonomy by Luo et al. [24], he leaves them as they are; if not, he assigns a temporary label to them. As an output, this step provides a draft categorization of the types of flakiness.

Iteration 2: The three researchers open a discussion on the draft taxonomy, with the aim of reaching a consensus on the assigned categories. Afterwards, the first author re-categorizes flaky tests according to the decisions taken during the discussion. Finally, the second author of the paper double-checks the classifications to reduce the risk of errors (he found no wrong classifications, thus reaching a total agreement).

The resulting extended taxonomy is then used to address **RQ$_1$**. Besides reporting the list of such categories, we also characterize them with additional information on (i) their frequency, computed on the basis of the distribution of the categories within the dataset, and (ii) the fixing strategies. As for the latter point, to characterize the fixing strategies adopted by developers when dealing with flaky tests, two authors of this paper conduct a new iterative content analysis that—similarly to the previous one—consists of two iterations. In the first iteration, the first author analyzes the patches associated to the resolution of the considered flaky tests to label them with a name and a short description of the fixing action performed by developers. In the second iteration, the two authors open a discussion on the initial labels assigned to reach a consensus. Then, the first author applies the changes according to the discussion while the second re-checks the final classifications to reduce threats in the interpretation of the results.

---

[1]https://wiki.mozilla.org/Sheriffing

2.4 Survey: Research Method

To answer **RQ$_2$** and **RQ$_3$**, we design and deploy an online survey, and analyze the collected answers.

Overall Survey Design. Following the guidelines provided by Flanigan et al. [15], we limit common issues possibly arising in survey studies and affecting the response rate: we keep the survey short, respecting the anonymity of participants and preventing our influence in the answers. We create an anonymous online survey using a professional tool (i.e., Surveygizmo [1]).

The survey first describes to the respondents the concept of flaky tests, following the definition accepted by the research community: *"[flaky tests are] intermittent tests that sometimes pass, sometimes fail, even though there are no changes in the code they test"*. Then, the survey asks for demographic information about the participants, including programming/testing experience as well as company and team size/domain. Subsequently, the survey contains other two main sections to collect data about the relevance of flakiness (**RQ$_2$**) and its challenges (**RQ$_3$**).

RQ$_2$ – Collecting Information About Relevance. We gather data on (i) how many times flaky tests occur, (ii) how problematic they are, and (iii) what are the top 3 to 5 problems caused by test flakiness. Participants can answer the first two questions by using a 5-point Likert scale indicating the extent to which the problem exists and how much it is serious, while they are free to write down the problems (if any) in a text box.

RQ$_3$ – Collecting Information About Challenges. We first conduct a multivocal literature review (MLR) [19]. This allows us to let emerge eight pieces of information that may be linked to challenges in the process of fixing flaky tests. Afterwards, we ask the survey respondents to rate how important is each of those information pieces from 'Not at all important' to 'Extremely important' (on a 4-point Likert scale) and the difficulty in obtaining it from 'Very easy' to 'Very difficult' (on a 5-point Likert scale). We also let the respondents indicate any information needs and/or challenges they face that are not included in the list (respondents indicated a total of another eleven needs/challenges).

For our MLR, we analyze (as described in the following protocol) both previously published papers on the topic (i.e., white literature) and online sources (e.g., blog posts, websites, and documents written by practitioners) related to flaky tests (i.e., gray literature).

- *Identifying relevant white literature:* We perform a keyword search on Google Scholar, IEEE Explore, and ACM Digital Library to create an initial selection of articles. As keywords we use: 'flaky tests', 'flaky test resolution practices', 'intermittent failures'. This step results in an initial set of 15 papers. We also perform forward and backward snowballing [38], inspecting sources referred or that refer to those belonging to the initial set of primary studies. At the same time, we consider the proceedings of all the relevant, top-tier conference venues in Software Engineering such as ICSE [7], ISSTA [3], ICST [8], ICSME [5], ESEC/FSE [2] and journals such as IEEE TSE [6], ACM TOSEM [4], and Springer's EMSE [9]. This step results in the inclusion of additional four papers. Once having this set of resources, we filter out the papers that do not report information on what are the practices used by developers to diagnose and/or fixing flaky tests: This is achieved by reading study setting, methodology, and conclusions made in

Moritz Eck,[1] Fabio Palomba,[1] Marco Castelluccio,[2] Alberto Bacchelli[1]

all the considered papers. The filtering process is jointly discussed among all the authors of the paper and, in the end, we agree on a list of three papers.

- *Identifying relevant gray literature:* We conduct the gray literature survey following the guidelines by Garousi et al. [19]. We query GOOGLE by using 'flaky tests', 'flaky test resolution practices', 'intermittent failures' as keywords. We perform our search on the results appearing in all the relevant GOOGLE pages until saturation is reached, i.e., until the search results contain references to blogs, documents, or websites reporting information on flaky tests. We take great care to ensure that considered sources originate from reputable companies, organizations, and authors. We also take steps from preventing our personal search bias from playing a role in the search results, hence we search on GOOGLE using the incognito mode. Overall, we find 56 relevant sources to further consider. Whenever possible, we also follow the web links contained in the initial set of sources as they might refer to other documents. This step results in the addition of three sources. The high number of relevant sources identified in the gray literature, as opposed to that of the white one, hints at the fact that the problem of understanding flaky tests has been only partially addressed by the research community. This situation calls for further empirical studies to create a stronger scientific knowledge around test flakiness.

