MASTER'S THESIS 2020

Mitigation and handling of non-deterministic tests in automatic regression testing

Axel Berglund, Oskar Vateman

ISSN 1650-2884
LU-CS-EX: 2020-54

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-54

Mitigation and handling of non-deterministic tests in automatic regression testing

Axel Berglund, Oskar Vateman

Mitigation and handling of non-deterministic tests in automatic regression testing

Axel Berglund
`ine15abe@student.lu.se`

Oskar Vateman
`ine15ova@student.lu.se`

September 1, 2020

Master's thesis work carried out at Axis Communications AB.

Supervisors: Helena Razdar, `helena.razdar@axis.com`
Per Runeson, `per.runeson@cs.lth.se`

Examiner: Emelie Engström, `emelie.engstrom@cs.lth.se`

Abstract

Iterative development is a principle of agile development, meaning that the steps of software development are performed over and over again in iterations. Performing these steps multiple times means that testing must also be done multiple times, which leads to the automation of these tests being necessary.

A flaky test, also called a non-deterministic test, is a test which can both pass and fail using the same version of the code. This non-determinism leads to testers not trusting their tests. Therefore our purpose with this thesis was to find out how to minimise flakiness in a test suite. To do so we used an iterative research approach, where different potential solutions to a problem were implemented and evaluated in iterations until we arrived at a working solution.

During the first iteration the approach was to attempt to fix flaky tests in a test suite and from this identify common underlying causes as well as solutions to the identified common causes. From this approach it was discovered that fixing flaky tests and identifying their causes is difficult and time consuming.

For the second iteration another solution was evaluated, which consisted of performing a cost-benefit analysis on different strategies of handling flaky tests. This analysis was done by conducting interviews with Axis employees and by using information from relevant literature.

The third iteration consisted of developing guidelines, based on information from the interviews as well as information from the literature, on how to avoid flaky tests, how to handle flaky tests, and how to mitigate flakiness during test case implementation. The guidelines were found to be effective in minimising flakiness through validation interviews with testers and experts at Axis Communications.

Keywords: non-determinism, flaky tests, regression testing, flakiness strategies, cost-benefit analysis

Acknowledgements

We would like to thank Axis Communications and especially the New Business department for welcoming us to your office and overall being very helpful with any problems we had.

The research in this thesis would not be possible without the input from the interviews we conducted. We would therefore like to extend our vast gratitude to all interviewees who invested their time in making this thesis a reality.

We would also like to thank Baldvin Gislason Bern for bringing us the expert perspective on flakiness and the intricacies of automated regression testing.

A special thanks to Per Runeson, our supervisor at LTH, for being extremely helpful with everything concerning the work and the report, including feedback and support. And also a special thanks to Helena Razdar, our supervisor at Axis, for being with us every step of the way.

Contents

Chapter 1
Introduction

This section aims to provide a quick introduction to the thesis and its' purpose, the context of the thesis as well as to the topic of flaky tests.

1.1 Background

Agile development is an approach to developing software which focuses on finding solutions through collaboration between and within self-organizing and cross-functional teams. A principle of agile development is iterative development. This means that software development activities, such as coding, integrating and testing, repeat over and over again [1]. Code is constantly changing and as such the step of testing is also repeated. This repeated testing is called regression testing, named so because it is used to protect against software regression, by testing that previous functionality is still working after a change to the production code. Since this testing is repeated it becomes beneficial to automate this process [30]. One might think that if a test works once then it will always work and therefore tests that work will never have to be fixed or replaced. This is, however, an incorrect assumption as tests may not have a strictly deterministic outcome.

Flaky tests, also called non-deterministic tests, are tests which can both pass and fail using the same version of the code (depending slightly on definition). This can happen for a variety of reasons, including asynchronous calls, network issues, test case timeouts etc. These tests are a problem for a test suite as they undermine the trust people have in their test results. In some cases this can lead to the results being ignored, defeating the purpose of running the test in the first place [32]. In the process of fixing these tests, reproducing the failure can be very difficult due to the failure not occurring reliably. Tests that exhibit flaky behavior are difficult to fix, and as such require disproportionate attention from test engineers [26]. Therefore, minimising flakiness is a worthwhile goal which has the potential to free up time for test engineers to work on other tasks.

1.2 Definition of flakiness

Different definitions of flakiness exist across the relevant literature. The most common definition of flakiness is a test case that both fails and passes while the codebase is static (i.e. does not change) [11, 29, 40]. Luo et al. [29] define flaky tests as "tests that can intermittently pass or fail even for the same code version" and Eck et al. [11] uses the following definition: "Software tests are flaky when they exhibit a seemingly random outcome (i.e. pass or fail) despite exercising code that has not been changed." Others, such as Fowler, add other conditions such as both the code and the environment have to be the same during both a success and a failure. We have decided to use the broader definition, used by Luo et al. [29] and Eck et al. [11]; *if the same test can both pass and fail for the same code version, it is flaky*. Thus we disregard the environment the test is run in when we define a test as flaky. We chose this definition because it is the most common one used in the relevant literature and also because we want the code base to be more robust against flakiness, and thus should be able to handle a changing test environment.

1.3 Context

Axis Communications AB is a Swedish company, founded in 1984 in Lund, which provides network solutions for security purposes. At the end of 2019 there were 3646 Axis employees in more than 50 countries. The largest office and headquarters is located in Lund. They currently sell network intensive products in the areas of video surveillance, access control and audio. Their regression tests are run over the network using Jenkins, an open source automation tool which is built with continuous integration in mind.

The Lund office consists of two departments: the main department which focuses on network cameras, and the New Business department which focuses on all other products such as access control and audio products. Our thesis was carried out at the New Business department.

1.4 Problem statement

Axis applies iterative development to their workflow and as such are negatively affected by flakiness. Our goal with this thesis is to better understand what causes flakiness and to use that information to develop guidelines for writing tests and handling tests in a way that minimises flakiness and its negative impacts. In order to achieve this goal we must answer the following three questions:

- **RQ1**: What are the common causes of flaky tests?
- **RQ2**: How can the flakiness of a test case be minimised?
- **RQ3**: How should flaky tests be handled in order to balance costs and test suite stability?

Chapter 2
Methodology

This chapter explains the methodology used to conduct the work in this report.

2.1 Research approach

The scientific disciplines can be categorized in the following way according to van Aken [3]:

1. Formal sciences (e.g. philosophy and mathematics)
2. Explanatory sciences (e.g. the natural sciences and a majority of the social sciences)
3. Design sciences (e.g. the engineering and medical sciences)

Formal sciences aim to build knowledge systems that has an internal logical consistency. *Explanatory sciences* aim to produce a model which describes and explains observable phenomena within its field. The research output in the explanatory sciences should lead to propositions which are majorly accepted by the scientific forum. *Design sciences* aim to develop the knowledge necessary to create, realize or improve human-made design. The research output in the design sciences should be knowledge that can be used to design solutions to problems.

The basis for our research strategy builds on methodologies used in the design sciences as the aim of our research is to develop the knowledge necessary to generate a solution for a problem. Further, Runeson et al. [35] argue that the design science paradigm is appropriate in the area of software engineering as the software itself as well as the surrounding tools and technologies are human-made constructs.

We are using the design science research model proposed by Runeson et al. [35] to guide us in our research endeavour. Figure 2.1 shows the model. Design science research contributions can be categorized into practical contributions, i.e. specific solutions to a specific problem, and theoretical contributions, i.e. generalized knowledge about the relationship between the problem and solution domain which can be expressed as a technological rule.

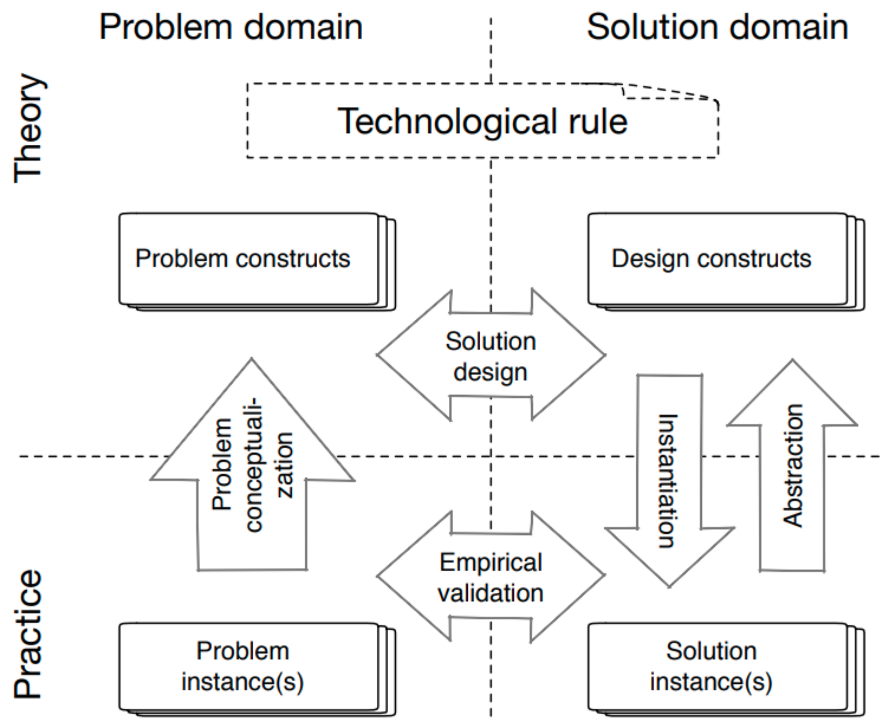

Figure 2.1: "An illustration of the interplay between problem and solution as well as between theory and practice in design science research." by Engström, Emelie, et al. "How software engineering research aligns with design science: a review." [12], DOI: 10.1007/s10664-020-09818-7, licensed under CC BY 4.0.

The arrows in figure 2.1 depict activities that are performed in an iterative manner in order to create knowledge.

- *Problem conceptualization* is the activity of capturing the problem and identifying its constituent parts.
- *Solution design* refers to the activity of formulating a solution that may be expressed as a technological rule.
- *Instantiation* refers to the activity of implementing a solution to a specific problem.
- *Abstraction* refers to the activity of describing the key design decisions for a solution.
- *Empirical validation* refers to the activity of evaluating the implemented solution in its context.

The model works in an iterative way which means that an activity will be revisited several times during the course of the research. Our research includes all activities.

2.2 Research design

As explained previously we follow an iterative manner to conduct our research. In our research we are mainly iterating on (i.e. revisiting) the activity of solution design, whereas the other activities remain static. The solution design will consist of three iterations of possible solutions.

Based on the research questions expressed in chapter 1 we have identified a number of steps to lead us to a solution for our problem.

The steps are :

1. **Literature review.** This step relates to the problem conceptualization and solution design activities.
2. **Interviews.** This step relates to the problem conceptualization and empirical validation activities.
3. **Analyzing flaky tests.** This is the first iteration for a potential solution. This step is mainly performed in the form of a workshop and relates to the problem conceptualization and solution design activities.
4. **Cost trade-offs for flaky tests.** This is the second iteration for a potential solution. This step relates to the solution design activity.
5. **Developing guidelines.** This is the third iteration for a potential solution. This step relates to the solution design and instantiation activities.
6. **Affirmation of research results.** This step relates to the empirical validation activity.

The following subsections explains each step in closer detail.

2.2.1 Literature review

In order to gain a deeper understanding of the subject it is often preferable to start with a literature review. In the book *Research Methods for Engineers*, David V. Thiel [39] notes that a literature review should be conducted using the following sequence of tasks :

1. Key word searching.
2. Selection of relevant papers (partly influenced by publication date and citation number).
3. Review of paper abstracts for relevance.
4. Review of complete papers for relevance.
5. Critical analysis of the results as they apply to the new research project.

Further, Thiel notes that when reading a scientific article the researchers should have the following things in mind :

- The relevance of the article to their research project.
- The research methods described in the article.
- The conclusions reached at the end of the article.
- The relationship of the article to other publications.

We conduct the literature review in two steps. First, we conduct a database search using relevant keywords. This is primarily done using the research platform EBSCOhost. We have identified a number of search terms aimed to find literature which has previously discussed flakiness in testing. The search terms are *flaky*, *flaky test*, *flaky tests*, *flaky testing*, *flakiness* and *non-deterministic tests*.

Second, after identifying relevant literature we further search for relevant research by exploring the references and citations in the found literature. This is known as snowballing [6] and is considered a good complement to a database search.

The initial database search complemented with the search through snowballing gives us a solid foundation of theoretical knowledge about flakiness on which we build upon.

2.2.2 Interviews

Common goals of all interviews are to gain insights through interviewing that we cannot directly observe and to gain new perspectives on a problem or area [34]. We use interviews in order to delve further into the practice domain as mentioned in chapter 2. More specifically we are using interviews as a way to further conceptualize the problem to gain an understanding of the problem from an industry perspective. The interviews also guide us in the empirical validation activity by confirming that the results are applicable in practice. The interviews are done with thirteen Axis Communication employees. 7 testers, 2 developers, and 4 experts within the area of software testing are interviewed. The validation interviews are done with four Axis Communication employees. All four interviewees for the validation interviews were testers.

Hughes [17] discusses three different types of interviews; the informal conversation interview, the general interview guide approach and the standard open-ended interview. As we want to gain a deeper understanding of the practical domain we are using the general interview guide approach so that we are collecting the same general areas of information from the interviewees but we still allow the interviewees to go into further detail in the areas most relevant in their daily work.

The interviews are conducted online through the Meeting function in Microsoft Teams. The choice to conduct the interviews in a digital setting comes from the current situation of the COVID-19 pandemic which complicates a face-to-face setting. Microsoft Teams was chosen as it is the software used on a daily basis by the interviewees and so it is a software they are comfortable using.

The interview starts with an introduction of the thesis work we are conducting as well as an introduction to flaky tests using the definition we established in section 1.2. The interviews then follows the interview guide found in Appendix A, but allows a certain degree of freedom as we may choose to explore an area further. The interview ends with the interviewee being able to ask questions or further clarify anything they said during the interview as this can sometimes lead to further information being provided [17]. All interviews are recorded so that we are able to confirm any given answers.

2.2.3 Analyzing flaky tests

In order to gain an understanding of the causes for flaky tests we attempt to analyse a number of real-life examples of flaky tests. This helps us get an understanding of and further conceptualizes the problem of flaky tests. The tests are part of a real test suite at Axis Communications.

