Escola Tècnica Superior d’Enginyeria Informàtica

Universitat Politècnica de València

Research on the detection of test smells and flaky tests

Master’s Final Thesis

Máster Universitario en Ingeniería Informática

Author: Alejandro Vicent Micó

Tutor: Manuela Albert Albiol

External assistant: Shingo Takada

Year: 2022-2023

Abstract

This project is an empirical study that aims to research about test smells and flaky tests. Software testing is a key part of the development process of every project. During the implementation of test code, test smells arise, which are poor programming practices present in the test cases. Because of that, not only the effectiveness of the tests is affected, but also the maintenance and readability. In addition, some tests may have non-deterministic outcomes. These tests are called flaky and often lead to confusion and unreliable results.

In this study, tools for the automated detection of test smells have been examined and used, which allow developers to find test smells in their projects in an efficient way. The tools were executed over a set of projects with flaky tests and the results were compared with the flaky results for analyzing the test methods with flaky behavior and test smells. The most common smells were found out and the relation between smells and flakiness was considered.

The reached conclusion was that there is not enough evidence to support a strong correlation between test smells and flaky tests. However, some test smells might be related to other patterns that cause flaky results, such as asynchronous behaviors and connections to external resources. For easing the process of detecting and comparing test smells and flaky tests, an application was developed. As a result, the introduction of these practices is a bad habit that should be avoided, and some guidelines are provided in this regard.

Keywords: test smells, flaky tests, software testing, detection tools, programming practices.

Resumen

Este proyecto es un estudio empírico que pretende investigar sobre los malos olores en pruebas y las pruebas inestables. Las pruebas de software son una parte fundamental del proceso de desarrollo de cualquier proyecto. Durante la implementación del código de prueba, surgen los malos olores, que son malas prácticas de programación presentes en los casos de prueba. Debido a ello, no sólo se ve afectada la eficacia de las pruebas, sino también su mantenimiento y legibilidad. Además, algunas pruebas pueden tener resultados no deterministas. Estas pruebas se denominan "inestables" y a menudo provocan confusión y resultados poco fiables.

En este estudio se han examinado y utilizado herramientas para la detección automatizada de malos olores, que permiten a los desarrolladores encontrar estos olores en sus proyectos de forma eficiente. Las herramientas se ejecutaron sobre un conjunto de proyectos con pruebas inestables y los resultados se compararon con los resultados defectuosos para analizar los casos de prueba con comportamiento inestable y malos olores. Se descubrieron los olores más comunes y se consideró la relación entre olores e inestabilidad.

La conclusión obtenida es que no hay pruebas suficientes que apoyen una fuerte correlación entre los malos olores en casos de prueba y las pruebas inestables. Sin embargo, algunos malos olores podrían estar relacionados con otros patrones que causan resultados inestables, como comportamientos asíncronos y conexiones a recursos externos. Para facilitar el proceso de detección y comparación de los "malos olores" y las "pruebas inestables", se ha desarrollado una aplicación. Como resultado, la introducción de estas prácticas es un mal hábito que debe evitarse, y se proporcionan algunas pautas en este aspecto.

Palabras clave: malos olores, pruebas inestables, pruebas de software, herramientas de detección, prácticas de programación.

Resum

Aquest projecte es un estudi empíric que preten investigar sobre les males olors en proves i les proves inestables. Las proves de software son una part fonamental del procés de desenvolupament de qualsevol projecte. Durant la implementació del códi de prova, sorgeixen les males olors, que son males pràctiques de programació presents en els casos de prova. Per aquesta raó, no sols es veu afectada l'eficàcia de les proves, sinó també el seu manteniment y legibilitat. Ademés, algunes proves poden tenir resultats no deterministes. Aquestes proves es denominen "inestables" i sovint provoquen confusió i resultats poc fiables.

En aquest estudi s'han examinat i utilitzat ferramentes per a la detecció automatitzada de males olors, que permeten els desarrolladors trovar aquestes olors en els seus projectes de forma eficient. Les ferramentes es van executar sobre un conjunt de projectes amb proves inestables i els resultats es van comparar amb els resultats defectuosos per a analizar els casos de prova amb comportament inestable i males olors. Es van descobrir les olors més comunes i es considerà la relació entre olors i inestabilitat.

La conclusión obtinguda es que no hi han proves suficiente que suporten una forta correlació entre males olors en casos de prova i proves inestables. No obstant, algunes males olors podrien estar relacionades amb altres patrons que provoquen resultats inestables, com comportaments asíncrons i conexions a recursos externs. Per a facilitar el procés de detecció i comparació de les "males olors" i les "proves inestables", s'ha desarrollat una aplicació. Com a resultat, la introducción d'aquestes practiques es un mal hàbit que es deuria d'evitar, i es proporcionen algunes pautes en aquest aspecte.

Paraules clau: males olors, proves inestables, proves software, ferramentes de detecció, pràctiques de programació.

Table of contents

List of figures

1. Introduction

Nowadays, software plays a central role in shaping the modern world. The rapid advancement of technology and the increasing reliance on digital solutions have significantly transformed the way we live, work, and interact. From artificial intelligence (AI) to blockchain, many developments have been accomplished during the past few decades, leading to a dynamic and continuously evolving software industry.

But if we want to ensure the quality, functionality and reliability of the software products, we must test it. As can be seen in *Figure 1*, software testing is a crucial phase in the software development life cycle that aims to identify defects and ensure these aspects. It involves the systematic evaluation of a software application or system to verify that it meets specified requirements and works as intended.

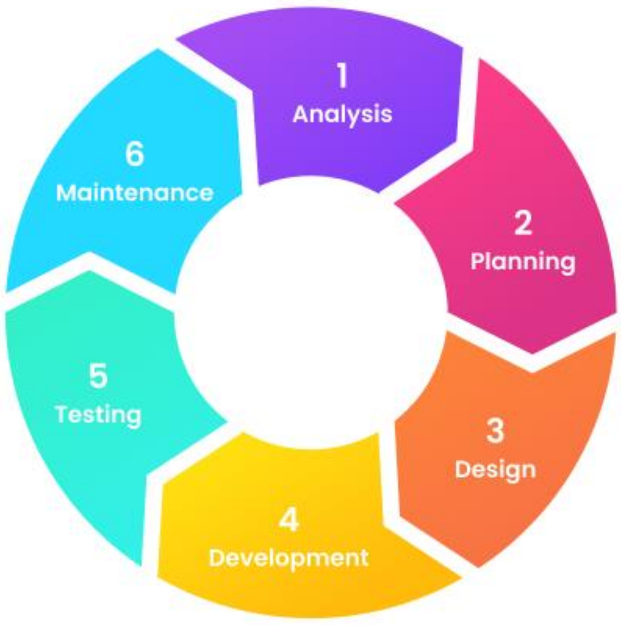


Figure 1. Software development life cycle

However, testing is not always performed in the best possible way. Test case quality is a critical aspect of the software testing process that directly influences the effectiveness and efficiency of testing efforts. High-quality test cases are essential for uncovering defects in the software and achieving accurate and reliable results.

This last aspect is the main point of this project. In this master’s final thesis, we focus on software testing quality. In particular, the focus is to study about the detection of test smells, flaky tests and the possible correlation between them. These particular details of software testing, which will be later explained in detail in section 3, are extremely important and should be taken into account when developing test code.

This final thesis has been developed as a result of the research done at Shingo Takada's software laboratory, in the Graduate School of Science and Technology, Keio University, Tokyo, Japan. Thanks to the Promoe scholarship, I have been able to spend one year as an exchange student at Keio University, where I could research at a laboratory and use the results to write this thesis.

1.1 Motivation

The main motivation of this project is to research about two key aspects of software testing. test smells and flaky tests, to learn about them, their detection and possible ways for avoiding and correcting them. Everyone wants to make a perfect software product from the beginning but that is not reasonable. After my years as a computer science student, I have realized that it is important to assume that some errors will appear during the different phases of the development.

When I first started programming, testing was not a part of the equation and, as a result, I encountered many errors in the later stages of the development. What do these errors mean? What is causing them? How can I correct them? These were some of the questions that arise in these cases. However, finding the answer is not easy. Without implementing tests, I ended up spending a lot of time reading the code I previously wrote (maybe weeks or months ago) to understand it and find the errors.

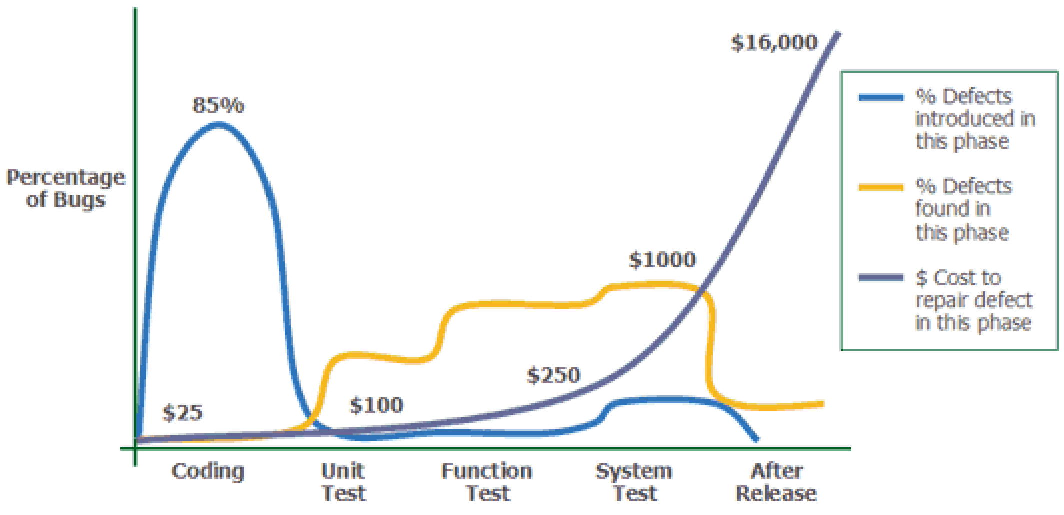


Figure 2. Cost of error repair

As we can see in *Figure 2*, the cost of finding and solving errors increase in the later stages of the development. There is more code to check, so more time has to be spent finding the errors, which means more money. Therefore, being able to detect the errors in an early stage and solve them before it gets too costly should be the path to follow when developing. This is the main objective of testing. In this regard, ensuring that the tests written are effective is very important, but some aspects, like test smells and flaky tests, make it difficult.

Ensuring that test smells and flaky tests do not appear in the testing code would improve the tests performance, but a manual detection is not feasible. In small projects with few test cases, a manual inspection of the tests in order to find test smells and flaky tests might be possible. However, in real projects with thousands of test cases, this is not possible. Instead, this process of detection has to be automated. After understanding the importance of these, I decided to focus on the detection of test smells, flaky tests and the relationship between them.

1.2 Justification of the subject

The quality of software, especially good programming practices, has been one of my main interests throughout my years as a student. Code smells are one of the most known issues regarding programming practices, so I got interested in them.

The automatic detection of code smells allows to save a lot of time when finding and correcting smells. Nevertheless, there has been a lot of research done about code smells. That is why I decided to focus on a specific type of code smells: test smells; that is, code smells in test cases. There are fewer articles about test smells, and even less about flaky tests. Then, I thought that it was a good opportunity to research about these topics, which are a recent concern in the last couple of decades.

1.3 Objectives

The main objective of this master's thesis is to research about the detection of test smells, flaky tests and the relation between them.

About specific objectives, the following ones have been established:

- Study about test smells to find out which test smells exist, which are the most common ones in existing projects and what is their definition.
- Investigate which are the current trends about the automatic detection of test smells and what techniques and tools are used in this regard.
- Research about flaky tests, what they are, what problems they can cause and how they can be detected.
- Find out which flaky test techniques exist nowadays, how they work and to what extent they can be used to automatically detect flakiness.
- Inquire if there is any correlation between test smells and flaky tests, to find out if there is a way of detecting flakiness using test smells as the source.

- Developing an application that helps me carry out the research, by being able to incorporate existing detection tools and use its results to graphically show the results so that they can be inspected and analyzed.

1.4 Structure of the thesis

Different sections will be presented throughout this memory, starting by an explanation of the methodology followed as part of the Promoe scholarship in an external university to carry out this thesis. In the next section, the state of the art will be introduced, with an explanation about code smells, test smells, flaky tests and current trends and tools about the automatic detection of these problems in software developing and testing. Conclusions about these matters will be mentioned at the end of the section.

After understanding the topics mentioned in the previous section and its importance, the problem of test smells and flaky tests will be analyzed and the possible solutions will be exposed. Then, a solution will be chosen, which Will be the path to follow in this research. In section 5, the proposed solution will be explained in detail, with the work plane expected, the design and the architecture of the system. In this section, some mock-ups and charts will be shown, which were done before the implementation of the solution to outline the shape of the solution.

Next, it is exposed the technologies used for the application development, as well as the reason why they were chosen and their impact in the development. In section 7, the implementation of the application will be documented, with an introduction to roughly explain the overall idea and different sections to explain in detail the functionality of the different parts of the application.

The next section is destined for validation, in which it will be checked if the work performed meets the expected results by performing some tests. Last but not least, in the final sections it is explained what conclusions can be obtained from the research done, the future projection and work to do and the final acknowledgements. Then, the bibliography with the references consulted, as well as the appendices with the abbreviations and acronyms used, and the slides employed for the final presentation.

2. Methodology

The purpose of this section is to explain in detail which has been the methodology followed during the time I have been working on this master's thesis. As I said in the introduction, I have been researching in a university outside from the UPV, so I think it is important to explain the path followed, as this is a case a bit special.

I received a scholarship by the Promoe program of the UPV to study abroad as an exchange student for one year at the Graduate School of Science and Technology, as part of Keio University, Tokyo, Japan. The methodology I followed in Keio was different from the one I would have followed in Valencia, so I am going to describe it in the following sections.

2.1 Research based

At Keio University and, as far as I know, at most universities in Japan, the bachelor's tesis and the master's thesis are more research based than in Valencia. This implies that first of all, you should read a lot of articles to understand the current state of the art and look for a possible topic for your thesis that is innovative and proposes some improvements to the current options.

At Keio University, to do research, you first have to find a professor that works on topics you are interested in, and then ask him if he allows you to join his laboratory. In a nutshell, joining a laboratory means that you are part of a research group about a certain sector. This group is formed by the professor and bachelor, master and PHD students. I was able to join the software and testing laboratory of professor Shingo Takada, where I started my investigation as an exchange student that would allow me to do my master's thesis for the UPV.

In this regard, I invested the first months in trying to find the best topic for my research. My initial goal was to research about good programming practices: refactoring, code smells, etc. My professor suggested me some topics related to his laboratory research topics, like fairness testing and code clones. Fairness tasting is based on finding discrimination towards certain individuals based on personal attributes such as race, gender or nationality. I read some papers regarding this topic [1], [2], [3] and found really interesting the things that were explained. Nevertheless, I was interested in the topic itself, but not in the techniques used to perform the study, based in machine learning approaches for testing. Also, I did not have much experience in machine learning, so I decided to drop this option.

For code clones, it was a topic much more related to good programming practices, which is what I was looking for. I also read many papers regarding this topic. Some of them are the following ones [4], [5] and [6]. Although I was interested in this topic, after a few weeks of discussion, I could not think of an innovative and new idea that could be a possible path to follow for my research, so I also decided to move on to a different topic. Finally, I started reading papers about code smells, which would be the starting point for my research, as I will explain in section 3.

As we can see in the timeline of *Figure 3*, the different phases of the research process are drawn. It gives an overall view of the topics I researched about to find the final topic for my thesis, from the ones I just mentioned in this section to the topics I will mention in section 3.

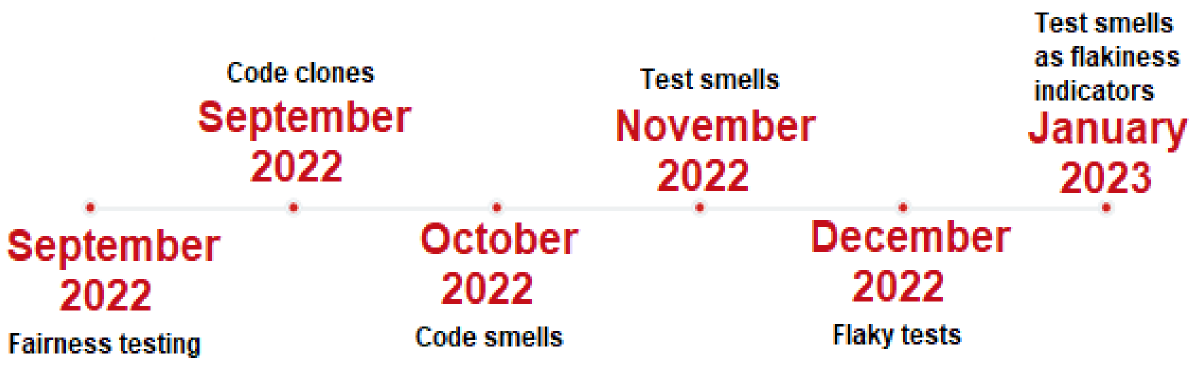


Figure 3. Timeline for the Research Process

2.2 Weekly meetings

The methodology followed in the laboratory is based on a meeting cycle that is repeated weekly throughout the year. Every week, two meetings take place. Even though you have to attend both meetings, you are able to decide which of these two days you want to make your presentation. The day you are not presenting, you hear to everyone else's presentation and give feedback whenever appropriate. The day you are presenting, you prepare a set of slides and explain to the professor the progress you were able to do during the whole week.

This methodology, though strict, helps you keep consistency among weeks, pushes you to work and allows a direct meeting with the professor, that gives constructive feedback every single week and helps you plan the next steps. First, I presented about the different papers I read and talked with the professor about what else should I investigate and how to proceed. Once I finally selected my thesis topic, I started making presentations about how I was progressing, which results were obtained and the things I had already implemented. The professor was always willing to help me and give me his opinion on my work.

3. State of the art

The objective of this section is to evaluate the current state of the art; that is, the highest level of advancement regarding test smells, flaky tests and its detection. The state of the art is continually evolving as new developments are made, and advancements are achieved. To know the specific state of the art in this specific field, I referred to many papers published the last years about code smells, test smells, flaky tests and its detection in software code.

3.1 Code smells

As I previously mentioned in section 2.1, I started researching about code smells before I decided to focus on test smells. It is important to know the definition and features of code smells, as well as the difference between code smells and test smells, before the state of the art about test smells is explained.

According to the paper published by José Amancio M. Santos et al. [7], code smells are potential problems in the design of software. The word potential is used because code smells themselves might not be the cause of errors, but they are initial indicators of possible future problems that not only affect the efficiency of the code, but also its readability and maintenance.

From the 1990s, the problems during the design of software started gaining importance and strategies have been discussed to deal with these problems. In 1999, Martin Fowler [8] published a book focused on refactoring, a technique used to improve the design of existing code. According to the following page [9], refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. In other words, refactoring means that some code that has been previously written is modified and written in a different way without changing what it does, its functionality.

In his book, Fowler adopted the name “code smell” to refer to these problems in the design of software. He paid special importance in the question “When should you start refactoring?”, and not only on “How should you refactor your code?”. He said that when to start refactoring and when to stop is just as important to refactoring as how to operate the mechanics of a refactoring.

In this regard, he wanted some clear steps that would help people understand when refactoring is needed and should be made.

Because of this reason, he elaborated a list of smells: a total of 22 code smells, with their characteristics and specific actions to deal with them. In *Figure 4*, the list with the code smells proposed by Fowler can be seen.

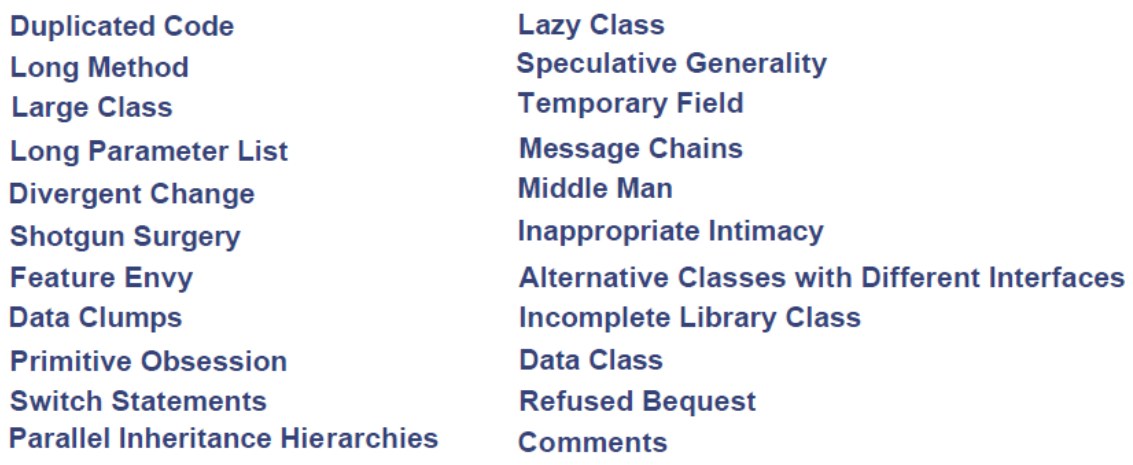

Duplicated Code
Long Method
Large Class
Long Parameter List
Divergent Change
Shotgun Surgery
Feature Envy
Data Clumps
Primitive Obsession
Switch Statements
Parallel Inheritance Hierarchies
Lazy Class
Speculative Generality
Temporary Field
Message Chains
Middle Man
Inappropriate Intimacy
Alternative Classes with Different Interfaces
Incomplete Library Class
Data Class
Refused Bequest
Comments

Figure 4. List of Code Smells proposed by Fowler

However, even though these smells, as well as other poor programming practices have been known for the past decades, and developers understand that they should be avoided, the reality is different. Nowadays, software systems become more and more complex and companies, in order to be competitive, have to continuously update their code within time limits [10]. This leads to a development team more focused on meeting these time limitations by writing code in a short time rather than taking into account good programming practices.

Code smells are not easy to find and correct manually, and as developers can't afford the time to focus on them, they leave them as they are, provoking some technical debt. Technical debt, which is explained in the work by W. Cunningham [11], can lead to an uncontrollable program and an inflexible product, when in excess quantities.

Because of the number of resources that have to be used in order to find these poor programming practices, researchers have studied about the possible implementation of an automated way of finding code smells. To achieve this, several code smells detection tools have been developed, that can automatically find smells in the code by using various techniques. In the paper published by Eduardo Fernandes [12], a study was done about the main characteristics of 84 different tools.

According to this study, the techniques used to discover code smells in these tools are based on metrics, abstract detection trees (AST), textual analysis, Program Dependence Graph (PDG) or token analysis. Out of the 84 smell tools, 31 are metric-based, 15 based on ASTs, 6 on textual analysis, 5 on PDGs and 3 on token analysis. In addition, 11 tools are based on machine learning techniques and the remaining 17 tools did not provide any information about the technique they used.

Also, some other information was provided, such as the programming language of the projects that the tools are able to analyze, the programming language that was used to develop the tools and the type of code smells that can be detected by using the tools.

First of all, and as we can see in *Figure 5*, the top ten smells that can be detected by the tools are shown. It can be observed that the smells Duplicated Code and Large class are the most common ones among the tools, followed by Long Method and Feature Envy. This high percentage might be because of the importance of these smells in order to write clean code.

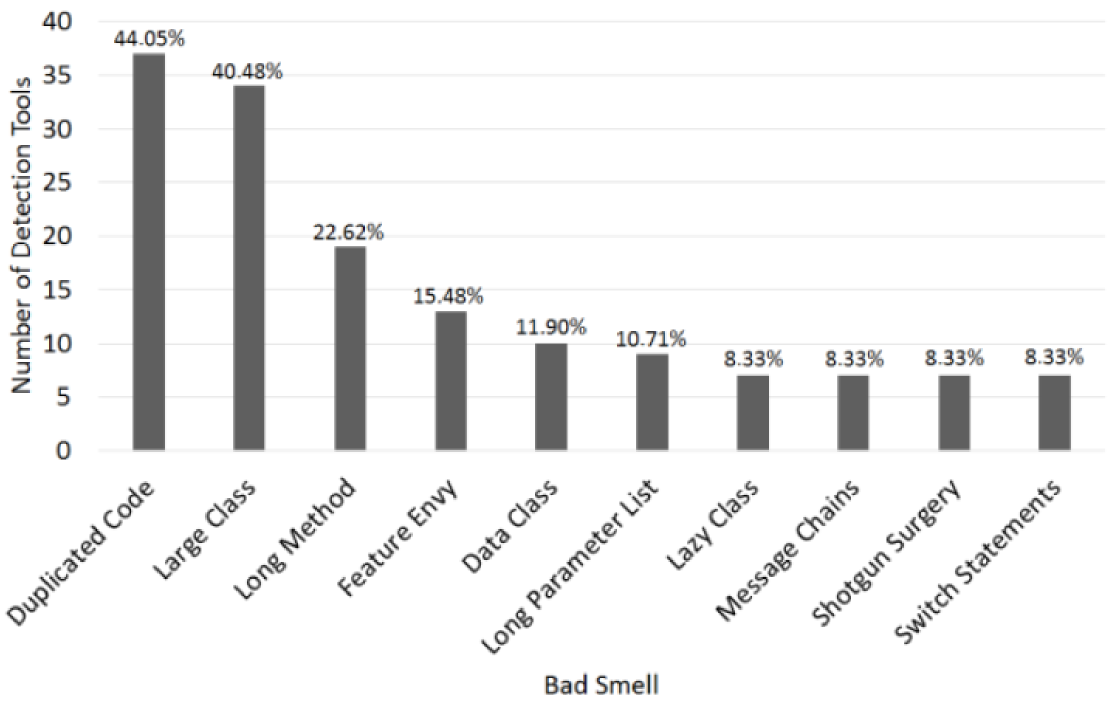


Figure 5. Most Common Code Smells that the Tools Can Detect

In addition, *Figure 6* shows the programming languages used to develop these tools. We can observe that Java is by far the preferred language when thinking about developing a code smell detection tool, followed by C and C++.

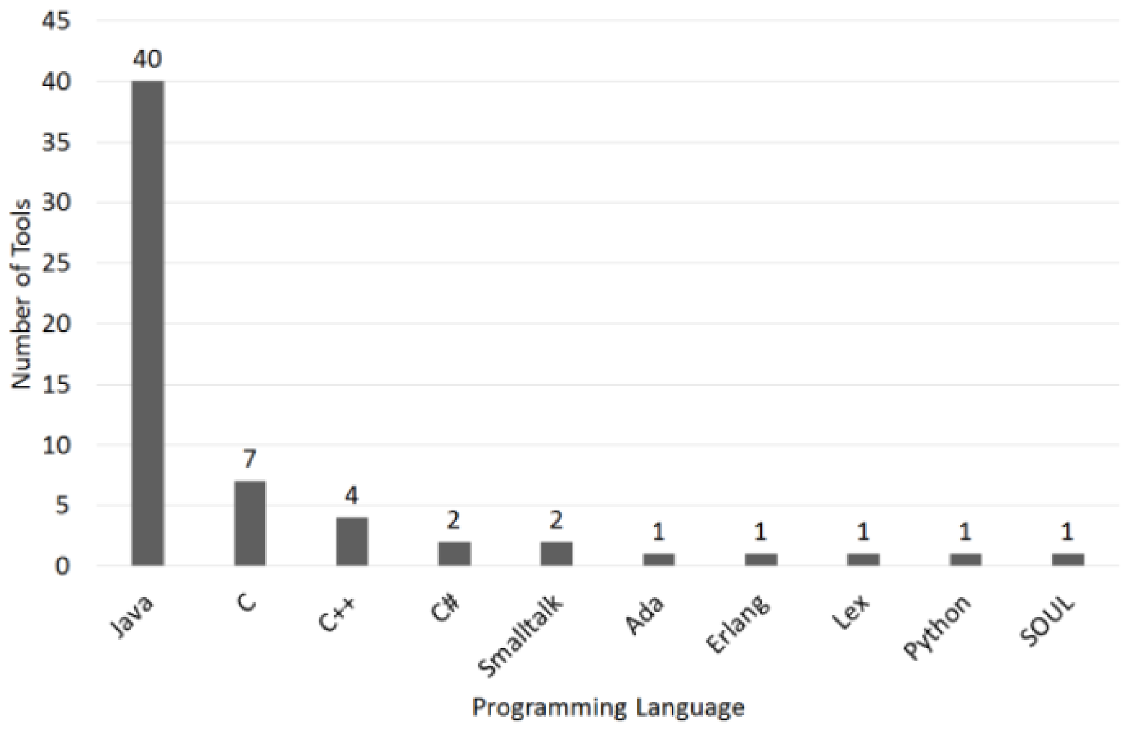


Figure 6. Programming Language Used to Develop the Tools

And in *Figure 7* we can see the programming language of the projects these tools aim at finding code smells. Java is again the predominant language, followed by C, C++, Python and C#. It is clear that Java is the easiest way to develop these tools, and that the most predominant languages are well-known ones.

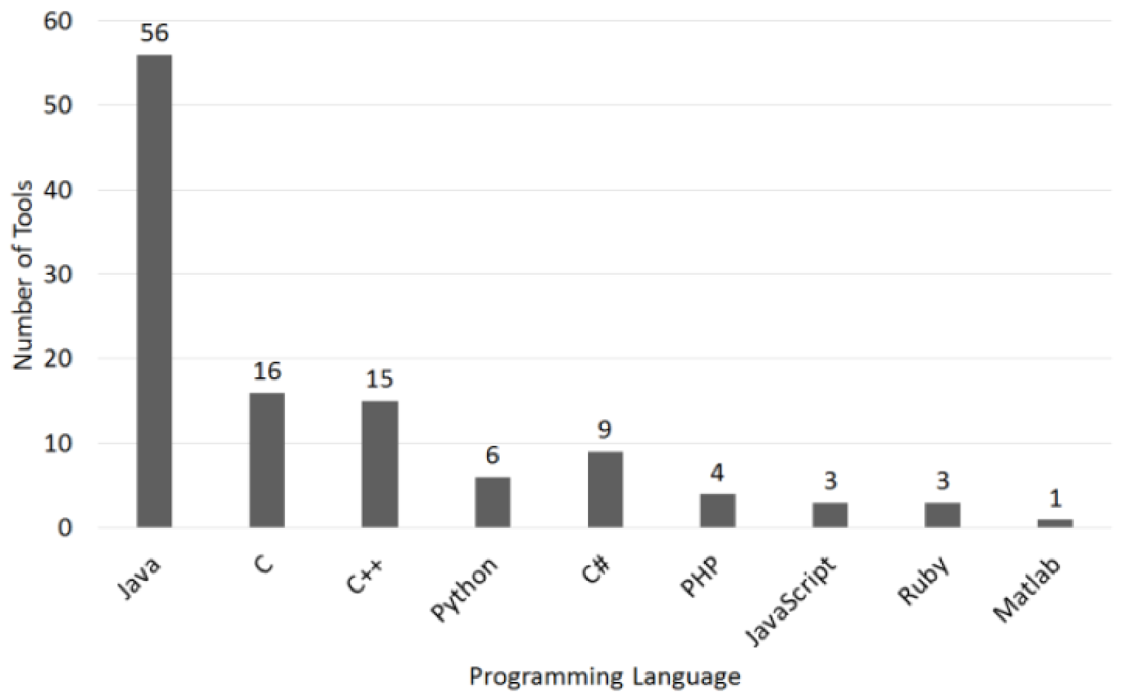
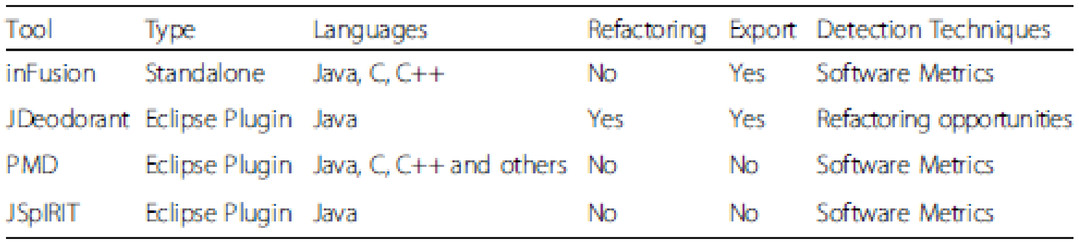


Figure 7. Programming Language that the Tools Can Analyze

Finally, and according to the paper by Thanis Paiva et al. [13], some of the code smell detection tools more known and used by developers are JDeodorant [14], inFusion [15], JSpIRIT [16] and PMD [15]. We can see some features of these tools in *Figure 8*. They can all detect test smells in Java programs, are free to download and their output allows to calculate recall, precision and agreement.



Tool	Type	Languages	Refactoring	Export	Detection Techniques
inFusion	Standalone	Java, C, C++	No	Yes	Software Metrics
JDeodorant	Eclipse Plugin	Java	Yes	Yes	Refactoring opportunities
PMD	Eclipse Plugin	Java, C, C++ and others	No	No	Software Metrics
JSpIRIT	Eclipse Plugin	Java	No	No	Software Metrics

Figure 8. Common Code Smells Detection Tools

A possible option for this thesis was to develop a machine learning approach that could be innovative in some way to exceed the results from already existing tools. However, there has been a lot of research done about code smells detection tools based on machine learning [15], [17], [18], and there is no way to know if the newly developed approach will improve the performance of the existing tools until it is finished and tested. As a result, this path was discarded.