- *Data Extraction:* We summarize each of the 62 identified relevant sources and create an enumeration of the aspects that are mentioned and may impact on the flaky tests fixing process. Then, we conduct another iterative content analysis sessions to group those aspects into the higher level themes that are related to the important information developers need to have when diagnosing and fixing test flakiness. This is a two-step process similar to the one conducted in the context of RQ_1: the second author of this paper performs a first iteration and then the draft themes are refined with the help of the last author. This iterative process results in eight relevant information types, which we list in the survey as previously mentioned.

Analysis of The Open Answers. For the open answers in the survey, we conduct a two-stage process comprising Descriptive and Pattern coding analysis to name them [13]. Specifically, in a first step two authors shortly summarize the topic of each passage from the open answers; then, they identify explanatory codes to create themes that are finally discussed among all the authors. For example, in case of the challenges, with this process we end up with nine additional challenges grouped into two categories that indicate the point in time in which they occur, i.e., (1) *Test case design* and (2) *Flaky test fixing.*

Attracting Participants. We advertised the online survey using the personal social network accounts of the authors (i.e., FACE-BOOK, TWITTER, and LINKEDIN), through private contacts, and also published on practitioners blogs (e.g., REDDIT). To stimulate the participation, we allow the participants to indicate a non-profit organization of their choice to which we donate 2 USD.

Respondents. As a result, we collect 188 answers: we filter out 68 of them because they are partial and, thus, we only consider 121 full responses. Among these respondents, 75% have more than 6 years of programming experience (median 10) and two years of

industrial programming as well as testing experience (median 5). Moreover, 45% respondents come from large companies having more than 250 employees. From these basic statistics, our sample is composed of developers who are experienced with (industrial) programming and tests, thus we consider the collected responses as valuable and valid for our research questions.

3 RQ_1 – NATURE, ORIGIN, AND FIXING

The first research question aims at describing the possible types of flakiness, as well as their origin, fixing effort and strategies.

The categories are reported by frequency of nature and, within those, by frequency of fixing strategies. Newly reported categories (i.e., natures of flakiness that were not included in the taxonomy proposed by Luo et al. [24]) are underlined and marked with a '*'.

Concurrency. A test that is flaky due to synchronization issues originating from multiple threads interacting in an unsafe manner is classified in this category. For example, a race condition occurs due to threads relying on an implicit ordering of the execution leading to a deadlock in certain situations. As observed by Luo et al. [24], this is a high diffused cause of flakiness. In our dataset, it is present in 61 cases (26%). Moreover, the median effort required to fix this problem—as assigned by developers—is 4 (high), indicating that it is problematic and further justifying the amount of research effort done so far to define automatic techniques able to solve it [26, 27]. Interestingly, not all flaky tests in this category origin in test code: indeed, the developers report that in 34% of the cases the fixing process requires the examination of the production code and not of the test. Thus, *test flakiness can be originated by the production code*; besides motivating further research on the origin of flaky tests, this finding indicates that automated approaches aimed at fixing this problem must consider whether the flakiness originates in test or production code. As a final note, there seem to be one key fixing strategy, namely the addition of a `waitFor` statement, while other operations, like the modification of the concurrency guard condition or the addition of `lock` statement to avoid other threads running at the same time represent less frequent strategies to remove this flakiness. In 5% of the cases, developers act by refactoring production code to remove concurrency, thus making the code deterministic. Finally, it is also interesting to observe that in 2% of the cases, developers either preferred to rewrite the test ot completely disable it.

Async Wait. This flakiness is characterized by a test performing asynchronous calls without waiting properly for their result to become available. As opposed to the `Concurrency` category presented above, which related to *local* synchronization issues, `Async Wait` refers to *remote* resources being unavailable. For example, a test performing a request to a remote server and continuing the execution without ensuring the requested resource is available results in the test failing or passing depending on how quickly the resource becomes available. We confirm the results by Luo et al. [24] on the high diffusion of this category, as it affects 52 of the validated tests (22%). Moreover, the median effort is 3 (medium). The origin of the flakiness for all these tests is in test code. As for the fixing strategy, in most of the cases (86%) developers fix it by means of a `waitFor` statement addition.

Table 1: Taxonomy of the nature of flaky tests with the corresponding fixing strategies, by frequency (N = 234, because developers were allowed to assign more than one nature to each of the 200 flaky tests they analyzed). The '*' and underlining indicates newly reported categories (i.e., they were not included in the taxonomy proposed by Luo et al. [24]).

Nature: Concurrency	Cases: 61 | Avg. Fixing Effort: 4.0 | Origin: 66% Test Code	
Fixing Strategies:		
Name	Description	Frequency
Wait For Addition	An `await` or `waitFor` promise is added to wait for the required event.	46%
Change Concurrency Guard Conditions	Modification of the constraints that may block the applicability of the clause under some conditions.	26%
Adding Lock	New locks are introduced to ensure mutual exclusion between threads.	21%
Make Code Deterministic	The concurrency is removed.	5%
Test Replacement	The flaky test is replaced with a brand new test.	1%
Disable Test	The flaky test is disabled and no longer executed.	1%

Nature: Async Wait	Cases: 52 | Avg. Fixing Effort: 3.0 | Origin: 100% Test Code	
Wait For Addition	An `await` or `waitFor` promise is added to wait for the required event.	86%
Reordering Threads Execution	The source code is reorganized or refactored to make the execution deterministic or less async.	13%
Disable Test	The flaky test is disabled and no longer executed.	1%