The analysis is done in two steps. The first step focuses on identifying the flaky tests. This includes both identifying flaky test candidates as well as confirming their flakiness. The second step focuses on identifying the causes of the tests that are confirmed flaky. This is mainly done through code analysis.

The following sections explain the steps in further detail.

Identifying flaky tests

When identifying flaky tests we use historic data consisting of over two million tests, run from late August of 2019 until late January of 2020. The information contained in this data set consist of: The date and time the test was run, the build, the test level, the test class, the test case, and the result for that test. From the data we sort out the test cases that have both passed and failed in the same day, thus presumably running the same code base. These are our candidates for flaky tests as they follow the definition we established in chapter 1.

After identifying potential flaky tests we verify their flakiness by running the tests a large number of times and observing the results. We first run the tests on the latest code base to see if the test case is still exhibiting flaky behavior. The test cases that both pass and fail at least once during the run are labeled as flaky on the latest code base.

If the tests that have been identified as flaky do not exhibit flaky behavior when running them with the latest version of the test and production code, we rerun the tests using the test code versions which were used the last time they exhibited flaky behavior. If a test case both passes and fails during this run we assume that the flakiness has been fixed and a manual code analysis is used to classify the cause of the flaky behavior.

Identification of causes of flaky tests

Using the test cases identified as flaky, we try to identify the causes of their flakiness. This is done to get a better understanding of the problem (i.e. conceptualizing the problem) and to be able to start the activity of solution design.

We do this using two different approaches.

1. For test cases which no longer exhibits flaky behavior, but were confirmed flaky in an older commit we attempt to identify the cause of the flaky behavior by studying the commit that is most likely to have fixed the test.

2. For test cases that are still flaky we attempt to categorize the cause of flakiness by using a static code analysis on the latest version of the code.

The following sections explains the approaches in further detail.

Identifying causes using commits
If a test case is confirmed flaky on an older version of the code, but does not exhibit flaky behavior on the latest version it is likely that a change in the code base was able to fix the flaky behavior.

In this case we attempt to identify the cause of the flaky behavior by analyzing the code changes. We do this both by exploring the commit message as well as inspecting the actual code change. Based on the code changes as well as any hints given in the commit message we attempt to classify the root cause of the flaky behavior.

Identifying causes using static code analysis
For the test cases that are still exhibiting flaky behavior in the latest version of the code base, we attempt to find the likely causes of the flakiness using static code analysis. This is done in the form of a workshop together with Axis QA employees.

During the workshop one test case at a time is studied. The participants are introduced to the test case and any accompanying log information about the failure. They then attempt to fix the test case (i.e. make it deterministic) and then classify the cause of the flaky behavior based on the fix. After the workshop the test cases are run again to verify that the proposed fix actually fixes the flaky behavior.

2.2.4 Cost trade-offs for flaky tests

Most of the research that currently exists regarding flaky tests, as summarised in chapter 3, focus on determining the cause of the non-deterministic behavior and on ways to fix it. There is however currently no research regarding the potential costs in fixing flaky tests and reaching a completely deterministic test suite. Similarly the current research on cost models in regression testing does not consider the case of flaky tests. We explore this area by proposing a cost trade-off model for flaky tests focusing on the costs and benefits of fixing flaky tests as opposed to removing them from the test suite.

As explained in chapter 2 we are exploring the problem of flaky tests both in the theoretical domain as well as in the practical domain. In a real-life situation the benefit should outweigh the cost associated with performing an activity in order to justify performing said activity. This is in literature often referred to as a cost-benefit analysis [8].

We perform a cost-benefit analysis for whether or not a flaky test should be fixed or if it should be thrown out and rewritten. Further, we weigh the benefits of increased code coverage but decreased reliability of test results against a lower test coverage but increased robustness of the test suite.

The cost-benefit analysis helps us answer RQ3, as expressed in section 1.4, and we are therefore focusing on the following questions:

- How much time should be spent trying to fix the flaky behavior of a test before the cost outweighs the benefits?
- Is it better to remove a flaky test from the test suite entirely or is it better to leave it in even if the test results are less reliable?

- Will it always be better to remove the test or does the test have to be certain level of flaky for it to be worth it?

We perform the analysis by comparing the effort required by an employee to fix a flaky test with the benefit gained by having fixed the test. We do this by conducting a literature review, as explained in subsection 2.2.1, as well as interviewing a number of employees at Axis Communications as explained in subsection 2.2.2.

2.2.5 Developing guidelines

As explained in chapter 2 we are working in both the theoretical and the practical domain. In order to make the information in this thesis available and suitable for practitioners as well, the final results are presented in the form of guidelines.

The cost of fixing a defect in software is larger the further into the software development process you get [38]. Detecting flaky behavior in test cases before they enter production can therefore be a huge cost saver. As such we develop guidelines meant to mitigate the risk of a flaky test case entering the production code. The guidelines also cover how to handle flaky tests once they have entered the production stage, building on the knowledge gained from the cost trade-offs analysis.

The guidelines are developed using the insights gathered from interviews with Axis employees, as explained in subsection 2.2.2, as well as insights gained from the literature review. The guidelines focus on methods of writing tests to reduce flakiness as well as how tests which are already flaky should be handled.

2.2.6 Evaluation of research results

The last step of any research endeavour is evaluation of the results. We evaluate the results by having a discussion of the methodology used as well as through a validation of the guidelines with test employees and managers.

In the discussion of the methodology used we critically analyse the assumptions made and discuss any factors that may have influenced the results. We also discuss the delimitations we have used in the research endeavour in order to evaluate how generalized the results can be.

The validation of the guidelines are done by evaluating how employees at the QA department at Axis Communications perceive the guidelines as well as what difficulties and risks they find in implementing the guidelines. This is done through interviews with Axis QA Employees. The interview guide used for the interviews can be found in appendix B.

Chapter 3
Theory

This chapter aims to provide an overview of the current research related to non-deterministic tests.

3.1 Literature review

Table 3.1: Relevant literature as well as type of literature.

Literature	**Type**
Empirical Analysis of Factors and their Effect on Test Flakiness - Practitioners' Perceptions [2]	Empirical study
DeFlaker : Automatically detecting flaky tests [7]	New technique
Understanding flaky tests: the developer's perspective [11]	Empirical study
Proactively detecting unreliable tests [14]	PhD Thesis
Towards a Bayesian Network Model for Predicting Flaky Automated Tests [21]	New technique
iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests [24]	New technique
Where do our flaky tests come from? [26]	Case study (Google)
An empirical analysis of flaky tests [29]	Empirical study
Can We Trust Test Outcomes? [28]	Empirical study
Flaky tests at Google and how we mitigate them [32]	Case study (Google)
An empirical study of flaky tests in Android apps [40]	Empirical study

This chapter builds upon the information found in the literature review, as described in subsection 2.2.1. After conducting a database search as well as snowballing we identified a number of journals/articles relevant to the topic at hand. We did not deem it necessary to make a selection from the identified literature as the number of articles was relatively small. The identified relevant literature is presented in Table 3.1.

The literature presented in Table 3.1 were all written by different authors (with the exception of [28, 29]. However, most identified research builds upon the empirical study conducted by Luo et al. [29], which was the first empirical study done on flaky tests.

In addition to the research presented in Table 3.1, we also identified a number of papers where flaky tests were discussed but where it was not the main focus of the paper, e.g. [4, 15, 18, 23, 27, 31].

When finding literature relevant to, for example, the cost-benefit analysis section (e.g. [8, 16, 20, 25]), we made more general searches and did not employ the practice of snowballing. Those searches were therefore not as extensive as they were not the main focus of the thesis.

3.2 Flaky or non-deterministic tests

Regression testing ensures that previously working code still works after performing changes in the code. This is done through performing tests on previous functionality in order to test that the functionality is still working after applying a change to the production code. It is an essential part of software development and a failure when performing regression testing should indicate that the newly introduced changes broke previously working functionality. This assumes that the tests are deterministic, i.e. the test should always pass or fail for the same code under test. However, in practice this is not always the case.

Flaky tests are tests that are non-deterministic in their behavior. This means that the test may pass or fail without any changes having been made to the test code or to the code under test. These types of failures are relatively common in large codebases. For example, according to Luo et al. [29] the TAP system at Google attributed 73K out of 1.6M failures (4.56%) to flaky tests. Google has also reported seeing a continual rate of 1.5% of all test runs reporting a flaky result and almost 16% of tests across Google's test suite has some level of flakiness associated with them [32]. Additionally, Labuschagne et al. [23] attributed 13% of test failures to flaky tests in their study on regression testing in Java projects.

There are several problems associated with flaky tests. Because of the non-deterministic behaviour inherent in flaky tests, test failures caused by flaky tests can be very hard to reproduce. This could potentially lead to a developer spending a lot of time trying to track down a bug, only to discover that it was caused by a flaky test. This is both time consuming and wastes a lot of machine resources [11, 29]. A consequence of this is that many developers instead ignore failures as they chalk it up to a flaky test.

Ignoring flaky tests could be a problem as they may hide real bugs. Luo et al. [29] found that 24% of flaky tests were fixed by also changing the code under test. They found three reasons for this:

- The source code is deterministic but contains a bug which is discovered thanks to a flaky test. In this case the test code is the cause of the flakiness, but the test case was still helpful in uncovering a real bug.
- The source code is non-deterministic and contains a bug (for example a race condition) which causes flakiness. This happens if the flakiness is in the production code rather than the test code.

- The source code is non-deterministic but does not contain a bug. In this case the test code may not cover all possible result values, resulting in non-deterministic behavior. If this happens developers sometimes attempt to make the code more deterministic to make it easier for the test cases to cover all possible correct results.

3.3 Root causes for flakiness

There have been several empirical studies to understand the causes for flakiness in testing suites. Luo et al. [29] studied 201 commits that fixed flaky tests in 51 different open-source projects. They identified ten different categories for causes of flaky tests, out of which the top three categories represented 77% of the studied commits.

In subsequent work, Eck et al. [11] asked developers to classify flaky tests they had previously fixed. In addition to the causes reported by Luo et al. they found a number of new causes of flakiness.

The root causes found in previous research are presented in Table 3.2.

Table 3.2: Root causes for flakiness.

Root cause	**Description**
Async wait	This type of flakiness arises when a test makes an asynchronous call and does not wait for the result of that call to become available before using that result. For flakiness to be classified as Async wait the flakiness must be the result of a remote resource being unavailable. An example of this is when the test waits for a fixed amount of time for a result from a remote server every time the test is run. Sometimes the remote server takes longer to respond than the test waits for, which leads to this type of flakiness. [29]
Concurrency	Much like async wait, this flakiness is caused by synchronization issues. However, unlike async wait, this flakiness is caused by a local synchronization issue in contrast to a remote one. This flakiness can be characterized as when the interaction of different threads lead to an undesirable outcome, for example when the threads execute in a different order than their dependencies would require. [29]
Test order dependency	As the name suggests, this flakiness is caused by the result of the test being dependant on which order the tests are run. This occurs when a shared state is not properly setup or cleaned. The failure would then occur if the test is run after another test has changed the state of the main memory or an external resource without cleaning up or resetting it to the expected state before the next test is run. [29]
Resource leak	This flakiness is caused by the application not managing its resources correctly, such as memory or database connections. [29]
Network	The network on which the tests are run can be unreliable at times and as such tests which execution depends on it may exhibit flaky behavior. [29]
Time	Flakiness may arise from the test relying on the system time. The failure could be caused by time zones or perhaps be due to the timing imprecision when compared with another platform. [29]

IO	Flakiness may also be found during I/O operations. An example of this is having a file reader that is not closed until it gets garbage collected. If a test would try to open the same file it could either pass or fail depending on whether or not the file reader had been garbage collected. [29]
Randomness	Tests that make use of a random number generator may exhibit flaky behavior. This might be caused by the test not accounting for all possible numbers generated. For example, the test might expect a value greater than zero, but the random number generated can assume a value of zero or greater. [29]
Floating point operations	Operations with floating point numbers can exhibit flakiness if the precision of the floating point numbers are not considered. Both potential over- and underflow needs to be considered for the test to behave properly. This type of flakiness also includes the case where the failure is caused by a differing number of significant digits in different test executions. [29]
Unordered collections	When iterating over a set, the elements can either be sorted or unsorted. If the code assumes that the elements are sorted while the set is unsorted, the test outcome can become non-deterministic. [29]
Too restrictive range	If the allowed range of output values is narrower than the valid range of output values the test will pass or fail at random, i.e. flaky behavior. [11]
Test case timeout	The max runtime value can be selected in such a way that the test experiences non-deterministic timeouts. This is usually the result of the test growing over time while the max runtime value not being increased to match the new size of the test. [11]
Platform dependency	A test may be called flaky if it fails on only certain platforms (e.g. a test might fail on Windows 10 but not Windows 7). This flakiness might be caused by the platform being unusually slow or the platform might have missing preconditions. [11]
Test suite timeout	Similar to Test case timeout, but the max runtime value is not properly adjusted for the entire test suite instead of the max runtime value not being adjusted for a single test. [11]
Hard to classify	There were also a number of cases where the cause could not be classified into a category. The reasons for this could be that there was not enough information provided in the commit or that the developer did not understand why the test code changes fixed the flakiness. [29]

It should be noted that sometimes a flaky test may be attributed to multiple root causes. For example, a flaky test caused by a network problem may manifest itself as a test case timeout, making both network and test case timeout the cause of the flakiness.

Thorve et al. [40] studied flakiness in Android projects. In addition to the aforementioned causes they also found three additional causes of flakiness, which can be seen in table Table 3.3. The most common root cause for flakiness found in Android projects was concurrency, similar to the findings of Luo et al.

Table 3.3: Root causes for flakiness.

Root cause	Description
Dependency	This type of flakiness is caused by certain hardware, OS version or third party libraries. For example, a test case may work on one device but fail or exhibit flaky behavior when run on a different device.
Program logic	This type of flakiness occurs when developers have wrong assumptions on program behavior.
UI	This refers to flakiness caused by the user interface of the application. For instance, if a test does not close the soft keyboard before invoking a click() event, the clicking gesture may be wrongly captured by the soft keyboard instead of as a click on the correct UI element.