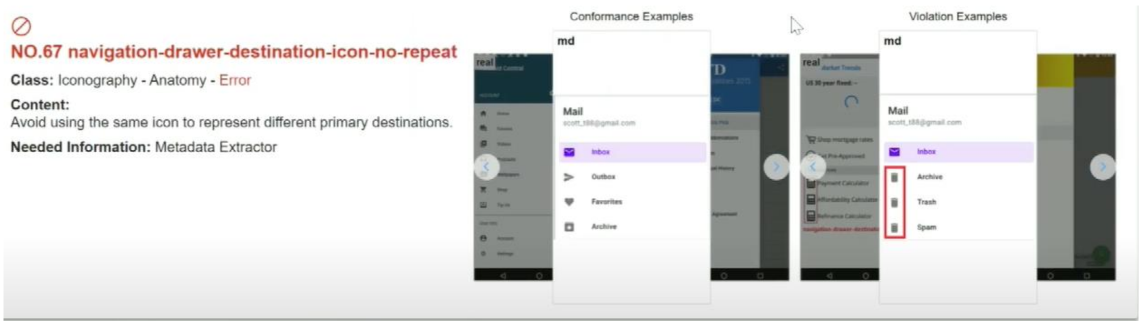
Another possible option was to develop a tool that could detect smells for a project written in a different language than the ones existing tools can analyze. But at the same time, the programming language should be known and used to some extent for the tool to be useful. After reading the above papers and realizing that so many tools were done for detecting smells in Java programs, C and its derivates, if the tool was at least for another language, it could be a good option, as there is not much competence in other languages.

Then, I realized that none of these papers talked about the devices that we most use and carry with us the whole day: mobile phones and its programming language, Android. The improvements in mobile technology and the large number of applications developed for Android in the last years are clear signs of the importance of software in these devices. These applications evolve faster than regular software applications, as deadlines are marked to meet the new requirements from the customers and rapidly fix possible bugs. This evolution may cause the introduction of bad design practices, in particular code smells.

Because of the increasing popularity of software apps for Android, I decided to make some research about the detection of code smells for this programming language. Although there is not as much research done as for desktop programming languages, I could find papers related to the detection of normal code smells in Android applications [19], [20]. In these cases, there is not much innovation rather than changing the language where the smells are found to Android.

On the other hand, some approaches focus on detecting Android-specific code smells [21], [22], [23], [24], [25], [26]. Although there is little knowledge on these types of smells, the tools just mentioned were already able to achieve a pretty high performance, so I also decided not to focus on this kind of smells for my study.

I also looked at some tools for detecting UI design smells. I read the paper by Bo Yang [27] about this topic. They developed a tool, named UIS-Hunter, for Android applications that had a database of guideline knowledge, processes the input UI, extracts the information from this UI and highlights the violated guidelines and provides useful corrections. We can see its behavior in *Figure 9*.



NO.67 navigation-drawer-destination-icon-no-repeat

Class: Iconography - Anatomy - Error

Content:
Avoid using the same icon to represent different primary destinations.

Needed Information: Metadata Extractor

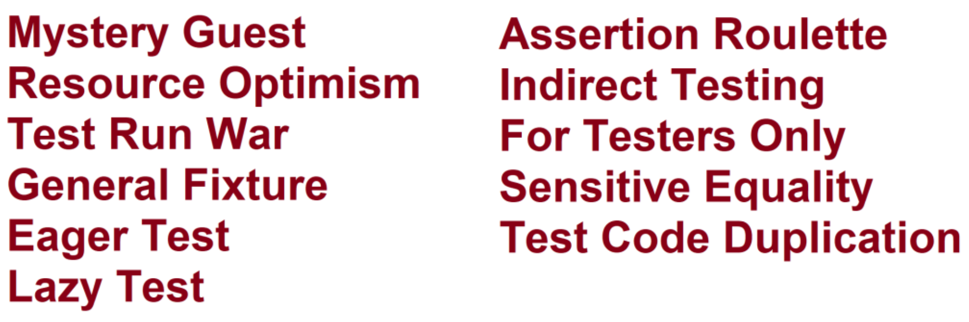
Figure 9. Behavior of the Tool UIS-Hunter

Nevertheless, even though I found this idea interesting, due to time constraints of this project and the complexity of a UI smells detection tool, I decided not to develop this kind of tool as the result of my thesis. Finally, I realized that there were only a few proposals regarding test smells, so I wanted to research about them, as I will explain in section 3.2, and I finally chose them as the topic of this thesis.

3.2 Test smells

It is known that good programming practices should be applied when coding but, as previously said in section 3.1, in practice sometimes it is not possible. That is why the code is revisited to look for errors and to be refactored. In the set of code smells proposed by Fowler [8], strategies are explained to remove these smells and refactor the code. However, he does not mention anything about the refactoring and application of good practices in test code.

Some papers talk about the importance of applying good programming practices to the test code [28], [29], [30] and finally, code smells in the test cases were introduced. It was done in the work of Arie van Deursen, called "Refactoring Test Code" [31]. According to him, refactoring test cases is different from refactoring production code. Tests are organized and implemented in a different way, and so a new set of test smells arise. The list of smells he proposed is the one shown in *Figure 10*.



Mystery Guest
Resource Optimism
Test Run War
General Fixture
Eager Test
Lazy Test

Assertion Roulette
Indirect Testing
For Testers Only
Sensitive Equality
Test Code Duplication

Figure 10. List of Test Smells by A. Deursen

And the explanation for every one of them, as well as the strategy used for removing them:

- **Test Smell 1. Mystery Guest:** A test method containing object instances of external resources such as files and databases classes. It is hard to understand the functionality of the test and hence use it, because there is not information about these extra resources. In addition, more dependences are added to the test, as modifying the external resources (file deletion, location change, etc.) might produce problems in the test, causing failures. Apply strategy 1 Inline Resource or strategy 2 Setup External Resource.
- **Test Smell 2. Resource Optimism:** A test method utilizes an instance of a File class without calling the method "exists()", "isFile()" or "notExists()"of the object. In this way, it is assumed that some external resources like files, folders or databases are available (or not). This might produce non-deterministic undesirable results, where the test succeeds and fails in different reruns. Apply strategy 2 Setup External Resource.
- **Test Smell 3. Test Run War:** Even though the test works if a single developer runs it, if multiple developers are running the test at the same time, it fails, mostly because there is some interference in the resources used by the test. Apply strategy 3 Make Resource Unique.
- **Test Smell 4. General Fixture:** When not all the fields instantiated within the "setup" method of a test class are utilized by all test methods in the same test class. If this method contains variables that are too generic and only some of them are used by the test cases, it makes thinks more difficult to understand. In addition, extra instructions can slow the performance of the tests. Use Fowler's Extract Method, Inline Method or Extract Class.
- **Test Smell 5. Eager Test:** When a test case contains multiple calls to different production methods. It lowers the readability and understandability, making documentation harder. It also adds dependencies among tests, affecting the maintainability. Apply Fowler's Extract Method to solve this.

- **Test Smell 6. Lazy Test:** When multiple test cases call the same production method using the same strategy to check it, even though they check different variables or resources. Apply Fowler's Inline Method.
- **Test Smell 7. Assertion Roulette:** It happens when a test case contains more than one assertion statement without an explanation (meaning the parameter that acts like a message in the assertion method). In case one of the assertions fails, there is no way to know which one was. Use strategy 5 Add Assertion Explanation to solve it.
- **Test Smell 8. Indirect Testing:** When a test case is actually testing other methods rather than the method it is supposed to be checking. It can be solved by applying Fowler's Extract Method and Move Method.
- **Test Smell 9. For Testers Only:** Observed when a production class has methods that do not do any specific functionality, but they are only being tested by the test cases. This means that these methods are useless and can be removed. One possible solution is to apply Fowler's Extract Subclass.
- **Test Smell 10. Sensitive Equality:** When a test method invokes the "toString()" method of an object. Even though the check through the "toString()" strategy can be seen as a simple solution, it causes problems when the method is changed, as it starts failing because it depends on irrelevant things, such as spaces or commas. Apply the strategy 6 Introduce Equality Method in this case.
- **Test Smell 11. Test Code Duplication:** When programming, sometimes the strategy of copy-pasting code is used. This can lead to undesirable situations and extra dependences later on. A possible solution is to apply Fowler's Extract Method.

In addition, van Deursen proposed a number of strategies to refactor test code without removing test cases and making the code more readable and maintainable:

- **Strategy 1. Inline Resource:** A variable in the test code is added to hold some external resource, so that the dependency between the test code and the external resource is avoided. For example, a variable could hold the contents of a file that will be used later.
- **Strategy 2. Setup External Resource:** Fix the test so that it correctly allocates the external resources that will be used (files, databases, etc.) and release them after using them.
- **Strategy 3. Make Resource Unique:** Make sure that unique identifiers are used for the different resources of the tests.
- **Strategy 4. Reduce Data:** Use only the essential data in the tests, which will improve the maintainability and documentation.

- **Strategy 5. Add Assertion Explanation:** Use the message in the optional first argument of the tests to note the difference among the assertions in the same test.
- **Strategy 6. Introduce Equality Method:** Add an “equals” method to an object structure that has to be checked in the tests, so that this method can be rewritten in the tests for its specific purpose.

After the work of van Deursen, some other papers proposed additional test smells, increasing the catalog of smells available [32], [33], [34], [35], [36], [37], [38]. Then, each researcher or developer can select the set of test smells he is interested in studying or using for their research or project/tool.

3.3 Test smells detection tools

Researchers all over the world have realized about the importance of avoiding these sub-optimal design choices that are called test smells and hence some detection tools for the automatic detection of test smells have been developed. In the work done by Wajdi Aljedaani et al. [39], a list of 22 test smells detection tools was constructed by reviewing many papers related to this topic. In this, list some aspects of the tools are analyzed, like the programming language, the detection strategy or the number of smells they are able to detect.

The tools taken into account in his study can be seen in *Figure 11*:

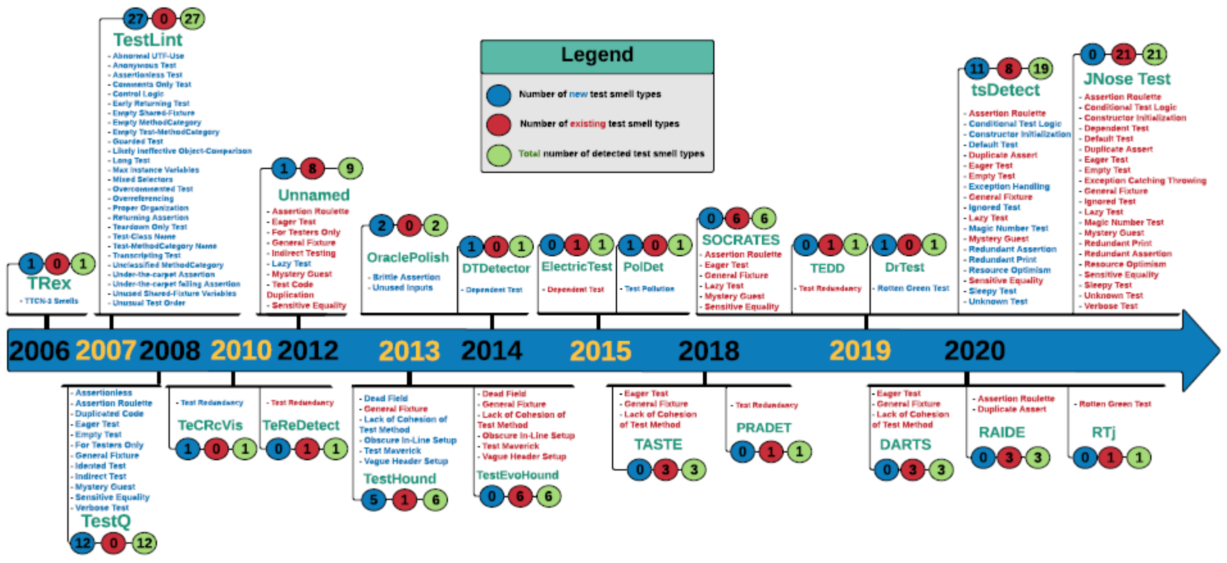


Figure 11. Test Smells Detection Tools

From this list, I made a selection of tools that were able to detect test smells in Java programs and were available online for download. In this selection, I included a total of

6 test smells detection tools: TsDetect [40], DARTS [41], TestHound [35], JNose [42], [43], RAIDE [44] and TestQ [45].

The performance of these tools is normally evaluated using precision and recall. To understand these, we first have to introduce the concepts of oracle, true positive, false positive, true negative and false negative:

- **Oracle:** it is a list containing the "true" result (smelly/not smelly) of the Java project instances. Normally, it is obtained by manually inspecting the code and deciding if each instance contains smells or not.
- **True positive:** it is an instance of code smell reported by the tool and present in the oracle.
- **False positive:** it is an instance of code smell reported by the tool but not present in the oracle.
- **True negative:** it is an instance of code smell not reported by the tool and not present in the oracle.
- **False negative:** it is an instance of code smell not reported by the tool but present in the oracle.

The following chart, called Confusion Matrix (*Figure 12*), better shows these terms in a graphical way:

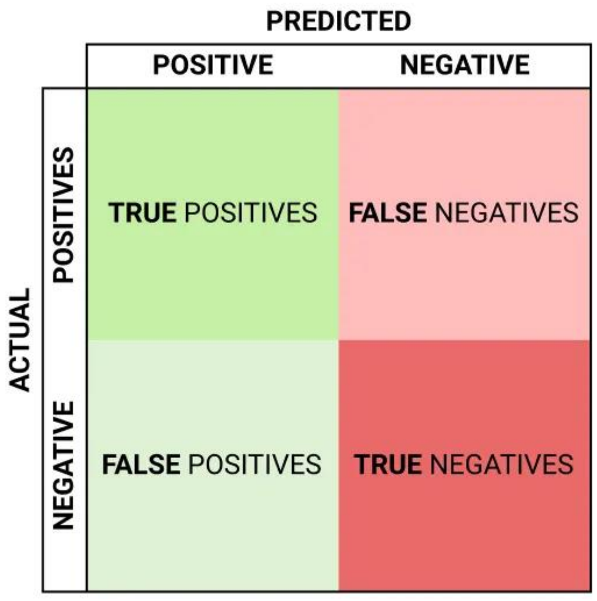


Figure 12. Confusion Matrix

With these concepts clear, we can now understand the definitions of precision and recall used in the test smells detection tools:

- **Precision:** in the simplest terms, precision is the ratio between the True positives and anything predicted as a positive. In this case, it would be the number of smells reported by the tool that are actually true smells out of all the smells found (*Figure 133*).

$$Precision = \frac{True\ Positive(TP)}{True\ Positive(TP) + False\ Positive(FP)}$$

Figure 13. Precision Formula

- **Recall:** the recall is the measure of our model correctly identifying True positives. For our case, it is the number of smells reported by the tool that are actually true smells out of all the true smells in the oracle (*Figure 14*).

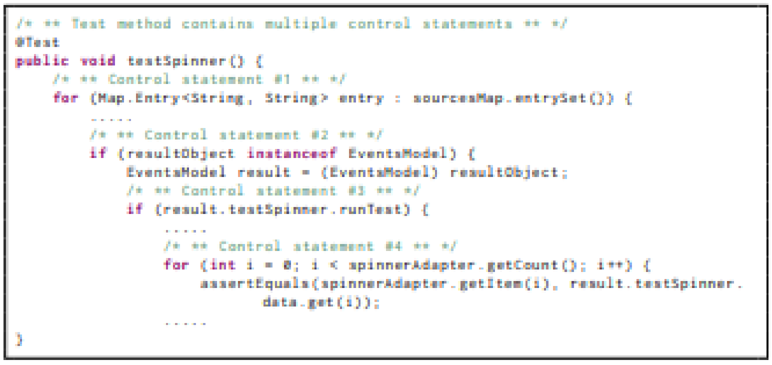
$$Recall = \frac{True\ Positive(TP)}{True\ Positive(TP) + False\ Negative(FN)}$$

Figure 14. Recall Formula

After understanding these terms, we can start analyzing the tools studied. Starting by TsDetect, it is a standalone command line application without user interface developed by Anthony Peruma et al. It can detect a total of 19 different types of test smells in Java programs. It achieved a precision score of 96% and a recall score of 97% and it is available in Github for free to download [46]. It calls the JavaParser library to parse the source code files. JavaParser builds an Abstract Syntax Tree from the unit test file that is under analysis. Then the AST is analyzed by each of the smell detection modules.

From the 19 test smells it is able to detect, 7 of them come from the work of van Deursen [31]: Assertion Roulette, Eager Test, General Fixture, Lazy Test, Mystery Guest, Resource Optimism and Sensitive Equality, whose definition was already explained in section 3.2. In addition, they propose 12 new smells. Because of the importance of this tool in my thesis, the new proposed smells will be explained as follows:

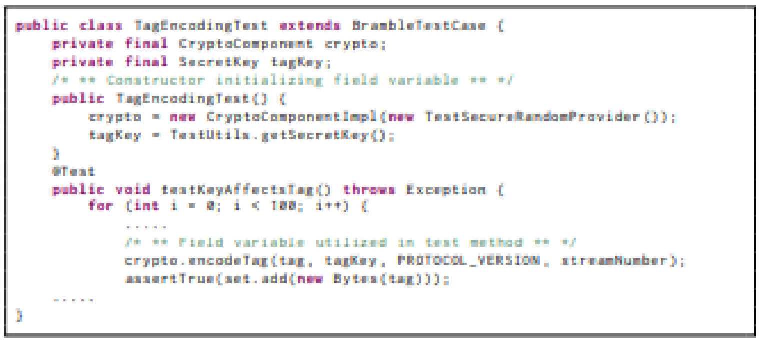
- **Conditional Test Logic:** When there are one or more control statements in a test case. TsDetect rules take into account if, switch, conditional expression, for, foreach and while as control statements. Test methods should not be complicated to understand, thus control statements should be avoided, leaving them to the production method. These conditions modify the functionality of the test and its expected output, causing some statements in the test to not be executed and then affecting the performance of the test. An example of this smell is provided in *Figure 15*.



```
/* ** Test method contains multiple control statements ** */
@Test
public void testSpinner() {
    /* ** Control statement #1 ** */
    for (Map.Entry<String, String> entry : sourcesMap.entrySet()) {
        .....
        /* ** Control statement #2 ** */
        if (resultObject instanceof EventsModel) {
            EventsModel result = (EventsModel) resultObject;
            /* ** Control statement #3 ** */
            if (result.testSpinner.runTest) {
                .....
                /* ** Control statement #4 ** */
                for (int i = 0; i < spinnerAdapter.getCount(); i++) {
                    assertEquals(spinnerAdapter.getItem(i), result.testSpinner.
                            data.get(i));
                .....
}
```

Figure 15. Conditional Test Logic Example

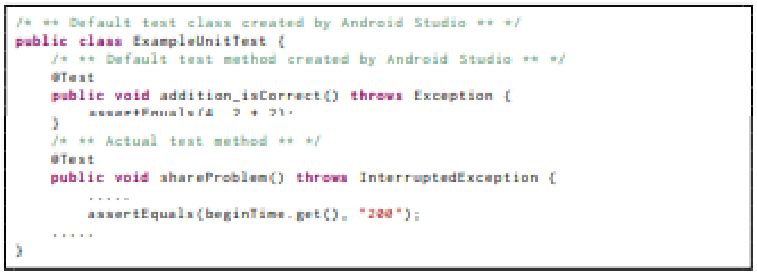
- **Constructor Initialization:** When there is a constructor declaration in the test case. Instructions related to initialization have to be coded in the "SetUp()" method and not in a specific test case. An example of this smell is provided in *Figure 16*.



```
public class TagEncodingTest extends BrambleTestCase {
    private final CryptoComponent crypto;
    private final SecretKey tagKey;
    /* ** Constructor initializing field variable ** */
    public TagEncodingTest() {
        crypto = new CryptoComponentImpl(new TestSecureRandomProvider());
        tagKey = TestUtils.getSecretKey();
    }
    @Test
    public void testKeyAffectsTag() throws Exception {
        for (int i = 0; i < 100; i++) {
            .....
            /* ** Field variable utilized in test method ** */
            crypto.encodeTag(tag, tagKey, PROTOCOL_VERSION, streamNumber);
            assertTrue(set.add(new Bytes(tag)));
    .....
}
```

Figure 16. Constructor Initialization Example

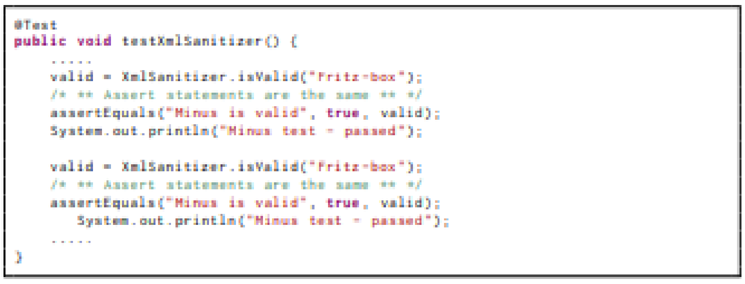
- **Default Test:** When the test case name is one of the following: "ExampleUnitTest" or "ExampleInstrumentedTest". These names are the ones created by default when a new project is started, and hence should be removed, and tests placed in a more logic and tidy way. An example of this smell is provided in *Figure 17*.



```
/* ** Default test class created by Android Studio ** */
public class ExampleUnitTest {
    /* ** Default test method created by Android Studio ** */
    @Test
    public void addition_isCorrect() throws Exception {
        assertEquals(4, 2 + 2);
    }
    /* ** Actual test method ** */
    @Test
    public void shareProblem() throws InterruptedException {
        .....
        assertEquals(beginTime.get(), "200");
    .....
}
```

Figure 17. Default Test Example

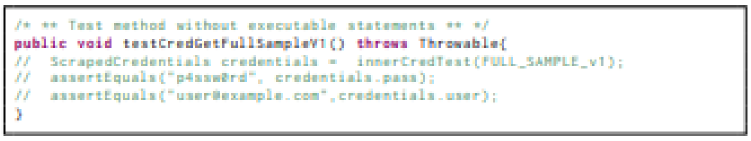
- **Duplicate Assert:** When there is more than one assertion with the same parameters in a test case. This means that the same test is checking the same condition more than once, when in this case separated test cases should be used. An example of this smell is provided in *Figure 18*.



```
@Test
public void testXmlSanitizer() {
    .....
    valid = XmlSanitizer.isValid("Fritz-box");
    /* ** Assert statements are the same ** */
    assertEquals("Minus is valid", true, valid);
    System.out.println("Minus test - passed");

    valid = XmlSanitizer.isValid("Fritz-box");
    /* ** Assert statements are the same ** */
    assertEquals("Minus is valid", true, valid);
       System.out.println("Minus test - passed");
    .....
}
```

Figure 18. Duplicate Assert Example

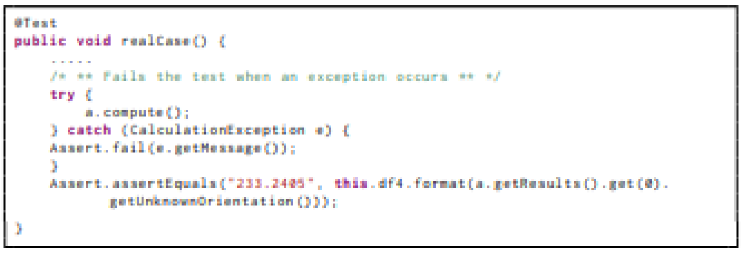
- **Empty Test:** When there is not any executable statement in a test case. This might be because it has been forgotten or the statements have been commented. In this case, the test will pass even when there are no statements, so this can cause confusion later on for developers. An example of this smell is provided in *Figure 19*.



```
/* ** Test method without executable statements ** */
public void testCredGetFullSampleV1() throws Throwable{
//  ScrapedCredentials credentials =  innerCredTest(FULL_SAMPLE_v1);
//  assertEquals("p4ssw0rd", credentials.pass);
//  assertEquals("user@example.com",credentials.user);
}
```

Figure 19. Empty Test Example

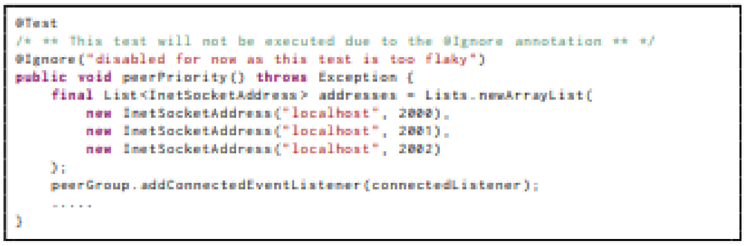
- **Exception Handling:** When there is a throw statement or catch clause in the test case. If the test result (pass or fail) depends on the production method throwing an exception, this can be complicated to understand when testing and can lead to more problems. An example of this smell is provided in *Figure 20*.



```
@Test
public void realCase() {
    .....
    /* ** Fails the test when an exception occurs ** */
    try {
        a.compute();
    } catch (CalculationException e) {
    Assert.fail(e.getMessage());
    }
    Assert.assertEquals("233.2405", this.df4.format(a.getResults().get(0).
           getUnknownOrientation()));
}
```

Figure 20. Exception Handling Example

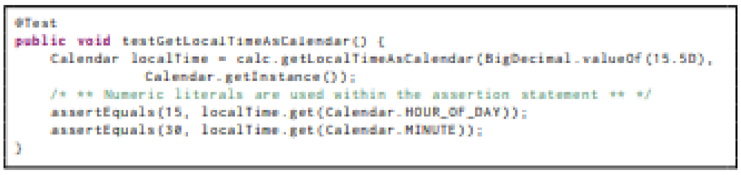
- **Ignored Test:** When the is an @Ignore annotation in the test case. This means that the test will not be run when executing the test suit, but it will still cause some problems, such as overhead in compilation time and an increase in complexity of the code. An example of this smell is provided in *Figure 21*.



```
@Test
/* ** This test will not be executed due to the @Ignore annotation ** */
@Ignore("disabled for now as this test is too flaky")
public void peerPriority() throws Exception {
    final List<InetSocketAddress> addresses = Lists.newArrayList(
        new InetSocketAddress("localhost", 2000),
        new InetSocketAddress("localhost", 2001),
        new InetSocketAddress("localhost", 2002)
    );
    peerGroup.addConnectedEventListener(connectedListener);
    .....
}
```

Figure 21. Ignored Test Example

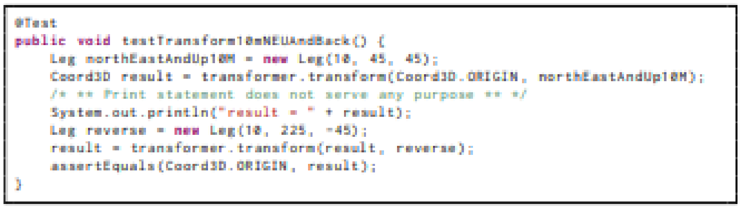
- **Magic Number Test:** When an assertion statement uses a numeric literal as an argument in the test case. Even though the assert will work the same, these numbers do not provide the developer any information about the meaning and purpose of the value, so it is harder to understand and should be replaced by a variable with its corresponding name. An example of this smell is provided in *Figure 22*.



```
@Test
public void testGetLocalTimeAsCalendar() {
    Calendar localTime = calc.getLocalTimeAsCalendar(BigDecimal.valueOf(15.5D),
             Calendar.getInstance());
    /* ** Numeric literals are used within the assertion statement ** */
    assertEquals(15, localTime.get(Calendar.HOUR_OF_DAY));
    assertEquals(30, localTime.get(Calendar.MINUTE));
}
```

Figure 22. Magic Number Test Example

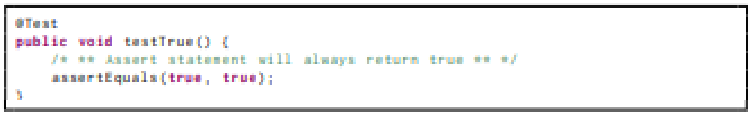
- **Redundant Print:** When one of the following instructions is used in the test case: “print”, “println”, “printf” or “write” method of the System class. Because the tests already provide the result, these statements are redundant and can increase the execution time. An example of this smell is provided in *Figure 23*.



```
@Test
public void testTransform10mNEUAndBack() {
    Leg northEastAndUp10M = new Leg(10, 45, 45);
    Coord3D result = transformer.transform(Coord3D.ORIGIN, northEastAndUp10M);
    /* ** Print statement does not serve any purpose ** */
    System.out.println("result = " + result);
    Leg reverse = new Leg(10, 225, -45);
    result = transformer.transform(result, reverse);
    assertEquals(Coord3D.ORIGIN, result);
}
```

Figure 23. Redundant Print Example

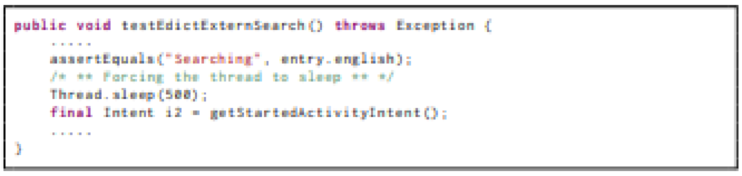
- **Redundant Assertion:** When an assertion statement with the same expected and actual parameters is used in a test case. When the statements of the method are always true or false regardless of the input, it is not useful and should be removed. An example of this smell is provided in *Figure 24*.



```
@Test
public void testTrue() {
    /* ** Assert statement will always return true ** */
    assertEquals(true, true);
}
```

Figure 24. Redundant Assertion Example

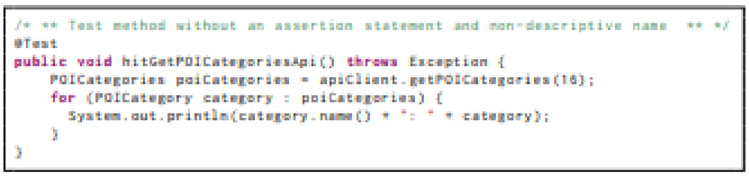
- **Sleepy Test:** When the instruction Thread.sleep() is invoked in the test case. This is used when the developer wants to stop the execution for a certain amount of time, but can lead to unexpected results depending on external causes. An example of this smell is provided in *Figure 25*.



```
public void testEdictExternSearch() throws Exception {
    .....
    assertEquals("Searching", entry.english);
    /* ** Forcing the thread to sleep ** */
    Thread.sleep(500);
    final Intent i2 = getStartedActivityIntent();
    .....
}
```

Figure 25. Sleepy Test Example

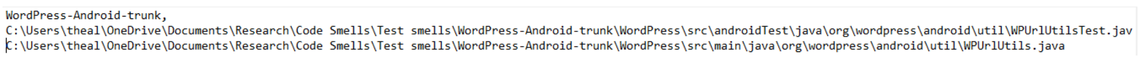
- **Unknown Test:** When there is not any assertion statement or "@Test(expected) annotation parameter in the test case. The assertion statement explains an expected condition for a test method and if it is missing, it becomes harder to understand the purpose of the test. An example of this smell is provided in *Figure 26*.



```
/* ** Test method without an assertion statement and non-descriptive name  ** */
@Test
public void hitGetPOICategoriesApi() throws Exception {
    POICategories poiCategories = apiClient.getPOICategories(16);
    for (POICategory category : poiCategories) {
      System.out.println(category.name() + ": " + category);
    }
}
```

Figure 26. Unknown Test Example

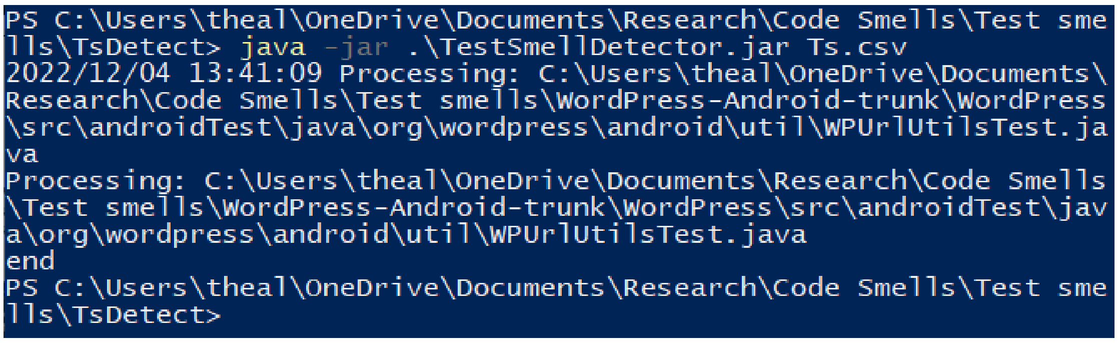
To execute the TsDetect tool, a csv file is needed as input containing, for every class that wants to be analyzed, the application name of the project, the path to the test file within the project and the path to the production file that is being tested. In the case of *Figure 27*, the test class WPUrlUtilsTest.java is being tested, so its path is provided, as well as the path of the production file WPUrlUtils.java, both belonging to the app WordPress-Android-trunk.



```
WordPress-Android-trunk,
C:\Users\theal\OneDrive\Documents\Research\Code Smells\Test smells\WordPress-Android-trunk\WordPress\src\androidTest\java\org\wordpress\android\util\WPUrlUtilsTest.jav
C:\Users\theal\OneDrive\Documents\Research\Code Smells\Test smells\WordPress-Android-trunk\WordPress\src\main\java\org\wordpress\android\util\WPUrlUtils.java
```

Figure 27. Example of Csv File for TsDetect

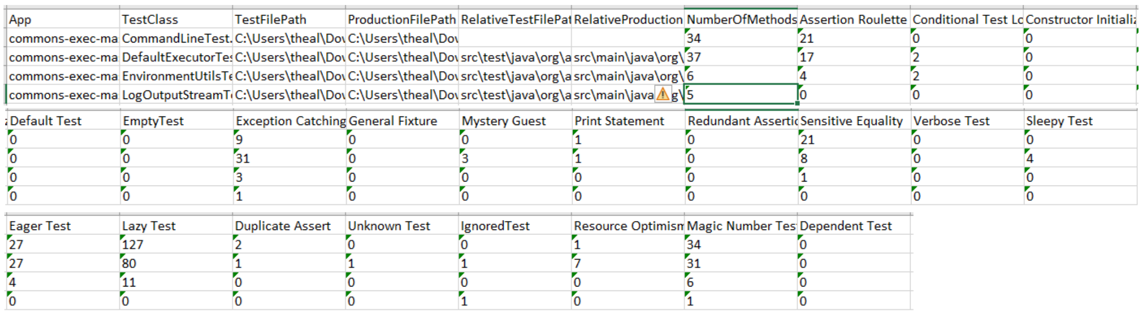
To run TsDetect, as we can see in *Figure 28*, the command line is necessary. First, the application is downloaded. Then, by writing the command "java -jar .\TestSmellDetector.jar Ts.csv", we execute the TsDetect smell detection over the classes specified in the csv file called Ts.csv, which in this case is only the class WPUrlUtilsTest.java.



```
PS C:\Users\theal\OneDrive\Documents\Research\Code Smells\Test sme
lls\TsDetect> java -jar .\TestSmellDetector.jar Ts.csv
2022/12/04 13:41:09 Processing: C:\Users\theal\OneDrive\Documents\
Research\Code Smells\Test smells\WordPress-Android-trunk\WordPress
\src\androidTest\java\org\wordpress\android\util\WPUrlUtilsTest.ja
va
Processing: C:\Users\theal\OneDrive\Documents\Research\Code Smells
\Test smells\WordPress-Android-trunk\WordPress\src\androidTest\jav
a\org\wordpress\android\util\WPUrlUtilsTest.java
end
PS C:\Users\theal\OneDrive\Documents\Research\Code Smells\Test sme
lls\TsDetect>
```

Figure 28. Example of TsDetect Execution

Finally, the result from the TsDetect test smell detection is saved into a new csv file. It contains, for every test class analyzed, the name of the app, the name of the test class, the path to the test class and to the production class (both absolute and relative), the number of methods in the test class and the number of smells found in each class, ordered by type. As we can see in this other example in *Figure 29*, four different test classes of the app commons-exec-main were analyzed. For example, for the first class called CommandLineTest.java, 34 methods where found. In this class, there were 21 Assertion Roulette smells, 0 Conditional Test Logic smells, 0 Construction Initialization smells, and so on.