Nature:	Cases: 40 | Avg. Fixing Effort: 1.0 | Origin: 100% Test Code	
Fix Assertion	The cause leading to the failure is diagnosed and fixed.	45%
Adjust Fuzzing	The `assert` statement is modified so that it provides less extreme values or allow larger difference when comparing expected and actual outcome.	20%
Disable Test	The flaky test is disabled and no longer executed.	16%
Missing Code Added	The code needed to check whether a certain condition is met is added.	1%

Nature: Test Order Dependency	Cases: 22 | Avg. Fixing Effort: 2.0 | Origin: 100% Test Code	
Remove Dependency	Changes aimed at (i) modifying the output directory for the test to use a separate directory or (ii) checking instance variables before that the test is accessed and executed.	100%

Nature: *Test Case Timeout	Cases: 18 | Avg. Fixing Effort: 4.0 | Origin: 100% Test Code	
Increase Timeout	The timeout time is increased to avoid the flakiness behavior.	85%
Skip Non-Initialized Part	Code added to skip non-initialized parts to make the test run faster and not timeout.	5%
Split Test	Split up all tests (run in parallel) into more groups to reduce risk of a timeout.	5%
Disable Test	The flaky test is disabled and no longer executed.	5%

Nature: Resource Leak	Cases: 14 | Avg. Fixing Effort: 3.0 | Origin: 85% Test Code	
Destroy Object	New code is added so that the conflicting object is destroyed before continuing the execution.	50%
Test Replacement	The flaky test is replaced with a brand new test.	36%
Disable Test	The flaky test is disabled and no longer executed.	14%

Nature: *Platform Dependency	Cases: 10 | Avg. Fixing Effort: 4.0 | Origin: 90% Test Code	
Run Additional Tests	New platform-specific tests are ran instead of the flaky one.	60%
Correct Directories	The test suite is modified so that it includes the correct platforms to run the tests on or to fix a directory issue.	40%

Nature: Float Precision	Cases: 6 | Avg. Fixing Effort: 4.0 | Origin: 100% Test Code	
Subtract Bytecode Offset	The floating point is corrected by subtracting the bytecode offset so that the precision is preserved.	100%

Nature: *Test Suite Timeout	Cases: 4 | Avg. Fixing Effort: 3.5 | Origin: 100% Test Code	
Skip Non-Initialized Part	Code added to skip non-initialized parts to make the test run faster and not timeout.	75%
Split Test Suite	Split up all tests (run in parallel) into more groups to reduce risk of a timeout.	25%

Nature: Time	Cases: 4 | Avg. Fixing Effort: 1.0 | Origin: 100% Test Code	
Disable Test	The flaky test is disabled and no longer executed.	75%
Preserve Time Precision	The test case is modified so that the time precision is preserved.	25%

Nature: Randomness	Cases: 3 | Avg. Fixing Effort: 1.0 | Origin: 100% Test Code	
Replaced Math.random	The `Math.random` call is replaced with more reliable random number generator.	100%

*Too Restrictive Range. This is the first category that was not originally included in the catalog by Luo et al. [24]. Some of the valid output values are outside the assertion range considered at test design time, so the test fails when they show up. Such test cases have a range of predefined values for which the test is allowed to pass. If this range is defined too restrictively, tests may start failing in a not deterministic way. This type of flakiness additionally includes failures originating from improperly placed

Moritz Eck,[1] Fabio Palomba,[1] Marco Castelluccio,[2] Alberto Bacchelli[1]

assertion statements causing the test to pass or fail independently of the test execution. In our dataset, 17% of the flaky tests belong to this category, indicating that this problem is not uncommon. On the other hand, developers find this type of problem easy to fix (median effort=1) as it is always located in test code and only requires the proper definition of a range of values: indeed, in 65% of the cases, developers address this problem by fixing or adjusting the range of values the assertion refers to. Interestingly, however, 16% of the times the corresponding tests were disabled: likely, this is due to the limited time developers have to spend in understanding the correct range to use within the assertion and, thus, they prefer to keep having a green test suite.

Test Order Dependency. The class is characterized by the result of the test run depending on the execution order of the tests. This occurs mostly due to tests not properly cleaning up after themselves (e.g., restoring common states and shared variables) or failing to setup the necessary preconditions (e.g., a test environment state) before starting to run [24]. 11% of the flaky tests are due to this reason, confirming its diffusion [30]. This is, by nature, a problem raising in test code. As for the effort, the median is equal to 2 (low)—thus, quite easy to fix for developers through the (i) modification of the output directory - so that the test uses a separate one, avoiding conflicts or (ii) checking the instance variables before the test is executed.

*Test Case Timeout. Flaky tests experiencing non-deterministic timeouts related to a single test belong to this category. It is comparable to the Test Suite Timeout (reported later), with the difference that the size of a single test grew over time without adjusting the max runtime value. Various reasons (e.g., failing to download prerequisites or a test not producing output for a fixed amount of time, which then is killed by the execution system assuming that the test stalled) can lead to the non-deterministic outcome. In our dataset, this flakiness type appears in 18 tests (8%). Moreover, it is associated with a high effort to fix: despite it might seem that this problem would be only concerned with the increasing of the timeout, practitioners explained that finding the right timeout is unexpectedly hard. The increase of the timeout time is, obviously, the most frequent solution to this type of flakiness (85%), however other fixing strategies such as (i) the addition of code that make the test faster or (ii) the extraction of a new test to reduce the risk of timeout are also sometimes taken into account.