Introduction of flaky tests

When exploring how flaky tests are introduced in a test suite Luo et al. [29] found that out of 161 studied flaky tests 126 were flaky from the first time they were written, 23 became flaky in a later commit and 12 cases were hard to determine as tests were refactored.

In the 23 cases where a flaky test was introduced in a later revision of the code it was mainly because of two reasons. The first reason is newly introduced tests that violate the isolation between tests. If a test case relies on a shared variable being a set value at the start of the test for the test to pass and a new test case changes the shared variable then the first test may become flaky. This would introduce a flaky test that falls under the test order dependency category.

The second reason is due to test code changes in the test itself. This could be due to, for instance, patching a new bug, refactoring the test or changing test functionality. The changed test code could introduce flakiness to a test case that was previously deterministic.

When asked if most flaky tests are flaky from the time they are written, Ahmad et al. [2] found that 40% of surveyed practitioners responded yes to the question. 40% of survey takers responded maybe, 10% responded that they did not know and 10% responded no.

3.4 Mitigation strategies for flakiness

There exist no general solution so far that completely eliminates flaky tests. The approaches that exist often have other problems associated with them. This section provides an introduction to the current approaches and solutions proposed in research to eliminate or decrease flakiness in test suites.

3.4.1 Rerunning flaky tests

The most common approach to combat flaky tests is to rerun a failing test multiple times. If the test passes any of the reruns the test is considered to have passed [29, 40].

There are a number of annotations that can be used to help with this. The Android annotation @FlakyTest [5] will rerun the test at failure until either the test passes or the number of runs exceeds the retry limit, @RandomlyFails [19] in Jenkins will rerun the test and ignore the test if it fails consistently and retry rules in JUnit will rerun tests that have failed.

This approach is often deemed unsatisfactory as it does not remove flaky tests. As such several of the problems associated with flaky tests still exist when using this approach. It is also considered costly and may slow down the development cycle. [7]

3.4.2 Improving implementation

A common way to fix flaky tests is to improve the implementation of the test cases. For instance, to fix issues related to concurrency a common strategy is to introduce locks, sleeps or increasing the current waiting or timeout time. Another improvement could be to add extra condition checks or making the software more resilient to non-determinism caused by third party libraries [40] .

Eck et al. [11] performed a study on flaky tests from the developers perspective. They identified the following pieces of information that helped practitioners in fixing flaky tests:

- **The context leading to failure.** As flaky tests are hard to reproduce it is often hard to identify the context that leads to the failing behavior. However, the context of the failure was reported as an important piece of information both in fixing a test case and in verifying that a change has fixed the failing behavior. This was rated as a important piece of information in order to fix a test, but also difficult to obtain.

- **The nature of the flakiness.** Understanding which of the causes, as discussed in section 3.3, is considered an important piece of information by practitioners but is also considered one of the hardest pieces of information to obtain.

- **The origin of the flakiness.** An important piece of information is the origin of the flakiness, i.e. whether the flakiness is caused by the testing code or by the production code.

- **The involved code elements.** This is mostly obtained by looking at the test code and log files. Practitioners find that the challenge of obtaining this piece of information is mainly how complex the source code is. The authors argue that this may indicate that keeping the code high quality may ease the fixing process.

- **The changes to perform the fix.** This piece of information is deemed slightly relevant for developers.

- **The context leading to passing.** This piece of information is considered slightly important by developers. The main reason for this is that developers mainly aim to identify cases where the test will fail.

- **The commit introducing the flakiness.** This piece of information is deemed slightly relevant for developers.

- **The history of this test's flakiness (previous causes and fixes).** This piece of information is deemed the least important. The authors argue that this is likely due to the cause of the flakiness being different, meaning that a past fix can not be used to fix the new type of flakiness.

The context leading to failure, and the nature of the flakiness was considered the most important pieces of information in order to fix a test, but also the most difficult to obtain. Similarly, the history of the test's flakiness, and the commit introducing the flakiness were considered the least difficult to obtain but also the least important when fixing a flaky test.

The survey sent out by Eck et al. also revealed that the risk of a test being flaky is reduced if good design principles are applied while the code is developed, according to the participants of the survey. They also mentioned that mocking dependencies and keeping tests decoupled are good design practices to reduce test flakiness. Decoupling test cases is a good practice to ensure that tests does not affect each other. The main benefit of mocking dependencies is that one is able to control the environment, by e.g. making sure installed packages are of the right version and mocking out dependencies on external services. They also note that in order to reduce test flakiness it is important that tests are not reliant on external dependencies.

Participants also reported major challenges in verifying if a change fixes the test (as it may disappear by chance), insufficient levels of details in log files as well as a lack of insight into the system.

Luo et al. [29] described a number of strategies used by developers to fix flaky tests for a number of root causes. They are briefly summarised in the following sections.

Async wait

Flaky tests caused by async wait are fixed by addressing the underlying order violation. An order violation is defined as "The desired order between two (groups of) memory accesses is flipped (i.e. A should always be executed before B, but the order is not enforced during execution.)" [27].

The following fixes were found for flaky tests caused by async wait issues :

- 57% of tests were fixed by calling waitFor. Often (36% of times) the waitFor replaced a sleep call.
- 27% of tests were fixed using sleep calls. Of the tests which were fixed this way, 60% of the fixes entailed increasing the waiting time of an existing sleep, while 40% of fixes introduced a new sleep call to the test.
- 3% of tests were fixed by reordering the code.
- 14% of tests had a fix specific to the project and was therefore hard to generalize.

All the fixes did not completely remove flakiness. Instead, some of the fixes simply reduced the frequency of the failures. For example, if a test consisted of increasing the waiting time in a sleep it may still be flaky when using slower hardware.

Using sleep is generally not an advised approach as it is inefficient and makes the test harder to grasp [18]. It is often hard to distinguish what order the sleep is trying to enforce. Further, delevopers tend to over-estimate the waiting time needed in a sleep, making the test more inefficient.

Using a mechanism similar to waitFor is preferred as it makes clear what condition needs to be satisfied before the program can proceed. This is also a more efficient approach as it allows the program to continue as soon as the condition is met. As a result a longer timeout can be used when using a mechanism similar to waitFor compared to sleep, which would reduce flakiness as it allows processes to finish that would timeout when using a sleep call. This is also supported by Luo et al. [29] who found that the average waiting time in a waitFor call was 13.04 seconds, compared to an average of 1.52 seconds for sleep calls.

Concurrency

Concurrency bugs are caused by threads interleaving in an undesired way. The concurrency bugs found in flaky tests are similar to concurrency bugs that can be found in production code.

The following fixes were found for flaky tests caused by concurrency issues [29]:

- 31% of tests were fixed by adding a lock to code that should only be accessed by one thread at a time. This strategy ensures mutual exclusion for the code.
- 25% of tests were fixed by making the execution deterministic. The fixes include, for instance, modifying code to eliminate concurrency and enforcing deterministic orders between thread executions.
- 9% of tests were fixed by changing the guard conditions in the test code.
- 9% of tests were fixed by changing assertions in the test code. For example, if non-determinism is permitted in the program the test assertion may not accept all valid behaviors. The fix to this is to change the assertion to accept all valid behaviors.
- 25% of tests had a fix specific to the project and was therefore hard to generalize.

All studied fixes for flaky tests caused by concurrency issues completely removed flakiness in the test case.

Test order dependency

Flakiness caused by test order dependency may be difficult to identify as it is usually not clear what tests are interdependent of each other. When developers figure out the dependency the fix is usually quite straight-forward.

The following fixes were found for flaky tests caused by test order dependency issues: [29]

- 74% of tests were fixed by setting up or cleaning up the shared state between tests.
- 16% of tests were fixed by removing the dependency (i.e. by creating a local copy of the shared variable).
- 10% of tests were fixed by merging dependent tests, thus eliminating the dependency between tests.

All studied fixes for flaky tests caused by test order dependency completely removed flakiness in the test case. The first two strategies are however preferred over the third strategy of merging the tests as merging tests hurts the readability and maintainability of the test suite.

Other root causes

Fixes that were found by Luo et al. [29] for flaky tests caused by other root causes than the ones disucussed previously are discussed here.

- **Resource leak**. Fixes for flaky tests caused by resource leak was hard to generalize. One way to handle resource leaks suggested by Fowler [13] is to manage resources through a resource pool. By reducing the pool size to 1 and make it throw an exception when a resource is requested and the pool is empty the test causing resource leak can be easier found.
- **Network**. There are two ways to fix flakiness caused by network issues. The recommended approach is to use mocks when dealing with networks. If mocks is not a viable approach for the project then flakiness can be alleviated by using a waitFor mechanism.
- **Time**. As time precision can differ between systems the recommended approach is to avoid using platform dependent values such as time.
- **IO**. The recommended approach when dealing with IO-related issues is to ensure that any opened resource gets closed and to ensure proper synchronization when multiple threads are sharing a resource.
- **Randomness**. To alleviate flakiness caused by randomness the recommended strategy is to control the seed of the random generator so that an individual run can be reproduced while the seed can be varied between runs. Developers should also ensure that edge cases (e.g. boundary values) are properly handled.
- **Floating point operations**. As floating point operations are non-deterministic by nature developers should be careful when dealing with floating point operations. It is recommended to make test assertions independent from floating point operations.
- **Unordered collections**. In order to reduce flakiness in tests caused by unordered collections developers should write tests in a way that does not assume a specific ordering on collections.

3.4.3 Removing flaky test cases

One way to eliminate flakiness in a test suite is to simply remove the test cases that exhibit flakiness. Thorve et al. [40] found in their study about flakiness in Android projects that 10 out of 77 studied commits that fixed flakiness simply commented out the flaky test code. This was mainly done when the root cause of the flakiness could not be found.

3.4.4 Replacing implementation

One strategy developers can take to fix flaky tests is to change the implementation (e.g. by refactoring code or implementing a test in an alternative way). This is a strategy used when for example a third party library or network is the cause of the flakiness. By, for instance, replacing an old version of a library with a newer version or by replacing a piece of code with a semantically equivalent version that is syntactically different, flakiness could be removed or reduced even though developers sometimes struggle to explain why. [40]

3.4.5 Using machine learning

Machine learning and artificial intelligence is a rapidly growing field with many use cases. One such use case could be to automatically detect flaky tests. A first exploration of this approach has been made by King et al. [21] who used Bayesian networks for classifying and predicting flaky tests.

King et al. constructed a Bayesian network by viewing flaky tests as a disease with a number of symptoms. After identifying symptoms, causes and supporting metrics they tested the Bayesian model through a case study where the model was tested on an actual product. The model correctly predicted 188/583 of true flaky test cases and 1880/2552 of false flaky test cases thus having a test flakiness prediction accuracy of 65.7%. The results of the case study were considered positive, resulting in a high improvement in stability for the company and an improvement in stability of up to 60% for the individual test teams. This points towards machine learning as an approach for flaky test prediction being viable and worthy of further research.

3.4.6 Assume tests are flaky

Harman and O'Hearn challenges the common notion of tests either passing or failing with no other possible outcomes from the system under test by proposing the view of "Assume Tests Are Flaky" (ATAF) [15]. As such, practitioners should not think of a test as having a boolean outcome (i.e. passing or failing) but rather having a probabilistic outcome. They note a number of consequences that this view has on testing:

- **Regression testing**: It is typical to optimize according to certain objectives when performing regression testing, e.g. execution time, test coverage and resource consumption. With the view of ATAF another objective that should be considered is test flakiness (e.g. a prioritisation might be to favour tests that are more deterministic over those that are less deterministic).

- **Mutation testing**: Fundamental concepts, such as the mutation score, needs to be adapted to fit with the view of ATAF.

- **Foundations**: The theoretical foundations of software testing needs to be revisited when using the view of ATAF, even for fundamental concepts such as test coverage as the coverage may differ on different executions of the system. For example, the authors note that when evaluating a weather application the statement coverage of the application depend on the current weather condition [4].

They also discuss that much of the previous research focuses on trying to control and reduce flakiness. They state that this is desirable in the cases where it is possible, but they also believe that hoping to return to the world of deterministic testing that was common in the early stages of software development is idealistic and in many cases unrealistic. They state that research focus should be on how to cope with flakiness, but also on constructing test automation approaches that benefit from it, rather than on trying to eliminate the problem completely. This view of flakiness gets support by John Micco at Google who states that flaky tests are inevitable and testing systems must be able to deal with a certain level of flakiness [31].

3.4.7 Test prioritisation and test selection

Certain test prioritisation and test selection techniques has been shown to reduce flakiness during test execution. Busjaeger and Xie presented an approach for test prioritisation that had the unexpected benefit of detecting flaky tests [9]. Similarly, Shi et al. [36], studied regression test selection (RTS) techniques and found that a majority of failures missed by RTS techniques were flaky tests. They argue that RTS is beneficial for avoiding flaky tests.

3.4.8 Tools

A number of tools have been proposed in previous research for identifying flaky tests.

DeFlaker

DeFlaker [7] is a tool proposed by Bell et al. that can be used to automatically identify flaky tests. DeFlaker identifies flaky tests by comparing both the current test code as well as the current code under test to the previous version. This outputs the difference in code between the two versions.

After a test run a report containing the following is produced:

- A test case which has previously passed but now fails, without covering any code that has been changed is marked as flaky.
- A test case that fails but passes when rerun on the same code is marked as flaky.
- If a non-code file has been changed since the last run, a failing test is marked as *might* be flaky and the non-code changes is pointed out as the potential cause.
- For failing tests that covered change code a message is printed out outlining each part of the changes that the test covered.
- For any changes that are not covered by any test, DeFlaker prints a warning that the changes are not being tested.

By comparing the changes made to the code the tool is able to detect flaky tests without rerunning them, saving a great amount of time and processing power. In the evaluation of the tool it managed to find 4846 flaky tests out of 5328 confirmed flaky tests (95.5%) with a false positive rate of 1.5%.

The implementation of the tool was done in Java and was used on projects using the continuous integration service Travis CI. The authors note that because of complex projects often including manual configuration steps before they can compile there can be a substantial human cost of running the tool on a new project.

iDFlakies

iDFlakies is a tool developed by Lam et al. [24] which automatically detects and partially classifies flaky tests. The tool reruns a test suite a specified number of rounds. For each round that contains a test failure the tool runs a classification step on the failed test cases to determine if the flakiness is due to test order dependency or not.