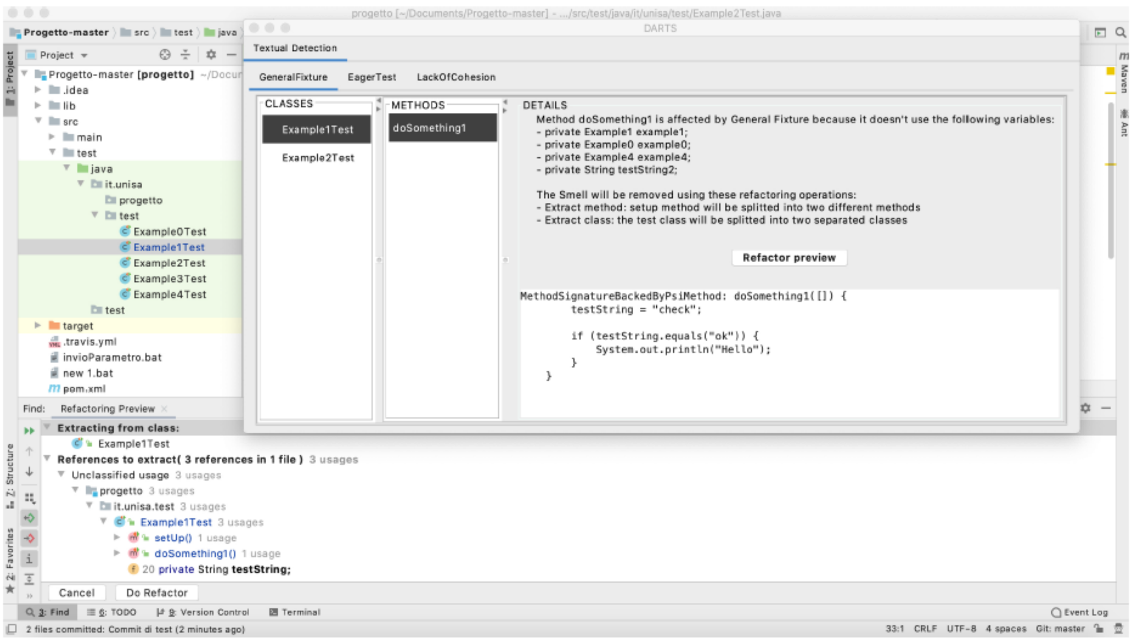
App	TestClass	TestFilePath	ProductionFilePath	RelativeTestFilePat	RelativeProduction	NumberOfMethods	Assertion Roulette	Conditional Test Lo	Constructor Initializ
commons-exec-ma	CommandLineTest.	C:\Users\theal\Dov	C:\Users\theal\Dov			34	21	0	0
commons-exec-ma	DefaultExecutorTe:	C:\Users\theal\Dov	C:\Users\theal\Dov	src\test\java\org\a	src\main\java\org\	37	17	2	0
commons-exec-ma	EnvironmentUtilsTe	C:\Users\theal\Dov	C:\Users\theal\Dov	src\test\java\org\a	src\main\java\org\	6	4	2	0
commons-exec-ma	LogOutputStreamT	C:\Users\theal\Dov	C:\Users\theal\Dov	src\test\java\org\a	src\main\java⚠g\	5	0	0	0

Default Test	EmptyTest	Exception Catching	General Fixture	Mystery Guest	Print Statement	Redundant Assertic	Sensitive Equality	Verbose Test	Sleepy Test
0	0	9	0	0	1	0	21	0	0
0	0	31	0	3	1	0	8	0	4
0	0	3	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	0

Eager Test	Lazy Test	Duplicate Assert	Unknown Test	IgnoredTest	Resource Optimism	Magic Number Tes	Dependent Test
27	127	2	0	0	1	34	0
27	80	1	1	1	7	31	0
4	11	0	0	0	0	6	0
0	0	0	0	1	0	1	0

Figure 29. Example of TsDetect Detection Result

Following it, we have DARTS, an IntelliJ plugin developed by Stefano Lambiase et al. It can detect three different types of smells: General Fixture, Eager Test, and Lack of Cohesion of Test Methods. TASTE detection rules are used, so its precision and recall are the same as this tool. It achieved a precision score of 76% and a recall score of 62% and it is available in Github for free to download [47]. *Figure 30* shows the plugin interface inside IntelliJ and how it works.



DETAILS
Method doSomething1 is affected by General Fixture because it doesn't use the following variables:
- private Example1 example1;
- private Example0 example0;
- private Example4 example4;
- private String testString2;

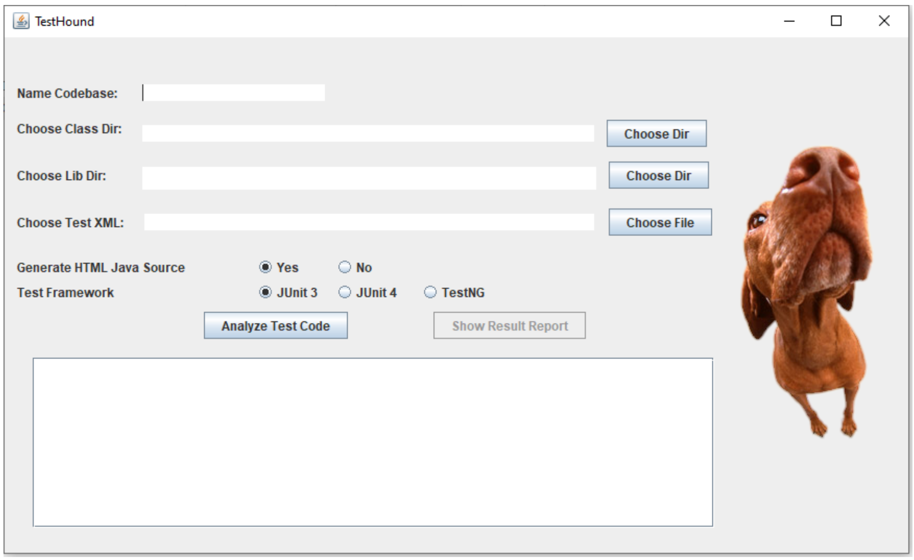
The Smell will be removed using these refactoring operations:
- Extract method: setup method will be splitted into two different methods
- Extract class: the test class will be splitted into two separated classes

```
MethodSignatureBackedByPsiMethod: doSomething1([]) {
        testString = "check";

        if (testString.equals("ok")) {
            System.out.println("Hello");
        }
    }
```

```
Extracting from class:
    Example1Test
References to extract( 3 references in 1 file ) 3 usages
  Unclassified usage 3 usages
    progetto 3 usages
      it.unisa.test 3 usages
        Example1Test 3 usages
          setUp() 1 usage
          doSomething1() 1 usage
          20 private String testString;
```

Figure 30. DARTS tool working on IntelliJ

TestHound is a standalone desktop application developed by Michaela Greiler et al. It can detect six different types of smells, all of them explained in [35]: General Fixture, Test Maverick, Dead Fields, Lack of Cohesion of Test Methods, Obscure In-Line Setup, Vague Header Setup. Its precision and recall scores and unknown and it is available in Github for free to download [48]. Although this tool could be opened (*Figure 31*), it had no documentation that explained how to use it and I could not analyze any project with it.

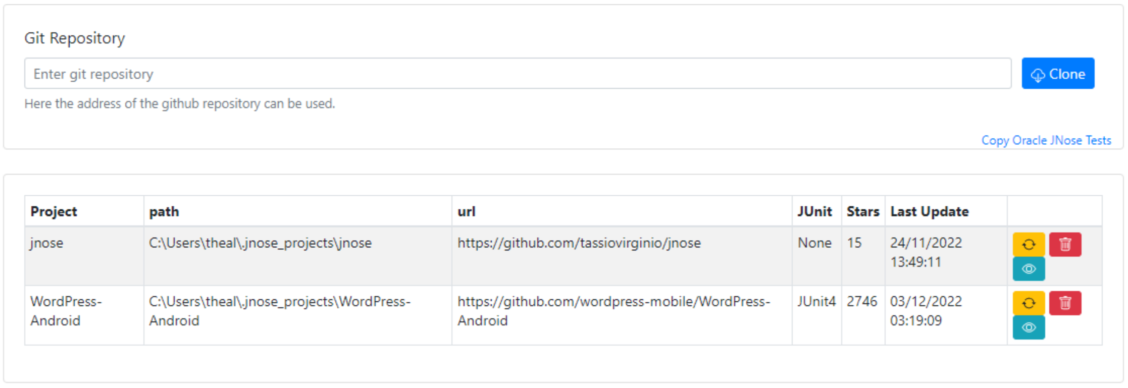


Name Codebase:
Choose Class Dir:
Choose Lib Dir:
Choose Test XML:
Generate HTML Java Source [✓ Yes] No
Test Framework [✓ JUnit 3] JUnit 4 TestNG

Figure 31. TestHound Tool UI

Next, there is JNose, a desktop application with user interface developed by Tássio Virgínio et al. It applies the TsDetect detection rules, so it achieves the same precision and recall, being a precision score of 96% and a recall score of 97%. It can detect 21 different test smells: the 19 smells that TsDetect could find plus two more that were added after. It is available in Github for free to download [49]. It performs a static analysis of the test code through an Abstract Syntax Tree generated by JavaParser. Then, it extracts information about the code structure to apply the TsDetector rules for the test smells detection.

To use the JNose tool, the needed input is the github URL of the program that is going to be analyzed for finding test smells. This URL is provided in the text field of the clone screen (*Figure 32*) and, by pressing the clone button, the project is downloaded into a local folder.



Git Repository

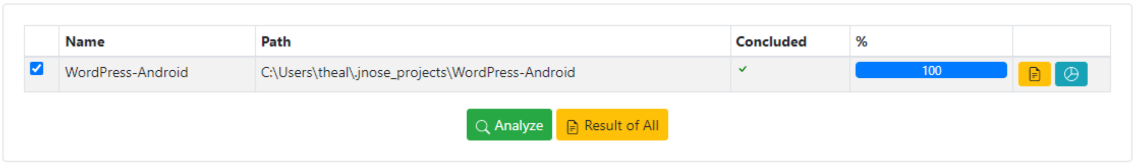
Enter git repository

Here the address of the github repository can be used.

Project	path	url	JUnit	Stars	Last Update	
jnose	C:\Users\theal\.jnose_projects\jnose	https://github.com/tassiovirginio/jnose	None	15	24/11/2022 13:49:11	
WordPress-Android	C:\Users\theal\.jnose_projects\WordPress-Android	https://github.com/wordpress-mobile/WordPress-Android	JUnit4	2746	03/12/2022 03:19:09	

Figure 32. JNose Project Clone Screen

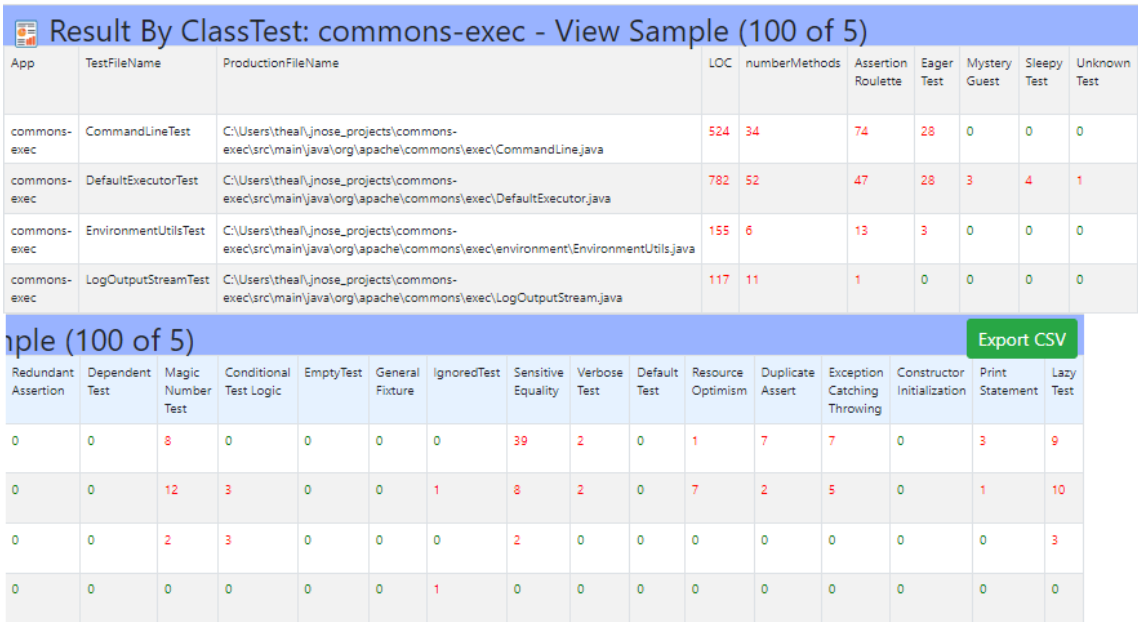
In *Figure 33*, we can see the analyze screen of JNose where, once the project is downloaded, you can run the test smell detector by pressing the analyze button. After some time, that depends on the number of classes and methods that need to be analyzed, the progress bar reaches 100% and the project is completely checked.



	Name	Path	Concluded	%	
✓	WordPress-Android	C:\Users\theal\.jnose_projects\WordPress-Android	✓	100	

Figure 33. JNose Project Analyze Screen

After the analysis, the results of the test smell detection can be checked by pressing the result button from the analyze screen. When doing so, a new screen will open (*Figure 34*), showing for every test class, the app name, the file name, the production file path, the lines of code, the number of methods and the number of smells found for all the different types of smell.



Result By ClassTest: commons-exec - View Sample (100 of 5)

Export CSV

App	TestFileName	ProductionFileName	LOC	numberMethods	Assertion Roulette	Eager Test	Mystery Guest	Sleepy Test	Unknown Test	Redundant Assertion	Dependent Test	Magic Number Test	Conditional Test Logic	EmptyTest	General Fixture	IgnoredTest	Sensitive Equality	Verbose Test	Default Test	Resource Optimism	Duplicate Assert	Exception Catching Throwing	Constructor Initialization	Print Statement	Lazy Test
commons-exec	CommandLineTest	C:\Users\theal\.jnose_projects\commons-exec\src\main\java\org\apache\commons\exec\CommandLine.java	524	34	74	28	0	0	0	0	0	8	0	0	0	0	39	2	0	1	7	7	0	3	9
commons-exec	DefaultExecutorTest	C:\Users\theal\.jnose_projects\commons-exec\src\main\java\org\apache\commons\exec\DefaultExecutor.java	782	52	47	28	3	4	1	0	0	12	3	0	0	1	8	2	0	7	2	5	0	1	10
commons-exec	EnvironmentUtilsTest	C:\Users\theal\.jnose_projects\commons-exec\src\main\java\org\apache\commons\exec\environment\EnvironmentUtils.java	155	6	13	3	0	0	0	0	0	2	3	0	0	0	2	0	0	0	0	0	0	0	3
commons-exec	LogOutputStreamTest	C:\Users\theal\.jnose_projects\commons-exec\src\main\java\org\apache\commons\exec\LogOutputStream.java	117	11	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

Figure 34. JNose Analysis Results Table

A chart showing the results from the detection in a graphical way is also generated after the process has finished (*Figure 35*). In this graph, a bar shows the number of smells found of each type.

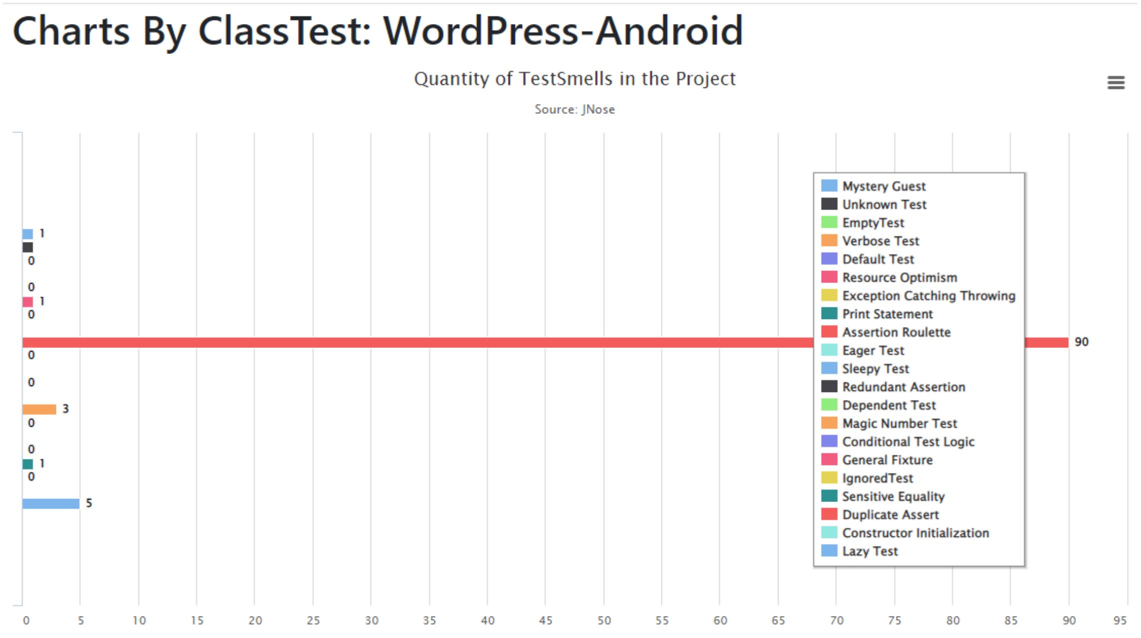
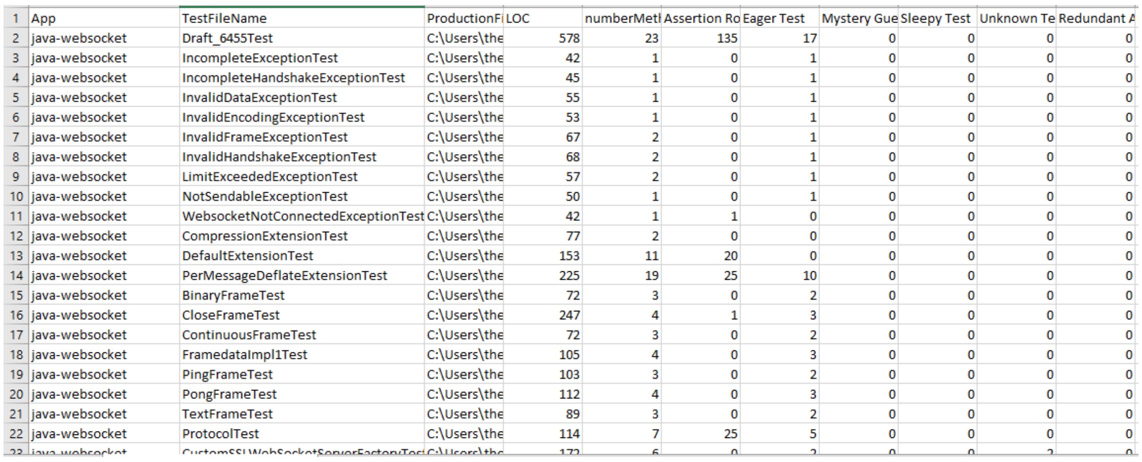


Figure 35. JNose Analysis Results Chart

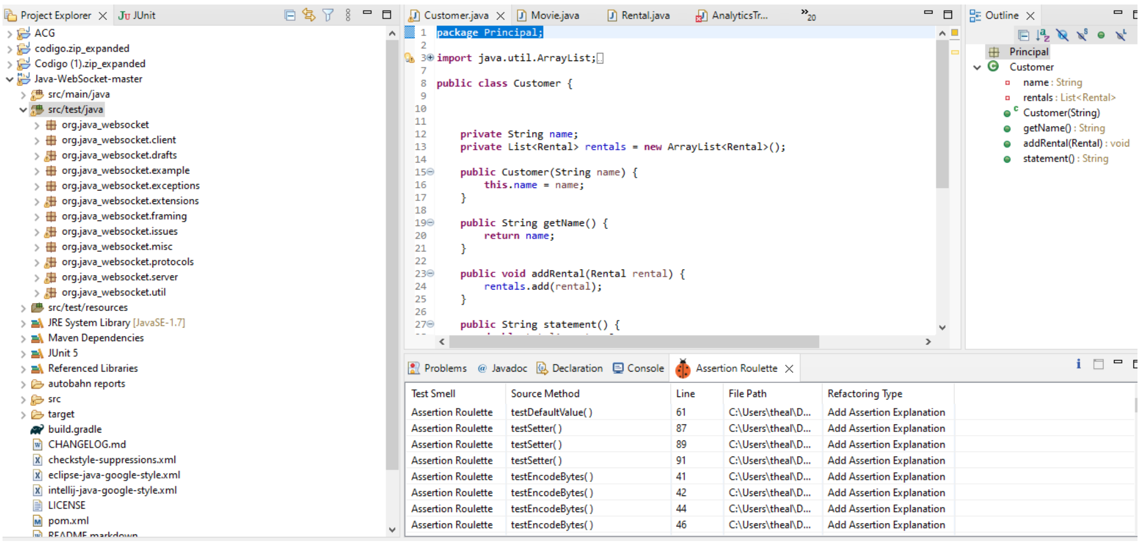
Finally, even though the results can be seen in the user interface of the JNose application, the option for downloading them into a csv file is also available (*Figure 36*). This is really helpful, as the results can be analyzed using other tools and used as input of other programs, as will do in the implementation of my application.



1	App	TestFileName	ProductionFi	LOC	numberMetl	Assertion Ro	Eager Test	Mystery Gue	Sleepy Test	Unknown Te	Redundant A
2	java-websocket	Draft_6455Test	C:\Users\the	578	23	135	17	0	0	0	0
3	java-websocket	IncompleteExceptionTest	C:\Users\the	42	1	0	1	0	0	0	0
4	java-websocket	IncompleteHandshakeExceptionTest	C:\Users\the	45	1	0	1	0	0	0	0
5	java-websocket	InvalidDataExceptionTest	C:\Users\the	55	1	0	1	0	0	0	0
6	java-websocket	InvalidEncodingExceptionTest	C:\Users\the	53	1	0	1	0	0	0	0
7	java-websocket	InvalidFrameExceptionTest	C:\Users\the	67	2	0	1	0	0	0	0
8	java-websocket	InvalidHandshakeExceptionTest	C:\Users\the	68	2	0	1	0	0	0	0
9	java-websocket	LimitExceededExceptionTest	C:\Users\the	57	2	0	1	0	0	0	0
10	java-websocket	NotSendableExceptionTest	C:\Users\the	50	1	0	1	0	0	0	0
11	java-websocket	WebsocketNotConnectedExceptionTest	C:\Users\the	42	1	1	0	0	0	0	0
12	java-websocket	CompressionExtensionTest	C:\Users\the	77	2	0	0	0	0	0	0
13	java-websocket	DefaultExtensionTest	C:\Users\the	153	11	20	0	0	0	0	0
14	java-websocket	PerMessageDeflateExtensionTest	C:\Users\the	225	19	25	10	0	0	0	0
15	java-websocket	BinaryFrameTest	C:\Users\the	72	3	0	2	0	0	0	0
16	java-websocket	CloseFrameTest	C:\Users\the	247	4	1	3	0	0	0	0
17	java-websocket	ContinuousFrameTest	C:\Users\the	72	3	0	2	0	0	0	0
18	java-websocket	FramedataImpl1Test	C:\Users\the	105	4	0	3	0	0	0	0
19	java-websocket	PingFrameTest	C:\Users\the	103	3	0	2	0	0	0	0
20	java-websocket	PongFrameTest	C:\Users\the	112	4	0	3	0	0	0	0
21	java-websocket	TextFrameTest	C:\Users\the	89	3	0	2	0	0	0	0
22	java-websocket	ProtocolTest	C:\Users\the	114	7	25	5	0	0	0	0
23	java-websocket	CustomSSLWebSocketServerFactoryTes	C:\Users\the	172	6	0	2	0	0	2	0

Figure 36. JNose Analysis Results Csv File

RAIDE is an Eclipse plugin developed by Railana Santana et al. It was done to detect test smells in Java programs and it can detect two different smells: Assertion Roulette and Duplicate Assert. It is an Abstract Syntax Tree based tool that reuses rule-based components from tsDetect to detect the test smells. Because of this, its precision and recall score are the same as the TsDetect tool and it is available in Github for free to download [50]. Its functionality inside Eclipse is shown in *Figure 37*.

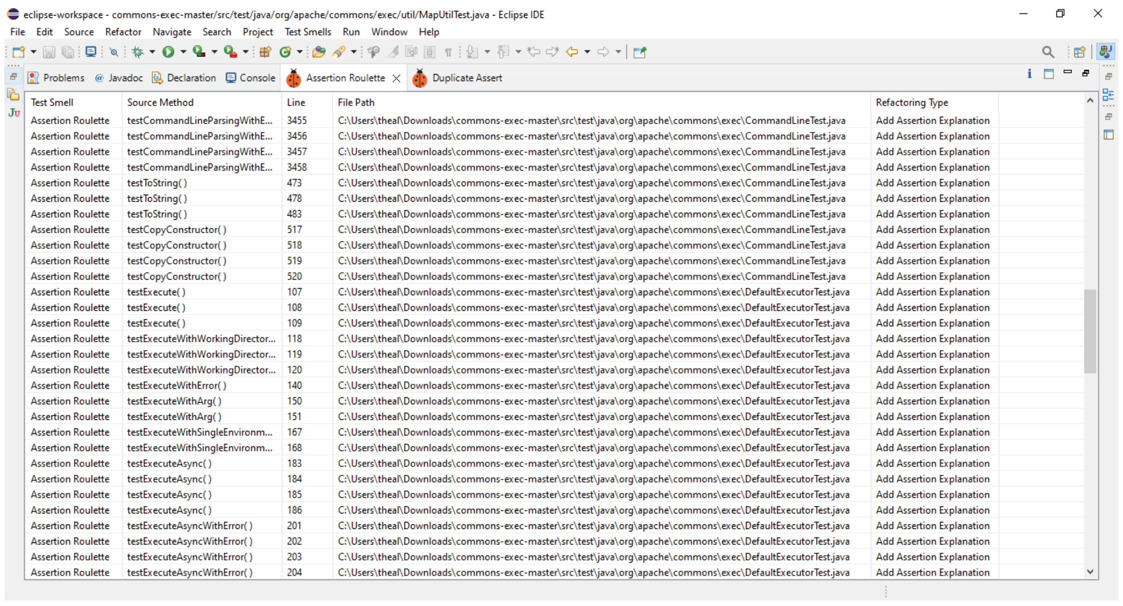


```java
package Principal;

import java.util.ArrayList;

public class Customer {



    private String name;
    private List<Rental> rentals = new ArrayList<Rental>();

    public Customer(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void addRental(Rental rental) {
        rentals.add(rental);
    }

    public String statement() {
```

Test Smell	Source Method	Line	File Path	Refactoring Type
Assertion Roulette	testDefaultValue()	61	C:\Users\theal\D...	Add Assertion Explanation
Assertion Roulette	testSetter()	87	C:\Users\theal\D...	Add Assertion Explanation
Assertion Roulette	testSetter()	89	C:\Users\theal\D...	Add Assertion Explanation
Assertion Roulette	testSetter()	91	C:\Users\theal\D...	Add Assertion Explanation
Assertion Roulette	testEncodeBytes()	41	C:\Users\theal\D...	Add Assertion Explanation
Assertion Roulette	testEncodeBytes()	42	C:\Users\theal\D...	Add Assertion Explanation
Assertion Roulette	testEncodeBytes()	44	C:\Users\theal\D...	Add Assertion Explanation
Assertion Roulette	testEncodeBytes()	46	C:\Users\theal\D...	Add Assertion Explanation

Figure 37. RAIDE in Eclipse

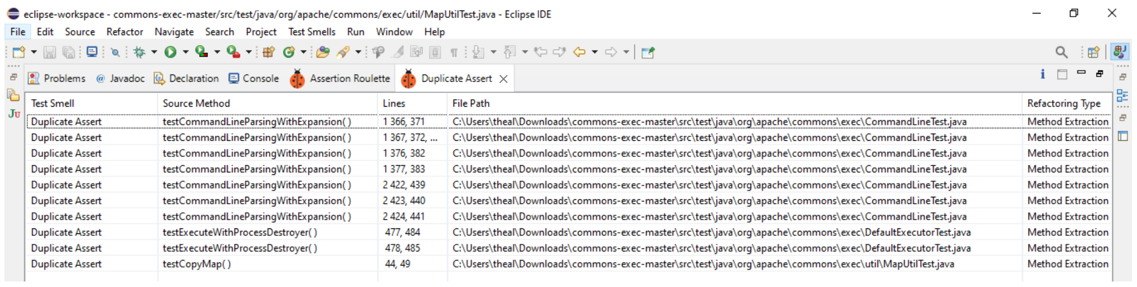
For Assertion Roulette, we can see the results in *Figure 38*. For example, it found the Assertion Roulette smell in the method called "testToString()" in lines 4473, 478 and 483 of the test class CommandLineTest.java (with its respective path to the test file) and the refactoring proposed is "Add Assertion Explanation".


eclipse-workspace - commons-exec-master/src/test/java/org/apache/commons/exec/util/MapUtilTest.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Test Smells Run Window Help
Problems
Javadoc
Declaration
Console
Assertion Roulette
Duplicate Assert

Test Smell	Source Method	Line	File Path	Refactoring Type
Assertion Roulette	testCommandLineParsingWithE...	3455	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Add Assertion Explanation
Assertion Roulette	testCommandLineParsingWithE...	3456	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Add Assertion Explanation
Assertion Roulette	testCommandLineParsingWithE...	3457	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Add Assertion Explanation
Assertion Roulette	testCommandLineParsingWithE...	3458	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Add Assertion Explanation
Assertion Roulette	testToString( )	473	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Add Assertion Explanation
Assertion Roulette	testToString( )	478	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Add Assertion Explanation
Assertion Roulette	testToString( )	483	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Add Assertion Explanation
Assertion Roulette	testCopyConstructor( )	517	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Add Assertion Explanation
Assertion Roulette	testCopyConstructor( )	518	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Add Assertion Explanation
Assertion Roulette	testCopyConstructor( )	519	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Add Assertion Explanation
Assertion Roulette	testCopyConstructor( )	520	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Add Assertion Explanation
Assertion Roulette	testExecute( )	107	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecute( )	108	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecute( )	109	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteWithWorkingDirector...	118	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteWithWorkingDirector...	119	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteWithWorkingDirector...	120	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteWithError( )	140	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteWithArg( )	150	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteWithArg( )	151	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteWithSingleEnvironm...	167	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteWithSingleEnvironm...	168	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteAsync( )	183	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteAsync( )	184	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteAsync( )	185	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteAsync( )	186	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteAsyncWithError( )	201	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteAsyncWithError( )	202	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteAsyncWithError( )	203	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation
Assertion Roulette	testExecuteAsyncWithError( )	204	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Add Assertion Explanation

Figure 38. RAIDE Results for Assertion Roulette

And for Duplicate Assert, the results are in *Figure 39*. For example, the smell Duplicate Assert is found twice, in the lines 477 and 484 in the test method "testExecuteWithProcessDestroyer()" of the test class DefaultExecutorTest.java, and the proposed refactoring type is "Method Extraction".


eclipse-workspace - commons-exec-master/src/test/java/org/apache/commons/exec/util/MapUtilTest.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Test Smells Run Window Help
Problems
Javadoc
Declaration
Console
Assertion Roulette
Duplicate Assert

Test Smell	Source Method	Lines	File Path	Refactoring Type
Duplicate Assert	testCommandLineParsingWithExpansion( )	1 366, 371	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Method Extraction
Duplicate Assert	testCommandLineParsingWithExpansion( )	1 367, 372, ...	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Method Extraction
Duplicate Assert	testCommandLineParsingWithExpansion( )	1 376, 382	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Method Extraction
Duplicate Assert	testCommandLineParsingWithExpansion( )	1 377, 383	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Method Extraction
Duplicate Assert	testCommandLineParsingWithExpansion( )	2 422, 439	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Method Extraction
Duplicate Assert	testCommandLineParsingWithExpansion( )	2 423, 440	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Method Extraction
Duplicate Assert	testCommandLineParsingWithExpansion( )	2 424, 441	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\CommandLineTest.java	Method Extraction
Duplicate Assert	testExecuteWithProcessDestroyer( )	477, 484	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Method Extraction
Duplicate Assert	testExecuteWithProcessDestroyer( )	478, 485	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\DefaultExecutorTest.java	Method Extraction
Duplicate Assert	testCopyMap( )	44, 49	C:\Users\theal\Downloads\commons-exec-master\src\test\java\org\apache\commons\exec\util\MapUtilTest.java	Method Extraction

Figure 39. RAIDE Results for Duplicate Assert

Finally, TestQ is a desktop application aimed at Linux with a UI that allows to detect 12 different types of test smells (*Figure 40*). It achieves a precision score of 89% and recall score of 52% and it is available online to download [51].