Resource Leak. Improper management of a external resources (e.g., failing to release previously allocated memory or dereference a pointer) characterize this category. Additionally, this includes test failing due to garbage collection processes removing parts of the test execution environment or required resource (e.g., a file stored in memory the test is reading multiple times) [24]. This category represents the root-cause of 7% of the flaky tests, and has a medium effort equals to 3. As for the origin, developers report that 15% of them are due to production code issues: thus, we confirm that test flakiness is not just a problem of tests. With respect to the possible fixing strategies, in 50% of the cases the test is modified so that the conflicting object is destroyed before continuing the execution. However, in the remaining cases, developers prefer to replace the test entirely or disable it: this result

seems to confirm that dealing with test flakiness can be annoying for developers and that, in several cases, they prefer to skip the problem rather than find a solution to deal with it.

*Platform Dependency. The Platform Dependency flakiness is: Non-deterministic test failures occurring only on specific platforms (e.g., a test only failing in debug builds or on 32-bit Windows 7 systems). While one can claim that a test failing consistently on a platform represents a permanent failure, developers consider it as flaky. If an organization employs cloud servers for builds/tests, machines with different characteristics (e.g., Linux kernel version) could be provisioned according to available resources, so the test fails intermittently. This type of flakiness occurs due to various reasons, such as a platform being unusually slow or missing preconditions (e.g., the test failed to setup or download a required resource). They appear 10 times in our dataset and originate mostly in the test code (90%), since they would most likely become permanent if they originated from a bug in the production code (as the tests would not run independently of what platform they are run on). The effort associated is 4 (high), as it turns to an infrastructure problem to be diagnosed and fixed. The high effort is also justified by the fixing strategies applied: indeed, 60% of the times brand new platform-dependent tests are added, while in 40% of the cases developers directly correct the existing code to avoid this flakiness.

Float Precision. As previously observed by Luo et al. [24], floating point operations may lead to non-deterministic test failures if potential precision over- and underflows are not considered (e.g., converting a float formatted as a string, rounding or cutting to a certain number of significant digits). Additionally, this includes tests failures occurring due to different number of significant digits being reported in different executions. Even if the diffusion of this flakiness type is pretty low (only 6 cases), the reported effort is high, which indicates that developers might spend considerable time in dealing with this type of flaky tests. The fixing operation associated with this flakiness relates to the modification of the test to preserve precision.

*Test Suite Timeout. Flaky tests originating because of the test suite non-deterministically timing out are classified as Test Suite Timeout flakiness. Test suites grow over time and the max runtime value is not always adjusted accordingly, leading to the test suites passing or failing depending on different random variables (e.g., the network congestion, the execution speed of the platform or the type of build). Note that in a `Test Suite Timeout`, no single test is the cause of flakiness, rather the whole execution; this makes this cause different from `Test Case Timeout`, whose cause is a specific test. This flakiness cause appears very rarely (only 4 cases in our dataset) and the developers assigned a median effort score of 3.5. As this is test issue-related problem, its origin is within test files. To fix the flakiness, developers frequently decide to take actions able to fasten the execution of the test suite (75% of the cases), while the remaining 25% of times they perform an Extract Class refactoring [17] to make the resulting suites more independent and faster.

Time. This cause of flakiness is characterized by tests relying on the executing system local time. For example, a test may fail due to the local system changing its day, i.e., reporting a different day

in the next iteration, or failing to take the timezone into account when comparing two timestamps. In line with Luo et al. [24], this kind of flakiness is generally not diffused; at the same time, we observe that it is always related to test code and is very easy to fix: the perceived easiness is due to the fact that in 75% of the cases developers just disable the test, and thus only in a few cases (25%) they actually try to preserve the time precision.

Randomness. The generation of random numbers may lead to non-deterministic failures if not all possible values generated including edge cases are considered (e.g., the generation of zero when any number greater than zero was expected or the random number is used in a mathematical operation). We confirm the results of previous work [24] on the very low diffusion of this flakiness cause. The associated effort is very low, the problem only appears in test files, and is always fixed by replacing the used random library call with a reliable number generator.

In our dataset, we do not find evidence of the presence of three out of the ten categories in the original taxonomy by Luo et al. [24], namely `Network`, `I/O`, and `Unordered Collections`. On the one hand, this may be due to the type of analysis we do in this study: instead of classifying ourselves the likely causes behind test flakiness, we prefer asking to the original developers the type of flakiness occurring in the tests they fixed: in this way, we rely on the expertise of people that actually dealt with the investigated problems. On the other hand, the missing observation of those three categories might be due to the nature of the system we exploit, i.e., the flakiness might manifest itself in different manners depending on the system taken into account. Our findings reveal the need for further research on the naturalness of flaky tests, where the role played by additional factors like project and organizational domains are assessed. At the same time, our taxonomy does not exclude the original one [24], rather complements it with additional unrevealed natures of test flakiness to be further studied.

> **Finding 1**. *We confirm the existence and frequencies of seven flakiness types revealed by Luo et al. [24]. We discover four additional categories, three of which developers consider as the most effort-prone types of flakiness to deal with. Finally, we present the fixing strategies developers put in place to deal with flakiness and provided evidence that flaky tests can be also caused by problems in production code.*

4 RQ$_2$: THE RELEVANCE OF FLAKY TESTS

Our second research question aims at understanding flaky tests prominence and how problematic they are according to developers. Figure 1 shows the survey respondents' answers in terms of how frequently they deal with flaky tests and how problematic these tests are for those who deal with them at least a few times a year.