The tool classifies the flaky tests as either an order-dependent test or a non-order-dependent test. An order-dependent test is flaky due to the order of the tests being run, whereas a non-order-dependent flaky test is flaky regardless of the test order. The tool does not classify the non-order-dependent tests further.

iDFlakies is implemented to run on maven-based Java projects using JUnit tests. When testing the tool Lam et al. used 183 projects containing 2921 modules and 1880362 tests. Because of problems relating to projects not using JUnit, modules being unable to be built by Maven and other problems the tool was only run on 945 of those modules. Of the 945 modules the tool detected 111 modules containing flaky tests.

3.5 Costs and benefits of test automation

This section provides an overview of the current research regarding costs and benefits in regards to test automation. The section focuses on research applicable to flaky tests.

Hoffman [16] notes that test automation is not always appropriate or desirable. For instance, one falsely expected benefit is often that all tests will be automated. This is not practical nor desirable as not all tests can be automated in a cost effective way. He also notes that there usually will not be an immediate payback from automating tests. Instead the benefits usually comes from running and rerunning the tests many times with little human intervention.

One of the first cost models for regression testing was proposed by Leung and White [25]. They identified the following four components related to the cost of testing a software systems:

1. **System analysis cost, Ca**. This is the cost associated with the test analyst becoming familiar with a system before being able to test it. The larger and more complex the system is the higher this cost will be.

2. **Test selection cost, Cs**. This is the cost associated with figuring out test input and corresponding output or system behavior. This cost will depend on the chosen test strategy.

3. **Test execution cost, Ce**. This is the cost associated with setting up the system and the cost of resources required to execute the system under test.

4. **Result analysis cost, Cr**. This is the cost associated with checking the output of the test and compare it to the desired or specified behavior. This cost depends on the time required by the tester to collect the required data, the time required to compare the collected data as well as the resources required for recording the required data in the system.

All cost components are related to the number of test cases.

As far as we could find, no cost models exist in regards to flaky tests, we therefor adapt the cost model of Leung and White for a flaky test context. The system analysis cost should not be affected by flaky tests as flaky tests are non-deterministic in its output, and as such a flaky test will not change the system itself. We therefore consider the system analysis cost to have a small impact when considering the cost of flaky tests.

Test selection cost is related to flakiness as figuring out the system behavior or corresponding output may become more time consuming. For example, consider a test case where the output is supposed to be non-deterministic, e.g. the possible (valid) outputs are A, B and C. If the test analyst forgets to check for one of the outputs, e.g. C, the test case will become flaky as the test case will pass when the output is A or B but fail when the output is C. This will lead to more time being spent trying to define the range of valid outputs, as opposed to a system where the output is deterministic (e.g. always A).

Having more flaky test cases in a test suite should also increase the test execution cost. The most common way currently to handle flaky test cases is to rerun the test case until it passes. This takes up more machine resources compared to having a deterministic test suite where each test case only needs to be run once.

The result analysis cost will also be effected by having flaky tests in the test suite. If a test case is non-deterministic then more resources need to be allocated to a failure in the system as these kind of defects are generally harder to track down. The result analysis cost can further be broken down into two separate components, namely the cost, Cu, for understanding the program and specification in order to evaluate if the program output is correct and the checking cost, Cc, for comparing each test output to the expected output.

This leads to the total cost, C, of implementing a test strategy, S, for a set of tests, T, with regards to flaky tests as

$$C(S) = Cs(T) + Ce(T) + Cu(T) + Cc(T) \tag{3.1}$$

In summary, the equation expressed in Equation 3.1 shows a first cost model for flaky tests. It is an adaptation of the cost model previously proposed by Leung and White, adapted to only include cost components relevant to flaky tests.

Chapter 4
Identifying and fixing flaky tests

The first iteration in our research endeavour consisted of trying to conceptualize and define the problem domain, as expressed in subsection 2.2.3. To do this we tried to identify flaky tests in the test suite at Axis and analyse the fixes to see if we could find a generalised pattern in either causes or fixes.

4.1 Approach

Our approach consisted of two steps. First, we needed to identify flaky tests in the test suite and then we needed to identify a fix to the test in order to categorize the cause of the flaky behavior. The following sections explains the steps in further detail.

4.1.1 Identifying flaky tests

The first step in our approach was to identify flaky tests. Through the historic test data we filtered out the test cases which had both failed and passed at least once during the same day. These became our candidates for flaky tests as the tests were presumably run on the same code base thus meeting our definition of a flaky test established in section 1.2. For each test case that passed the criteria we noted the name of the test case as well as the date on which the test case had last failed.

After identifying our flaky test candidates we wanted to confirm their flakiness. We started by running the tests on the latest production code as well as the latest test code 100 times in a row. If the tests passed our criterion for flakiness (i.e. passing at least once and failing at least once), we marked the tests as flaky in the latest code version. If the tests did not pass our criterion we marked them as not flaky on the latest code version.

For the tests which we could not confirm as flaky on the latest version we instead tried to confirm their flakiness on the version used on their last failure date as previously noted. We did this by using the code version of the test code that was used on the noted last date.

We again ran the tests 100 times in a row in order to identify flaky behavior. The tests were noted as either flaky on that version or not able to confirm flakiness depending on whether or not they passed our criteria for flakiness.

4.1.2 Fixing flaky tests

In order to gain a deeper understanding of the complexity in turning a test case with a non-deterministic output into a test with a deterministic output we hosted a workshop. The participants in the workshop were three QA employees at Axis Communications. The goal of the workshop was to perform a static code analysis of the tests we had identified as flaky in subsection 4.1.1. This effort proved futile as at best, an educated guess could be made about what could have caused the flakiness, and at worst not even such a guess could be made. The team spent more than an hour on each test case identified as flaky, never finding the true cause but instead moving on to the next test case when it was recognized that they would not be able to solve the problem by just looking at the code. We also analysed the commits which were suspected to have fixed the flakiness. This also did not yield very useful results. Almost all of the commits which had resulted in the flakiness being fixed were not made with the purpose of fixing the flakiness, which meant that the commit messages almost never specified how the flakiness had been fixed. It was also very difficult to analyse how the commits had fixed the flakiness by looking at the changes to the code, even though one tester who had written much of the code we analysed was participating in the workshop. The few commits which were clear what the cause and fix of the flakiness was, were not enough to base our analysis on.

We abandoned this approach when we realized that even the most educated guess would have to be used to create a fix, which would then have to be verified to have fixed the flakiness, to be of any value. While this could perhaps be accomplished for some of the tests, the time required to write a fix and to verify that it had worked would mean that only a couple of tests could be fixed. With this data, no meaningful conclusions could be drawn.

4.2 Results

After analysing the test data we identified 74 test cases that passed the criterion of failing and passing at least one test run during the same day. 10 of the 74 tests had the last failure on a test run using the latest commit (i.e. using the latest codebase).

After finding our candidates we attempted to confirm their flakiness by running the test cases a number of times in a row. The following results were found:

- 25 tests were confirmed flaky on the latest version of the codebase.
- 22 tests were confirmed flaky on an older commit.
- 15 tests could not be tested as the the tests could not be run.
- 12 tests could not be confirmed as flaky.

This gave a total of 47 identified flaky tests, out of which 25 were still flaky on the latest code base. Fixing the tests turned out to be more complex than first expected and during the 6 hour workshop we only managed to look at 5 test cases.

There was no conclusive root cause found for any of the test cases. For 4 of the test cases there were several proposed causes which would require additional time to verify whether or not they would serve as a solution to fixing the flaky test. For one test case there was no proposed fix as there was not enough information (e.g. logs and error data) available to identify the potential cause of the problem.

4.3 Discussion

The results point towards the time requirement of fixing flaky tests being quite large. It should also be noted that test cases that are easily fixed have presumably already been fixed as the sample of tests are taken from a live test suite where tests are maintained and updated continuously. A more accurate conclusion of the results may therefore be that the time required to fix *all* flaky tests in a test suite is very large, and sometimes not feasible.

This conclusion is reinforced by the interviews we conducted where several of the interviewees pointed to a fix being significantly variable in length. The interviewees all agreed that a flaky test case generally takes longer to fix than a test case that is defective but deterministic (i.e. wrongly implemented). The reasons given were mainly that a flaky test takes longer to investigate as it may be harder to reproduce, and a potential fix is harder to verify as the test needs to be run many times before the test can be verified as now being deterministic.

From the workshop we were unable to draw any conclusions on the causes of the flaky tests. During the interviews we also asked about what causes had been seen previously for flaky behavior. The following causes were found, with the number of interviewees giving a specific answer in parentheses:

- Asynchronous waits/sleeps (10/13)
- Network problems/timeouts (9/13)
- Dependencies (e.g. third party software) (6/13)
- Environment issues (e.g. power outages) (4/13)
- Bugs in the production code (3/13)
- Unknown/unable to identify a cause (3/13)
- The test case being too complex (3/13)
- Teardown wrongly implemented (e.g. not resetting changed values after a test run) (3/13)
- Problems with Jenkins (2/13)
- Badly maintained tests (1/13)
- Too many open resources (1/13)

Some of these categories can relate to the same issue. For example, a problem in the network could cause timeouts to happen.

The majority of the interviewees noted timeouts and network problems as the major causes of flaky tests. These issues have not been as prevalent in previous studies conducted on flaky tests [11, 29, 40]. This leads us to believe that the causes of flaky tests are often context-dependent. The products studied in this study are heavily network-reliant, and as such so are the tests. This leads to network problems being a large cause of flakiness in the test suite.

Chapter 5
Cost trade-offs when fixing flaky tests

This chapter describes the second iteration in our research endeavour, as expressed in subsection 2.2.4, focused on the costs and benefits of keeping flaky tests in a test suite versus removing them. As noted in section 4.3 the time requirement for fixing flaky tests is quite large. This begs the question of how beneficial a fix to a flaky test actually is. When taking the research into the practical domain, as mentioned in chapter 2, there needs to be a clear benefit to justify performing an action. The current research on flaky tests, as summarised in chapter 3, mainly focuses on different ways to fix tests that are flaky. If, however, there is a large cost associated with fixing the tests (e.g. in setting up a framework, investigating root causes, etc.) then this may not be practical in a real life situation.

As noted in section 3.5 one of the benefits of test automation comes from being able to rerun tests without the manual labor that a manual test would require. If a test case is flaky this benefit is diminished as the test will require human interaction. For example, someone needs to investigate the cause of the test failure and propose a course of action for the failing test case. We have found the following three strategies used by practitioners, through our interviews with Axis employees, to handle flaky tests:

- Keeping the test case in the test suite and fixing it (i.e. making it deterministic)
- Keeping the test case in the test suite but not fixing it (i.e. keeping it non-deterministic)
- Removing the test case from the test suite

Depending on which strategy is chosen a different set of costs and benefits will be incurred. The following sections go into the costs and benefits of employing the different strategies in further detail.

5.1 Costs and benefits of fixing flaky tests

One strategy to handle flaky tests is to fix them so that the output of the test case becomes deterministic. Through our interviews with Axis Communications employees we identified the following steps that were followed when fixing a test:

1. Investigating the cause of the problem
2. Implementing the fix
3. Verifying the fix

The implementation of the fix were often found to be quick whereas steps 1 and 3 were considered very time consuming.

In the interviews practitioners noted that investigating the cause of the problem is often the most time consuming action, in terms of working hours spent. This step involves looking at any accompanying logs for information about the cause of the problem, trying to reproduce the failure and finding where the problem lies (i.e. in the test code, the code under test or elsewhere). Fixing a non-deterministic test was considered harder than fixing a deterministic test. The reason given for this is that a non-deterministic test is generally harder to reproduce and so the root cause is often harder to find.

Implementing the fix was often considered a quick step by practitioners. Once the cause has been found the actual coding of the fix is generally done quickly. Some interviewees noted that this step may also depend on the cause of flakiness. In some instances the fix may be as simple as increasing the time of a timeout, whereas for more complex tests it may require code refactoring and more complex logic to be written. However, the general consensus was that this step is less time consuming compared to investigating and verifying the fix.

The main reason given in the interviews that verifying the fix is time consuming for a flaky test is that in order to verify that the test is now deterministic, a test case would need to be run many times. The time required to verify that a test is deterministic depends on a number of factors. For example, if a test fails 1% of the times (i.e. 1 out of every 100 runs), assuming a normal distribution of the failures, it would take an average of 100 runs before a failure has occurred. If we run this test, e.g., 10 times in a row there is still a 90% probability that the test will pass all 10 runs even if the test is still flaky.

Compare this to a test which fails 50% of the time. In this case the test would only take an average of two runs before we expect a failure to occur. If this test was to be run 10 times in a row there is only a 0.1% probability that the test will pass all 10 runs if the test is still flaky. This shows that the cost of verifying the fix increases when the flaky test has a lower failure rate as the test will need to be run less often thus decreasing the test execution cost. In other words, for a test that fails often, a proposed fix will have a low cost to verify whereas a test that fails rarely will have a high cost.

There is also a possibility that the fix did not completely eliminate the flakiness but instead made it fail less often (i.e. requiring more test runs to verify the fix). However, several interviewees noted that once the cause of the flakiness had been found and understood it is generally easy to fix the issue in a way that eliminates flakiness completely.

The cost of fixing a faulty test case will also depend on the cause of the problem. For some of the root causes listed in section 3.3 we found that the cost was very low. Some other root causes however, e.g. test order dependency, are harder to investigate as accompanying logs often does not give much information related to the cause and are harder to reproduce. Those tests will often take a significantly larger amount of time to fix.

The interviews we conducted lead us to three main findings in regards to fixing flaky tests.

First, the time taken to fix a flaky test (i.e. from the first failure to verifying an implemented fix) has a high variance. It was found that a fix could vary from a few hours to several weeks in terms of calendar time, depending on the complexity of the problem. The simplest fixes were considered to be timeout issues, where the most common solution was to simply increase the time before a timeout happens. The hardest fixes were considered to be the ones where the root cause was hidden (i.e. where the logs does not give much information about the problem).

Secondly, several of the interviewees mentioned that the time it would take to fix a problem was often predictable. Although it is often hard to give an exact estimate of a fix it was mostly easy to distinguish the easy problems (e.g. timeout issues) from the more complex. This is further reinforced by the findings reported in chapter 4, where the flaky tests still present in the test suite were hard to fix as the root cause was hard to identify and the failure was hard to reproduce. Presumably the tests in the test suite that were less complex to fix had already been fixed and implemented.