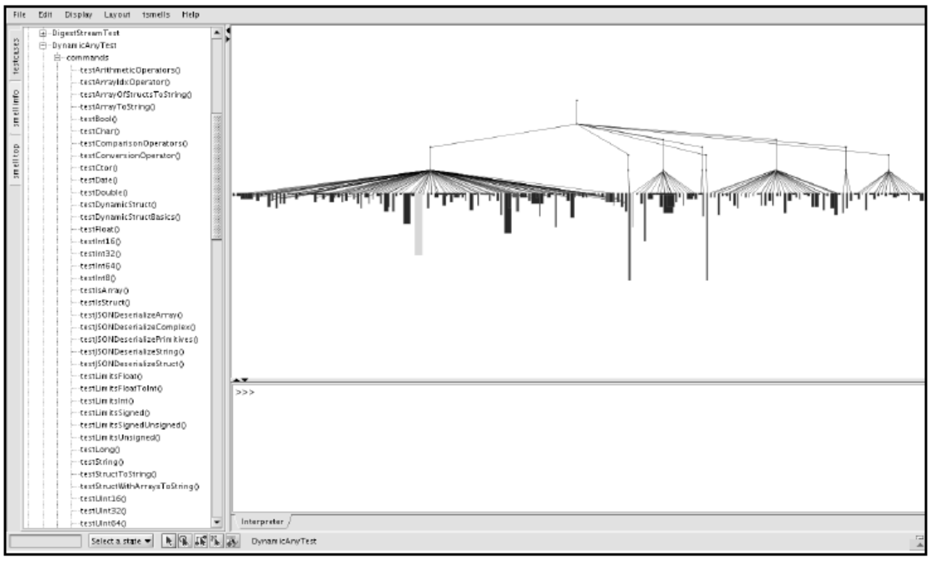


Figure 40. Interface of the TestQ Tool

3.4 Flaky tests

Flakiness is also a big issue when implementing tests. Flaky tests are tests that can intermittently pass or fail even when the source code has not been changed and the code version remains the same. Flaky tests can lead to confusion and unreliable results because developers must waste time debugging in order to find the problem, and then they realize that it did not occur because the recent changes, but because of a flaky test. In addition, they are hard to reproduced because of their randomness and they might hide other errors. As a result, people have researched about these kinds of tests, their characteristics, their impact, and their possible causes [52], [53].

The most common sources of flakiness are the following ones:

- **Asynchronous behaviors:** when a function that is asynchronous is called but the program is still able to execute other instructions before the result from the function is obtained. If the following instructions use the result from the asynchronous function as an input, the availability of this result may provoke the pass or fail of the test case depending on how fast this result is obtained. In addition, asynchronous instructions such as "Thread.sleep()" are the cause of unpredictable results.
- **Network connections:** connections to the network normally depend on waiting times, which makes it a dependency hard to control. When the server does not respond in a timely manner, because of thread scheduling or network delay, the result of the test can be affected. Because of this reason, if the delays are not considered, some "pass" and "fail" results can be obtained intermittently.
- **Input/Output operations:** when relying on external resources obtained via I/O, test cases should be coded carefully. For example, if the application was expecting to find a

file in a specific folder but it is not there, or the format is not the expected one. In these cases, the test can produce different results based on the availability and format of the expected resources.

- **Time instructions:** if the program relies on time instructions such as "java.time.LocalDateTime" or "java.time.Clock", the test depends on the time zone and it can fail, for example, when the midnight changes or the precision of the obtained result is not the expected one.
- **Test order dependency:** when implementing the test cases, we might think that they will be executed in a specified order, but the truth is that there is no specific order, and they are executed randomly. When the outcome of some tests is affected by other tests because, for example, they share some common variables, the order of execution affects the test results. This behavior should be avoided by isolating the different tests.
- **Randomness:** if the test cases make use of random numbers, there might be some cases that were not considered and hence produce an unexpected result. It is the case of a random number generator that, when obtaining exactly a zero and then dividing any number by it, produces as a result NaN.
- **Platform dependency:** a project, together with its test cases, is normally developed in a specific environment: a specific operating system (OS), computer hardware, screen size, etc. If this is not taken into account, the result of the test can be affected, as different platforms mean different delays and specification details.

3.5 Flaky tests detection tools

As we have explained in the previous section 3.4, flaky tests are a problem present in every real software project, and they are difficult to detect and deal with (even more manually). The most common way to find them is to rerun the tests that fail multiple times to check if they produce a different result. However, this consumes a lot of resources and, even if you run a flaky test 10000 times, it might not give any sign of flakiness or different result, so you would need more reruns to confirm that it is flaky. Because of that, there has been some research done on the automatic detection of these type of tests, and tools have been developed to perform this task.

DeFlaker is a tool developed by Jonathan Bell et al. [54] that can detect flaky tests without the need to rerun the failing tests multiple times, considerably reducing the spent resources. It is based on a technique that keeps track of the changes done in the code. If a test that was always passing and checks code that was not modified fails, then it is marked as flaky. It is a useful option that reduces the overhead, as it instantly notifies if a test that failed is flaky, avoiding extra reruns.

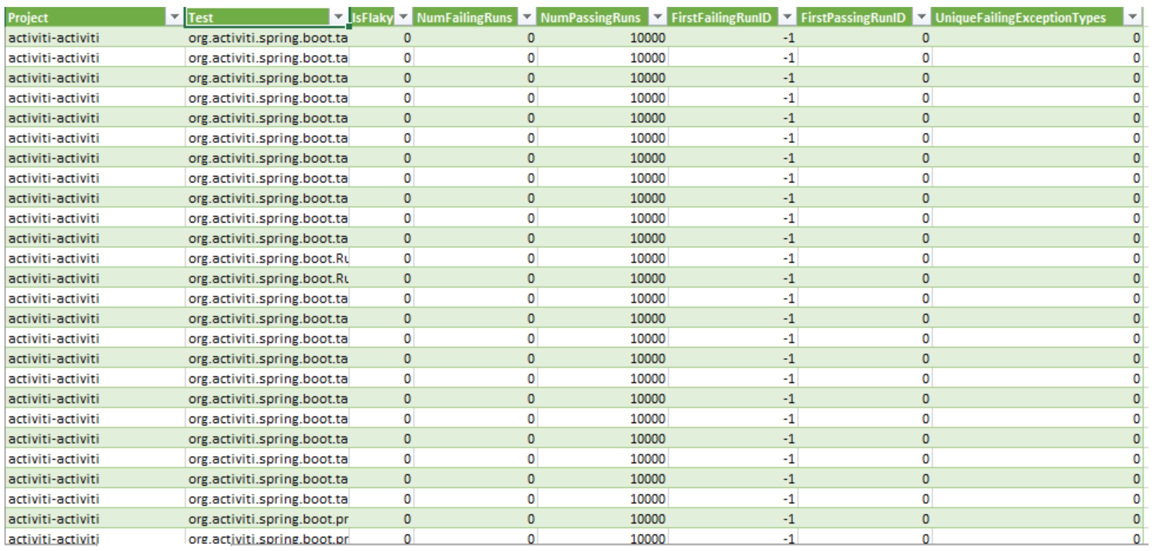
However, as it only aims at test that were passing but suddenly failed, it is not covering all possible flaky tests, are tests that still did not produce a different result are not checked. Nevertheless, even in the case where the developer wants to rerun the test to personally check for different results and address it as flaky, the test is already identified by the tool, so the developer only needs to rerun that test and not the complete test suit.

This tool was made available for everyone on Github [55]. When downloaded and installed through Maven, a.git directory has to be put in the same folder as the root POM of the build. Then, by running the command “mvn test” or “mvn verify”, a file will be obtained as output with four values: testClass, coveredClass, isCoverageClassLevel and lineNumber.

The tool FlakeFlagger is developed by Abdulrahman Alshammari et al. [56] and its based in a technic that focuses on collecting behavioral features of the tests to then predict if they are flaky or not. The tool can be downloaded from [57]. It provides a specific dataset that contains a total of 24 projects with over 22,000 test methods. The tests of these projects were classified as flaky or non-flaky by automatically executing them 10,000 times each. If a test method gave a different result at least once, it was declared as flaky. In the end, a total of 811 test methods were marked as flaky.

This dataset was used by the developers of FlakeFlagger to evaluate the tool. It contains the following columns (*Figure 41*):

- **Project:** the name of the project that the test method belongs to.
- **Test:** the path to the test method.
- **IsFlaky:** if the method analyzed is flaky or not. It is set to 0 in case of non-flaky and to 1 in the case of flaky.
- **NumFailingRuns:** out of the 10000 runs, the number of times the test method resulted in a failure.
- **NumPassingRuns:** out of the 10000 runs, the number of times the test method resulted in passing.
- **FirstFailingRunID:** the number of the repetition that failed for the first time. For example, if it failed three times and the first time failed in the repetition number 2386, that is the number. In case it did not fail, it is set to -1.
- **FirstPassingRunID:** the number of the repetition that passed for the first time. In most of the cases, as it passes more than fails, the first repetition is usually a pass, so that is why it is set to 0. In case it did not pass, it is set to -1.
- **UniqueFailingExceptionTypes:** the different number of exceptions that were thrown in case it failed multiple times. If it did not fail, it remains 0.



Project	Test	IsFlaky	NumFailingRuns	NumPassingRuns	FirstFailingRunID	FirstPassingRunID	UniqueFailingExceptionTypes
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.Ru	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.Ru	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.ta	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.pr	0	0	10000	-1	0	0
activiti-activiti	org.activiti.spring.boot.pr	0	0	10000	-1	0	0

Figure 41. Dataset of FlakeFlagger

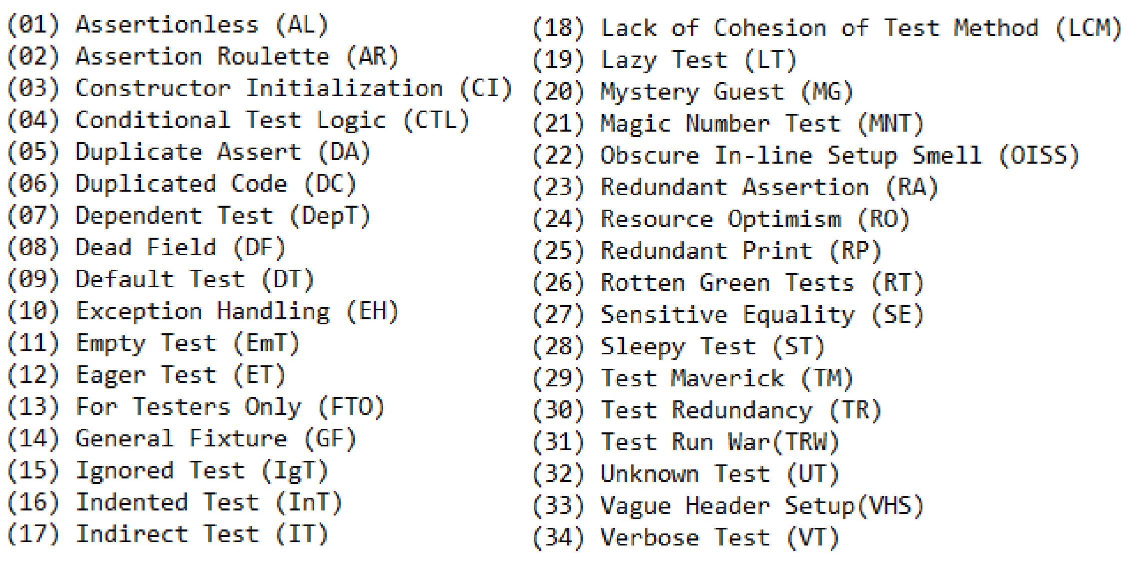
iDFlakies is developed by Wing Lam et al. [58] and can detect flaky tests by reordering and rerunning the test cases of a software system. The tool is also able to classify flaky tests as order-dependent when the cause of flakiness is the order in which tests are executed and non-order-dependent otherwise. For the order-dependent tests, they are flaky because the order in which they are executed cannot be controlled. However, they can deterministically pass or fail if we take the order into account. For the non-order-dependent tests, its causes are harder to control, such as depending on external resources on asynchronism.

To run the tool, we first download the project, which is available in Github [59]. Then, we need to create a csv file with, for every project we want to analyze, the Github URL and its SHA, separated by a comma. Next, we move to scripts/docker/create_and_run_dockers.sh within the iDFLakes project and run the following command: "bash create_and_run_dockers.sh <path to csv> <round num> <timeout (seconds)>. After that, a folder is created for every project in the csv file, containing a docker image. Inside, a csv file with the results of the flaky test analysis will be obtained.

4. Analysis of the problem

4.1 Test smells detection tools comparison

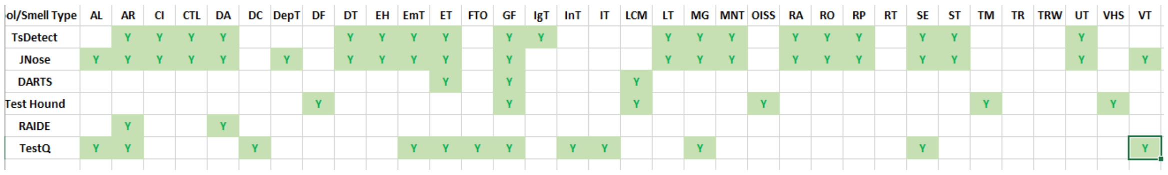
A comparison among the test smell detection tools mention in section 3.3 was made. To do that, the smells that the different tools were able to analyze were compiled into a table in *Figure 42*, with a total of 34 smells combining all the tools.



(01) Assertionless (AL)
(02) Assertion Roulette (AR)
(03) Constructor Initialization (CI)
(04) Conditional Test Logic (CTL)
(05) Duplicate Assert (DA)
(06) Duplicated Code (DC)
(07) Dependent Test (DepT)
(08) Dead Field (DF)
(09) Default Test (DT)
(10) Exception Handling (EH)
(11) Empty Test (EmT)
(12) Eager Test (ET)
(13) For Testers Only (FTO)
(14) General Fixture (GF)
(15) Ignored Test (IgT)
(16) Indented Test (InT)
(17) Indirect Test (IT)
(18) Lack of Cohesion of Test Method (LCM)
(19) Lazy Test (LT)
(20) Mystery Guest (MG)
(21) Magic Number Test (MNT)
(22) Obscure In-line Setup Smell (OISS)
(23) Redundant Assertion (RA)
(24) Resource Optimism (RO)
(25) Redundant Print (RP)
(26) Rotten Green Tests (RT)
(27) Sensitive Equality (SE)
(28) Sleepy Test (ST)
(29) Test Maverick (TM)
(30) Test Redundancy (TR)
(31) Test Run War(TRW)
(32) Unknown Test (UT)
(33) Vague Header Setup(VHS)
(34) Verbose Test (VT)

Figure 42. List of Test Smells for the Six Tools

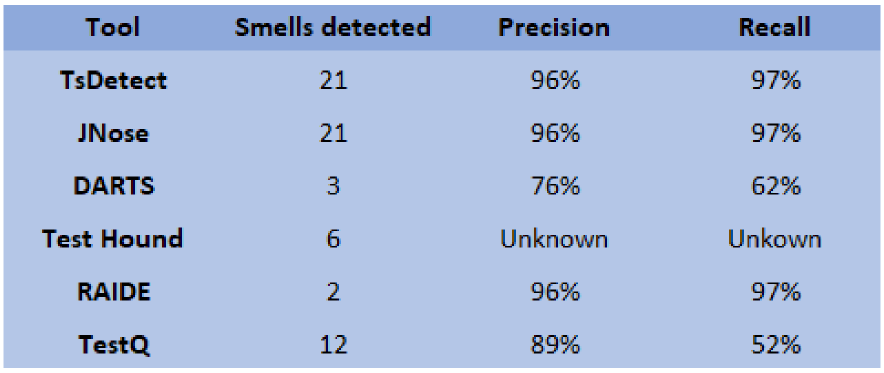
The tools selected can be seen in this table (*Figure 42*) represented as rows. The columns are all the test smells mentioned in the previous list. If the tool is able to detect a specific smell, it is marked with a green "Y".



ol/Smell Type	AL	AR	CI	CTL	DA	DC	DepT	DF	DT	EH	EmT	ET	FTO	GF	IgT	InT	IT	LCM	LT	MG	MNT	OISS	RA	RO	RP	RT	SE	ST	TM	TR	TRW	UT	VHS	VT
TsDetect		Y	Y	Y	Y				Y	Y	Y	Y		Y	Y				Y	Y	Y		Y	Y	Y		Y	Y				Y		
JNose	Y	Y	Y	Y	Y		Y		Y	Y	Y	Y		Y					Y	Y	Y		Y	Y	Y		Y	Y				Y		Y
DARTS												Y		Y				Y																
Test Hound								Y						Y				Y				Y							Y				Y	
RAIDE		Y			Y																													
TestQ	Y	Y				Y					Y	Y	Y	Y		Y	Y			Y							Y							Y

Figure 43. Test Smells Tools and its Detected Smells

In this other table (*Figure 44*), for every one of the tools, the number of different test smells that can be detected is written in the second column, while the precision and recall can be seen in the third and fourth column.



Tool	Smells detected	Precision	Recall
TsDetect	21	96%	97%
JNose	21	96%	97%
DARTS	3	76%	62%
Test Hound	6	Unknown	Unkown
RAIDE	2	96%	97%
TestQ	12	89%	52%

Figure 44. Test Smells Tools Comparison

Some of these tools were too old or had no clear documentation for me to make them work. It is the case of TestHound and TestQ. Others, such as DARTS and RAIDE, were only able to detect a few smells and are only plugins, which makes it difficult to incorporate them into a new project. Finally, TsDetect and JNose were the best choice to detect test smells in Java programs, as they can detect the larger number of smells, as well as being the easiest and most convenient to use. JNose uses TsDetect detection rules but builds an interface on top of that and improves the way of providing the input: it only needs the Github URL, in contrast to TsDetect, which needs a csv file with multiple fields. Because of that, JNose was the test smell detection tool selected for my study.

4.2 Test smells as indicators of flakiness

Current approaches to combat flaky tests are rather unsatisfactory. The most common approach is to run a flaky test multiple times, and if it passes in any run, declare it passing, even if it fails in several other runs. But these approach wastes a lot of machine resources and reduces the test suite effectiveness if the flaky test is excluded.

Some tools, like FlakeFlagger, DeFlaker and iDFlakies, propose different approaches that are not based on rerunning tests a lot of times. However, these tools are more like a complement to other techniques for finding flakiness and there is still a lot of work to do in this filed if flaky tests are to be found in an efficient way.

On the other hand, there has been a lot of research done about the automatic detection of test smells. In general, the results of the different detection tools are pretty accurate. If the points mentioned before are considered, that test smell detection tools are efficient and accurate and flaky test detection tools are rather unsatisfactory, then we can think of a possible next step. In this line and following the objectives I stated in section 1.3, I decided to conduct an empirical study to find out if there is a correlation between test smells and flaky tests and if test smell detection tools can be used to find flaky tests.

5. Proposed solution

In this section, the proposed solution for this study will be explained, as well the design and architecture of its system. As I mentioned in section 4.1, JNose was the tool selected for analyzing the projects of this study. On the other hand, I also had to select the dataset with flaky tests that I wanted to use as input of the JNose tool. FlakeFlagger already provides a dataset with the needed information; that is, the tests classified as flaky or non-flaky. Because this dataset was very large and contained a total of 24 projects with over 22,000 test methods, I decided to choose it for my empirical study.

5.1 Architecture of the system

The empirical study was conducted by developing a tool that is able to run JNose for detecting the test smells. This tool is developed using React, which means the use of Javascript, HTML and CSS. It is called Smell Flaky Analyzer and it is available in Github [60]. In addition, some Python scripts were implemented, that are necessary for running JNose and obtaining the parsed results from JNose and the flaky dataset.

React applications are web applications, which means that because of browser security policies, they cannot execute command line instructions. I needed to do that in order to execute the Python scripts, so I decided to convert it into a desktop application by using Electron.

To start with, the dataset resulting from the test smell detection and the dataset containing the flaky tests must be obtained. As we will be working with the projects from the FlakeFlagger study, its dataset will be used. The instructions to be followed are explained in my tool and a button directly sends you to the FlakeFlagger page where you can download the dataset csv file. Moreover, the csv file has to be uploaded to the tool, which parses it so that is has the desired format for combining in later with the JNose detection results.

Then, another button allows to open the JNose tool, which has been added to the project folder, into a new window. By using JNose, the URL of the projects present in the flaky dataset are provided, and then the JNose tool is used to analyze them for detecting test smells. When the detection process is completed, the results are downloaded into another csv file.

After that, the test smells csv file obtained from JNose and the parsed csv file with the flaky dataset are uploaded to the tool and merged into a single file. The results are shown in the tool with a table and different graphs. This tool wrapped all the process I would have to make manually into a single application, which made the process easier and more convenient to replicate, and it also provided a better way of inspecting the results rather than a raw csv file, so that I could reach some conclusions.

An overview of the research process that was done can be seen in *Figure 45*:

1. The test smell detection tool and the flaky dataset had to be selected.
2. The dataset was used as input of the test smell tool.
3. The results of step 2 and the flaky dataset were mixed into one single file.
4. This file was analyzed to look for some correlation between test smells and flaky tests.

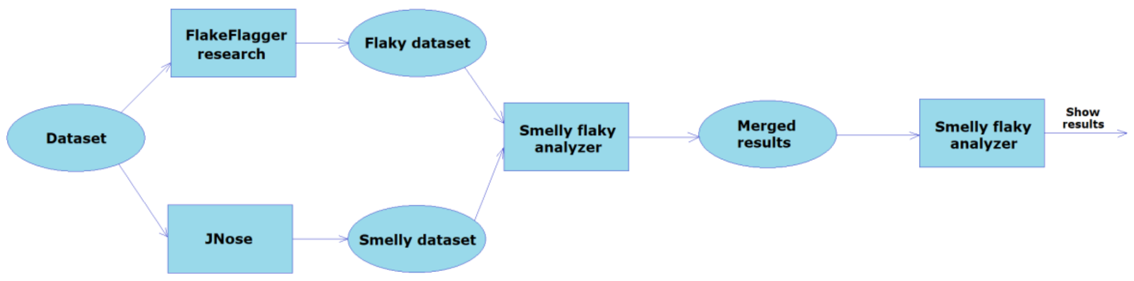


Figure 45. Architecture of Smell Flaky Analyzer Tool

5.2 Design of the system

5.2.1 Mock-ups

Mock-ups are a model of the graphical user interface of an application, in a partial and preliminary version. They can be designed on paper, on a whiteboard or even using software tools that facilitate their creation. They are said to be partial and preliminary because designing a mock-up does not imply that its final version will be the same. On the contrary, mock-ups are conceived with the aim of showing customers and other developers, using the prototyping technique, what the initial design idea for an interface is.

By having graphical references and not just descriptions and text, users are able to better understand how the application will work, and thus give feedback to the developers on which functionalities they would like to introduce, and which ones will be left out for the time being. Normally, the initial idea will undergo modifications and the final interface will end up being different from the first version shown. The mock-up of the Smell Flaky Analyzer tool that was created before its implementation is shown in *Figure 46*.

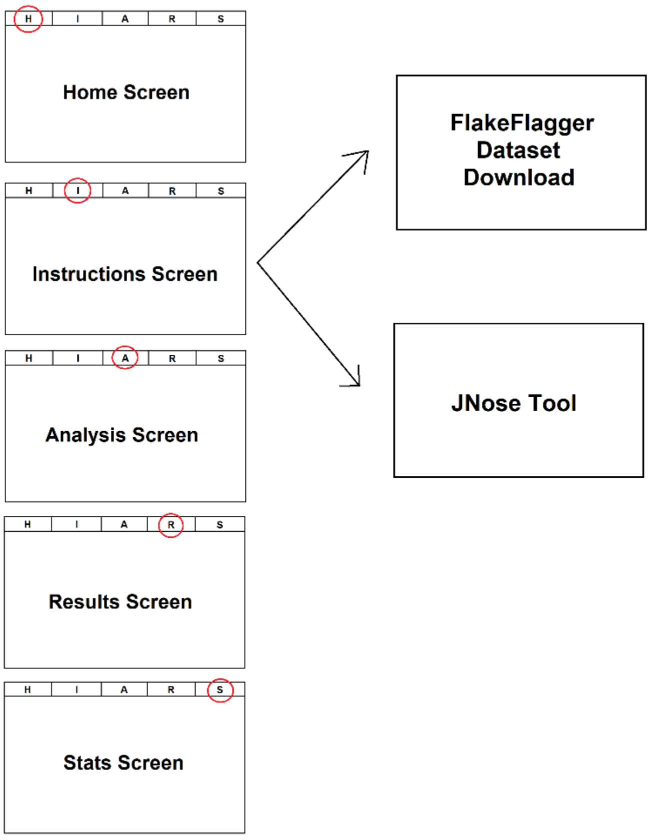
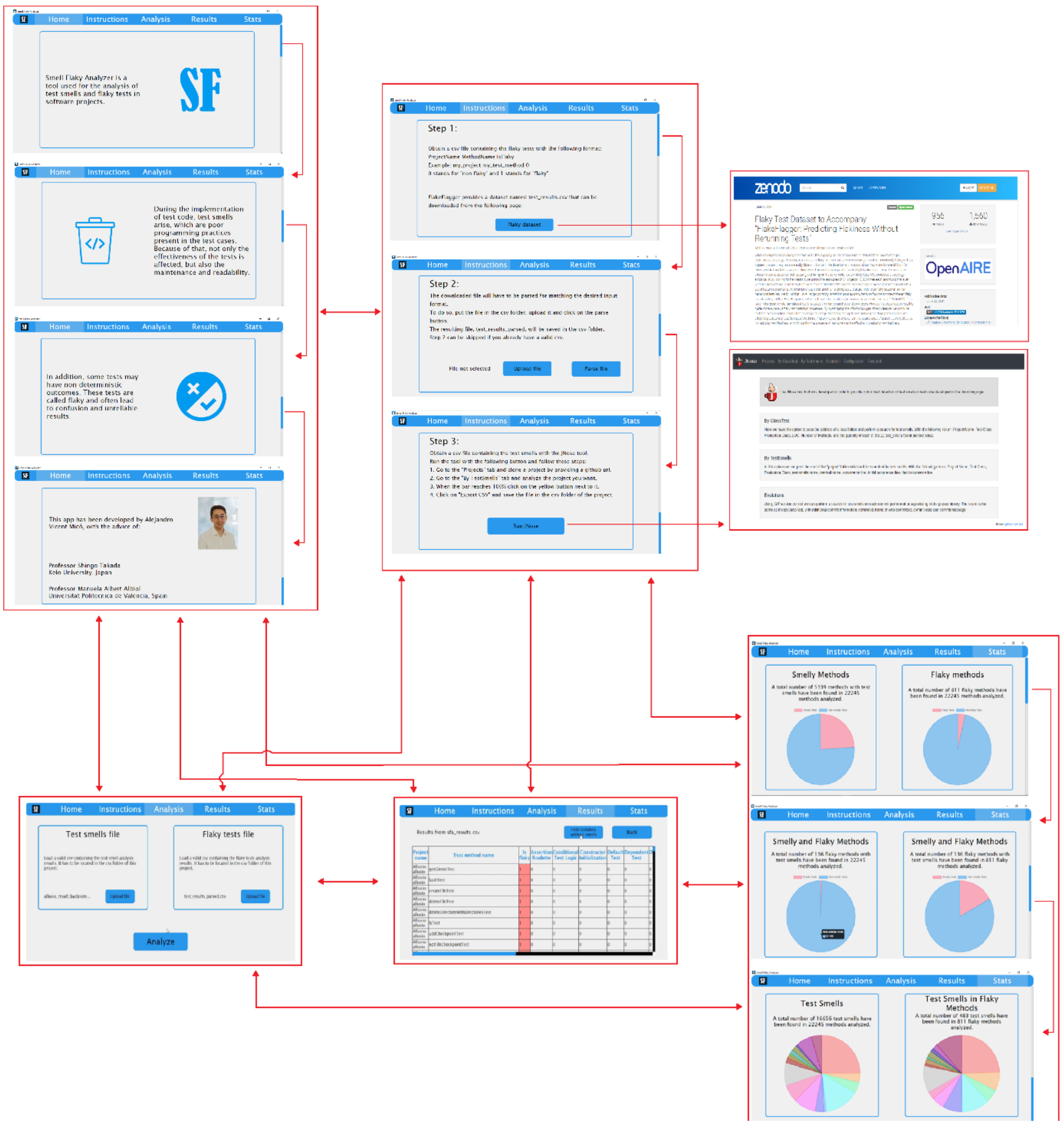


Figure 46. Mock-up Smell Flaky Analyzer Tool

5.2.2 Flowchart

A flowchart is a graphical representation that shows the logical sequence or steps to follow to navigate between the different screens of the application. These are connected by arrows, which indicate that the target window can be accessed from the source window. This type of diagrams allows to show the whole application and helps to understand how it works and what possibilities it offers. Additionally, it presents the style of the product as a whole, since you can see all the windows and how they combine with each other.

Below, in *Figure 47*, the flowchart of the Smell Flaky Analyzer tool is shown, pointing out the navigation between windows. Starting from the top-left corner, the first screen accessed is the home menu. It shows basic information of the application and its development. If we use the scrollbar, we can go down in the home menu and see the different screens. By clicking on a button from the top bar, we can access the five different menus: home, instructions, analysis, results and stats. Then, from each of the five screens, four arrows exit, showing the possible navigation to the other four screens. Finally, from the instructions menu, we can also access the website of FlakgeFlagger for downloading its dataset, as well as to the JNose tool to run the test smell detection.



Smell Flaky Analyzer is a tool used for the analysis of test smells and flaky tests in software projects.

During the implementation of test code, test smells arise, which are poor programming practices present in the test cases. Because of that, not only the effectiveness of the tests is affected, but also the maintenance and readability.

In addition, some tests may have non deterministic outcomes. These tests are called flaky and often lead to confusion and unreliable results.

This app has been developed by Alejandro Vicent Micó, with the advice of:

Professor Shingo Takada
Keio University, Japan

Professor Manuela Albert Albiol
Universitat Politecnica de Valencia, Spain

Step 1:

Obtain a csv file containing the flaky tests with the following format:
ProjectName MethodName IsFlaky
Example: my_project my_test_method 0
0 stands for "non flaky" and 1 stands for "flaky"

FlakeFlagger provides a dataset named test_results.csv that can be downloaded from the following page:

Step 2:

The downloaded file will have to be parsed for matching the desired input format.
To do so, put the file in the csv folder, upload it and click on the parse button.
The resulting file, test_results_parsed, will be saved in the csv folder.
Step 2 can be skipped if you already have a valid csv.

Step 3:

Obtain a csv file containing the test smells with the JNose tool.
Run the tool with the following button and follow these steps:
1. Go to the "Projects" tab and clone a project by providing a github url.
2. Go to the "By TestSmells" tab and analyze the project you want.
3. When the bar reaches 100% click on the yellow button next to it.
4. Click on "Export CSV" and save the file in the csv folder of the project.

Test smells file

Load a valid csv containing the test smell analysis results. It has to be located in the csv folder of this project.

Flaky tests file

Load a valid csv containing the flaky tests analysis results. It has to be located in the csv folder of this project.

Results from sfa_results.csv

Project name	Test method name	Is flaky	Assertion Roulette	Conditional Test Logic	Constructor Initialization	Default Test	Dependent Test
Alluxio-alluxio	getClientsTest	1	0	0	0	0	0
Alluxio-alluxio	faultTest	1	0	0	0	0	0
Alluxio-alluxio	createFileTest	1	0	0	0	0	0
Alluxio-alluxio	deleteFileTest	1	0	0	0	0	0
Alluxio-alluxio	deleteDirectoryWithDirectoriesTest	1	0	0	0	0	0
Alluxio-alluxio	isTest	1	0	0	0	0	0
Alluxio-alluxio	addCheckpointTest	1	0	0	0	0	0
Alluxio-alluxio	notFileCheckpointTest	1	0	0	0	0	0

Smelly Methods

A total number of 5339 methods with test smells have been found in 22245 methods analyzed.