The vast majority (109 respondents) encounter the problem at least some times a year, and 58% at least every month. Among these 109 respondents, 79% find flaky tests to be at a moderate to serious problem, while 21% a minor to not a problem, despite having to deal with them at least a few times per year. The analysis of the open text comments received by the participants about the top problems
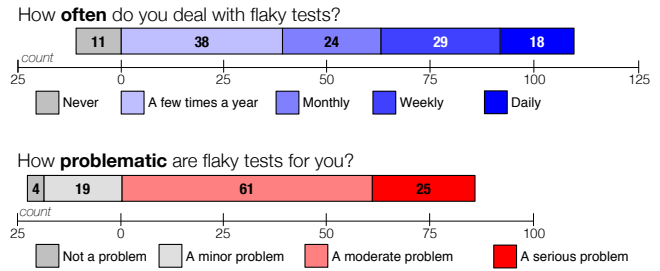
Figure 1: Frequency and relevance of the problem according to the respondents to our online survey.

they have with flaky tests reveal the presence of issues that can be grouped in three major themes:

Scheduling: According to the opinions of 92 developers (77%), flaky tests are *time consuming* since reproducing the test failure is not easy and not always guaranteed to be possible. The debugging process includes multiple reruns of the same test while varying the level of log output and context taken into account. Additionally, since flaky tests fail intermittently, their priority is often lower than those of permanent failures, i.e., developers need to find the time and request manager's permission to care for the flaky tests and the resources for the refactoring effort.

Test Suite Reliability: Once a test becomes flaky, 88 developers (73%) report that *the test is no longer fully reliable*. Thus, developers start to trust the test output less and, therefore, may start disregarding it, potentially leading to ignoring an actual failure. Moreover, the unexpectedness of flaky tests is an additional issue since it is unclear which tests will start to fail intermittently next.

Developer Allocation: 85 developers (71%) mention that it is important to keep in mind that some flaky tests are more likely to occur (e.g., tests relying on external dependencies) than others and, therefore, to distinguish how much attention shall be directed toward a specific type of failure. Since flaky tests are often intertwined with other tests, they require a certain *level of knowledge to be able to fix them* and, thus, intermittent tests may lead to problems in the allocation of the available resources.

> **Finding 2**. *The collected developers' opinions indicate that flaky tests are rather frequent and a non-negligible problem, with possibly important consequences on resource allocation and scheduling, as well as on the reliability of the test suite.*

5 RQ$_3$: THE CHALLENGES OF FLAKY TESTS

To answer what are the main challenges developers face when dealing with test flakiness, we further analyze the results of our online survey. When quoting developers, we refer to them by their ID, which corresponds to the incremental number assigned by the survey creation system.

Table 2 lists the eight pieces of information that emerged as potentially relevant for fixing a flaky behavior of a test from the

Table 2: Pieces of information for fixing a flakiness (emerged from the multivocal literature review, Section 2.4), by their importance (0–3) and difficulty (0–3) in obtaining, as rated by the survey respondents.

Information on the flaky test	Importance for fixing	Difficulty in obtaining
C_f - The context leading to failure	2.69	1.65
N - The nature of the flakiness	2.40	1.71
O - The origin of the flakiness	2.22	1.17
E - The involved code elements	2.21	1.13
C - The changes to perform the fix	2.08	1.59
C_p - The context leading to passing	1.95	1.20
Co - The commit introducing the flakiness	1.89	0.75
H - The history of this test's flakiness (previous causes and fixes)	1.79	0.83

multivocal literature review (Section 2.4). As explained in the methodology, we asked participants to rate these pieces of information in terms of importance for fixing and difficulty in obtaining. Therefore, the rows are sorted according to the weighted average importance[2] according to our survey's respondents; the table also details the reported weighted average difficulty.[3] In this way, we identify pieces of information that developers actually perceive as challenging to obtain, thus indicating potential pain points to address with future research or better practices.

C_f) **Failing Context.** The first important and challenging piece of information to obtain is understanding what is the context that leads to the failing behavior. As P_{161} puts it: "the most difficult operation is reproducing a flaky test, as sometimes only 1/20 fails". Additionally, P_{174} reports: "our UI tests are fairly slow, and the only real way to verify if the flakiness was resolved is to re-run the tests several times in the CI environment". In other words, the verification of the failing behavior is even more complicated by some co-factors, such as the slowness of tests and the environment in which they are run.

N) Nature. The second major challenge to the fixing process is the timely identification of the nature of the flakiness affecting a test. Understanding which of the causes discussed in **RQ$_2$** and in previous work [24] is the one leading to a flaky test represents a critical challenge for developers. Indeed, 52% of the survey participants consider this information as 'extremely important' and 37% of them as 'moderately important'. Similarly, 68% of the developers report that figuring out the flakiness root-cause is 'difficult' to 'very difficult'. For instance, P_{166} explains: "a big challenge is to detect the root cause that leads to a flaky test. You need to check concurrency issues or cache related problems that might be common causes of flakiness". This finding supports the need for empirical analyses aimed at further studying the phenomenon of test flakiness and the way it manifests itself.

---

[2]We assigned a value of 0 (minimum) to answers stating 'Not important at all' and of 3 (maximum) to answers stating 'Extremely important', and 1 or 2 to the values in between.
[3]We assigned 0 (minimum) to both the values 'Very easy' and 'Easy', and 3 (maximum) to 'Very difficult', and 1 or 2 to the values in between.