Thirdly, it was noted that it was possible to learn how to fix flaky tests more efficiently by doing it often. The more time one had spent fixing flaky tests the less time it took to implement a subsequent fix. This was mainly due to learning patterns in the logging information which gave hints to the root cause of the problem. However, it should be noted that this was mainly for problems where the root causes were similar. For complex and new failures the time taken for a fix was approximately the same no matter how many times one had fixed a different type of flaky test before. The conclusion we draw from this is that the cost associated with fixing a flaky test will decrease over time. However, for complex flaky tests the cost will stay approximately the same over time.

5.2 Costs and benefits of keeping flaky tests

As mentioned in subsection 3.4.1 a common way to deal with flaky tests is to keep them in the test suite and rerun them once they fail. If the test passes at least one of the reruns it is considered to have passed. If the reruns reaches the set limit of maximum reruns the test is considered to have failed and is handled in the same way as one would normally handle a failing test.

By keeping the tests non-deterministic there is an increased cost in the factors mentioned in section 3.5, i.e. test selection cost, test execution cost and result analysis cost.

5.2.1 Test selection cost

The test selection cost is the cost of selecting the tests that should be run (i.e. selecting inputs and outputs). A source of flakiness is that not all possible outputs of a test case has been considered.

5.2.2 Test execution cost

The test execution cost is the cost of setting up the system and the resources required to execute the system under test. The usage of machine resources and setting up the test environment would be examples of test execution costs. This cost varies a lot between products and companies and can be quite high for some applications [25].

The test execution cost will also depend on the failure rate of the test case. If a test fails regularly, e.g. once a day, the test will have a higher cost than a test that fails irregularly, e.g. once a month as it will require more reruns thus increasing the test execution cost.

5.2.3 Result analysis cost

If the test is set to automatically rerun at failure the result analysis cost may not be high. However, sometimes a flaky test might fail all of the reruns as well. In that case the result analysis will incur a cost as someone will have to look at the result manually in order to identify whether the failure is due to the non-determinism in the test or if it is an actual failure.

5.2.4 Other costs

Another problem with keeping flaky tests in the test suite without fixing them is that they may hide real defects. Luo et al. [29] report that if a test fails regularly practitioners tend to ignore it, and as a result might ignore real defects. In such case the value of the test being in the test suite is not very high as it does not convey a lot of information.

5.3 Costs and benefits of removing flaky tests

Another strategy to handle flaky tests is to simply remove them from the test suite. This eliminates the test flakiness but may introduce other costs. This section focuses on the costs and benefits associated with this strategy.

According to Kazmi et al. [20] 70% to 90% of new defects are found by manual testing. This points towards the benefit, regarding finding defects, of a single test case remaining in a test suite not being very high as a test suite can often contain hundreds of test cases.

A flaky test does not necessarily mean that the test case is the problem. Luo et al. [29] argue that in some cases the failure may occur because of a defect in the source code. In other instances the test case may be faulty but the code under test also contains a bug which is not discovered by other tests. In these cases removing the flaky test from the test suite would lead to the bug being undiscovered and hidden. In such cases a removed flaky test case would need to be replaced by a different test with the same test coverage, or with a manual test, in order to uncover the bug.

If there already exists a different test case covering the same feature or code under test then the test case does not provide much benefit and as such the cost for removing it will be low. If the test case needs to be replaced the cost would most likely correlate with the complexity of the test case as a complex test generally takes longer to write and is harder to verify.

There is also an option of removing a test case that is not covered by other tests but not replacing it. In this case the cost would be a lower test coverage. It is hard to quantify the value of test coverage. From our interviews the consensus seemed to be that a stable test suite is more important than a high test coverage. It was also made clear from the interviews that internal stakeholders often expect a certain amount of test coverage before doing a new release of a product and so going below that limit is not an option.

5.4 Discussion

There are clear costs and benefits associated with all three proposed strategies. In the case of fixing flaky tests, our research suggests that there is a relationship between the cost of fixing a test and its complexity. A less complex test case could take a few hours to fix whereas a test with a higher complexity could take weeks (i.e. tens of hours or more). This is also supported by our findings in chapter 4.

In the case of removing flaky tests from the test suite we also saw somewhat of a relation between the cost and the complexity of the test case. The reason for this is that there is a need to replace a removed test case in order to satisfy internal stakeholder demands on test coverage. The higher the complexity of the test case the more difficult and time consuming it will be to replace.

When employing the strategy of keeping flaky tests in the test suite we did not see a relationship between the cost and the complexity of the test. The general strategy in this case is to rerun the failing test until the test passes at least once. The highest costs of this strategy will be incurred from test execution as the tests will generally be rerun many times thus increasing the test execution cost whereas the test selection cost will only be incurred once, when selecting tests, and the result analysis cost will only be incurred in the event of a test failing all reruns.

It should also be noted that the higher the maximum number of reruns is of a test case, the lower the probability is that the test will fail all reruns, assuming a normal distribution of the failures. The interviews indicated that when employing the rerun strategy the probability of a test case failing all times is generally very low. Therefore the result analysis cost will in most cases also be low.

The test execution cost should not depend on the complexity of the test case as the same machine resources will be used for both a high and low complexity test case. As such a low complexity test case should have approximately the same cost as a high complexity test case and so our research suggests that the cost is not dependent on complexity when using this strategy.

Figure 5.1 shows a conceptual model of the cost of different test strategies compared to each other, based on the complexity of the test case. The intersections shows when it is a good idea to switch from one strategy to another. If the complexity of the test case falls in the area denoted with an A (i.e. low complexity) the best course of action is generally to fix the test. If the test falls inside the area denoted with a B the best course of action is generally to remove the flaky test and replace it with either a different automatic test or a manual test. If the test complexity falls in the area denoted with a C the best course of action is generally to keep the test case in the test suite and simply rerun the test when it fails.

It should be noted that the slope of the lines will vary depending on a number of factors,

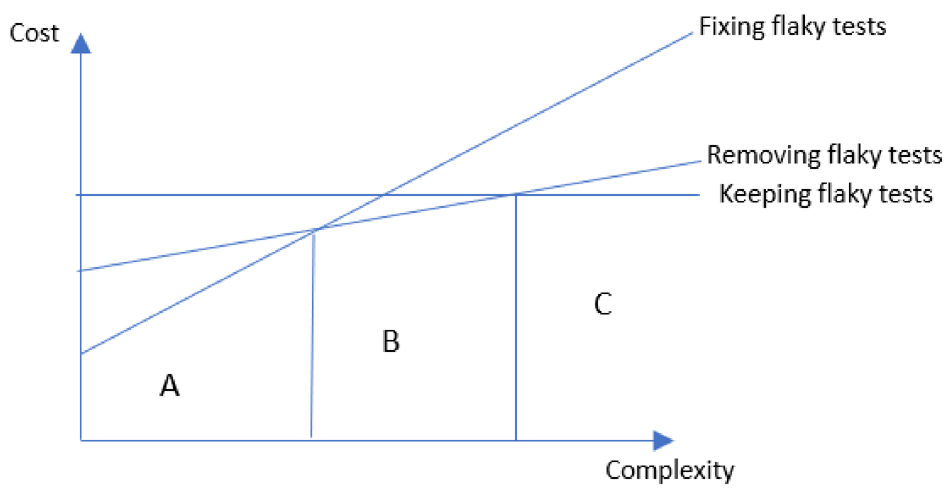

Figure 5.1: The three test strategies and their cost compared to the complexity of a test case.

e.g. the type of product and the cost of labour. It should also be noted that all three areas will not exist for all products and companies. For instance, the strategies of keeping flaky tests and removing flaky tests could intersect before intersecting with the strategy of fixing flaky tests which in turn would eliminate the B area completely, i.e. making the strategy of removing flaky tests redundant.

It should also be noted that this is a simplified model that only takes costs based on complexity of the test case into account. There may be other scenarios where the cost is more dependant on something other than the complexity. For example, if a test case is flaky but has a very low failure rate the better solution may be to keep the test in the test suite and rerun it once it fails. If the tests are run daily and the test case will fail once a month then the cost to handle the failure will be relatively low and so the best strategy may be to keep it in the test suite.

It may often be hard to determine exactly how the curve looks and what the optimal strategy is. Practitioners should however be aware of all strategies available and the general costs associated with them in order to make an informed decision.

Chapter 6
Guidelines for flaky tests

Previous research indicates that most flaky tests are flaky from the first time they are written [29]. This suggests that effort should be concentrated on new test cases to ensure that they are deterministic from the start. However, this will only diminish the risk of a flaky test entering the test suite. However, Harman and O'Hearn [15] suggest that it is inevitable for flaky tests to enter the test suite at some point and so it is also important to have a plan for how to handle flaky tests that are detected in the test suite. Further, when a flaky test has entered the test suite a preferred method is often to fix it.

Therefore, we present three sets of guidelines. One for mitigating the risk of a test case being flaky when writing it (i.e. before it is implemented in the test suite), one for how to handle flaky tests that are detected in a test suite, and lastly one set of guidelines for recommended implementation of test cases related to common root causes. The following sections explain the guidelines and any assumptions made and a condensed version of the guidelines can be found in appendix C, D and E.

6.1 Guidelines for avoiding non-deterministic tests

Generally, the cost to fix a defect in a software increases the further you get into the software development process [38]. This principle should also apply to flaky tests. If a flaky test enters the test suite, resources will need to be allocated in order to analyse the problem, investigate the cause of the problem and formulate a course of action. Therefore, there is a large benefit to be gained from finding and eliminating flaky tests early in the automatic test case development process.

Because of this, we believe that practitioners should be proactive, rather than reactive, and a big focus should be put on writing deterministic tests the first time the tests are written.

These guidelines are meant to guide practitioners in avoiding non-deterministic tests by avoiding common pitfalls related to non-determinism in tests. The following subsections explains the reasoning, as well as providing more context and further detail, about each guideline.

A summary of the guidelines and the source of the recommendations can be found in Table 6.1. The guidelines is also presented in appendix C.

Table 6.1: Guidelines for avoiding non-deterministic tests, and the source of the recommendation.

Guideline	**Source**
Be aware of the context	Interviews and literature review [11, 29, 40]
Minimise flakiness caused by the testing environment	Interviews
Avoid asynchronous implementation	Literature review [11, 29, 40]
Adhere to good coding practices	Literature review [11, 13, 14, 18, 28, 29, 40, 33]
Make tests explicit	Workshop
Keep test cases simple and test in isolation	Interviews
Strive for deterministic assertions	Literature review [15, 29, 31]
Limit third party dependencies	Interviews and literature review [11, 29, 40]

6.1.1 Be aware of the context

Through our interviews we found that network problems was considered a big cause for flakiness at the New Business test department at Axis Communications, although it has not been reported as frequently in previous empirical studies done on flakiness [11, 29, 40]. We believe that this is due to the nature of the software developed at Axis Communications, where there is a lot of network interaction. We believe that this means that the root causes for flakiness are quite general, but the probability of a certain root cause being present in a test suite will differ based on context. This is further reinforced by the findings of Thorve et al. [40] that found a number of root causes specific to the Android platform when studying flakiness in Android applications.

Our conclusion is that practitioners should be aware of the context of their products and a greater focus should be on minimising flakiness caused by root causes common in that context. For example, the studied company in this report produces products that are network intensive, which leads to more flakiness caused by, e.g., networking problems. By using good coding practices related to networking, found in appendix E, the risk of a flaky test entering the test suite will be greatly reduced.

6.1.2 Minimise flakiness caused by the testing environment

From our interviews we could see that the environment was a common cause for flakiness at Axis Communications. Network problems was one of the most common, but 4 out of 13 interviewees also noted other environment issues as a cause of flakiness. One example that was given by an interviewee was power outages in the testing environment.

Practitioners should strive to minimise flakiness caused by the testing environment by making sure that the environment is stable (i.e. being able to handle the testing requirements in terms of e.g. network load and power consumption). Any changes in the environment should be handled in a proper manner by, for example, throwing exceptions. Environment issues are not typically seen as a root cause of flaky tests as a power outage would affect all test cases equally and as such it is not the test case that is flaky but rather the environment. However, it will manifest itself in the same way as a flaky test (i.e. a test both failing and passing depending on if it can handle the load), and as such we believe that it is important to minimise variations in testing environment in order to ensure that any flaky behaviors are caused by the tests themselves and not surrounding factors.

6.1.3 Avoid asynchronous implementation

From the literature review we found that previous empirical studies done on flaky tests all found asynchronous implementation as the culprit for a large amount of explored flaky tests [11, 29, 40]. Our interviews indicate that asynchronous waits are a big problem at Axis Communications as well. The conclusions we draw from this is that practitioners should be careful with asynchronous implementations, and avoid them as much as possible.

From the literature review we found three fixes for flaky tests caused by asynchronous implementation; reordering the source code to make execution less asynchronus, using a waitFor mechanism, and adding a sleep. However, it was noted that sleeps should be used as a last resort and waitFor is the preferred mechanism of the two to reduce flakiness.

6.1.4 Adhere to good coding practices

To write deterministic tests it is important to avoid the common pitfalls of flakiness. Through the literature review we found a number of recommended fixes to common root causes of flakiness. These include, for example, using waitFor promises in order to mitigate flakiness caused by async wait and concurrency, and using resource pools to avoid flakiness caused by resource leaks. A full summary of the fixes can be found in appendix E.

We believe that by following these recommendations while writing the test code, one would be able to largely avoid flaky tests. By proactively adhering to the guidelines in appendix E, instead of applying them after the fact that a flaky test has appeared, we believe that it is possible to avoid a large amount of flaky tests early in the testing process.

6.1.5 Make tests explicit

From the workshop we found that many tests were not explicit in what they were testing. Although we could not prove that the tests were flaky because of this, it did bring up a discussion amongst the participants. The consensus was that tests that are not explicit have a higher tendency to be flaky, as it is harder to cover all possible outcomes.

Therefore, the tests should be specific in what it is testing. For example, if a test is sending a network request and verifying that the response from the server is the expected response, then the test might also implicitly test that there is a network connection. If this is not handled correctly the test may fail due to a different criteria than the one specified in the test. This could potentially lead to flaky behavior that is not related to the test itself. By making tests explicit, and handling errors caused by other factors correctly, a failure in the test will reflect a failure in what the test is supposed to test and not something else.