Flaky methods

A total number of 811 flaky methods have been found in 22245 methods analyzed.

Smelly and Flaky Methods

A total number of 136 flaky methods with test smells have been found in 22245 methods analyzed.

Smelly and Flaky Methods

A total number of 136 flaky methods with test smells have been found in 811 flaky methods analyzed.

Test Smells

A total number of 16656 test smells have been found in 22245 methods analyzed.

Test Smells in Flaky Methods

A total number of 483 test smells have been found in 811 flaky methods analyzed.

Figure 47. Flowchart Smell Flaky Analyzer Tool

6. Used technologies

6.1 Git/Github

Figure 48. Git and GitHub logos, respectively

Git (*Figure 48*) [61], developed by Linus Torvalds, is undoubtedly the most widespread version control software. This tool allows you to work on projects as a team and monitor changes to source code files. GitHub, using Git, is a collaborative software development platform that allows a team to work in parallel on a project, hosted in a cloud repository, which can be downloaded, modified and updated by its members.

I have chosen to use Git and GitHub so that the development of the project can be done in a comfortable, secure and stable way. This tool facilitates the saving of the project in a cloud repository, allowing other people to see and use it if necessary. In addition, it offers the possibility of controlling different versions of the code in case of any conflict or unforeseen event. Another functionality is to create different branches to work on new and untested features without affecting the code of the main branch, which is in a more stable and verified state.

6.2 React

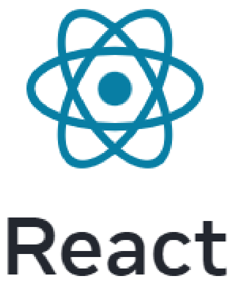


Figure 49. React Logo

React (*Figure 49*) [62] is an open-source JavaScript library for building user interfaces that was developed by Facebook and released in 2013. Its main focus is to create dynamic and interactive web applications by building reusable UI components. These components encapsulate the visual and functional aspects of an application's user interface, allowing developers to create complex UIs by composing these smaller building blocks together.

It was chosen for the development of this thesis’ application because it is a relatively simple way for creating attractive user interfaces. I have already used it before and had some experience with it, which made the implementation quicker, so that I could focus on other aspects of this project, such as the smells detection, rather than in the application’s user interface.

6.3 Javascript

Figure 50. JavaScript Logo

JavaScript (*Figure 50*) [63] is a versatile and widely used programming language that powers the interactive and dynamic elements of most modern websites and web applications, although its versatility extends beyond the web browser, allowing developers to build a wide range of applications across different platforms. It was initially created in the mid-1990s by Brendan Eich while he was working at Netscape, and it has since evolved into a fundamental technology for web development. JavaScript is the main programming language for this application, as it’s the base language for the React library to work.

6.4 HTML

Figure 51. HTML Logo

HTML (*Figure 51*) [64] is a fundamental language for creating and structuring content on the World Wide Web. It forms the backbone of web pages, defining the structure, layout, and elements that make up the content you see when you visit a website. HTML works in conjunction with other technologies like CSS (Cascading Style Sheets) and JavaScript to create visually appealing and interactive web experiences. It is the language used for defining the structure and components of the application of this thesis, as it works in conjunction with the React library.

6.5 CSS

Figure 52. CSS Logo

CSS (*Figure 52*) [65] is a crucial technology used in web development to control the presentation and visual styling of HTML documents. While HTML defines the structure and content of web pages, CSS focuses on how those elements should be displayed, including aspects like layout, colors, fonts, and spacing. CSS allows developers to separate the structure of a webpage from its design, making it easier to maintain and create consistent styles across a website.CSS is the technology that brings visual appeal and design to web pages. It's an essential tool for web developers, enabling them to create beautiful and responsive layouts, define typography, apply colors, and create a consistent user experience across different devices and screen sizes. It is used in this project because it works together with React and HTML to develop the appealing screens of the application.

6.6 Electron

Figure 53. Electron Logo

Electron (*Figure 53*) [66] is an open-source framework developed by GitHub. It enables developers to create desktop applications using web technologies such as HTML, CSS, and

JavaScript. Electron packages your web application along with a custom browser engine and exposes APIs that allow interaction with the underlying operating system. This way, you can create applications that work on Windows, macOS, and Linux using the same codebase. It has been chosen in this project because it allows to convert the web application developed using React into a desktop application, needed to execute the Phyton scripts that will run command line instructions.

6.7 Python

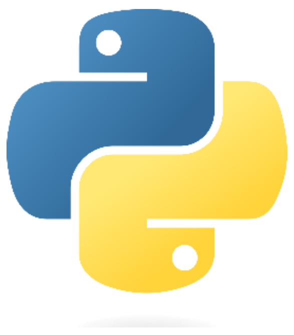


Figure 54. Python Logo

Python (*Figure 54*) [67] is a versatile, high-level programming language known for its simplicity, readability, and broad range of applications. Created by Guido van Rossum and first released in 1991, Python has gained immense popularity among developers due to its clean syntax, ease of use, and extensive standard library. It's used for web development, data analysis, scientific computing, artificial intelligence, automation, scripting, and more. It has been chosen in this project for executing some tasks, from creating the scripts for running command line instructions that allow to open browser windows (for executing JNose or accessing the FlakeFlagger webpage), to load and save csv files into local folders.

6.8 Visual Studio Code

Figure 55. Visual Studio Code Logo

Visual Studio Code (*Figure 55*) [68] is a widely used source code editor developed by Microsoft. It has gained immense popularity among developers due to its versatility, efficiency, and a vast ecosystem of extensions and features. Launched in 2015, VS Code has quickly become a go-to choose for programmers working with various programming languages and technologies. It has been chosen for this project because it provides a practical and useful way of programming using React and it is the most used editor in this regard.

6.9 Photoshop

Figure 56. Photoshop Logo

Adobe Photoshop (*Figure 566*) [69] is an image and graphics editing and retouching tool developed by Adobe Systems Incorporated. Photoshop has been chosen as the editor for the images used in the application's user interface, which have been specifically modified for the purpose of this project.

7. Implementation

7.1 Introduction

In the following chapter, the work done for the implementation of the solution of this project will be explained, as well as the development of the application obtained as a result of it. The main points of the development are the following ones:

- Implement a set of algorithms in the form of Python scripts that can use the output of a test smell detection tool (JNose in this case) and a flaky tests detection tool (FlakeFlagger in this case) as input for parsing them, merging them, and providing a csv file with the combined results, to later analyze them.

- Combine the different aspects of the process in a single application: the use of the test smell detection tool JNose for obtaining the test smells, the access to the FlakeFlagger webpage for obtaining the flaky tests, the execution of the Python scripts, the loading of input files and saving of output files, etc.

- Design and develop the user interface of the application for studying the relationship between code smells and flaky tests. This means creating the screens of the application that the user can interact with to access the set of functionalities that it offers and showing the analyzed output in a graphical and easy to understand way to the user.

7.2 Algorithms for parsing and merging

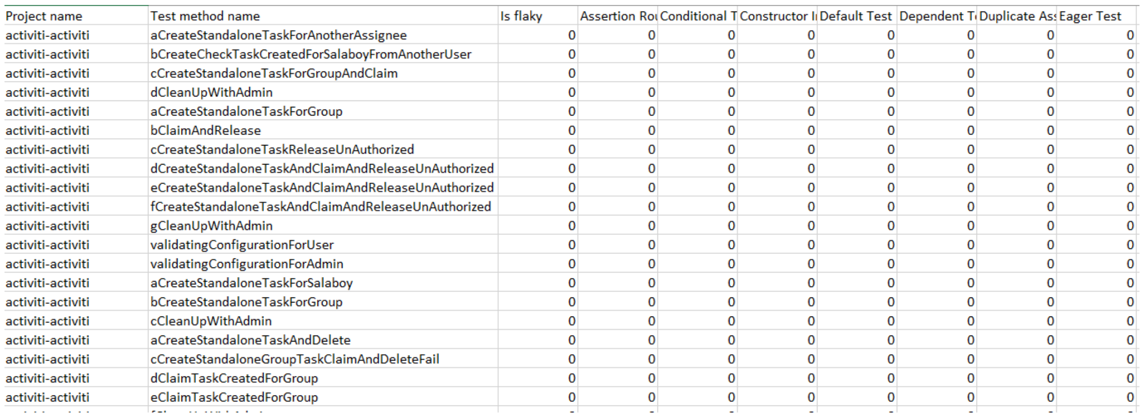
The first thing that was done before the implementation of the actual React application with its user interface was to do the parsing and merging of the results from JNose and FlakeFlagger by developing some Python scripts.

On the one hand, the first script is the one in charge of reading the csv file obtained from the FlakeFlagger project. This file, as it is obtained, contains a lot of unnecessary information when combined with the smell detection results (the number of times the test passed and failed, the exact run that failed for the first time, etc.), so some data is cut.

First, the script asks for the csv file from FlakeFlagger. We execute it and provide the file using the following command: "python get_test_flakiness.py test_results.csv". By using the csv.reader() function, each row of the file is read (which are the different tests of the dataset) and the columns that are not necessary are removed. Then, a new csv file named test_results_parsed.csv is saved into a local folder with the flaky tests information containing, for every test, the project name it belongs to, the test name and if it is flaky (marked as a 1) or not (marked as a 0).

On the other hand, a second script was created to parse the test smell results obtained from the JNose tool and merge them with the csv file obtained in the first script. To do that, it asks for both the smells csv file and the flaky tests csv file. It can be executed by writing the command: python get_test_smells.py <test smells file> <flaky tests file>".

This script creates an array of elements. Every element will contain the combined information for every test case. This is, the project name it belongs to, the test name, if it is flaky or not and the 21 different smells (marked as 1 for smelly and 0 for non-smelly). To do that, the flaky information from the previous step is obtained and added to the element and the file with the test smells is traversed. When a smell is found, the name of the test where the smell is can be read and the column of that test is updated to 1 in the new array. Finally, the result is saved in a file named sfa_results.csv. The contents of this file can be seen in *Figure 577*.



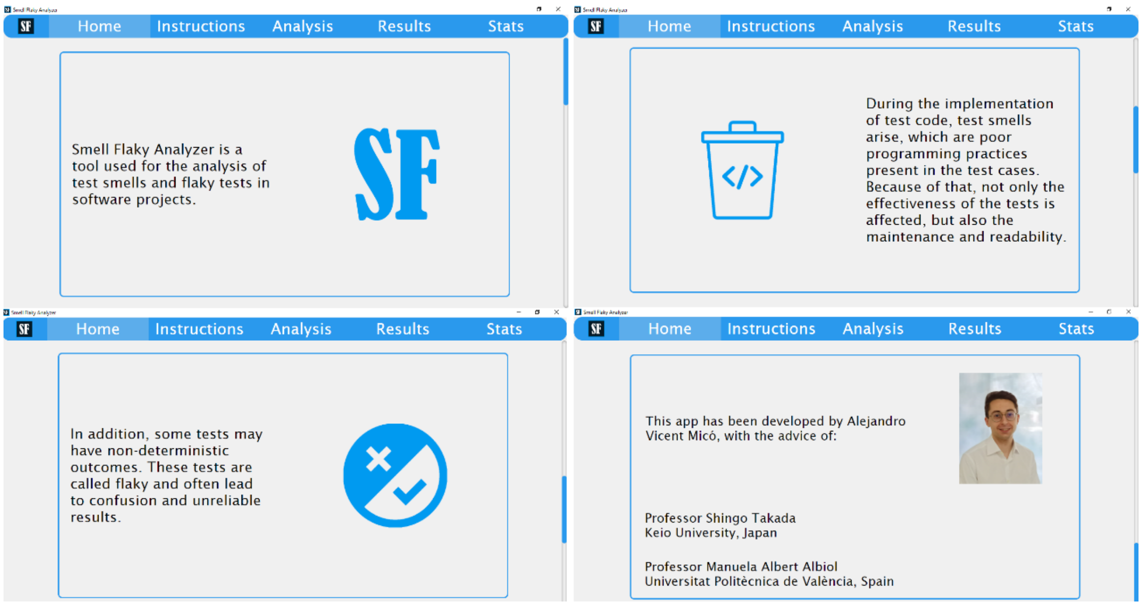
Project name	Test method name	Is flaky	Assertion Ro	Conditional T	Constructor I	Default Test	Dependent T	Duplicate As	Eager Test
activiti-activiti	aCreateStandaloneTaskForAnotherAssignee	0	0	0	0	0	0	0	0
activiti-activiti	bCreateCheckTaskCreatedForSalaboyFromAnotherUser	0	0	0	0	0	0	0	0
activiti-activiti	cCreateStandaloneTaskForGroupAndClaim	0	0	0	0	0	0	0	0
activiti-activiti	dCleanUpWithAdmin	0	0	0	0	0	0	0	0
activiti-activiti	aCreateStandaloneTaskForGroup	0	0	0	0	0	0	0	0
activiti-activiti	bClaimAndRelease	0	0	0	0	0	0	0	0
activiti-activiti	cCreateStandaloneTaskReleaseUnAuthorized	0	0	0	0	0	0	0	0
activiti-activiti	dCreateStandaloneTaskAndClaimAndReleaseUnAuthorized	0	0	0	0	0	0	0	0
activiti-activiti	eCreateStandaloneTaskAndClaimAndReleaseUnAuthorized	0	0	0	0	0	0	0	0
activiti-activiti	fCreateStandaloneTaskAndClaimAndReleaseUnAuthorized	0	0	0	0	0	0	0	0
activiti-activiti	gCleanUpWithAdmin	0	0	0	0	0	0	0	0
activiti-activiti	validatingConfigurationForUser	0	0	0	0	0	0	0	0
activiti-activiti	validatingConfigurationForAdmin	0	0	0	0	0	0	0	0
activiti-activiti	aCreateStandaloneTaskForSalaboy	0	0	0	0	0	0	0	0
activiti-activiti	bCreateStandaloneTaskForGroup	0	0	0	0	0	0	0	0
activiti-activiti	cCleanUpWithAdmin	0	0	0	0	0	0	0	0
activiti-activiti	aCreateStandaloneTaskAndDelete	0	0	0	0	0	0	0	0
activiti-activiti	cCreateStandaloneGroupTaskClaimAndDeleteFail	0	0	0	0	0	0	0	0
activiti-activiti	dClaimTaskCreatedForGroup	0	0	0	0	0	0	0	0
activiti-activiti	eClaimTaskCreatedForGroup	0	0	0	0	0	0	0	0

Figure 57. Combined Results from Script

7.3 Smell Flaky Analyzer Tool

To combine all the steps done, from the use of JNose to run the test smell detection to the execution of the scripts, a React tool with user interface named Smell Flaky Analyzer was developed. It is made up by a central screen that changes depending on the menu selected and a top bar menu with five buttons that allow the navigation among the different menus of the application: home, instructions, analysis, results and stats. The color palette selected is a combination of light blue, light grey and white, which combined form an appealing and easy to understand interface.

The first of the menus is the "Home" screen (*Figure 588*), which provides basic information about test smells, flaky tests, the application, and its developer, me. The different sections of the menu can be seen by using the right scrollbar to move between them.



Smell Flaky Analyzer is a tool used for the analysis of test smells and flaky tests in software projects.

During the implementation of test code, test smells arise, which are poor programming practices present in the test cases. Because of that, not only the effectiveness of the tests is affected, but also the maintenance and readability.

In addition, some tests may have non-deterministic outcomes. These tests are called flaky and often lead to confusion and unreliable results.

This app has been developed by Alejandro Vicent Micó, with the advice of:

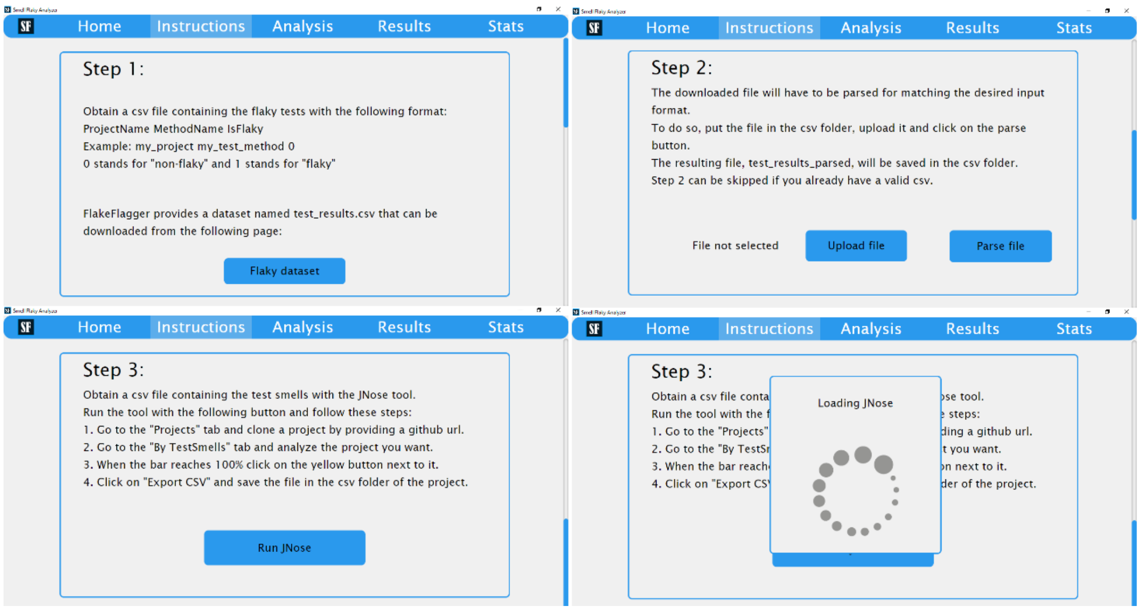
Professor Shingo Takada
Keio University, Japan

Professor Manuela Albert Albiol
Universitat Politècnica de València, Spain

Figure 58. Home Screen

The second menu is the “Instructions” screen (*Figure 59*), that shows all the steps that need to be done prior to the final merging between flaky test and test smells results. It is divided in:

- **Step 1:** Obtention of the flaky tests results from the FlakeFlagger tool. A button can be pressed to access the tool’s webpage, where the flaky dataset can be downloaded.
- **Step 2:** Parsing of the flaky results. The first of the python scripts is used in this step. To do that, the “Upload file” button must be pressed to load the flaky dataset. Then, the “Parse file” button is pressed to run the script through a command line instruction.
- **Step 3:** Obtention of the test smells detection results from JNose. When the”Run JNose” button is pressed, the tool is opened in a new screen. The user has to follow the steps explained in the screen to load projects and download the resulting csv file.



Step 1:

Obtain a csv file containing the flaky tests with the following format:
ProjectName MethodName IsFlaky
Example: my_project my_test_method 0
0 stands for "non-flaky" and 1 stands for "flaky"

FlakeFlagger provides a dataset named test_results.csv that can be downloaded from the following page:

Step 2:

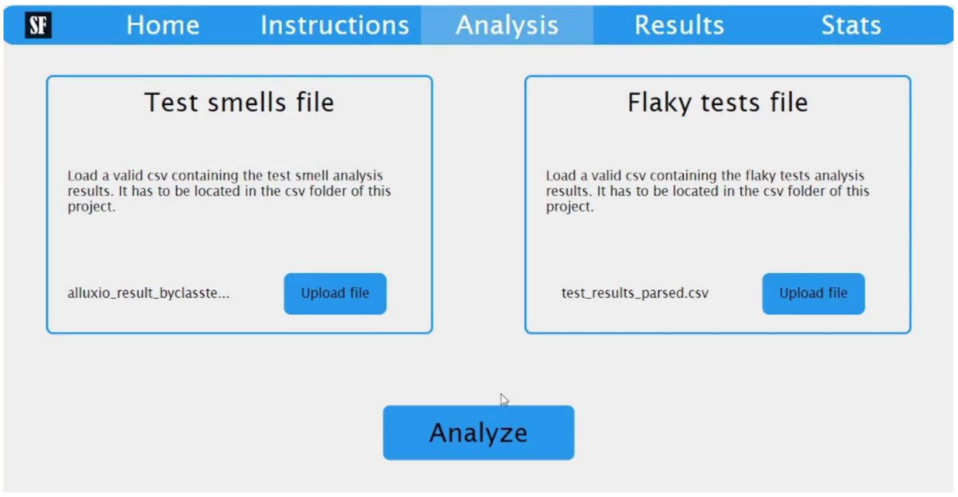
The downloaded file will have to be parsed for matching the desired input format.
To do so, put the file in the csv folder, upload it and click on the parse button.
The resulting file, test_results_parsed, will be saved in the csv folder.
Step 2 can be skipped if you already have a valid csv.

Step 3:

Obtain a csv file containing the test smells with the JNose tool.
Run the tool with the following button and follow these steps:
1. Go to the "Projects" tab and clone a project by providing a github url.
2. Go to the "By TestSmells" tab and analyze the project you want.
3. When the bar reaches 100% click on the yellow button next to it.
4. Click on "Export CSV" and save the file in the csv folder of the project.

Figure 59. Instructions Screen

The third menu is the “Analysis” screen (*Figure 60*), where the results from the test smells analysis and the flaky tests are combined. The “Upload file” button on the left asks for the csv file that was obtained in step 3 of the instructions screen. And the “Upload file” button on the right asks for the flaky tests csv file downloaded from the FlakeFlagger page. Then, by clicking the “Analzye” button, both files are taken as input of the second Python script, which creates a new file named “sfa_results.csv” with the combination of test smells and flaky tests and saves it in a local folder.



Load a valid csv containing the test smell analysis results. It has to be located in the csv folder of this project.

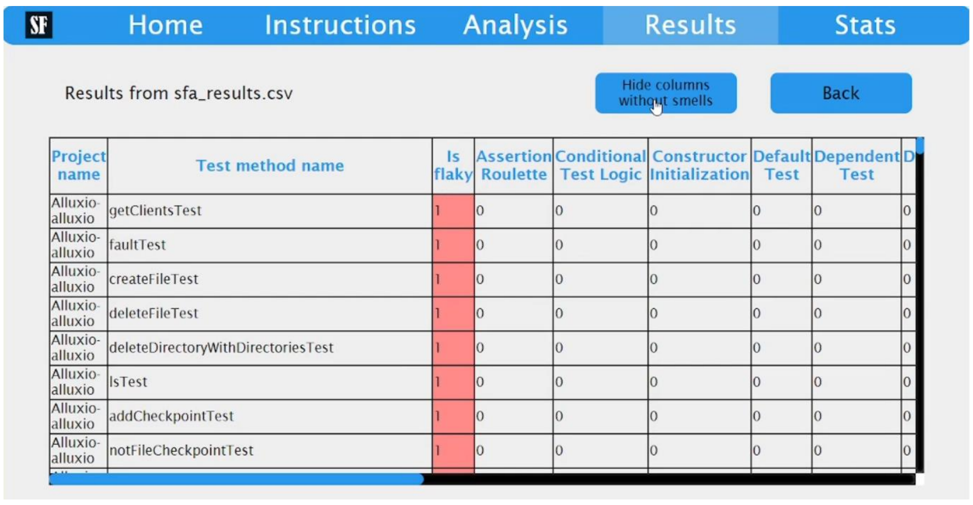
alluxio_result_byclasste...

Load a valid csv containing the flaky tests analysis results. It has to be located in the csv folder of this project.

test_results_parsed.csv

Figure 60. Analysis Screen

The fourth menu is the “Results” screen (*Figure 61*), where the results of the previous step can be seen. First, a button on the left will allow you to upload the csv file from a local folder to the application. Then, this file will be read, and the results will be put in a table that is comprehensible and straightforward. It shows, for every test case, the project name, the method name, if it is flaky “1” or not “0” and for every test smell if, it was found in the test case “1” or not “0”. For the third column and the remaining columns of test smells, the box is painted in red in case of a “1”, which makes it easier to read and understand.

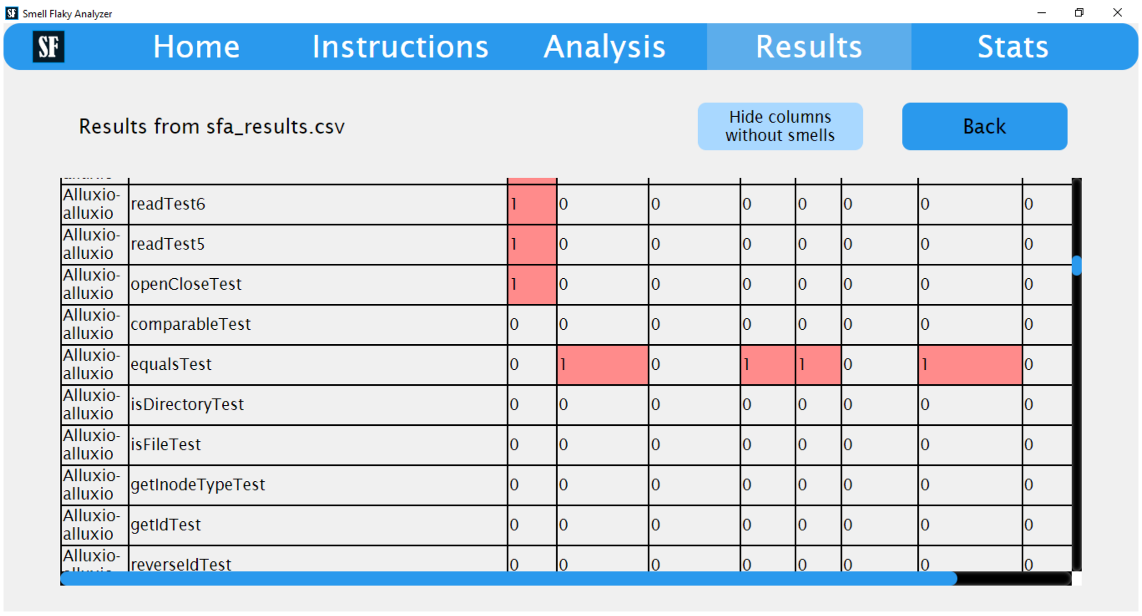


Results from sfa_results.csv

Project name	Test method name	Is flaky	Assertion Roulette	Conditional Test Logic	Constructor Initialization	Default Test	Dependent Test	D
Alluxio-alluxio	getClientsTest	1	0	0	0	0	0	0
Alluxio-alluxio	faultTest	1	0	0	0	0	0	0
Alluxio-alluxio	createFileTest	1	0	0	0	0	0	0
Alluxio-alluxio	deleteFileTest	1	0	0	0	0	0	0
Alluxio-alluxio	deleteDirectoryWithDirectoriesTest	1	0	0	0	0	0	0
Alluxio-alluxio	IsTest	1	0	0	0	0	0	0
Alluxio-alluxio	addCheckpointTest	1	0	0	0	0	0	0
Alluxio-alluxio	notFileCheckpointTest	1	0	0	0	0	0	0

Figure 61. Results Screen

When analyzing large projects, this table tends to become also very large, so a "Hide columns without smells" button was added. When pressed, it will hide all the columns whose test smell was not found in any method, so that it is easier to red and navigate through the table (*Figure 62*).

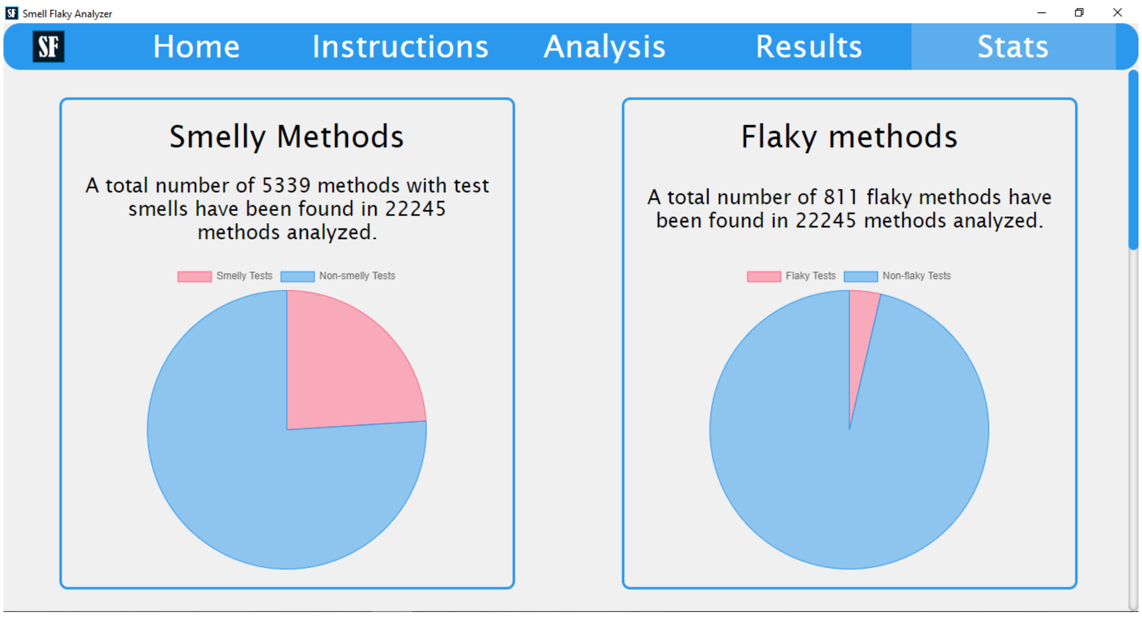


Results from sfa_results.csv

Alluxio-alluxio	readTest6	1	0	0	0	0	0	0	0
Alluxio-alluxio	readTest5	1	0	0	0	0	0	0	0
Alluxio-alluxio	openCloseTest	1	0	0	0	0	0	0	0
Alluxio-alluxio	comparableTest	0	0	0	0	0	0	0	0
Alluxio-alluxio	equalsTest	0	1	0	1	1	0	1	0
Alluxio-alluxio	isDirectoryTest	0	0	0	0	0	0	0	0
Alluxio-alluxio	isFileTest	0	0	0	0	0	0	0	0
Alluxio-alluxio	getInodeTypeTest	0	0	0	0	0	0	0	0
Alluxio-alluxio	getIdTest	0	0	0	0	0	0	0	0
Alluxio-	reverseIdTest	0	0	0	0	0	0	0	0

Figure 62. Results Screen with Hidden Columns

Finally, the fifth menu is the "Stats" screen (*Figure 63*), where some graphs are displayed to show additional information about the results of the detection. The first graph shows the total number of methods with test smells found (in red) in comparison to all the methods analyzed (in blue). The second one shows the total number of flaky methods found (in red) in comparison to all the methods analyzed (in blue).



Smelly Methods

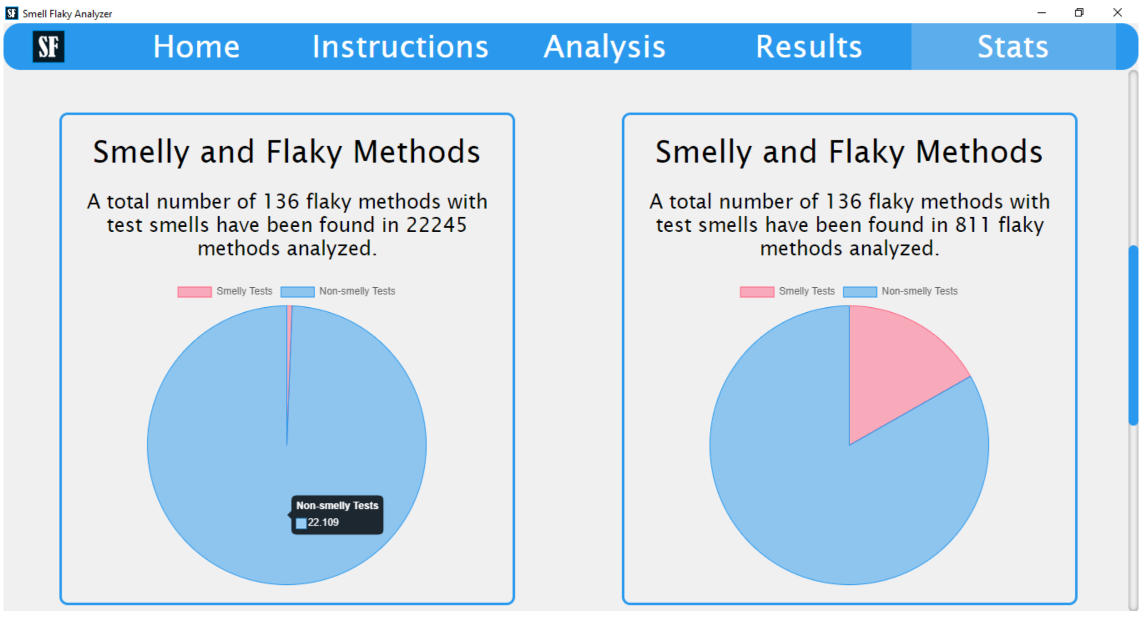
A total number of 5339 methods with test smells have been found in 22245 methods analyzed.

Flaky methods

A total number of 811 flaky methods have been found in 22245 methods analyzed.

Figure 63. Stats Screen 1

In the second part of this screen (*Figure 64*) the left graph shows the number of methods that were flaky and had at least some test smells at the same time (in red) in comparison to the total number of methods (in blue). The right graph shows the number of flaky methods with some test smells (in red) in comparison to all the methods analyzed (in blue).