E) Involved Code Elements. Detecting the code elements involved in the flakiness is valuable information to obtain for 83% of the developers, but most of them (75%) consider it as not hard to gather. Therefore, this does not seem to be a major pain point, as the information is at hand by looking at the test code and log files. Nevertheless, there are still developers who consider this as major difficulty in the whole fixing process; looking more in depth into the reported opinions, we discover that the main factor characterizing the easiness of such information retrieval is how complex the source code actually is. As P_{166} puts it: "I think that for example finding the involved code might be rather easy if your code is well designed and written or might proof to be a complete nightmare if your codebase is a mess". This may indicate that keeping test code quality high may naturally ease the flaky test fixing process.

O) Origin. The final potential challenge based on the information needs we identified in literature is whether the flakiness is caused by the test or by the production code. As shown in **RQ$_2$**, the flakiness can be even caused by problems in the production code and, apparently, having this information is important to speed-up the diagnosing of flaky tests. Indeed, 79% of the participants evaluated this challenge as 'moderately important' to 'extremely important'. However, in this case, most of the developers (66%) consider it pretty easy to obtain.

C_p,Co,H) Passing Context, Commit, and History. Finally, the context that leads to the passing behavior is considered only slightly or moderately important in 69% of the cases because developers mainly aim at identifying the cases where behavior of the test leads to a failure. Similarly, the past flakiness history of a test seems to be poorly relevant during the fixing process for 72% of the developers: This is likely due to the tests being subject to different flakiness types, meaning that the information from past history cannot be effectively reused to solve the current cause.

Table 3: Futher challenges due to flaky tests, as reported.

Category	Challenge	# of mentions
Design	Mocking Dependencies	6
	Keeping Tests Decoupled	6
	Reliance on External Dependencies	3
	Too Much Setup Code	3
	Common Coding Style / Etiquette	1
Fixing	Uncertainty of Changes Fixing The Test	7
	Not Detailed Enough Log	4
	Lack of Insight Into The System	4
	Restructuring The Tests	2

Table 3 presents the additional challenges mentioned by the developers in the survey, by their reporting frequency. Overall, the first observation is related to the development phases mentioned by the survey participants. While we expect to find critical challenges arising during the diagnosing and fixing process, the fact that developers report the presence of challenges at design-time is somehow surprising and unexpected. OLur findings indicate how the problem of flakiness is *spread over the entire life-cycle of tests.*

Design. According to our participants, the probability for a test to be flaky can be reduced if good design principles are applied while

developing it, thus confirming previous findings in the field [31]. Six developers report that keeping tests decoupled and mocking dependencies are major challenges when designing test to be less prone to flakiness. In the former case, keeping low the coupling between tests is a key property because there is the risk that, quoting P_{130}, "one test sets something up that affects another test, and so on". In the second case, understanding when it is convenient to mock external objects can be useful to "control the environment (making sure packages installed are all the right version, mocking time, mocking/stubbing out dependencies on external services)" [P_{80}]. Additional challenges are represented by (i) designing systems that do not rely too much on external dependencies, to reduce the risk of intermittent failures due to not controllable sources; (ii) reducing the amount of setup code required, so that flakiness eventually due to problems with the setup of tests can be more easily diagnosed; and (iii) establishing a common coding style and etiquette, to ease the understanding of test code.

Fixing. Seven participants report as a main challenge the uncertainty of not knowing whether a code change actually fixes the flakiness (it may disappear by chance). This is currently solved by manually re-running multiple times the supposedly fixed flaky tests, but it is not considered as an optimal solution. Finally, log files with insufficient levels of detail and a lack of insights into the system are also reported as a major challenge, as well as the process of restructuring the test to remedy the flakiness.

> **Finding 3**. *Reproducing the context leading to the test failure and understanding the nature of the flakiness are the most important, yet challenging needs according to developers. Moreover, designing test code properly to avoid flakiness emerged as an additional challenge not mentioned in the reviewed academic and gray literature.*

6 DISCUSSION AND IMPLICATIONS

Our results highlighted a number of points to be further discussed and several implications for the research community:

Flaky tests as a relevant research problem. Besides confirming seven of the root-causes behind flaky tests originally defined by Luo et al. [24], our work let emerge the existence of four additional types of flakiness previously unknown, i.e., Too Restrictive Range, Test Case Timeout, Platform Dependency, and Test Suite Timeout. We found them to be highly diffused over our dataset, and the latter three causes are those that developers associated with a higher fixing effort. This finding calls for further research in this area to both understand *how* such flaky tests are introduced and managed, as well as devise automatic solutions to locate and fix them. The worrisome finding that, in a not negligible amount of cases, developers prefer to just disable flaky tests rather than properly deal with them is a urgent call for researchers to find solutions to help practitioners dealing with developers' information needs (as emerged from this study) to reduce their efforts.

An organizational view. The presence of intermittent test cases leads to several undesired problems and consequences connected to organizational aspects (e.g., resource allocation issues).

This finding highlights the current inability of development teams to properly manage flaky tests; thus, we argue that a broader view of test flakiness should not only involve source code and its management, but also how flaky tests impact teams' organization and how their resolution process may be optimized. For instance, research can be conducted to investigate approaches able to recommend *who* should fix a flaky test based on developers' knowledge/experience, flakiness type, and the available time.

Preventing Flaky Tests. As test flakiness is problematic, it is of the greatest importance the definition of methods able to *prevent* the introduction of flaky tests. We envision the adoption of machine learning techniques—properly trained on the features characterizing each flakiness type—to be a promising way to help developers in preventively spotting tests that have the highest risk to become flaky in the future.