By making tests explicit it is easier to distinguish the cause of the failure and it also ensures that flakiness experienced in the test suite is related to the test case and not to e.g. a problem in the hardware.

6.1.6 Keep test cases simple and test in isolation

Several interviewees noted that a test case being too complex (e.g. trying to test many features at once or performing complex calculations that are hard to verify) was a cause they had seen for flakiness. The reasons given for this were mainly that having a test case that is complex reduces the readability and maintainability of the test case. We believe that complexity is not the root cause of flakiness, but rather keeping good coding practices is harder when the test case becomes larger and more complex which in turn increases the risk of flakiness.

It was also noted that when a test case is more complex the risk of a reviewer missing a fault in the test code becomes larger, as it is harder for the reviewer to understand what each part of the code does. Therefore, practitioners should strive to keep test cases simple and test features in isolation by dividing tests that test multiple things into several, smaller, test cases. This will help in both keeping flakiness away from the test suite, as well as making flaky tests easier to investigate and fix.

6.1.7 Strive for deterministic assertions

A common cause of flakiness is that not all outputs are accounted for. For example, flakiness caused by floating point operators could be caused by different machines calculating floating points differently, resulting in two different outputs [29]. A way to fix this is to make the assertion independent from the floating point result.

Similarly, one interviewee gave an example of a flaky test with a non-deterministic output. The test was testing if the bit rate was within a certain range when performing an action. The particular test sometimes passed and sometimes failed as the bit rate depended on the rate of the network. He noted that in his opinion the test case was not suited as a regression test, but rather a performance test that should be tested in a separate, dedicated, testing environment.

This means that practitioners should always strive to minimise the amount of non-determinism in assertions in order to avoid flaky tests. However, both Harman and O'Hearn [15] and John Micco [31] believe that a certain amount of non-determinism in test suites is inevitable and should be expected. For tests where it is not possible to produce an entirely deterministic output, it is important to ensure that all possible values are handled in assertions. This means making more relaxed assertions, accepting a wider range of values, and making sure that edge cases are covered. Timeouts should also be set in a way that ensures there is enough time to perform the feature under test.

However, one interviewee also noted that it is important not to relax assertions too much as this could lead to false negatives (i.e. the test passing when it should have failed). Therefore, we believe that assertions should be relaxed, but they should not be relaxed in a calculated rather than random way so that you do not introduce false negatives in the test suite.

6.1.8 Limit third party dependencies

Flakiness due to third party dependencies were a common cause of flakiness both in literature [11, 29, 40] as well as in our interviews. Third party dependencies are often hard to maintain and a change in the third party dependency may cause flakiness in the software. Therefore, third party dependencies should be kept to a minimum and any third party dependency should be properly documented so that the cause of the failure can be easily traced.

6.2 Guidelines for handling flaky tests

We propose a model for handling flaky tests that are in a test suite, which can be seen in Figure 6.1, expressed as a flowchart. The next sections describes the different steps in the flowchart in further detail.

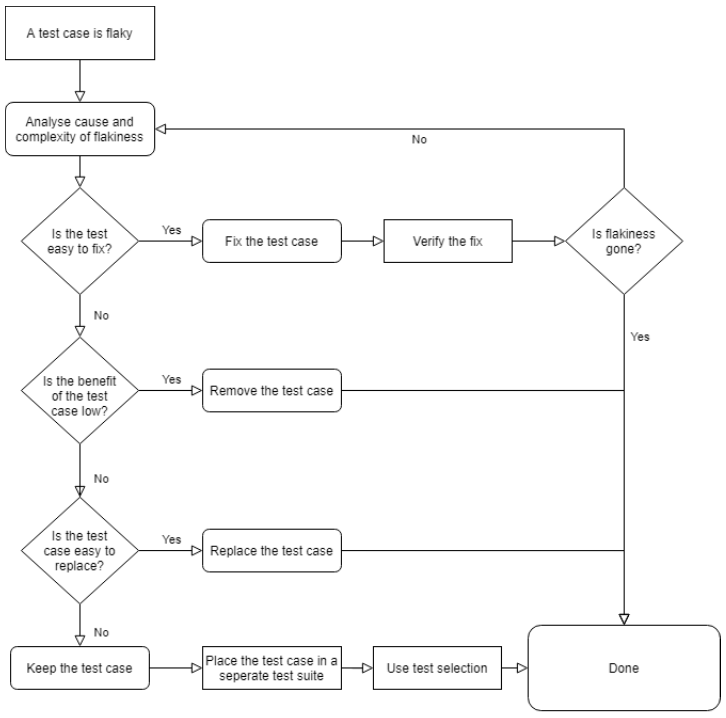

Figure 6.1: Flowchart depicting the recommended steps to be taken to handle flaky tests in a test suite.

6.2.1 Analysing the test case

In order to determine how to handle a flaky test case, the first step should be to analyse the test case. From the interviews, we gathered that it is generally easy for practitioners to judge how complex a problem is, and to estimate the time to fix. From the interviews we also gathered that the value of the test case is generally the determining factor on whether or not a test case can be removed from the test suite or if the test case is necessary in order to pass a certain acceptance criteria set by the product owners. When analysing the test case, practitioners should therefore take note of two things; the difficulty of fixing the test case (i.e. the complexity of the problem) and the value of the test case.

It is important to note that the time spent on the analysis of the test case should be limited. If a lot of time is spent on this step, the time savings of choosing the correct strategy is diminished as the time consuming action will be switched from handling the flaky test to analysing the best course of action. From our interviews the general consensus amongst practitioners seemed to be that judging the complexity of the issue is quite quick. From the interviews the general consensus appeared to be that for an easily fixed test, the fix could be quickly found and the time to fix was easily estimated. Therefore, if one is unable to judge the complexity of the problem quickly then it should be assumed that it has a high complexity.

6.2.2 Choosing strategy

The next step when handling a flaky test should be to determine the strategy that should be taken. Our research suggests three different strategies to handle flaky tests, as expressed in chapter 5, namely *fixing the test*, *removing the test* and *keeping the test*. Removing the test can be further divided into removing the test case and replacing it, and removing the test case without replacing it.

The findings in chapter 5 shows that a low complexity (i.e. “easy to fix”) test case should be handled by fixing it, while a high complexity test case should be handled by either removing it from the test suite completely or by keeping it in the test suite but using a rerun annotation.

If the complexity of the issue is found to be low, i.e. the test is estimated to be easy to fix, the test case should generally be fixed. For a test case that is complex the course of action should mainly depend on two things :

1. **The benefit of the test case.** The benefit of the test case is judged by how much value it provides. A high value test case would be one that covers features or parts of the code that no other test case in the test suite covers. Another example of a high value test case would be a test case which covers a feature required to be thoroughly tested in order to release the product. If the benefit of the test case is considered low the general course of action should be to remove it from the test suite as it does not provide much value.

2. **The difficulty of replacing the test case.** If the benefit of the test case is high and the test case is easily replaceable (i.e. with a low cost) by a deterministic test providing the same test coverage as a flaky test, it is generally the best course of action to replace it. However, sometimes the cost for replacing a test may be high. In such cases it is generally best to keep the test in the test suite but mark it as flaky.

The next steps will depend on the chosen strategy.

6.2.3 Fixing the test case

If the test case is deemed to be of low complexity, the test case should be fixed (i.e. made non-deterministic).

Eck et al. [11] identified a number of pieces of information that aids developers to fix flaky tests. When employing the strategy of fixing a flaky test it is important that this information is readily available, or that the infrastructure is in place to record the necessary information. The information deemed important by developers is:

- The context leading to failure
- The nature of the flakiness
- The origin of the flakiness
- The involved code elements
- The changes to perform the fix
- The context leading to passing
- The commit introducing the flakiness
- The history of this test's flakiness

By setting up a testing environment that continuously records the needed information it will be easier to gather the necessary information once a flaky test needs to be fixed. Again, we believe that it is important to have a proactive rather than reactive approach and so a testing environment should be set up in a way that automatically records these pieces of information.

Once the information is gathered, the tester should analyse the pieces of information in order to figure out the cause of the failure. From the interviews it was made clear that the steps taken when attempting to fix a flaky test case is different depending on the preference of the tester, but two strategies that we found to be the most common, and most helpful in fixing a flaky test case, was looking at the produced log information and trying to reproduce the failure if possible.

Once a probable root cause has been identified a fix needs to be coded and implemented. We recommend referring to the guidelines in test case implementation that can be found in appendix E for ways to fix flaky tests based on common root causes.

Verifying the fix

After the fix has been implemented the developer should verify that the fix actually eliminates the flakiness. Generally this is done by rerunning the test a number of times and verifying that the test passes all runs.

If the test is still flaky after the proposed fix is implemented, the test might be more complex than first expected. In this case, a new analysis should be made on how to handle the test.

6.2.4 Keeping flaky tests

If the test case is deemed to have a high value, and also a high complexity then the strategy should be to keep the test case.

A potential consequence of keeping flaky tests is that you risk incurring technical debt. The term technical debt was originally introduced by Ward Cunningham [10] and has come to mean a future cost that is incurred by using a simple, but limited, solution to a current problem [22]. The debt in this case is the cost of having to rerun failing tests. Because of technical debt it may seem contradictory to ever want keep flaky tests in the test suite. However, as flaky tests may be inevitable in certain software [31, 15], there may be cases where this strategy is necessary. If the cost of rerunning the test is low, and the test does not need to be rerun often (i.e. the failure rate is low), the cost may not be very high even across a long period of time. In such cases the strategy of keeping flaky tests in the test suite would still hold merit.

Keeping the test in a separate test suite

Flaky tests that are kept should be continuously evaluated so that the value of the test case has not decreased (and thus the test should be removed from the test suite), and that the cost of keeping the test is still low relative to the cost of fixing or replacing the test. It is also recommended not to keep flaky tests in the test suite responsible for acceptance testing [37]. We therefore propose to keep flaky tests in a separate test suite that is continuously monitored and evaluated in order to not interfere with normal operations.

When a test case annotated as flaky fails it should be rerun, until it passes or the maximum number of reruns is exceeded. There is little guidance in previous research on how many times a flaky test should be rerun or how they should be performed. Google reruns their flaky tests 10 times and does so outside of peak execution time [31]. Bell et al. note that reruns sometimes will need to be delayed in order for the cause of the failure to be resolved (e.g. a network outage) [7].

Using a test selection policy

If there are flaky tests in the test suite it is beneficial to employ a test prioritisation and test selection policy as it has been shown that employing such techniques are good at finding real failures while skipping failures that are caused by flakiness [9, 36].

Bell et al. [7] propose a method that analyses the statement coverage for a failed test run. If the test case does not cover any recently changed code, the failure is attributed to flakiness. We believe that by using a test selection technique that only select tests that covers newly changed code we would get the same effect as this method. By excluding those tests without changed code we would not need to rerun the test cases in the case of a potential flaky failure, and as such machine resources are saved.

6.2.5 Removing flaky tests

The strategy of removing flaky tests should be employed when either the flaky test does not provide enough benefit to warrant being in the test suite, or when the flaky test is easily replaceable. If the test does not provide any benefit to the test suite no further action needs to be taken after removing it. If the test does provide some benefit that needs to be maintained after removing the test (e.g. by providing test coverage necessary for releasing the product), the test should be replaced by a new test.

When writing the new test it is important to write it in a way that does not reintroduce the flakiness. To do this it is important to adhere to the guidelines for avoiding non-deterministic tests, found in appendix C. A guideline that we believe is especially important when replacing a flaky test case is *Keep test cases simple and test in isolation*. If it is possible, the flaky test case should be divided into several smaller tests with one assertion each, rather than one large test case with multiple assertions. This increases the probability of the flakiness being isolated to one test case, which will be easier to handle.

An important input we got through the interviews was that if a test case is divided into smaller test cases, then it is important to reset the state after a test case has been run. Files that were created should be destroyed, and variables should be reset to their default state. If this is not done there is a risk of introducing new flakiness caused by test order dependency.

After the test case has been replaced, the new test case(s) should be verified to not contain flakiness. This is generally done by rerunning the test a large number of times. If the new test case is also flaky a new analysis needs to be made.

6.3 Guidelines on test case implementation

In order to effectively fix a test case, it is important to know how to make the implementation more deterministic for different root causes. We therefore present a set of guidelines where various recommended implementations are presented for each root cause.

The recommendations are taken from the literature review, and are expanded upon in subsection 3.4.2. The guidelines on test case implementation serve both as an aid in fixing a test once a root cause has been identified, but can also help as recommendations when writing a test case for the first time. By keeping the common root causes, as well as the recommended implementations, in mind when writing a test case we believe that flakiness in many cases can be stopped before entering the test suite.

Chapter 7 Validation

This chapter discusses the validity of the thesis, actions taken to validate the work and any limitations used when performing the thesis work.

7.1 Validation interviews

In order to validate the results found in this report, we conduct interviews with four Axis QA Employees as the last step in our research endeavour, as explained in subsection 2.2.6. The interview guide used for the interviews can be found in appendix B. The employees being interviewed consisted of three test engineers and one expert in testing and had experience, to varying degrees, with, regression testing, UI testing, testing in continuous integration and automation of tests.

Overall, all four interviewees were positive when seeing the guidelines and thought the guidelines would have a positive impact in reducing flakiness at the company. All four also agreed that the guidelines should be realistic to implement at the company.

When asked about which guidelines were seen as the most important, the answers were varying. Three interviewees answered both *Minimise flakiness caused by the testing environment* and *Avoid asynchronous implementation* as most important. This is consistent with the common causes for flakiness at Axis, as presented in section 4.3, where asynchronous sleeps and network problems were noted as the most common answers for causes of flakiness by interviewees. Therefore, this result may be an effect of the context where the work was conducted.

Keep test cases simple, *Adhere to good coding practices* and *Make tests explicit* were also answered, but only by one interviewee each. From this we can see that several of the guidelines are considered important, however which ones are considered important may vary between roles

One controversial suggestion was found in mocking the network, suggested in e.g. the *Network* section of appendix E. Three of the interviewees saw risks in mocking the network, pointing out that there is a large cost associated with it in terms of, e.g., maintenance, whereas one interviewee argued that the benefits would most likely outweigh the risks in his case.