Smelly and Flaky Methods

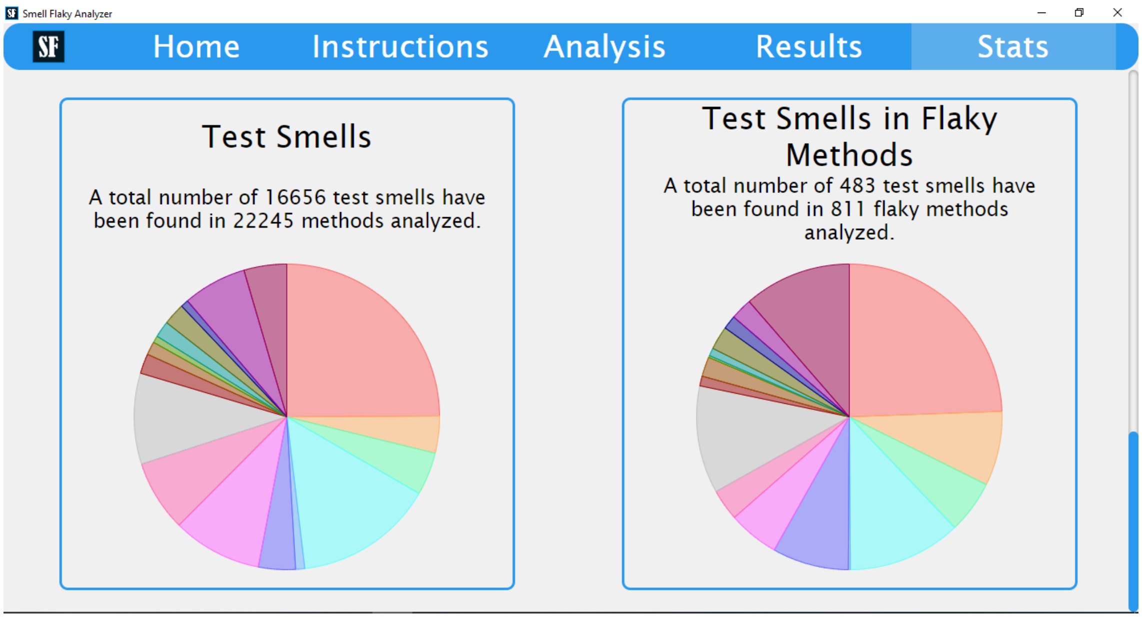
A total number of 136 flaky methods with test smells have been found in 22245 methods analyzed.

Smelly and Flaky Methods

A total number of 136 flaky methods with test smells have been found in 811 flaky methods analyzed.

Figure 64. Stats Screen 2

And the third part of the menu shows the final two graphs (*Figure 65*). On the left, the different number of smells found in all the test methods are shown in different colors. If the cursor is put on one color, it shows which type of smell it is. On the right, the test smells found in the flaky methods are shown, divided also by color.



Test Smells

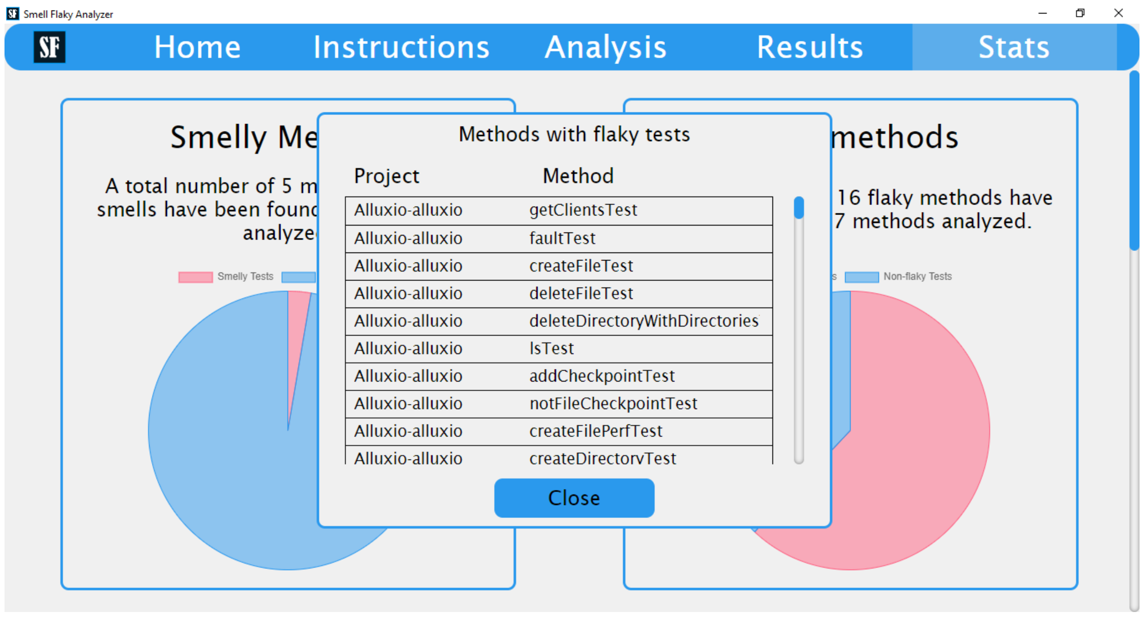
A total number of 16656 test smells have been found in 22245 methods analyzed.

Test Smells in Flaky Methods

A total number of 483 test smells have been found in 811 flaky methods analyzed.

Figure 65. Stats Screen 3

When a slide of the graph is clicked, the details on which project and method are part of that slide are shown. For example, in the first graph on the right, if the red slide is clicked, it will show the name of all the flaky methods that were found (*Figure 66*).



Methods with flaky tests

Project	Method
Alluxio-alluxio	getClientsTest
Alluxio-alluxio	faultTest
Alluxio-alluxio	createFileTest
Alluxio-alluxio	deleteFileTest
Alluxio-alluxio	deleteDirectoryWithDirectories
Alluxio-alluxio	lsTest
Alluxio-alluxio	addCheckpointTest
Alluxio-alluxio	notFileCheckpointTest
Alluxio-alluxio	createFilePerfTest
Alluxio-alluxio	createDirectoryTest

Figure 66. Stats Screen Explanatory Menu

The development of the application was done using the Visual Studio Code editor, which allows to organize the different folders and files of the project in a practical way (*Figure 67*):

- **node_modules folder:** stores all the Javascript and React libraries that are installed in the project.
- **public folder:** contains basic files, such as index.html or main.js, which will be executed at the beginning of the application start.
- **src folder:** contains all the elements necessaries to build the application:

  -assets folder: the images used in the project, such as logos or icons.

  -components folder: the Javascript and CSS component files that are used to build the different screens of the application.

  -csv folder: the csv files used as input in the application and the ones obtained as output.

  -reducers folder: the reducers files used by React Redux (explained after this part).

  -scripts folder: the Python scripts executed from the UI of the application.

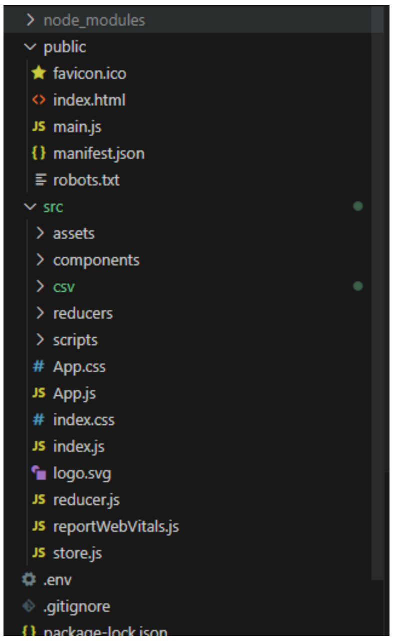
  -index.js and index.css files: the root files that start calling the other components so that they are displayed on the screen.

  -App.js and App.css: the main component of the application, called from index.js, that starts having functionality and organizing the other components.

-reducer.js and store.js files: the Javascript files that allow the React Redux store to work properly.

-logo.svg file: the image used as icon of the application.

- **package.json file:** contains the information of the project, such as the version, the dependencies and the scripts that can be executed.



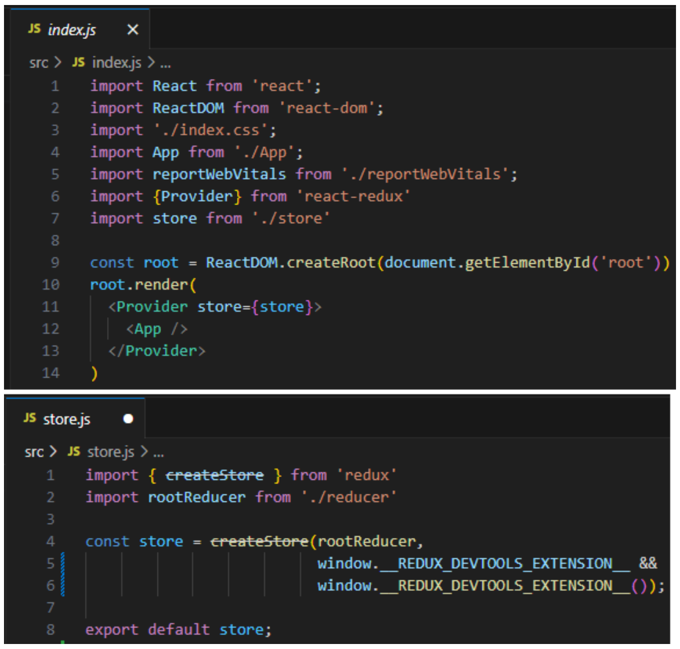
```
> node_modules
v public
    favicon.ico
    index.html
    main.js
    manifest.json
    robots.txt
v src
  > assets
  > components
  > csv
  > reducers
  > scripts
    App.css
    App.js
    index.css
    index.js
    logo.svg
    reducer.js
    reportWebVitals.js
    store.js
  .env
  .gitignore
  package-lock.json
```

Figure 67. Project Folders and Files

React uses “states” to store and keep track of the current state of the application. In this way, when the variables that are stored in the state of React are modified, the components are updated and the user interface is reloaded, so that the user is able to see the changes to the state. For example, the state can store a boolean variable called “buttonPressed” that controls if a button has been pressed or not. This variable is initialized in the state to false, meaning that it still has not been pressed. In addition, a popup menu component conditionally renders based on the “buttonPressed” variable: when the button is clicked, the menu activates. Then, if we click the button, the handler method associated to it will modify the state of the component, updating the variable to “true” and rendering the component again so that the popup menu is now shown.

This procedure, although a bit hard to get familiar with at the beginning, becomes really useful when a bit of experience on React is acquired. However, when large applications are developed and a lot of components are created, it becomes trickier. This is because each component has its state, so if multiple components want to communicate to change someone else’s state, a lot of code has to be added, and it becomes harder to understand.

To solve this problem, many people choose to use React Redux instead of the default React state mechanism. Redux allows to encapsulate your application inside a “store” which will manage the state of the application (*Figure 68*).



index.js

src > index.js > ...

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import {Provider} from 'react-redux'
import store from './store'

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(
  <Provider store={store}>
    <App />
  </Provider>
)
```

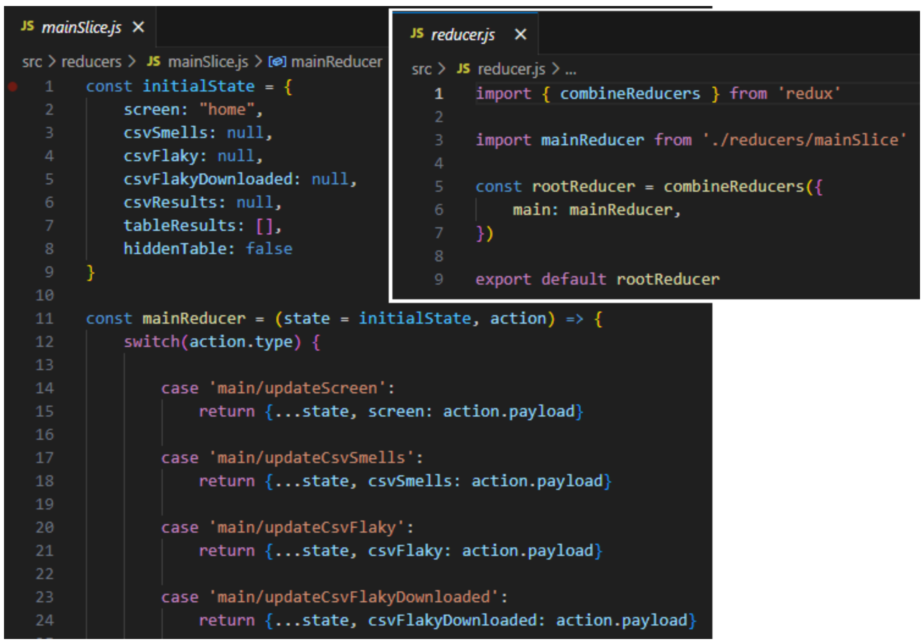
store.js

src > store.js > ...

```
import { ~~createStore~~ } from 'redux'
import rootReducer from './reducer'

const store = ~~createStore~~(rootReducer,
                          window.__REDUX_DEVTOOLS_EXTENSION__ &&
                          window.__REDUX_DEVTOOLS_EXTENSION__());

export default store;
```

Figure 68. Store Class

This store allows the use of “reducers” which will contain the information of the global state of the application. By default, a single reducer is created, but “combineReducers” can be used, which is a technique that permits to split the global state into different slices, allowing the developer to organize the different variables of the project depending on the purpose they serve (*Figure 69*).



mainSlice.js

src > reducers > mainSlice.js > mainReducer

```
const initialState = {
    screen: "home",
    csvSmells: null,
    csvFlaky: null,
    csvFlakyDownloaded: null,
    csvResults: null,
    tableResults: [],
    hiddenTable: false
}

const mainReducer = (state = initialState, action) => {
    switch(action.type) {

        case 'main/updateScreen':
            return {...state, screen: action.payload}

        case 'main/updateCsvSmells':
            return {...state, csvSmells: action.payload}

        case 'main/updateCsvFlaky':
            return {...state, csvFlaky: action.payload}

        case 'main/updateCsvFlakyDownloaded':
            return {...state, csvFlakyDownloaded: action.payload}
```

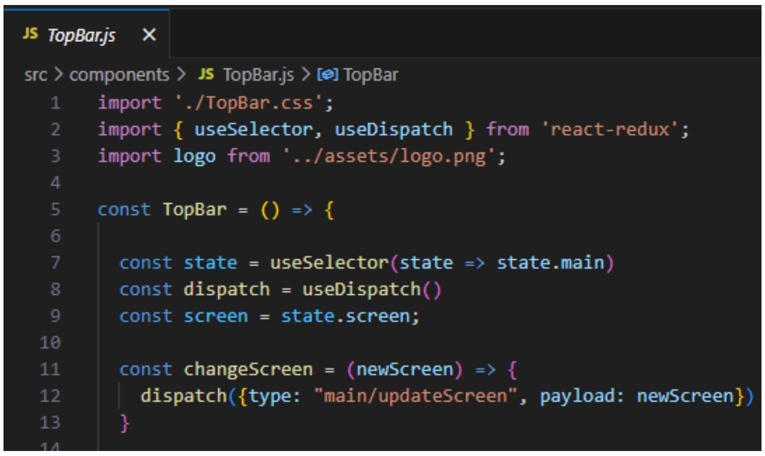
reducer.js

src > reducer.js > ...

```
import { combineReducers } from 'redux'

import mainReducer from './reducers/mainSlice'

const rootReducer = combineReducers({
    main: mainReducer,
})

export default rootReducer
```

Figure 69. Reducers Classes

The variables that are part of these global state in the reducers can be accessed from any of the components of the application, by using the React Hooks “useSelector” and “useDispatch” (*Figure 70*).



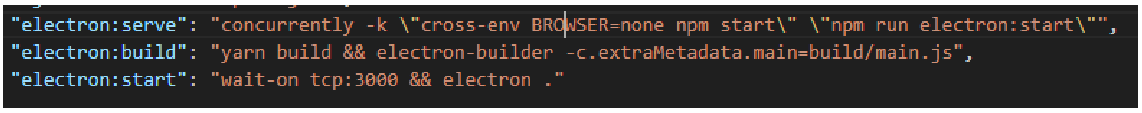
TopBar.js

src > components > TopBar.js > TopBar

```
import './TopBar.css';
import { useSelector, useDispatch } from 'react-redux';
import logo from '../assets/logo.png';

const TopBar = () => {

  const state = useSelector(state => state.main)
  const dispatch = useDispatch()
  const screen = state.screen;

  const changeScreen = (newScreen) => {
    dispatch({type: "main/updateScreen", payload: newScreen})
  }
```

Figure 70. Use of useSelector and useDispatch

Finally, the execution of the Python scripts has been possible thanks to the introduction of Electron to this project, which allows the execution of the React application as a desktop application instead of a web one. To run it, it is enough to navigate to the project’s folder and write the command “npm run electron:serve”, which will cancel the launch of the app in the navigator and will instead lunch it as a desktop app on port 3000 (*Figure 71*).



```
"electron:serve": "concurrently -k \"cross-env BROWSER=none npm start\" \"npm run electron:start\"",
"electron:build": "yarn build && electron-builder -c.extraMetadata.main=build/main.js",
"electron:start": "wait-on tcp:3000 && electron ."
```

Figure 71. Electron Scripts

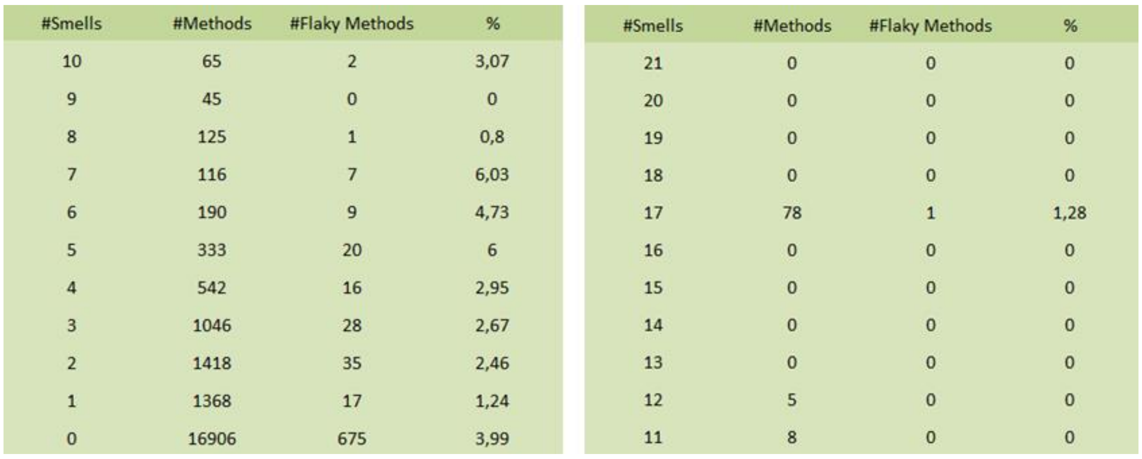
8. Validation

At the end of the implementation, we must verify the utility of the application and the work done, as well as the obtained results. A total of 24 projects with over 22,000 test methods were used as input for obtaining the results. First, the obtained results from the execution of the python scripts will be explained, followed by the results shown once the application was finished. Finally, some findings obtained during the validation regarding sources of flakiness will be explained.

8.1 Obtained results

By executing the Python scripts developed and explained before, a csv file is obtained, where the results from the analysis of the test smells and flaky tests can be studied. This file was opened in an excel sheet and some data was obtained. First of all, *Figure 72* shows the number of methods and flaky methods that had a specific amount of test smells, and its percentage. For example, 1368 methods had one test smell. From these number of methods, 17 of them were flaky, which means only the 1,24% of the 1368 methods were flaky. As another example, 542 methods had 4 test smells. Of these methods, only 16 were flaky, which means the 2,95% of the total amount of tests.

We can observe that the number of methods decrease when more smells are found in the same method. This is expected, as it might be common to find one, two or a few smells in a test case, but finding many smells in the same method is more complicated.



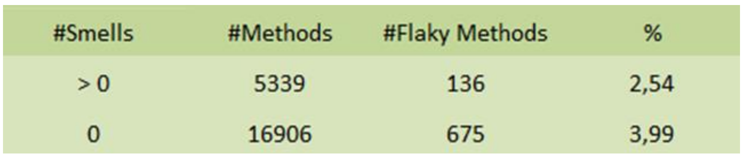
#Smells	#Methods	#Flaky Methods	%
10	65	2	3,07
9	45	0	0
8	125	1	0,8
7	116	7	6,03
6	190	9	4,73
5	333	20	6
4	542	16	2,95
3	1046	28	2,67
2	1418	35	2,46
1	1368	17	1,24
0	16906	675	3,99

#Smells	#Methods	#Flaky Methods	%
21	0	0	0
20	0	0	0
19	0	0	0
18	0	0	0
17	78	1	1,28
16	0	0	0
15	0	0	0
14	0	0	0
13	0	0	0
12	5	0	0
11	8	0	0

Figure 72. Methods and Flaky Methods

In *Figure 73*, we can see that there are 16906 methods that did not have any test smell. From these, only 675 were flaky which means the 3,99% of them. On the other hand, 5339 methods had at least one or more test smells. From this amount, 136 were flaky. That is 2,54% of the methods with test smells. It can be observed that the percentages of flaky methods with test smells is pretty low (2,54%). It is even less than the percentage of flaky methods without any test smell, which is

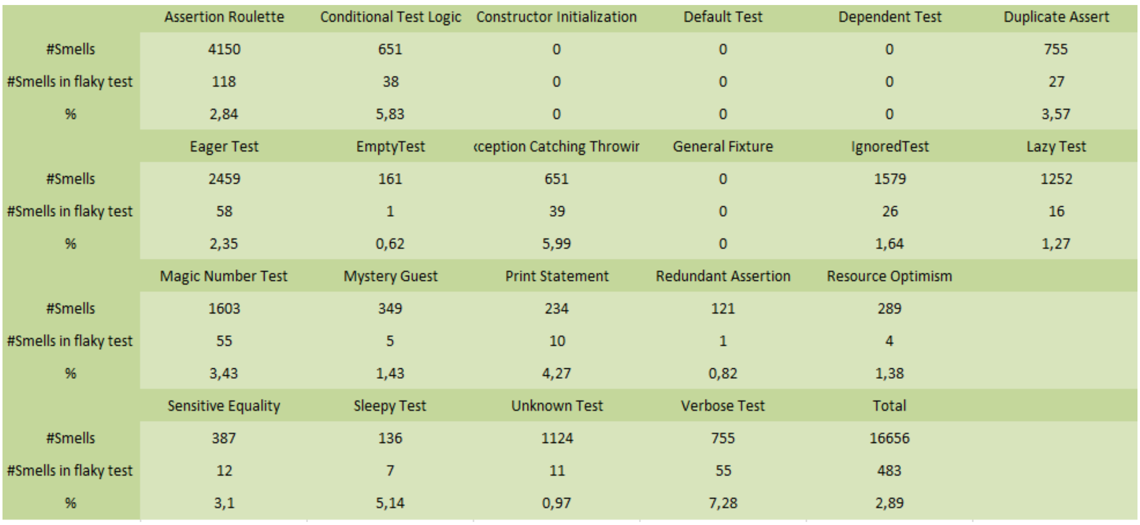
3,99%. In this first analysis, it looks like there is no clear correlation between a method been flaky and having smells.



#Smells	#Methods	#Flaky Methods	%
> 0	5339	136	2,54
0	16906	675	3,99

Figure 73. Methods with and without Smells

Next, the numbers were split depending on the different 21 smells (*Figure 74*). For every test smell, the number of methods that were found with that specific smell are shown in the "#Smells" row and from that number, the amount of flaky methods are shown in the "#Smells in flaky test" row. And in the third row, the percentage of that smell being in a flaky test is displayed. Some smells score higher percentages than others, but in general, they are still very low.



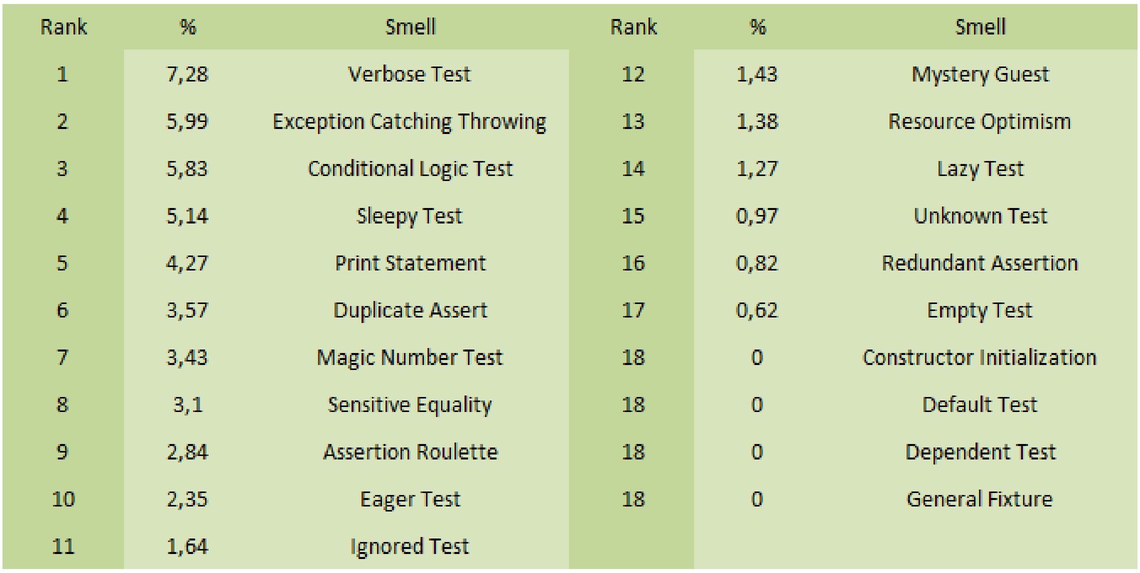
	Assertion Roulette	Conditional Test Logic	Constructor Initialization	Default Test	Dependent Test	Duplicate Assert
#Smells	4150	651	0	0	0	755
#Smells in flaky test	118	38	0	0	0	27
%	2,84	5,83	0	0	0	3,57
	Eager Test	EmptyTest	ception Catching Throwir	General Fixture	IgnoredTest	Lazy Test
#Smells	2459	161	651	0	1579	1252
#Smells in flaky test	58	1	39	0	26	16
%	2,35	0,62	5,99	0	1,64	1,27
	Magic Number Test	Mystery Guest	Print Statement	Redundant Assertion	Resource Optimism	
#Smells	1603	349	234	121	289	
#Smells in flaky test	55	5	10	1	4	
%	3,43	1,43	4,27	0,82	1,38	
	Sensitive Equality	Sleepy Test	Unknown Test	Verbose Test	Total	
#Smells	387	136	1124	755	16656	
#Smells in flaky test	12	7	11	55	483	
%	3,1	5,14	0,97	7,28	2,89	

Figure 74. Test Smells in Flaky Tests by Type

Finally, the rank of smells in flaky tests was created (*Figure 76*). Here, we can more clearly see that all the percentages are below 8%, which means that test smells do not seem to strongly affect the creation of flaky tests. For a given smell, the percentage means the number of times that smells appears in a flaky test divided by the number of times it appears in total (*Figure 75*). For the smell with highest presence Verbose Test, only 7,28% of the smells were in a flaky test. Most of them (17/21) were below 5%.

$$\% = \frac{\#\text{smells in flaky tests}}{\#\text{smells in total}}$$

Figure 75. Test Smells Percentage Formula

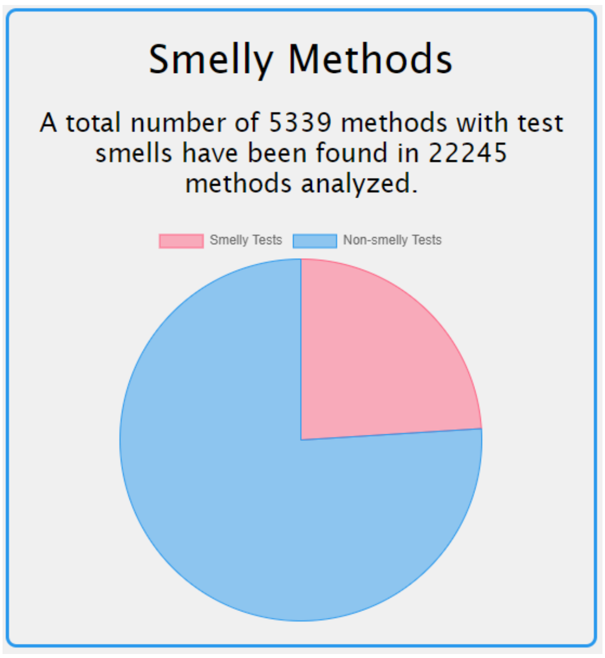


Rank	%	Smell	Rank	%	Smell
1	7,28	Verbose Test	12	1,43	Mystery Guest
2	5,99	Exception Catching Throwing	13	1,38	Resource Optimism
3	5,83	Conditional Logic Test	14	1,27	Lazy Test
4	5,14	Sleepy Test	15	0,97	Unknown Test
5	4,27	Print Statement	16	0,82	Redundant Assertion
6	3,57	Duplicate Assert	17	0,62	Empty Test
7	3,43	Magic Number Test	18	0	Constructor Initialization
8	3,1	Sensitive Equality	18	0	Default Test
9	2,84	Assertion Roulette	18	0	Dependent Test
10	2,35	Eager Test	18	0	General Fixture
11	1,64	Ignored Test			

Figure 76. Rank of Test Smells in Flaky Tests

8.2 Application results

After developing the application, the results of analyzing more than 22000 methods can be better displayed and studied. From the "Stats" screen, we can see if test smells and flaky tests are common in software projects. For test smells yes, as 24.00% of the test methods had at least one test smell (*Figure 77*).



Smelly Methods

A total number of 5339 methods with test smells have been found in 22245 methods analyzed.

Figure 77. Test Smells in Test Methods

For flaky tests not so much, as only 3.64% of the test methods were flaky (*Figure 78*).

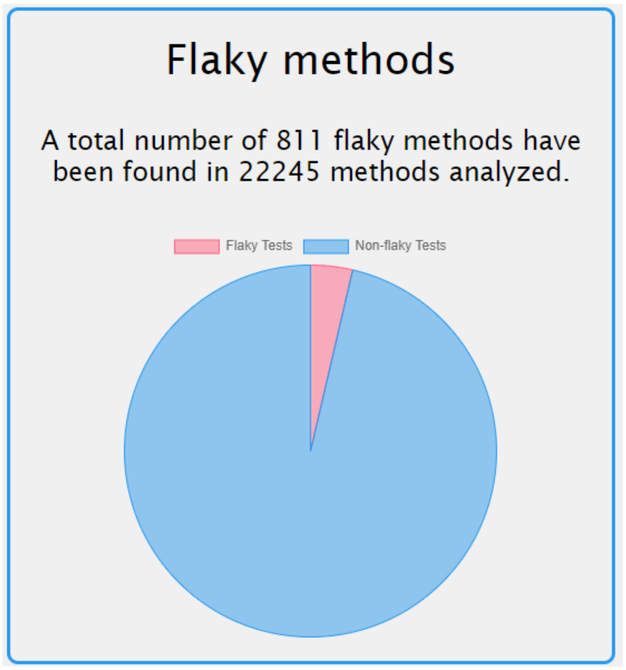


Figure 78. Flakiness in Test Methods

From the next two graphs, we can see which test smells are the most common in software projects (*Figure 79*). For the 24 projects analyzed, these smells are Assertion Roulette with a percentage of 24.91%, Eager Test with a percentage of 14.76%, Magic Number Test with a percentage of 9.62% and Ignored Test with a percentage of 9.48%.

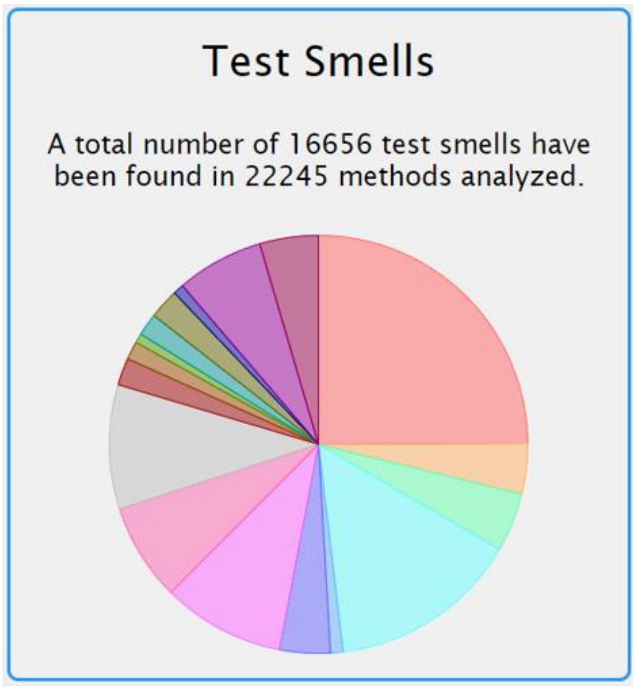


Figure 79. Test Smell Types in Software Projects

And which test smells are the most common in flaky tests (*Figure 80*). These are Verbose Test in 7.28% of the flaky tests, Exception Catching Throwing in 5.99% of the tests, Conditional Logic Test in 5.83% of the tests and Sleepy Test in 5.14% of the tests.

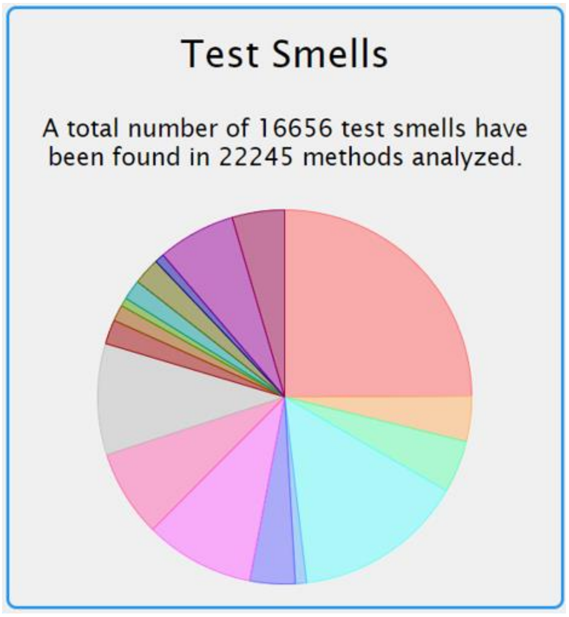


Figure 80. Test Smell Types in Flaky Tests

And from the last two graphs, we can observe if test smells are good indicators of flakiness in the test methods. Only 136 flaky methods had test smells, which is the 16,76% of the 811 flaky methods (*Figure 82*), in contrast with 24% of the total 22245 methods (*Figure 81*). In addition, and as we could see in *Figure 76*, all the percentages are below 10%. Therefore, there is not enough evidence to support a strong correlation between test smells and flaky tests.

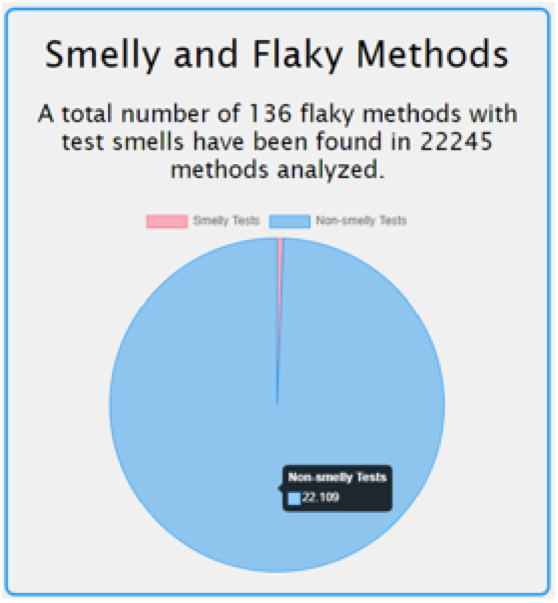


Figure 81. Test Smells in Flaky Methods 1

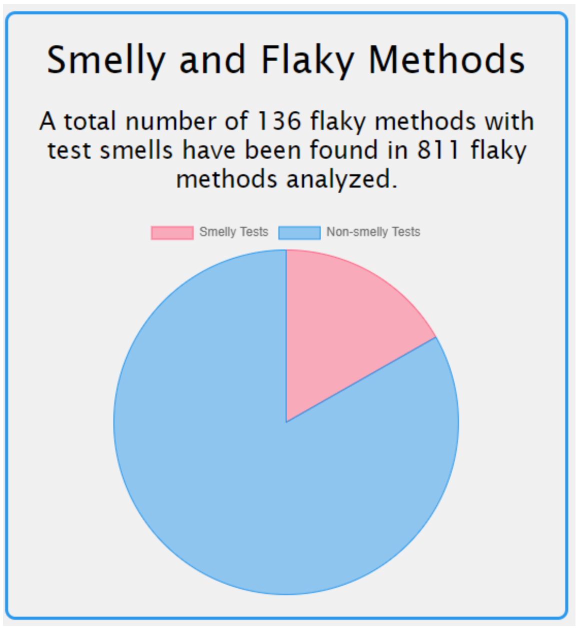
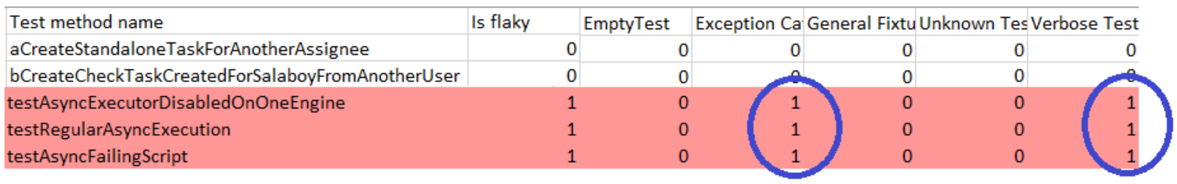


Figure 82. Test Smells in Flaky Methods 2

8.3 Manual inspections and findings

After the previous step, I decided to manually inspect some of the methods marked as flaky to look for some common indicators of flakiness. A lot of the reviewed flaky tests (*Figure 83*) contained one or more of the first four smells in the *Figure 76*.

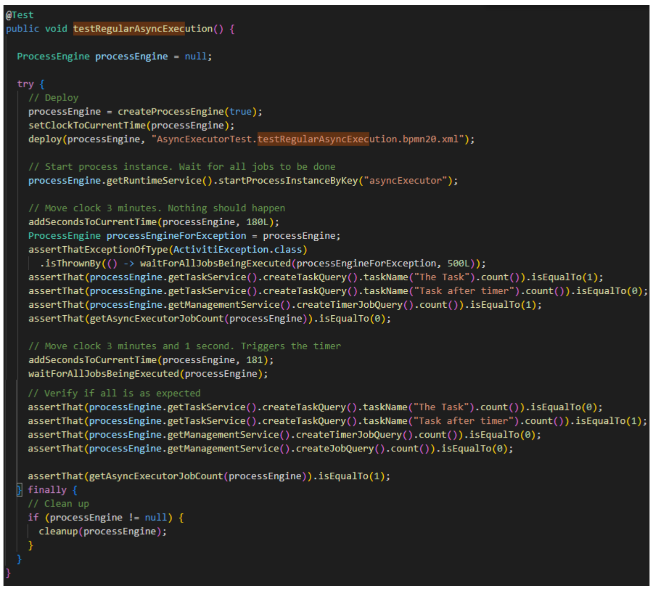


Test method name	Is flaky	EmptyTest	Exception Ca	General Fixtu	Unknown Tes	Verbose Test
aCreateStandaloneTaskForAnotherAssignee	0	0	0	0	0	0
bCreateCheckTaskCreatedForSalaboyFromAnotherUser	0	0	0	0	0	0
testAsyncExecutorDisabledOnOneEngine	1	0	1	0	0	1
testRegularAsyncExecution	1	0	1	0	0	1
testAsyncFailingScript	1	0	1	0	0	1

Figure 83. Tests with Most Common Smells

Some examples of the smells found are:

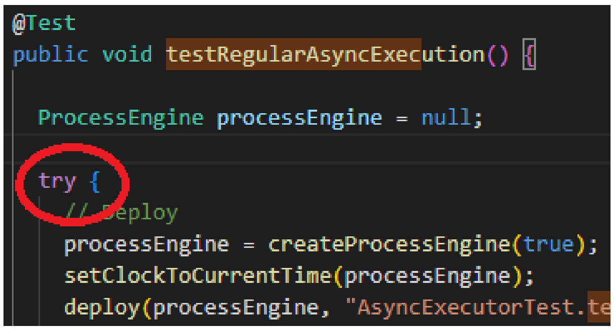
- **Verbose test:** a method containing a large number of statements (JNose establishes the threshold in 30 statements per method), found in 7.28% of the flaky tests (*Figure 84*).