Design for Testability. Our participants revealed that finding how to design a test code to avoid flakiness is an important challenge to face. This motivates the growing research area around test code quality [18, 29, 33–35] and provides two promising directions that the research community can focus on: (i) the definition of a set of *design patterns* that can support the creation of deterministic tests; (ii) the definition of a set of flakiness-related *anti-patterns* that practitioners should avoid when writing test cases. While some initial steps have been done about the relation between test smells and flaky tests [31, 32], further investigation is necessary.

Flaky Test Prioritization. The results on the effort required to fix each flaky test type, as well as their perceived importance, indicated that not all flaky tests are equally problematic. As large codebases might contain a higher number of tests suffering flakiness issues, approaches that can rank them by exploiting effort-related information would help developers in maintaining high quality test suites. Machine learning is a promising approach to exploit to accomplish this task, as it could spot patterns in the code that are the fingerprints of certain types of flakiness.

7 THREATS TO VALIDITY

This section discusses the threats that might have influenced our findings, despite our mitigation strategies.

Construct Validity. Threats in this category are mainly concerned to the way we built the dataset of flaky tests (RQ_1), the relevance of the problem (RQ_2), and the potential challenges (RQ_3). To obtain a reliable analysis of the flaky test types, we directly involved the professional MOZILLA developers and asked them to analyze the flakiness cases they fixed themselves. Although we considered flaky tests they fixed in a recent time and provided them with the links to their own patches as well as issue reports, we cannot exclude that they could not recall the required fixing effort precisely. To collect information about the frequency and relevance of flakiness for developers, we used an online survey. Although we spread the survey through several channels, our survey responses may suffer from a self-selection or voluntary response bias: People who volunteered to respond may be more involved with flakiness than the average developer. To mitigate this bias, we introduced a donation-based incentive of 2 USD to a charity per valid respondent. Although our results concerning the frequency are aligned with those reported by Luo et al. [24], it is still possible that we have

Moritz Eck,[1] Fabio Palomba,[1] Marco Castelluccio,[2] Alberto Bacchelli[1]

more extreme results about flakiness frequency and harmfulness than what would be in the average practice. Finally, besides inquiring developers on the major challenges they face when dealing with test flakiness, in **RQ**$_3$ we came up with a set of information needs anecdotally considered relevant by practitioners. We did this by surveying both white and gray literature following the validated guidelines by Garousi et al. [19]. However, we cannot exclude that other sources presented further potential information needs and challenges when dealing with test flakiness.

Conclusion Validity. As for the analysis methods exploited in our study, in **RQ**$_1$ we computed statistics on the answers provided by the participants of our survey. In the context of **RQ**$_2$ we defined a taxonomy of flaky test types by relying on an iterative content analysis that involved three software engineering researchers having experience with flaky tests. Although such iterative process is designed to mitigate the possibility of classification errors, subjective judgement may have played a role in the elaboration of the themes. A similar process has been done in the context of **RQ**$_3$ to classify the challenges emerged from the multivocal literature review. We cannot exclude human errors or that some important challenges, not present in the sources examined, were not reported in the survey. Further replications and analyses would be beneficial to corroborate our findings.

External validity. Threats related to the generalizability of our findings concern size and nature of the dataset exploited. We considered 200 flaky tests coming from Mozilla. Although the company has been the subject of several software engineering studies, which later also generalized to other contexts, the taxonomy we created may (and may not) be applicable to other systems and, therefore, further studies are needed to investigate this aspect.

8 RELATED WORK

A number of researchers and practitioners reported that some flaky tests can be a serious problem in automated regression testing [10, 16, 24, 26, 27, 31, 39]. Memon and Cohen [26] described multiple negative effects that flaky tests can create, discovering that their occurrence may even lead to features not being included in a release as they were not able to be tested sufficiently beforehand [26]. Marinescu et al. [25] and Hilton et al. [20] 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. [24] 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. While the taxonomy of flaky test root causes built in the context of **RQ**$_1$ is mostly aligned with the one presented by Luo et al. [24], our study reveals the existence of additional causes for flakiness. In this sense, our paper can be seen as an additional source aimed at better understanding the phenomenon of flaky tests.

Also other researchers investigated the motivations behind flaky tests as well as devised strategies for their automatic identification. For instance, Palomba and Zaidman [31, 32] discovered that test

smells [36] may induce test code flakiness; moreover, they found that the removal of such smells through refactoring also induce fixing flaky tests. Zhang et al. [39] focused instead on test suites affected by test dependency issues, reporting methodologies to identify these tests. Muslu et al. [27] found that test isolation may help in fault localization, while Bell and Kaiser [10] proposed a technique able to isolate test cases in Java applications by tracking the shared objects in memory. Bell et al. [11] proposed DeFlaker, an automated technique that identifies flaky tests by running a mix of static and dynamic analyses.

Another well-studied root cause of flaky test is concurrency. Farchi et al. [14] identified a set of common erroneous patterns in concurrent code, while Lu et al. [23] reported a comprehensive study into the characteristics of concurrency bugs. Jin et al. [21] devised a technique for automatically fixing concurrency bugs by analyzing the single-variable atomicity violations. With respect to the studies discussed above, our work can be considered as complementary as it provides, for the first time, an in-depth investigation of the developer's perspective, which includes the nature of flaky tests and fixing efforts, the relevance of the problem, and the most important challenges that face when handling flaky tests.

9 CONCLUSION

We presented an empirical study aimed at improving our scientific knowledge around the problem of test flakiness. We first assembled a dataset of 200 flaky tests, asking the original developers that fixed the problem to analyze them in terms of nature, fixing effort, and origin of the flakiness. Then, we used this annotated dataset (which we also make publicly available) to define a taxonomy of flaky test types, which comprises seven categories already known in literature [24] as well as four additional ones; out of these four, we found that three are the most demanding overall in terms of fixing efforts. In parallel, we performed a survey study having the goal of collecting developers' opinions on the relevance of flakiness in practice, as well as major challenges and information needs they have when dealing with flaky tests. This study revealed a set of problems and needs that the research community should carefully look at as to improve the way test flakiness is managed by practitioners.