The interviewee who saw a benefit in mocking the network was working mainly with UI tests. He mentioned that mocking the network (or rather mocking the data that would be the result of the network call) is something he has proposed before, as he does not care about whether the network works or not, only the results. However, the test managers said no as they thought it was more important to test the entire system. As an example on how mocking could help, the interviewee mentioned a test run he had done the previous week where 80 UI tests out of 140 failed because the API had stopped working. Because of this a lot of resources were wasted that could have been saved if he had been able to mock the data instead of using the network.

The other three interviewees were more hesitant in mocking the network. One interviewee mentioned that mocking the network has a large cost associated with it in terms of maintainability as it is important to make sure that the mock is updated when the underlying resource that is being mocked is changed. However, he was slightly positive to mocking when things are harder to control, such as networks. One interviewee mentioned that there is a risk associated with creating a new dependency, and that from experience it is often cheaper to buy more hardware than to mock the network. The other interviewee did not go into closer detail.

This highlights the importance of having everyone on the same page in a team, and how management may impose a risk in implementing certain guidelines. It also shows that the benefits of specific guidelines may differ based on the role of the tester, or the type of test that is being performed. This balancing act was further reinforced during the interviews as all four interviewees mentioned that there are risks associated with the guidelines, and that there are cases where implementing a certain guideline may not be preferred. One example that was given was that the cost of setting up a stable testing environment may be too large making it unrealistic to completely eliminate flakiness from that source. It is therefore important to stress that there are exceptions to the guidelines.

All four interviewees agreed that the flowchart, depicting how to handle a flaky test that is already in the test suite, seemed reasonable. Two of them specifically mentioned that moving a flaky test case into a separate test suite is a good idea. Therefore we believe that the process described in the flowchart should be reasonable to follow in a practical context.

From the interviews we draw the conclusion that the guidelines should impact a company positively. We also note that there is a balancing act that needs to be considered. Sometimes the guidelines may not be feasible, or may carry a too large cost, in a practical context. However, we believe that the guidelines would still have a positive effect even when they can not be fully implemented, as it makes the testers think twice when implementing a test that has a high risk of being flaky.

7.2 Threats to validity

When performing research it is important for the research to have both internal and external validity. Internal validity refers to how other factors and variables may have influenced the results, whereas external validity refers to how generalized the results are (i.e. if they can be applied to other situations). This chapter discuss both the internal and external validity of the thesis.

7.2.1 Internal validity

This section discuss the validity of each step of the research endeavour, as explained in section 2.2. As a reminder, the following steps were taken to conduct the work presented in this thesis:

1. **Literature review.**
2. **Interviews.**
3. **Analyzing flaky tests.**
4. **Cost trade-offs for flaky tests.**
5. **Developing guidelines.**
6. **Affirmation of research results.**

During the literature review we followed a very systematic approach by first performing a database search using relevant keywords and subsequently finding more relevant literature through snowballing. During the database search we found the majority of the papers that are used in our work. We judged their relevance by first looking at the title, secondly reading through the abstract, and last reading the entire paper. When searching for literature through the snowballing technique we only found a handful relevant papers that we had not already read. We believe this was due to the database search being very extensive and thus covering most of the relevant literature. As such, we believe that we found most literature relevant to the work conducted in this thesis.

When conducting interviews the answers will naturally be subjective, which could be seen as a threat to the validity. We tried to alleviate this in a few different ways. First, we tried to follow the interview guide listed in appendix A as closely as possible for all interviews. This led to the interview answers being more easily comparable, and differences between answers could be found easier. If something was unclear we asked for clarification, or for the interviewee to elaborate on his or her answer. Second, we tried to interview as broad a range of roles as possible in order to get different perspectives on issues. We only interviewed people in roles which had a relevant technical opinion on testing. Testers, developers, test team leaders, and experts in different areas of expertise relevant to software testing were interviewed. Last, we tried to compare any given answers with relevant literature to validate the answers and find abnormalities. We therefore believe that the interviews are trustworthy enough to draw the conclusions that we did.

During the analysis of the flaky tests we encountered a few problems which may affect the validity. The data set we received was very large, and contained real test runs, which means that quantity wise it was a very good data set to use. However, the data set suffered from a lack of information which made it hard to analyse. The manual analysis of the flaky test cases also suffered from a few problems in terms of validity. As the test cases were taken from a real test suite it meant that the probability of a flaky test case being fixed increases the easier it is to fix it. As such, certain causes for flakiness might be more common when writing a test than is apparent when only looking at the flaky tests left in the test suite. Since certain causes might be easier to fix and as such those cause of flakiness are not accurately represented in the test suite. However, these problems were discussed in the relevant chapters, and as this approach was abandoned after we could not draw any conclusions based on the inadequate data this should not have had a big effect on the outcome of the thesis.

When performing the cost trade-offs analysis we mainly used data and conclusions from the interviews held earlier. The biggest threat to validity of this step is the shape of the diagram shown in Figure 5.1. As there was not enough data or previous research on cost models in regards to flaky tests we had to make assumptions on the shapes of the curves. However, this is also discussed in section 5.4, where we also note that the important takeaways of the chapter is not the shape of the curves, but rather the underlying analysis.

The guidelines were developed following the conclusions drawn in previous chapter. As no new data was used when developing the guidelines there should not have been any variables affecting the conclusions. We believe that the biggest threat to validity of this step were assumptions we made in order to produce the guidelines. However, the source of the guidelines as well as any assumptions made are clearly presented in chapter 6. Therefore, each guideline can be easily traced back to its source and as such we believe that this threat to validity is relatively small.

When affirming the research results we, again, used interviews to do so. Therefore, this step suffers from the same threats to validity as discussed earlier (i.e. answers being subjective). Further, the validation interviews also suffers a bias in that two out of four interviewees were also interviewed in the first interview. Ideally the validation interviewees should be different from the interviewees used to produce the results, as otherwise there is a risk of the interviewees validating their own answers. However, due to limitations in availability of participants this was not possible. Similar to the earlier interview, we preferred having interviewees from a range of roles in order to receive input from many different perspectives. However, unlike the previous interview we did not interview any developers, which might have an impact on the validity.

Overall, we believe that the internal validity is strong. As we did not overly rely on data and had a large number of interviewees the results of the thesis should be trustworthy. Any limitations we did have are discussed and taken into account in the thesis and as such we believe that we have minimised the threat of validity in the areas where it is possible.

7.2.2 External validity

As the time frame of the thesis did not allow us to evaluate the guidelines' effect on the company we were not able to confirm the effect the guidelines would have in practice. However, the validation interviews were overall very positive, which gives some insights into the validity of the guidelines.

All interviews were conducted with employees of the same company, which naturally creates a bias. This bias could lead to a lower generalisability. However, we tried to alleviate this bias by interviewing employees in a broad selection of roles. When conducting the interviews some employees also brought up examples from previous companies they had worked for and the problems encountered at Axis Communications and previous companies were often times similar. We therefore believe that this threat to validity is not very strong.

In addition, we also tried to find support for any findings in interviews through the literature study as well. This further reinforces the conclusions drawn in this report. The main findings that we could not find support for in the literature was the cost analysis, and the expenditure of time in regards to fixing flaky tests. As most interviewees agreed on the time it takes to fix a flaky test, and we did not receive any major critique during the validation interviews, we believe that the conclusions are still valid.

During the cost analysis, found in section 5.4, an assumption was made about the cost in relation to the complexity of a test case. We argued that there is a relationship between the cost of fixing a flaky test and the complexity as well as the cost of removing a flaky test and the complexity. This relationship may not always hold true, or may not always be linear, depending on context. However, we also discuss this in the chapter. We believe that even though this relationship may not always hold true, the simplified model is still a good enough basis to support the conclusions drawn from it in section 6.2.

Axis Communications works according to common software principles, which means that other companies working according to the same principles should be able to apply these guidelines in the same manner. There are some guidelines which may be more difficult in implementing in a different company. For example, limiting third party dependencies may not be possible for a smaller company that do not have the resources to code and maintain certain features. However, the guidelines presented in this thesis allow a certain degree of freedom and should be adaptable enough to work in such context as well. We therefore believe that the work and results of this thesis is applicable on other companies as well, and that the generalisability is quite high. We were also able to find support for most of our conclusions in previous research and literature, which also points towards the conclusions drawn in the thesis being generally applicable. However, due to the time limitations of the thesis we are unable to guarantee generalisability.

Chapter 8
Discussion

This chapter discusses the results found in this thesis as well as how the thesis can be expanded upon in future work.

8.1 Discussion of results

Our first approach to minimise flakiness was to find the causes and solutions to flakiness through fixing tests. This approach mostly failed, but we learned more about flakiness. Flaky tests are hard to identify and hard to find the cause of, and while relatively easy to implement a fix for a flaky test, the entire process of identifying a flaky test to the test no longer being flaky is very time consuming. With this information we postulated that guidelines, which took the time spent on flaky tests in account, would be very useful. Therefore we have presented a cost model comparing different strategies to handle flaky tests. The cost analysis resulted in a flowchart on how to handle flaky tests. We also presented two additional sets of guidelines meant to aid practitioners in reducing flakiness in a test suite.

We have arrived at the presented guidelines through a combination of interviews with employees at Axis and from conclusions drawn from related work. Our validation suggests that the guidelines will have a positive effect on the company, however this needs to be confirmed in future work.

Compared to related work, our report not only focuses on minimising flakiness in a single test case, it also address the balancing act between costs and minimising flakiness. Our guidelines view a test as being worth running if the benefits outweigh the costs. How exactly to measure the benefit is highly subjective and a task left for the practitioners. Our contribution to the research of flakiness is the combined overview of how to write deterministic tests, and a novel approach on how to handle flaky tests.

We believe that all three research questions we originally formulated has been answered.

RQ1, regarding common causes of flaky tests, were answered through the guidelines expressed in appendix E, where common fixes to the causes are also listed. The different categories of flakiness were taken from literature, and the categories presented in this thesis are the ones most commonly agreed upon. The most common categories in literature are Async Wait, Concurrency and Test Order Dependency. At Axis, Test Order Dependency was less common and network issues were a lot more common. This could be explained by guideline number 1: Be aware of the context. Axis' products are run over the network and the products themselves are more network intensive than average. Axis has also put in a lot of work to minimise flakiness, which might also explain why the most common causes of flakiness are slightly different, if they targeted certain types of flakiness more than others.

RQ2, regarding how flakiness can be minimised, was answered through the guidelines presented in appendix C. These guidelines were based on information from the literature review, from the interviews and from the workshop. Through the validation interviews we learned that most of the guidelines describe activities that experienced testers are already doing. Therefore the consensus among the interviewees was that the guidelines would be very useful for new testers to get up to speed and that the guidelines would have a positive effect on minimising flakiness.

RQ3, regarding how flaky tests should be handled, was answered through the flowchart presented in appendix D, and is explained more in depth in section 6.2. This flowchart was based primarily on the cost-benefit analysis and the interviews. The work flow at Axis is already very similar to the flowchart, and they claim to have relatively few problems with flaky tests. Probably the biggest difference is that testers might be more hesitant to throw out tests than our flowchart would suggest to be optimal. However, that is hard to know for certain since the benefit is difficult to measure. In summary, the people we interviewed for our validation generally agreed that the flowchart was useful and would have a positive effect on flakiness in the test suite.

The validation of the thesis was overall positive, however we could not evaluate the guidelines in practice due to time limitations. It should also be noted that the guidelines are now implemented at Axis and used by the QA Department, which also points towards testers finding value in using the guidelines.

8.2 Future work

We believe that there are many ways this thesis can be expanded upon. The first way would be to implement the guidelines in practice and evaluate them during a longer time period. This would also be a good way to see which guidelines has the biggest effect on flakiness, and the effort required to implement the guidelines.

It could also be interesting to further expand the guidelines on test case implementation to cover more cases, as research is furthered in the field, in order to get an even more comprehensive list on root causes and recommendations related to them.

In this thesis we have mainly focused on automated regression test suites. Flakiness is also present in other types of testing and an interesting topic would be to explore how these guidelines carry over into other types of test suites.

References

[1] Agile 101. `https://www.agilealliance.org/agile101/`. Online; accessed 5 June 2020.

[2] Azeem Ahmad, Ola Leifler, and Kristian Sandahl. Empirical analysis of factors and their effect on test flakiness-practitioners' perceptions. *arXiv preprint arXiv:1906.00673*, 2019.

[3] Joan E van Aken. Management research based on the paradigm of the design sciences: the quest for field-tested and grounded technological rules. *Journal of management studies*, 41(2):219–246, 2004.

[4] Nadia Alshahwan and Mark Harman. Automated web application testing using search based software engineering. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 3–12. IEEE, 2011.

[5] Android FlakyTest annotation. `https://developer.android.com/reference/android/support/test/filters/FlakyTest.html`. Online; accessed 1 July 2020.

[6] Deepika Badampudi, Claes Wohlin, and Kai Petersen. Experiences from using snowballing and database searches in systematic literature studies. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10, 2015.

[7] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. DeFlaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 433–444. IEEE, 2018.

[8] Anthony E Boardman, David H Greenberg, Aidan R Vining, and David L Weimer. *Cost-benefit analysis: concepts and practice*. Cambridge University Press, 2017.

[9] Benjamin Busjaeger and Tao Xie. Learning for test prioritization: an industrial case study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 975–980, 2016.

[10] Ward Cunningham. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

[11] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: the developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 830–840, 2019.

[12] Emelie Engström, Margaret-Anne Storey, Per Runeson, Martin Höst, and Maria Teresa Baldassarre. How software engineering research aligns with design science: a review. *Empirical Software Engineering*, 25(4):2630–2660, 2020.

[13] Martin Fowler. Eradicating non-determinism in tests. `https://martinfowler.com/articles/nonDeterminism.html`, 2011. Online; accessed 1 July 2020.

[14] Alex Gyori. *Proactively detecting unreliable tests*. PhD thesis, University of Illinois at Urbana-Champaign, 2017.

[15] Mark Harman and Peter O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23. IEEE, 2018.

[16] Douglas Hoffman. Cost benefits analysis of test automation. *STAR West*, 99, 1999.

[17] Mark Hughes. Interviewing. pages 264–274. Wiley Online Library, 2016.

[18] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo, Grigore Rosu, and Darko Marinov. Improved multithreaded unit testing. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 223–233, 2011.

[19] Jenkins RandomlyFails annotation. `https://javadoc.jenkins.io/component/jenkins-test-harness/org/jvnet/hudson/test/RandomlyFails.html`. Online; accessed 1 July 2020.