```java
@Test
public void testRegularAsyncExecution() {

  ProcessEngine processEngine = null;

  try {
    // Deploy
    processEngine = createProcessEngine(true);
    setClockToCurrentTime(processEngine);
    deploy(processEngine, "AsyncExecutorTest.testRegularAsyncExecution.bpmn20.xml");

    // Start process instance. Wait for all jobs to be done
    processEngine.getRuntimeService().startProcessInstanceByKey("asyncExecutor");

    // Move clock 3 minutes. Nothing should happen
    addSecondsToCurrentTime(processEngine, 180L);
    ProcessEngine processEngineForException = processEngine;
    assertThatExceptionOfType(ActivitiException.class)
      .isThrownBy(() -> waitForAllJobsBeingExecuted(processEngineForException, 500L));
    assertThat(processEngine.getTaskService().createTaskQuery().taskName("The Task").count()).isEqualTo(1);
    assertThat(processEngine.getTaskService().createTaskQuery().taskName("Task after timer").count()).isEqualTo(0);
    assertThat(processEngine.getManagementService().createTimerJobQuery().count()).isEqualTo(1);
    assertThat(getAsyncExecutorJobCount(processEngine)).isEqualTo(0);

    // Move clock 3 minutes and 1 second. Triggers the timer
    addSecondsToCurrentTime(processEngine, 181);
    waitForAllJobsBeingExecuted(processEngine);

    // Verify if all is as expected
    assertThat(processEngine.getTaskService().createTaskQuery().taskName("The Task").count()).isEqualTo(0);
    assertThat(processEngine.getTaskService().createTaskQuery().taskName("Task after timer").count()).isEqualTo(1);
    assertThat(processEngine.getManagementService().createTimerJobQuery().count()).isEqualTo(0);
    assertThat(processEngine.getManagementService().createJobQuery().count()).isEqualTo(0);

    assertThat(getAsyncExecutorJobCount(processEngine)).isEqualTo(1);
  } finally {
    // Clean up
    if (processEngine != null) {
      cleanup(processEngine);
    }
  }
}
```

Figure 84. Verbose Test Smell Found Example

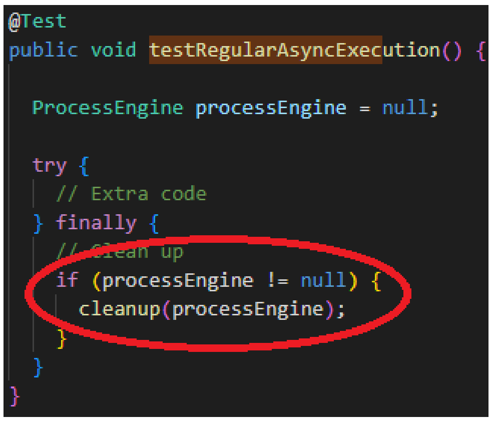
- **Exception catching throwing:** a test method that contains either a throw statement or a catch clause, found in 5.99% of the flaky tests (*Figure 85*).



```java
@Test
public void testRegularAsyncExecution() {

  ProcessEngine processEngine = null;

  try {
    // Deploy
    processEngine = createProcessEngine(true);
    setClockToCurrentTime(processEngine);
    deploy(processEngine, "AsyncExecutorTest.te
```

Figure 85. Exception Catching Throwing Smell Found Example

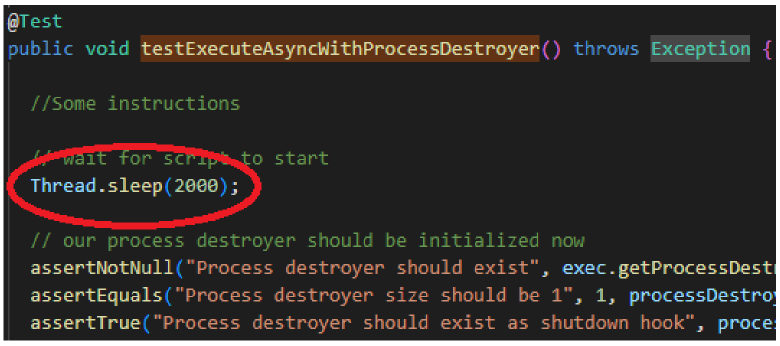
- **Conditional logic test:** a test method containing one or more control statements, found in 5.83% of the flaky tests (*Figure 86*).



```java
@Test
public void testRegularAsyncExecution() {

  ProcessEngine processEngine = null;

  try {
    // Extra code
  } finally {
    // Clean up
    if (processEngine != null) {
      cleanup(processEngine);
    }
  }
}
```

Figure 86. Conditional Logic Test Found Example

- **Sleepy test:** a method that invokes the “Thread.sleep()” instruction, found in 5.14% of the flaky tests (*Figure 87*).

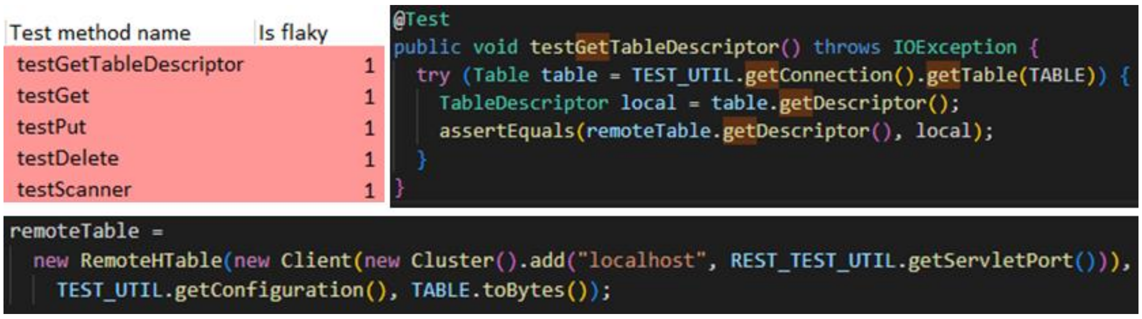


```
@Test
public void testExecuteAsyncWithProcessDestroyer() throws Exception {

  //Some instructions

  // wait for script to start
  Thread.sleep(2000);

  // our process destroyer should be initialized now
  assertNotNull("Process destroyer should exist", exec.getProcessDest
  assertEquals("Process destroyer size should be 1", 1, processDestro
  assertTrue("Process destroyer should exist as shutdown hook", proce
```

Figure 87. Sleepy Test Found Example

In addition, some sources of flakiness mentioned in section 3.4 were also predominant in the test cases marked as flaky:

- **REST connections**; that is, relying on external resources (*Figure 88*).



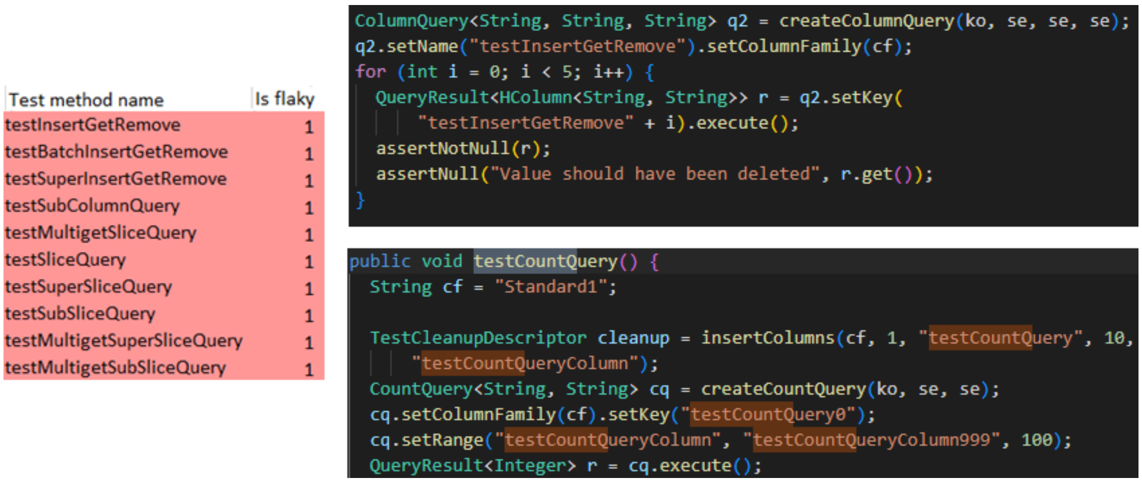
Test method name	Is flaky
testGetTableDescriptor	1
testGet	1
testPut	1
testDelete	1
testScanner	1

```
@Test
public void testGetTableDescriptor() throws IOException {
  try (Table table = TEST_UTIL.getConnection().getTable(TABLE)) {
    TableDescriptor local = table.getDescriptor();
    assertEquals(remoteTable.getDescriptor(), local);
  }
}
```

```
remoteTable =
  new RemoteHTable(new Client(new Cluster().add("localhost", REST_TEST_UTIL.getServletPort())),
    TEST_UTIL.getConfiguration(), TABLE.toBytes());
```

Figure 88. REST Connections Found Example

- **Database queries**, which is also relying on external resources (*Figure 89*).



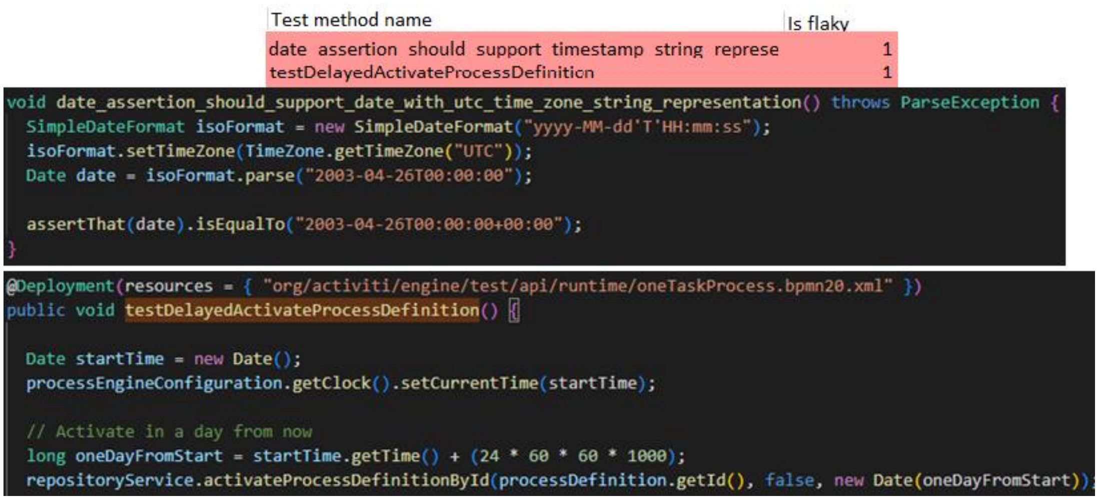
Test method name	Is flaky
testInsertGetRemove	1
testBatchInsertGetRemove	1
testSuperInsertGetRemove	1
testSubColumnQuery	1
testMultigetSliceQuery	1
testSliceQuery	1
testSuperSliceQuery	1
testSubSliceQuery	1
testMultigetSuperSliceQuery	1
testMultigetSubSliceQuery	1

```
ColumnQuery<String, String, String> q2 = createColumnQuery(ko, se, se, se);
q2.setName("testInsertGetRemove").setColumnFamily(cf);
for (int i = 0; i < 5; i++) {
  QueryResult<HColumn<String, String>> r = q2.setKey(
      "testInsertGetRemove" + i).execute();
  assertNotNull(r);
  assertNull("Value should have been deleted", r.get());
}
```

```
public void testCountQuery() {
  String cf = "Standard1";

  TestCleanupDescriptor cleanup = insertColumns(cf, 1, "testCountQuery", 10,
      "testCountQueryColumn");
  CountQuery<String, String> cq = createCountQuery(ko, se, se);
  cq.setColumnFamily(cf).setKey("testCountQuery0");
  cq.setRange("testCountQueryColumn", "testCountQueryColumn999", 100);
  QueryResult<Integer> r = cq.execute();
```

Figure 89. Database Queries Found Example

- **The Date class**, with its time dependent instructions (*Figure 90*).



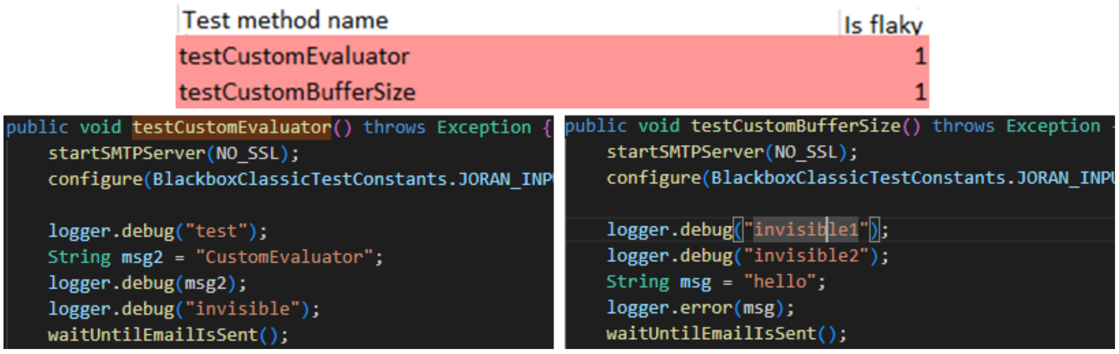
Test method name	Is flaky
date assertion should support timestamp string represe	1
testDelayedActivateProcessDefinition	1

```
void date_assertion_should_support_date_with_utc_time_zone_string_representation() throws ParseException {
  SimpleDateFormat isoFormat = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss");
  isoFormat.setTimeZone(TimeZone.getTimeZone("UTC"));
  Date date = isoFormat.parse("2003-04-26T00:00:00");

  assertThat(date).isEqualTo("2003-04-26T00:00:00+00:00");
}
```

```
@Deployment(resources = { "org/activiti/engine/test/api/runtime/oneTaskProcess.bpmn20.xml" })
public void testDelayedActivateProcessDefinition() {

  Date startTime = new Date();
  processEngineConfiguration.getClock().setCurrentTime(startTime);

  // Activate in a day from now
  long oneDayFromStart = startTime.getTime() + (24 * 60 * 60 * 1000);
  repositoryService.activateProcessDefinitionById(processDefinition.getId(), false, new Date(oneDayFromStart));
```

Figure 90. Date Class Found Example

- **SMTP connections**, that means relying on network connections (*Figure 91*).



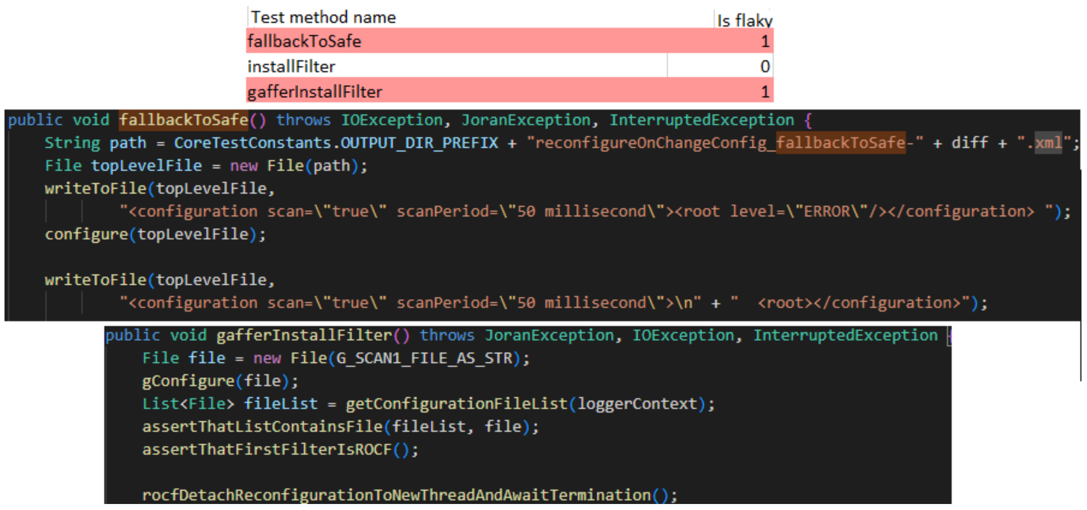
Test method name	Is flaky
testCustomEvaluator	1
testCustomBufferSize	1

```
public void testCustomEvaluator() throws Exception {
    startSMTPServer(NO_SSL);
    configure(BlackboxClassicTestConstants.JORAN_INP

    logger.debug("test");
    String msg2 = "CustomEvaluator";
    logger.debug(msg2);
    logger.debug("invisible");
    waitUntilEmailIsSent();
```

```
public void testCustomBufferSize() throws Exception
    startSMTPServer(NO_SSL);
    configure(BlackboxClassicTestConstants.JORAN_INPU

    logger.debug("invisible1");
    logger.debug("invisible2");
    String msg = "hello";
    logger.error(msg);
    waitUntilEmailIsSent();
```

Figure 91. SMTP Connections Found Example

- **File read/write**, which is again relying on external resources (*Figure 92*).



Test method name	Is flaky
fallbackToSafe	1
installFilter	0
gafferInstallFilter	1

```
public void fallbackToSafe() throws IOException, JoranException, InterruptedException {
    String path = CoreTestConstants.OUTPUT_DIR_PREFIX + "reconfigureOnChangeConfig_fallbackToSafe-" + diff + ".xml";
    File topLevelFile = new File(path);
    writeToFile(topLevelFile,
            "<configuration scan=\"true\" scanPeriod=\"50 millisecond\"><root level=\"ERROR\"/></configuration> ");
    configure(topLevelFile);

    writeToFile(topLevelFile,
            "<configuration scan=\"true\" scanPeriod=\"50 millisecond\">\n" + "  <root></configuration>");
```

```
public void gafferInstallFilter() throws JoranException, IOException, InterruptedException {
    File file = new File(G_SCAN1_FILE_AS_STR);
    gConfigure(file);
    List<File> fileList = getConfigurationFileList(loggerContext);
    assertThatListContainsFile(fileList, file);
    assertThatFirstFilterIsROCF();