(1) Evidence on four new categories of flaky tests—important for both researchers (called to devise techniques to deal with these new types) and practitioners (called to fix these issues);

(2) First reported evidence that flakiness is also caused by problems in production code;

(3) Surprising evidence from our analyses on the effort required to fix flaky tests, like the timeout problem;

(4) Evidence on unexpected side-effects of flaky tests, such as the issues they induce to organizational aspects (e.g., resources allocation). This gives insights to practitioners to reflect on their practices with flaky tests.

The outputs of our study represent the main input for our future research agenda, which is mainly oriented to the addressing of the challenges identified as well as the corroboration of our analyses on different projects and developers. It is our hope that the insights we have discovered lead to more improved techniques and tools, based on research, to aid developers handle flaky tests.

REFERENCES

[1] 2019. SurveyGizmo. https://www.surveygizmo.com.

[2] 2018. 2018. ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. http://www.esec-fse.org

[3] 2018. 2018. ACM SIGSOFT International Symposium on Software Testing and Analysis. https://conf.researchr.org/series/issta

[4] 2018. 2018. ACM Transactions on Software Engineering and Methodology. https://tosem.acm.org

[5] 2018. 2018. IEEE TCSE International Conference on Software Maintenance and Evolution. http://conferences.computer.org/icsm/

[6] 2018. 2018. IEEE Transactions on Software Engineering. https://www.computer.org/web/tse

[7] 2018. 2018. IEEE/ACM International Conference on Software Engineering. http://www.icse-conferences.org

[8] 2018. 2018. International Conference on Software Testing. https://www.es.mdh.se/icst2018/

[9] 2018. 2018. Springer's Empirical Software Engineering Journal. https://link.springer.com/journal/10664

[10] Jonathan Bell and Gail Kaiser. 2014. Unit Test Virtualization with VMVM. In *Proceedings of the International Conference on Software Engineering (ICSE)*. ACM, 550–561. https://doi.org/10.1145/2568225.2568248

[11] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*. To Appear.

[12] blinded authors. 2018. Data and materials. https://tinyurl.com/yaem95a5.

[13] Juliet M Corbin and Anselm Strauss. 1990. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology* 13, 1 (1990), 3–21.

[14] E. Farchi, Y. Nir, and S. Ur. 2003. Concurrent bug patterns and how to test them. In *Proceedings International Parallel and Distributed Processing Symposium*. 7 pp.–. https://doi.org/10.1109/IPDPS.2003.1213511

[15] Timothy S Flanigan, Emily McFarlane, and Sarah Cook. 2008. Conducting survey research among physicians and other medical professionals: a review of current literature. In *Proceedings of the Survey Research Methods Section, American Statistical Association*, Vol. 1. 4136–47.

[16] Martin Fowler. [n. d.]. Eradicating non-determinism in tests. https://martinfowler.com/articles/nonDeterminism.html

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

[18] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[19] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2016. 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, 26.

[20] Michael Hilton, Jonathan Bell, and Darko Marinov. 2018. A Large-Scale, Longitudinal Study of Test Coverage Evolution. In *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2018)*. http://jonbell.net/publications/coverage

[21] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-violation Fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 389–400. https://doi.org/10.1145/1993498.1993544

[22] R Burke Johnson and Anthony J Onwuegbuzie. 2004. Mixed methods research: A research paradigm whose time has come. *Educational researcher* 33, 7 (2004), 14–26.

[23] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. 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, 329–339. https://doi.org/10.1145/1346281.1346323

[24] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 643–653. https://doi.org/10.1145/2635868.2635920

[25] Paul Marinescu, Petr Hosek, and Cristian Cadar. 2014. 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, 93–104. https://doi.org/10.1145/2610384.2610419

[26] Atif M. Memon and Myra B. Cohen. 2013. Automated Testing of GUI Applications: Models, Tools, and Controlling Flakiness. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1479–1480.

[27] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. 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, 496–499. https://doi.org/10.1145/2025113.2025202

[28] A. N. Oppenheim. 1992. *Questionnaire Design, Interviewing and Attitude Measurement.* Pinter Publishers.

[29] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2016. Automatic test case generation: What if test code quality matters?. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 130–141.

[30] Fabio Palomba and Andy Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*. 1–12. https://doi.org/10.1109/ICSME.2017.12

[31] Fabio Palomba and Andy Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 1–12.

[32] Fabio Palomba and Andy Zaidman. 2019. The smell of fear: On the relation between test smells and flaky tests. *Journal of Empirical Software Engineering* (2019).

[33] Fabio Palomba, Andy Zaidman, and AD Lucia. 2018. Automatic test smell detection using information retrieval techniques. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

[34] Davide Spadini, Maurício Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To Mock or Not To Mock? An Empirical Study on Mocking Practices. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 402–412.

[35] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the relation of test smells to software code quality. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.

[36] Arie van Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. 2001. Refactoring Test Code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*. 92–95.

[37] Marilyn Domas White and Emily E Marsh. 2006. Content analysis: A flexible methodology. *Library trends* 55, 1 (2006), 22–45.

[38] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. ACM, 38.

[39] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muslu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 385–396. https://doi.org/10.1145/2610384.2610404