[20] Rafaqut Kazmi, Imran Ghani, Radziah Mohamad, Murad Tariq, Imran Sarwar Bajwa, and Seung Ryul Jeong. Trade-off between automated and manual testing: a production possibility curve cost model. *Int. J. Advance Soft Compu. Appl*, 8(1):12–27, 2016.

[21] Tariq M King, Dionny Santiago, Justin Phillips, and Peter J Clarke. Towards a bayesian network model for predicting flaky automated tests. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 100–107. IEEE, 2018.

[22] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE software*, 29(6):18–21, 2012.

[23] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: a study of java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 821–830, 2017.

[24] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. iDFlakies: A framework for detecting and partially classifying flaky tests. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 312–322, 2019.

[25] Hareton KN Leung and Lee White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance*, volume 91, pages 201–208, 1991.

[26] Jeff Listfield. Where do our flaky tests come from? *Online] https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html*, 2017.

[27] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, 2008.

[28] Qingzhou Luo, Lamyaa Eloussi, Farah Hariri, and Darko Marinov. Can we trust test outcomes? `https://pdfs.semanticscholar.org/a4b2/f4b9bcfdd0e83323570c40b893310f41e979.pdf`, 2014. Online; accessed 13 July 2020.

[29] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653, 2014.

[30] Kent McDonald. Agile Q&A: How is testing incorporated into agile software development? `https://www.agilealliance.org/agile-qa-testing-in-agile-software-development/`, 2020. Online; accessed 5 June 2020.

[31] John Micco. The state of continuous integration testing@ google. `http://aster.or.jp/conference/icst2017/program/jmicco-keynote.pdf`. ICST, 2017.

[32] John Micco. Flaky tests at google and how we mitigate them. `https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html`, 2016. Online; accessed 1 July 2020.

[33] Steve Neely and Steve Stolt. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *2013 Agile Conference*, pages 121–128. IEEE, 2013.

[34] Michael Quinn Patton. *Qualitative research & evaluation methods: Integrating theory and practice*. Sage publications, 2014.

[35] Per Runeson, Emelie Engström, and Margaret-Anne Storey. The design science paradigm as a frame for empirical software engineering. In Michael Felderer and Guilherme Horta Travassos, editors, *Contemporary Empirical Methods in Software Engineering*. Springer, 2020.

[36] August Shi, Peiyuan Zhao, and Darko Marinov. Understanding and improving regression test selection in continuous integration. In *International Symposium on Software Reliability Engineering*, pages 228–238, 2019.

[37] Managing test flakiness. `https://smartbear.com/resources/ebooks/managing-ui-test-flakiness/`. Online; accessed 1 July 2020.

[38] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. Technical Report RTI Project Number 7007.011, National Institute of Standards and Technology, 2002.

[39] David V. Thiel. *Research Methods for Engineers*. Cambridge University Press, 2014.

[40] Swapna Thorve, Chandani Sreshtha, and Na Meng. An empirical study of flaky tests in android apps. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 534–538. IEEE, 2018.

Appendices

Appendix A
Interview guide

1. What is your role at Axis Communications?
2. How and when do you notice that a test case is flaky?
3. How do you currently deal with flaky tests?
4. What is your opinion on removing a flaky test case from the test suite vs. trying to fix it?
5. What root causes have you seen for flaky tests?
6. How much of your working time is spent writing and fixing tests?
7. How much of your working time is spent on flaky tests?
8. Do you prefer a flaky test over a lower test coverage?
9. Do you prefer test cases that have a low complexity but that are more stable over test cases with a higher complexity but with a higher probability of being flaky?
10. How much flakiness have you experienced?
11. How much of a problem do you consider flakiness to be?
12. How has the amount of flakiness in the test suite progressed over time?
13. How many of the flaky tests that you have attempted to fix have you succeeded in doing so?
14. Do you have any questions, or something that you think is important to mention that has not been covered?

Appendix B
Interview guide - validation

1. What is your role at Axis Communications?
2. Are the guidelines something you are already thinking of when you are writing tests?
3. If you have investigated a flaky test for several hours, how would you normally move on?
4. What are your general thoughts on the guidelines?
5. Which of the guidelines do you find the most important in reducing flakiness?
6. What effect do you think the guidelines will have on Axis Communications?
7. Do you think the guidelines are realistic to implement?
8. Do you have any questions, or something that you think is important to mention that has not been covered?

Appendix C
Guidelines for avoiding non-deterministic tests

The following guidelines should be adhered to in order to decrease the risk of a non-deterministic test entering a test suite.

1. **Be aware of the context**
   Be aware of the context your applications are operating in and focus on minimising flakiness caused by root causes common in such context. For example, if the application is operating in a context where tests rely on files and resources being opened regularly then there should be a greater focus on making sure that such resources are handled correctly (e.g. closed after use).
   Example : If you are testing a piece of software with extensive file handling, then pay special attention to guidelines referring to resources and IO as there is a higher risk of flakiness being introduced from those sources.

2. **Minimise flakiness caused by the testing environment**
   Failures caused by the testing environment (e.g. power outages and network load) should be minimised as much as possible. Ensure a stable testing environment with as low variations as possible. Use mocks to have as much control over the environment as possible.
   Example : If the network load in your testing environment is large, minimise the strain on the network by using mocks or by investing in more hardware.

3. **Avoid asynchronous implementation**
   Asynchronous implementation should be avoided by reordering the code to make the execution less asynchronous. Replace sleeps with waitFor promises wherever possible.
   Example : Reorder the execution order of threads to make the execution less asynchronous.

4. **Adhere to good coding practices**
   Adhere to the guidelines in appendix E when writing tests that relates to a certain root

cause.
Example : If you are writing a test case that needs to make use of a sleep, then try to use a waitFor mechanism instead.

5. **Make tests explicit**
Ensure that the goal of the test is defined and specific. Ensure that the result of the test will reflect the goal of the test and that errors that are not related to the goal of the test are handled as errors and not as test failures.
Example : Specify each step of the test case and ensure that the test case will only cause a failure if the goal criteria is not met.

6. **Keep test cases simple and test in isolation**
Each test should ideally only test one thing. Strive to keep test cases small by dividing larger tests into several smaller tests. Avoid multiple assertions in a test case when possible.
Example : If a test case has several goal criteria that can be divided into separate test cases, then the test case should be divided into several smaller test cases.

7. **Strive for deterministic assertions**
Non-determinism should be minimised in assertions. Relax assertions to accept a wider range of values for test cases where non-determinism is unavoidable (but ensure that you do not introduce false negatives). Ensure that all possible outputs are accounted for, and that edge cases are properly handled. Ensure timeouts give enough time to receive a response.
Example : If there is a risk of two platforms producing different output based on, for example, how they handle floating point operations, then assertions should be relaxed to allow a wider range of outputs.

8. **Limit third party dependencies**
Keep third party dependencies to a minimum and ensure that any dependencies are properly documented so that the cause of the failure can be easily traced.
Example : If the cost of writing functionality that now comes from a third party library is small, then the third party library should be removed.

Appendix D
Guidelines for handling flaky tests

The following flowchart should be used to determine how to handle flaky tests that are present in a test suite.

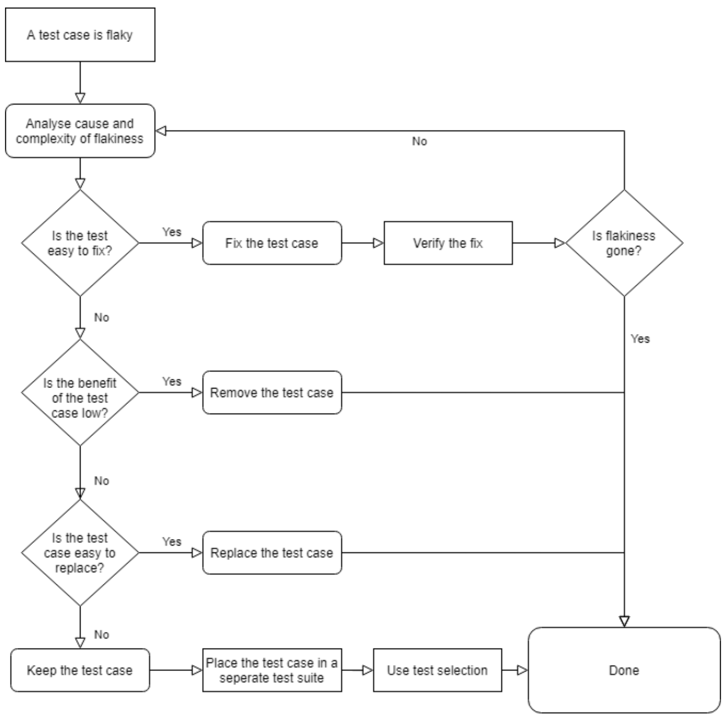

Figure D.1: Flowchart depicting the recommended steps to be taken to handle flaky tests in a test suite.

Appendix E
Guidelines on test case implementation

Table E.1: Guidelines on test case implementation.

Root cause	Recommended implementation
Async wait	• Use an *await*, *waitFor* or similar promise to wait for the required event. [13, 11, 29] • Reorder the source code to make execution less asynchronous. A delay can be achieved through executing other parts of the code while waiting, which makes it more likely that the wait is long enough.[11, 29] • Add a sleep, or increase the waiting time of an already existing sleep [11, 29, 40]. However, sleeps should be avoided or replaced by *wait-For* whenever possible as the use of sleep makes the test more unreliable, unintuitive and inefficient [13, 18].

Concurrency	• Use an *await*, *waitFor* or similar promise to wait for the required event. [11] • Add locks to ensure mutual exclusion [11, 29]. • Make execution deterministic by e.g. eliminating concurrency, enforcing a deterministic order between thread executions [11, 29, 33]. • Change concurrency guard conditions to ensure it includes all possible scenarios [11, 29]. • Change assertions to ensure all valid outcomes are accounted for [29].
Test order dependency	• Modify the output directory for the test to use a separate directory, in order to avoid conflicts between test cases [11]. • Ensure that shared states are setup before executing the test and ensure to clean up shared states after the test has been run [29, 11]. • Remove the dependency completely by making copies of the shared variable [29].
Resource leak	• Manage relevant resources through a resource pool. This way the first test to request a resource after ta resource leak, will fail. Which makes it significantly easier to find the problem [13]. • Destroy the conflicting object before continuing the execution [11].
Network	• Mock the network. Mocks can also be used to simulate, e.g., a network outage [29, 14] for tests that are meant to test for that. • Using waitFor, if one is unable to mock the network [29]. • Add connection retries [28].
Time	• Avoid, if possible, using platform dependent values such as time since time precision differs between systems [29, 13].

IO	• Close opened resources (e.g. files and databases) after use [29]. • Use proper synchronisation between threads that use shared resources [29].
Randomness	• Control the seed of the random number generator so that an individual run can be reproduced yet the seed can be varied across runs [29]. • Handle the boundary values, e.g. zero, that the random number generator can return.
Floating point operations	• Make assertions independent from floating point results [29]. • Relax assertions to accept a wider range of values (as floating point operations can differ depending on hardware) [28].
Unordered collections	• Write tests that do not assume any specific ordering on collections [29].
Too restrictive range	• Ensure that all possible outputs are accounted for [11].
Test case timeout	• Increase the time until timeout [11].
Platform dependency	• Make platform-specific test cases for the platform that exhibits flaky behavior [11].

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-08-27

EXAMENSARBETE Mitigation and handling of non-deterministic tests in automatic regression testing

STUDENTER Axel Berglund, Oskar Vateman
HANDLEDARE Per Runeson (LTH), Helena Razdar (Axis Communications)
EXAMINATOR Emelie Engström (LTH)

Minimering och hantering av programvarutester med odeterministiskta utfall

POPULÄRVETENSKAPLIG SAMMANFATTNING **Axel Berglund, Oskar Vateman**

Ett programvarutest körs två gånger på raken. Ena gången passerar testet, för att i nästa körning fallera utan att någon ändring har skett däremellan. Det kan låta som en omöjlighet, men faktum är att detta är ett stort problem för programvarutestare. I vårt arbete har vi tagit fram riktlinjer för hur man kan skriva tester för att få ett deterministiskt utfall samt hur man hanterar tester som är odeterministiska.

För att veta om en programvara fungerar som den ska så måste man testa den. Om man dessutom kontinuerligt uppdaterar programvaran så måste man också kontinuerligt testa den för att säkerställa att gammal funktionalitet fortfarande fungerar efter införande av ny funktionalitet. Detta brukar kallas för *regressionstestning* och är en viktig aktivitet vid programvaruutveckling. Regressionstesterna är ofta automatiserade då de exekveras ofta. Ett problem som länge gäckat programvarutestare är att dessa automatiserade regressionstester ibland kan ge olika resultat trots att ingen ändring av programkod eller tester har skett. Ett sådant test med odeterministiskt utfall brukar kallas för ett *flaky test.*

Ett *flaky test* är problematiskt då programvarutestare inte längre kan lita på sina testresultat, vilket i sin tur kan leda till att man inte kan lita på programvaran eller att buggar och felaktigheter smyger sig in i programvaran. Att minimera andelen *flaky tests* skulle därför göra testresultaten mer värdefulla och leda till mer tillförlitlig programvara.

Genom att intervjua testare, utvecklare, och experter inom programvarutestning har vi i vårt examensarbete försökt förstå vad som orsakar *flaky tests* och utifrån detta utvecklat riktlinjer för hur man minimerar andelen *flaky tests* samt hur man hanterar dem.

För hanteringen av *flaky tests* har vi tagit fram en flowchart där man beroende på bland annat testets komplexitet får rekommendationen att antingen fixa testet, ta bort testet, ersätta testet eller ha kvar testet oförändrat.

För att stödja dem som skriver tester tog vi fram åtta riktlinjer. Generellt säger riktlinjerna att man ska göra testerna och testmiljön mindre komplex, t.ex. genom att minska fel som uppstår p.g.a. nätverksanslutning eller att man vill minimera tredjepartsberoenden. Riktlinjerna har valideratats av testare, utvecklare och experter inom programvarutestning men vidare studier bör göras där man testar riktlinjerna i praktiken under en längre period.

Riktlinjerna bör leda till mer tillförlitliga tester inom programvaruutvecklingen, och i förlängningen även mer robust och tillförlitlig programvara.