    rocfDetachReconfigurationToNewThreadAndAwaitTermination();
```

Figure 92. File Read/Write Found Example

9. Conclusions

The accomplishment of this project has allowed me to fulfill successfully the objectives stablished in section 1.3 for the reasons explained as follows. The first objective consisted in studying about the definition, types and presence of test smells. This has been done by researching and Reading state of the art papers about test smells, as explained in section 3, as well as by implementing my tool and obtaining the results over a set of projects. As a result, it can be affirmed that this objective has been met. The second objective was focused on the detection of test smells. This has also been fulfilled, by researching about it and studying and executing current tools with different approaches for the automatic detection of smells in software projects.

Moreover, the third and fourth objectives focused on flaky tests. The study of papers related to flaky tests allowed me to better understand the importance of avoiding and correcting them. I learned about what problems they can cause, how they can be detected and how to properly refactor the code to correct them. In addition, some tools that could automatically detect flaky tests were also studied, and its results were examined. This allowed me to achieve the third and fourth objectives successfully.

The fifth objective is related to find if there is any correlation between test smells and flaky tests. Because flaky tests are hard to find, I wanted to know if test smells were a good indicator to find flaky tests. In the end, I conclude that test smells might not be the source of flakiness themselves, so that is why the percentages are so low. However, some of them might be related to other flakiness indicators. It is the case for example of the Sleepy test smell, exception catching throwing and conditional test logic. I manually inspected some of the tests where these smells are present, and I could find other indicators of flakiness. For example, a sleepy test smell combined with asynchronous behavior can cause flakiness and a conditional statement or a try catch clause that depends on network connection or input/output operations can also cause flakiness. Because of that, even if the result was not the desired one, I could study the relation between them and hence complete the fifth objective.

And for the sixth objective, I developed a tool that provides an easy way of inspecting test smells and flaky tests, with all the steps explained. State of the art tools requires to spend time learning how to use them and what kind of input is needed. Some of these tools provide the results in an unclean and difficult way to understand. With my tool, test smells and flakiness are brought together to the same table, for analyzing if the test smells of the project under test are in flaky methods, and graphs are also provided to get a better idea of the situation of the projects. Then, I can say that this objective has also been successfully achieved.

As a conclusion, I conducted an empirical study concerning test smells and flaky tests. I learned a lot about code smells, test smells, flaky tests and its detection and I came up with an application capable of showing results in a clear and user-friendly way. I think that this research will provide a new path for thinking about the best way of detecting flakiness, by using flakiness indicators mentioned previously or test smells in combination with other techniques.

10. Future work

The final project resulted in an application completely finished, with the basic functionality necessary for combining test smell detection results with flaky tests detection results and then studying the combination of both in a convenient and straightforward way. Starting from this point, we can divide the future work in three different sections.

First, if the application was to be used by other developers to find test smells and flaky tests, the offered features would be extended. The application would be improved so that it can detect the exact point in the methods were test smells have been found. This would be shown to the user in a new screen, maybe similar to a code editor, with remarks and recommendations on how to refactor it so that the user could see the smell and correct them when appropriate.

Secondly, a flaky tests detection tool could be developed. This tool would use the flakiness indicators mentioned in step 3.4 to look for flakiness in the tests. These indicators seem to be a better way of finding flaky tests than test smells, so a tool that used them would seem to achieve higher performance than existing approaches. Test smells could also be used to find flakiness indicators, and then find flaky tests, if considered a good idea.

Finally, more research could be done on the detection of flaky tests. It is still a considerably new field, and there is still room to improve and new approaches to be discovered. These bad programming practice, as explained before, is something to take into account and developers should avoid introducing them in their code and, if this has already happened, use an efficient way for detecting and refactoring them.

11. Acknowledgments

I would like to thank professor Takada for his help throughout the year. It has not been an easy year for me as an exchange student in a completely different country, but professor Takada has always been willing to evaluate the work I have done, give me constructive feedback all the weeks and propose possible ideas I could look into. His support during my year at Keio University was one of the main reasons this project has been completed.

In addition, I would also like to thank professor Manuela Albert. Being a thesis advisor from the other side of the planet is not an easy thing to do, but she always responded to by messages and doubts, and did not have any problems in arranging meetings with me to check my progress and help me with the project.

12. Bibliographical references

[1] Daniel Perez Morales et al., "Coverage-Guided Fairness Testing", International Conference on Intelligence Science (ICIS) 2021 <https://link.springer.com/chapter/10.1007/978-3-030-79474-3_13>

[2] Sakshi Udeshi et al., "Automated Directed Fairness Testing", International Conference on Automated Software Engineering (ASE) 2018 <https://dl.acm.org/doi/10.1145/3238147.3238165>

[3] Shinya Sano et al., "An efficient discrimination discovery method for fairness testing", International Conference on Software Engineering and Knowledge Engineering (SEKE) 2022 <https://ksiresearch.org/seke/seke22paper/paper064.pdf>

[4] Francesco Leone et al., "Overcoming Type Limitations in Semantic Clone Detection", International Workshop on Software Clones (IWSC) 2022 <https://ieeexplore.ieee.org/document/9978229>

[5] Yu-Liang Hung et al., " CPPCD: A Token-Based Approach to Detecting Potential Clones", International Workshop on Software Clones (IWSC) 2020 <https://ieeexplore.ieee.org/document/9047636>

[6] Muhammad Waseem Anwar et al., "A Systematic Review on Code Clone Detection", Emerging Technologies in Data Mining and Information Security 2020 <https://ieeexplore.ieee.org/abstract/document/8719895>

[7] José Amancio M. Santos et al., "A Systematic Review on the Code Smell Effect", The Journal of Systems and Software 2018 <https://www.sciencedirect.com/science/article/abs/pii/S0164121218301444?via%3Dihub>

[8] Martin Fowler et al., "Refactoring: Improving the Desing of Existing Code", Object Techonology Series. Addison-Wesley, 1999 <https://books.google.co.jp/books?hl=es&lr=&id=2H1_DwAAQBAJ&oi=fnd&pg=PT14&ots=NgFqvho2OV&sig=AIX0x40WQXm93L7-BMRHT0ShOng&redir_esc=y#v=onepage&q&f=false>

[9] Refactoring.com, a webpage about refactoring <https://refactoring.com/#:~:text=Refactoring%20is%20a%20disciplined%20technique,of%20small%20behavior%20preserving%20transformations.>

[10] M. M. Lehman, "Programs, life cycles, and laws of software evolution”, International in Proceedings of the IEEE, vol. 68, no. 9, 1980 <https://ieeexplore.ieee.org/document/1456074>

[11] W. Cunningham, "The WyCash portfolio management system" OOPS Messenger, vol. 4, no. 2, 1993 <https://dl.acm.org/doi/pdf/10.1145/157710.157715>

[12] Eduardo Fernandes et al., "A Review-based Comparative Study of Bad Smell Detection Tools" International Conference on Evaluation and Assessment in Software Engineering (EASE) 2016 <https://dl.acm.org/doi/10.1145/2915970.2915984>

[13] Thanis Paiva et al., "On the evaluation of code smells and detection tools" International Journal of Software Engineering Research and Development 2017 <https://jserd.springeropen.com/articles/10.1186/s40411-017-0041-1>

[14] Nikolaos Tsantalis et al., "JDeodorant: Identification and Removal of Type-checking Bad Smells" European Conference on Software Maintenance and Reengineering (CSMR) 2008 <https://ieeexplore.ieee.org/document/4493342>

[15] F. A. Fontana et al., "Code Smell Detection: Towards a Machine Learning-Based Approach" International Conference on Software Maintenance 2013 <https://ieeexplore.ieee.org/document/6676916>

[16] Santiago A. Vidal et al., "An Approach to Prioritize Code Smells for Refactoring" Automated Software Engineering 2014 <https://dl.acm.org/doi/10.1007/s10515-014-0175-x>

[17] Dario Di Nucci et al., "Detecting code smells using machine learning techniques: Are we there yet?" International Conference on Software Analysis, Evolution and Reengineering (SANER) 2018 <https://ieeexplore.ieee.org/abstract/document/8330266>

[18] Seema Dewangan et al., "A Novel Approach for Code Smell Detection: An Empirical Study" IEEE Access 2021 <https://ieeexplore.ieee.org/document/9641807>

[19] Oumayma Hamdi et al., "An Empirical Study on Code Smells Co-occurrences in Android Applications" ACM International Conference on Automated Software Engineering Workshops (ASEW) 2021 <https://ieeexplore.ieee.org/document/9680287>

[20] Dustin Lim, "Detecting Code Smells in Android Applications" Master’s Thesis 2018 <http://resolver.tudelft.nl/uuid:bab69ac3-07b7-4dae-8b80-670443af2faa>

[21] Geoffrey Hecht et al., "Tracking the Software Quality of Android Applications Along Their Evolution (T)" ACM International Conference on Automated Software Engineering (ASE) 2015 <https://ieeexplore.ieee.org/document/7372012>

[22] Fabio Palomba et al., "Lightweight detection of Android-specific code smells: The aDoctor project" International Conference on Software Analysis, Evolution and Reengineering (SANER) 2017 <https://ieeexplore.ieee.org/document/7884659>

[23] Marouane Kessentini et al., "Detecting Android Smells Using Multi-Objective Genetic Programming" International Conference on Mobile Software Engineering and Systems (MOBILESoft) 2017 <https://ieeexplore.ieee.org/document/7972726>

[24] Jehan Rubin et al., "Sniffing Android code smells: an association rules mining-based approach" International Conference on Mobile Software Engineering and Systems (MOBILESoft) 2019 <https://dl.acm.org/doi/10.5555/3340730.3340753>

[25] Chenguang Mao et al., "Droidlens: Robust and Fine-Grained Detection for Android Code Smells" International Symposium on Theoretical Aspects of Software Engineering (TASE) 2020 <https://ieeexplore.ieee.org/document/9405264>

[26] Jing Yu et al., "A Novel Tree-based Neural Network for Android Code Smells Detection" International Conference on Software Quality, Reliability and Security (QRS) 2021 <https://ieeexplore.ieee.org/document/9724832>

[27] Bo Yang et al., "Don’t Do That! Hunting Down Visual Design Smells in Complex UIs Against Design Guidelines" International Conference on Software Engineering (ICSE) 2021 <https://ieeexplore.ieee.org/document/9402139>

[28] Kent Beck, "Test Driven Development: By Example" Addison-Wesley Longman Publishing Co. 2002 <https://github.com/clarabez/SoftwareTestingBooks/blob/master/Test-Driven%20Development%20By%20Example%20(Kent%20Beck).pdf>

[29] A. Schneider et al., "JUnit best practices" Java World 2000 <https://www.infoworld.com/article/2076265/junit-best-practices.html>

[30] Abdallah Qusef et al., "SCOTCH: Test-to-code traceability using slicing and conceptual coupling" International Conference on Software Maintenance (ICSM) 2011 <https://ieeexplore.ieee.org/document/6080773>

[31] Arie van Deursen et al., "Refactoring test code" Technical Report CWI Centre for Mathematics and Computer Science 2001 <https://dl.acm.org/doi/10.5555/869201>

[32] Gerard Meszaros, "xUnit Test Patterns: Refactoring Test Code" Addison-Wesley 2007 <https://github.com/ahmedfarhat/software-development-ebooks-1/blob/master/%5BxUnit%20Test%20Patterns%20Refactoring%20Test%20Code%20(Addison-

Wesley%20Signature%20Series%20(Fowler))%20Kindle%20Edition%20by%20Gerard%20Meszaros%20-%202007%5D.pdf>

[33] Helmut Neukirchen et al., "Utilising code smells to detect quality problems in ttcn-3 test suites" International Conference on Testing of Communicating Systems and International Workshop on Formal Approaches to Testing of Software (TestCom/FATES) 2007 <https://link.springer.com/chapter/10.1007/978-3-540-73066-8_16>

[34] Bart Van Rompaey et al., "Characterizing the relative significance of a test smell" International Conference on Software Maintenance (ICSM) 2006 <https://ieeexplore.ieee.org/document/4021366>

[35] Michaela Greiler et al., "Automated Detection of Test Fixture Strategies and Smells" International Conference on Software Testing, Verification and Validation 2013 <https://ieeexplore.ieee.org/document/6569744>

[36] Davide Spadini et al., "On the Relation of Test Smells to Software Code Quality" International Conference on Software Maintenance and Evolution (ICSME) 2018 <https://ieeexplore.ieee.org/document/8529832>

[37] Michele Tufano et al., "An empirical investigation into the nature of test smells" International Conference on Automated Software Engineering (ASE) 2016 <https://dl.acm.org/doi/abs/10.1145/2970276.2970340>

[38] Manuel Breugelmans et al., "TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites" 2008 <https://api.semanticscholar.org/CorpusID:9839569>

[39] Wajdi Aljedaani et al., "Test Smell Detection Tools: A Systematic Mapping Study" International Conference on Evaluation and Assessment in Software Engineering (EASE) 2021 <https://dl.acm.org/doi/10.1145/3463274.3463335>

[40] Anthony Peruma et al., "TsDetect: an open source test smells detection tool" ACM Joint Meeting on ESEC/FSE 2020 < https://dl.acm.org/doi/10.1145/3368089.3417921 >

[41] Stefano Lambiase et al., "Just-In-Time Test Smell Detection and Refactoring: The DARTS Project" International Conference on Program Comprehension 2020 <https://dl.acm.org/doi/10.1145/3387904.3389296 >

[42] Tássio Virgínio et al., "On the influence of Test Smells on Test Coverage" Brazilian Symposium on Software Engineering (SBES) 2019 <https://dl.acm.org/doi/10.1145/3350768.3350775>

[43] Tássio Virgínio et al., "JNose: Java Test Smell Detector" Brazilian Symposium on Software Engineering (SBES) 2020 < https://dl.acm.org/doi/10.1145/3422392.3422499>

[44] Railana Santana et al., "RAIDE: A Tool for Assertion Roulette and Duplicate Assert Identification and Refactoring" Brazilian Symposium on Software Engineering 2020 <https://dl.acm.org/doi/abs/10.1145/3422392.3422510>

[45] Manuel Breugelmans et al., "TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites" International Workshop on Advanced Software Development Tools and Techniques (WASDeTT) 2008 <https://api.semanticscholar.org/CorpusID:9839569>

[46] Github repository to download the test smell detection tool TsDetect <https://github.com/TestSmells/TestSmellDetector>

[47] Github repository to download the test smell detection tool DARTS <https://github.com/StefanoLambiase/DARTS>

[48] Github repository to download the test smell detection tool TestHound <https://github.com/SERG-Delft/TestHound>

[49] Github repository to download the test smell detection tool JNose <https://github.com/arieslab/jnose>

[50] Github repository to download the test smell detection tool RAIDE <https://github.com/arieslab/raide>

[51] Webpage link to download the test smell detection tool TestQ <https://code.google.com/archive/p/tsmells/downloads>

[52] Moritz Eck et al., "Understanding flaky tests: the developer's perspective" ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) 2019 <https://dl.acm.org/doi/10.1145/3338906.3338945>

[53] Qingzhou Luo et al., "An empirical analysis of flaky tests" ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE) 2014 <https://dl.acm.org/doi/10.1145/2635868.2635920>

[54] Jonathan Bell et al., "DeFlaker: Automatically Detecting Flaky Tests" International Conference on Software Engineering (ICSE) 2018 <https://ieeexplore.ieee.org/document/8453104>

[55] Github repository to download the flaky tests detection tool DeFlaker <https://github.com/gmu-swe/deflaker>

[56] Abdulrahman Alshammari et al., "FlakeFlagger: Predicting Flakiness Without Rerunning Tests" International Conference on Software Engineering (ICSE) 2021 <https://ieeexplore.ieee.org/document/9402098>

[57] Github repository to download the flaky tests detection tool FlakeFlagger <https://github.com/AlshammariA/FlakeFlagger>

[58] Wing Lam et al., "iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests" Conference on Software Testing, Validation and Verification (ICST) 2019 <https://ieeexplore.ieee.org/document/8730188>

[59] Github repository to download the flaky tests detection tool iDFlakies <https://github.com/iDFlakies/iDFlakies>

[60] Github repository to download the tool of this project, Smell Flaky Analyzer <https://github.com/TheAlexet/smell-flaky-analyzer>

[61] Git webpage <https://git-scm.com>

[62] React webpage <https://react.dev>

[63] Javascript <https://developer.mozilla.org/en-US/docs/Web/JavaScript>

[64] HTML <https://developer.mozilla.org/en-US/docs/Web/HTML>

[65] CSS <https://developer.mozilla.org/en-US/docs/Web/CSS>

[66] Electron webpage <https://www.electronjs.org>

[67] Python webpage <https://www.python.org>

[68] Visual Studio Code webpage <https://code.visualstudio.com>

[69] Photoshop webpage <https://www.adobe.com/es/products/photoshop.html>

13. Appendices

13.1 List of abbreviations and acronyms

AI: Artificial Intelligence

UPV: Universitat Politècnica de València

PHD: Philosophy Doctorate

AST: Abstract Detection Trees

PDG: Program Dependance Graph

UI: User Interface

CSV: Comma-Separated Values

URL: Uniform Resource Locator

I/O: Input and Output

NaN: Not a Number

OS: Operating System

SHA: Secure Hashing Algorithm

HTML: Hypertext Markup Language

CSS: Cascading Style Sheets

REST: Representational State Transfer

SMTP: Simple Mail Transfer Protocol

13.2 Slides used for the final presentation

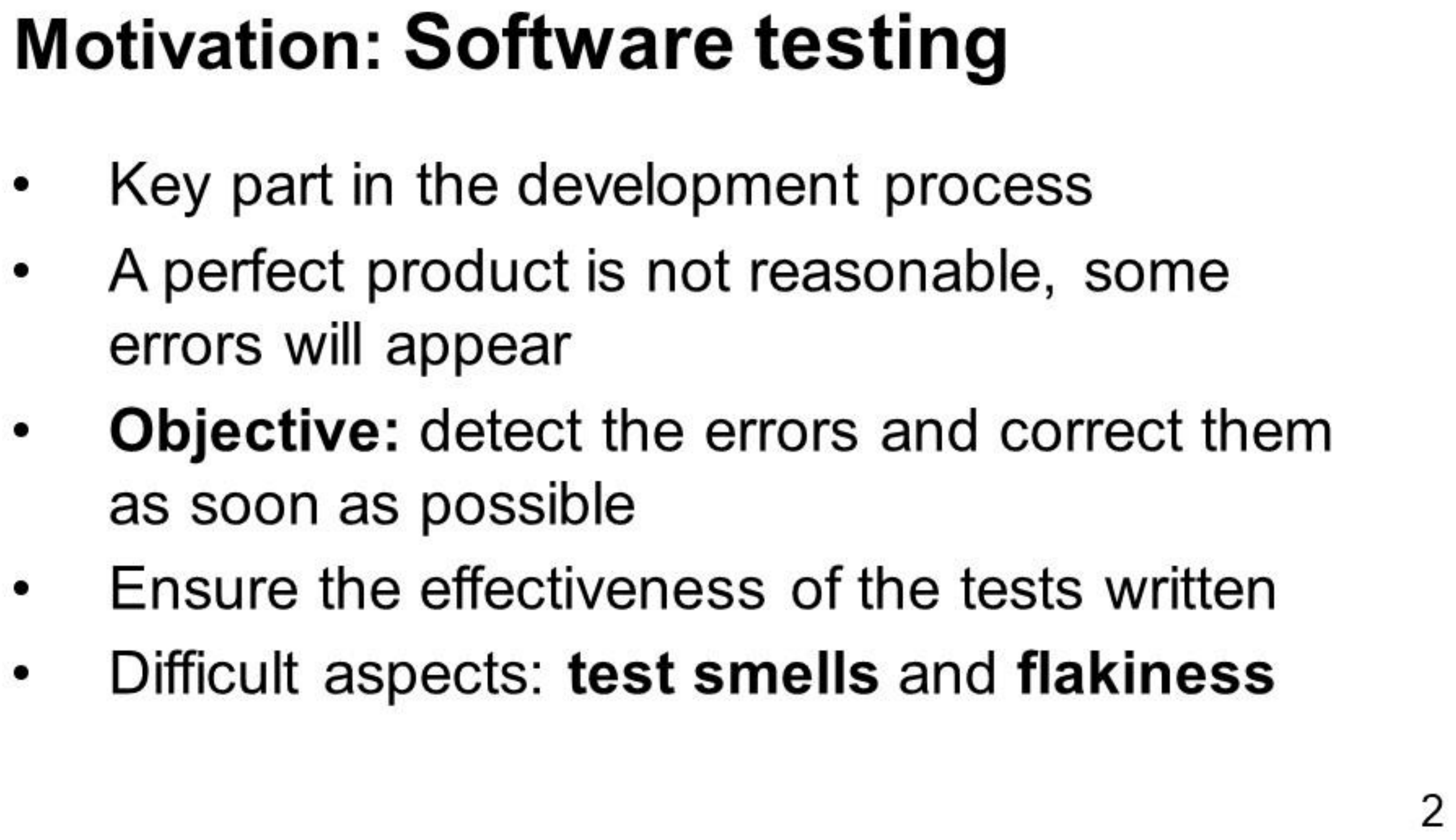
Research on the correlation of test smells and flaky tests

Alejandro Vicent Micó
ID: 92255237

Independent Study for Exchange Student

1

Figure 93. Slide 1: Title

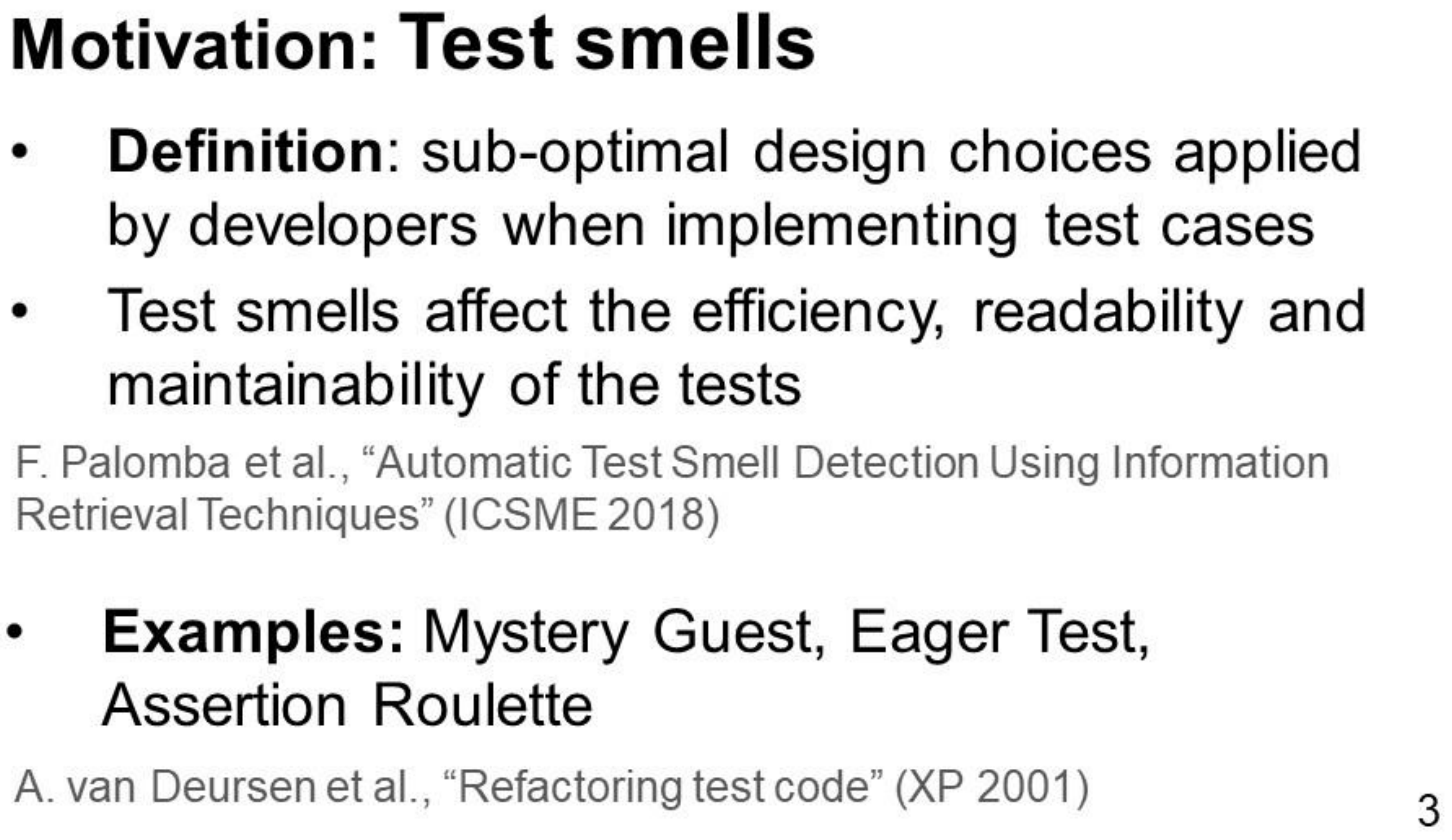


Motivation: Software testing

- Key part in the development process
- A perfect product is not reasonable, some errors will appear
- **Objective:** detect the errors and correct them as soon as possible
- Ensure the effectiveness of the tests written
- Difficult aspects: **test smells** and **flakiness**

2

Figure 94. Slide 2: Motivation Software Testing



Motivation: Test smells

- **Definition**: sub-optimal design choices applied by developers when implementing test cases
- Test smells affect the efficiency, readability and maintainability of the tests

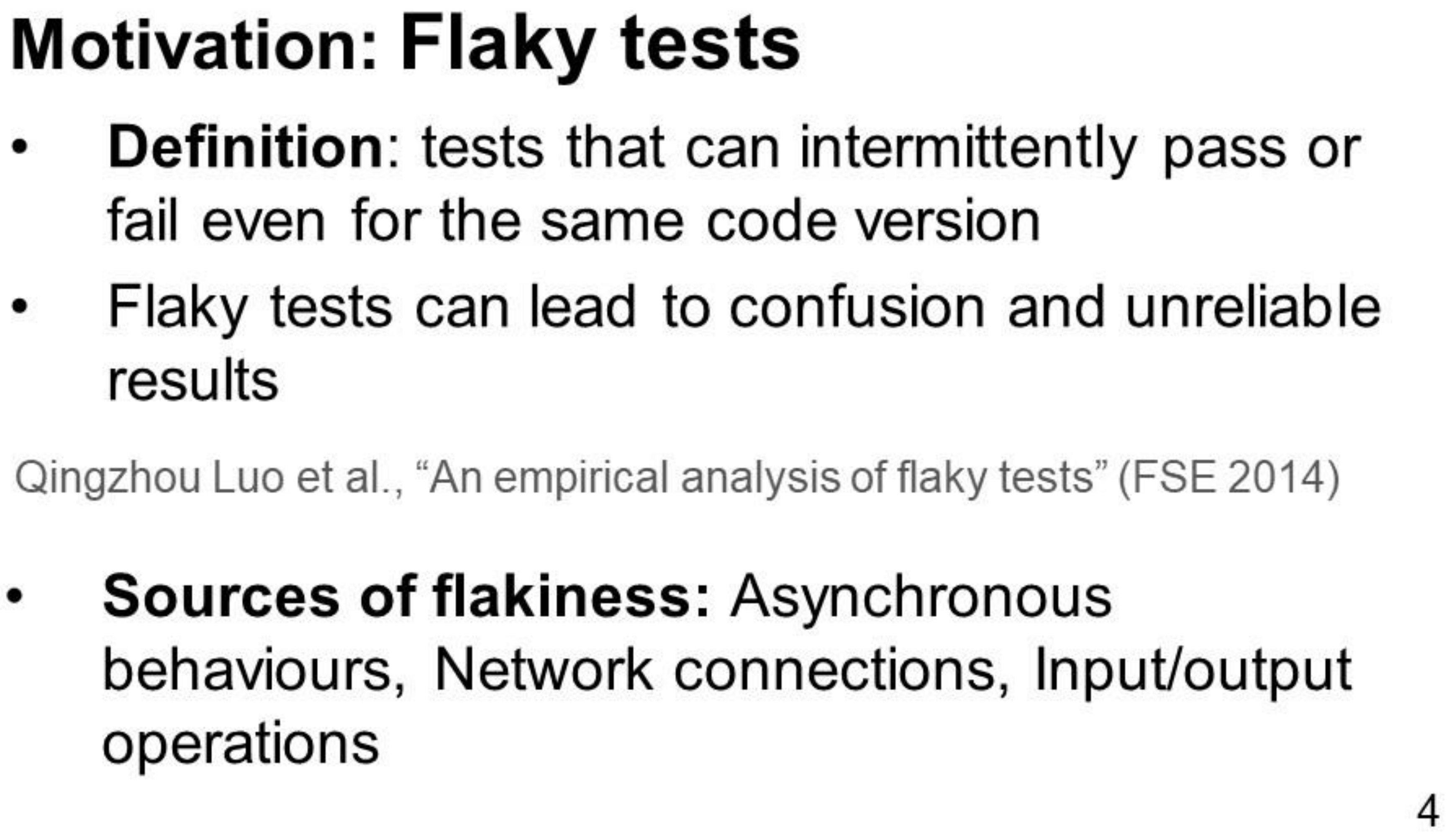
F. Palomba et al., “Automatic Test Smell Detection Using Information Retrieval Techniques” (ICSME 2018)

- **Examples:** Mystery Guest, Eager Test, Assertion Roulette

A. van Deursen et al., “Refactoring test code” (XP 2001)

3

Figure 95. Slide 3: Motivation Test Smells



Motivation: Flaky tests

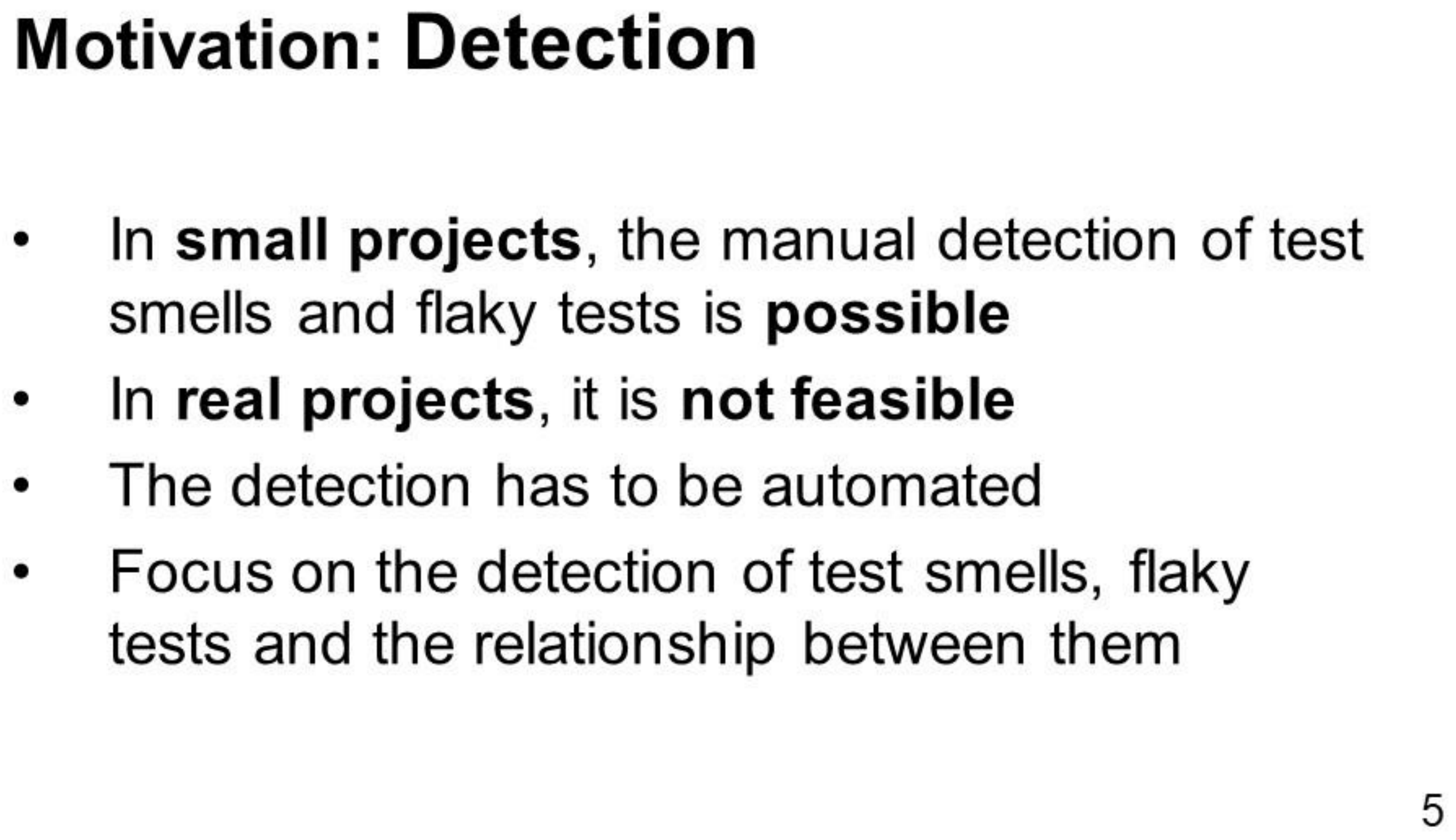
- **Definition**: tests that can intermittently pass or fail even for the same code version
- Flaky tests can lead to confusion and unreliable results

Qingzhou Luo et al., "An empirical analysis of flaky tests" (FSE 2014)

- **Sources of flakiness:** Asynchronous behaviours, Network connections, Input/output operations

4

Figure 96. Slide 4: Motivation Flaky Tests

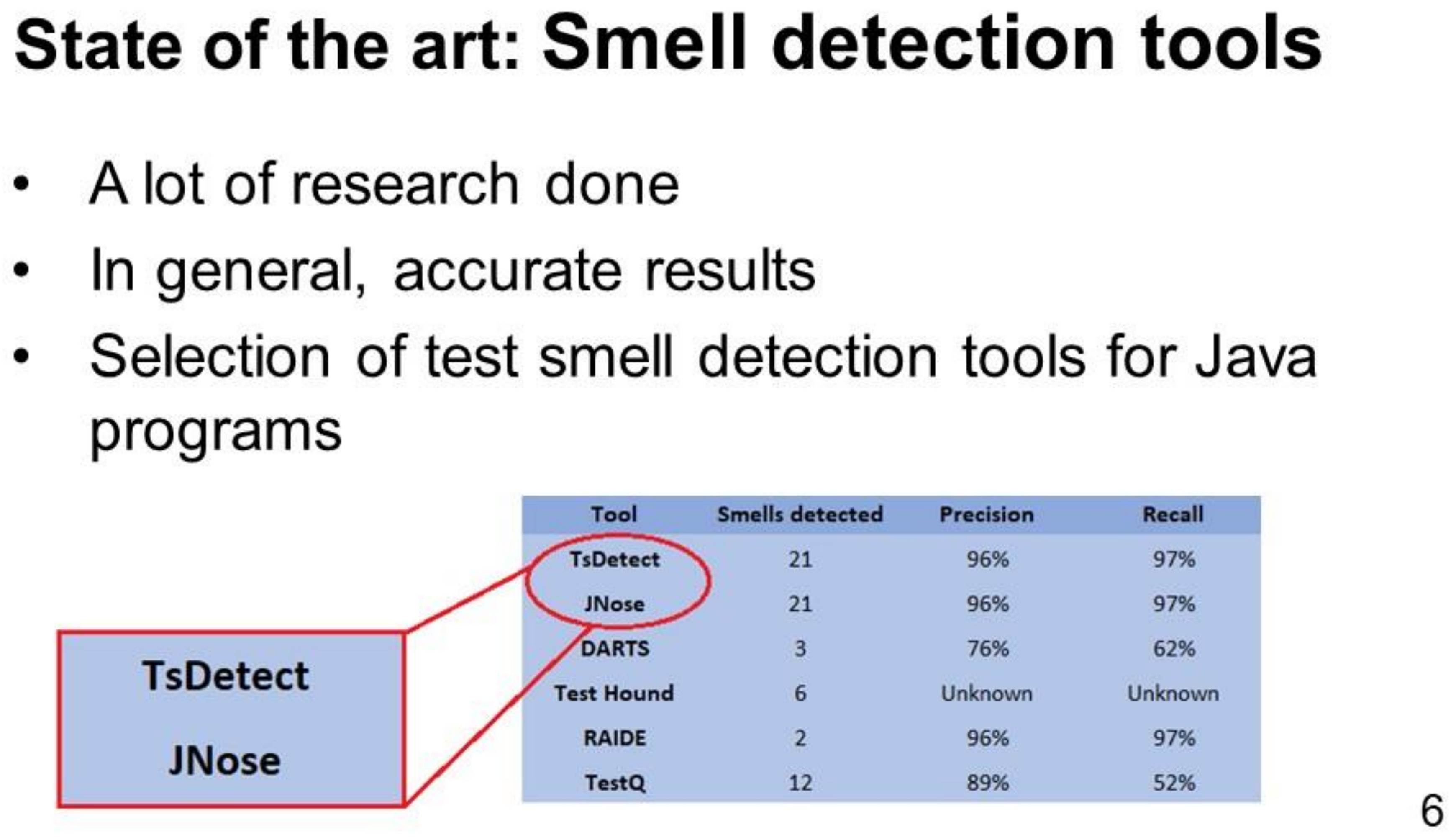


Motivation: Detection

- In **small projects**, the manual detection of test smells and flaky tests is **possible**
- In **real projects**, it is **not feasible**
- The detection has to be automated
- Focus on the detection of test smells, flaky tests and the relationship between them

5

Figure 97. Slide 5: Motivation Detection



State of the art: Smell detection tools

- A lot of research done
- In general, accurate results
- Selection of test smell detection tools for Java programs

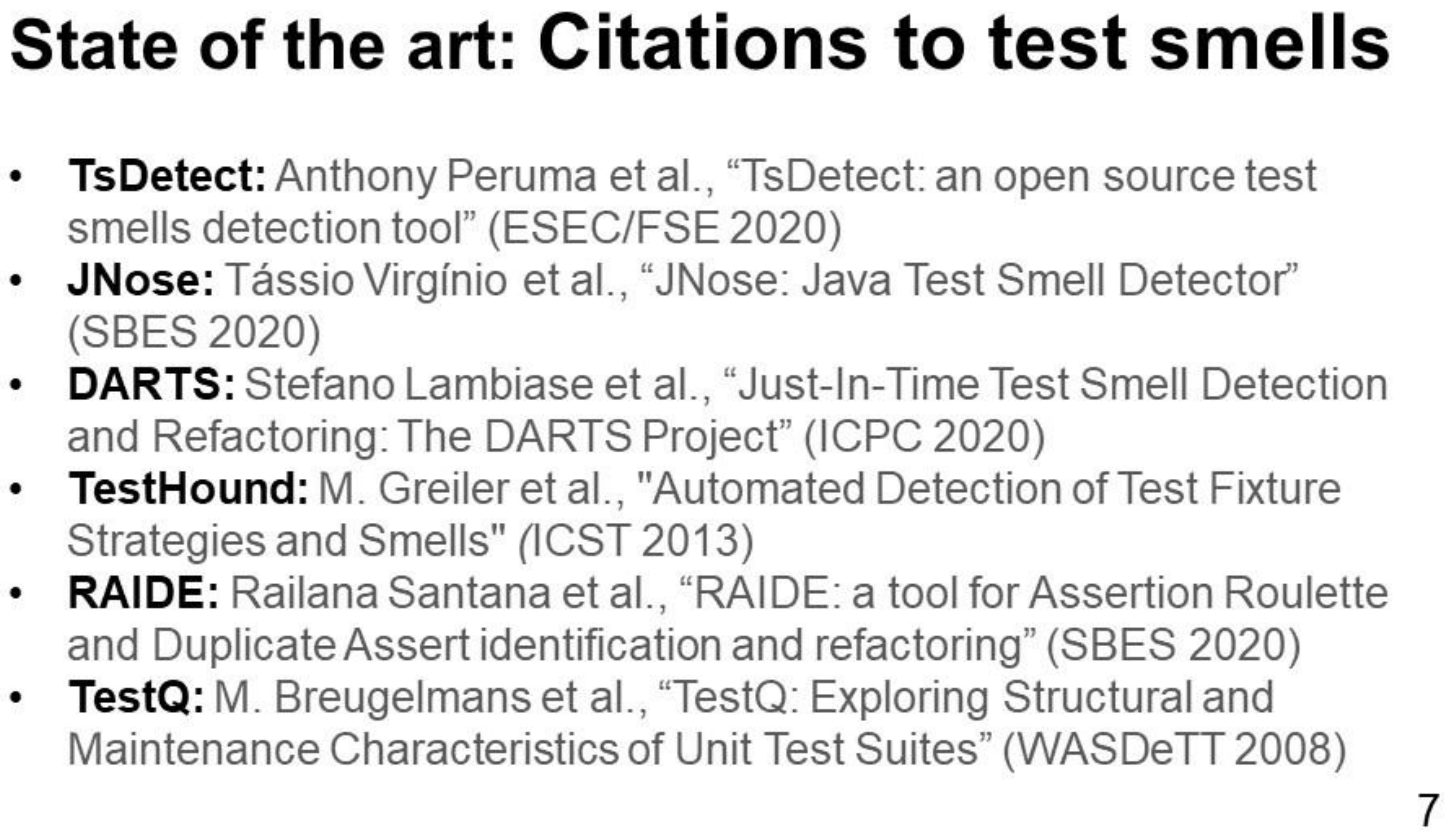
Tool	Smells detected	Precision	Recall
TsDetect	21	96%	97%
JNose	21	96%	97%
DARTS	3	76%	62%
Test Hound	6	Unknown	Unknown
RAIDE	2	96%	97%
TestQ	12	89%	52%

TsDetect

JNose

6

Figure 98. Slide 6: State of the Art Smell Selection Tools

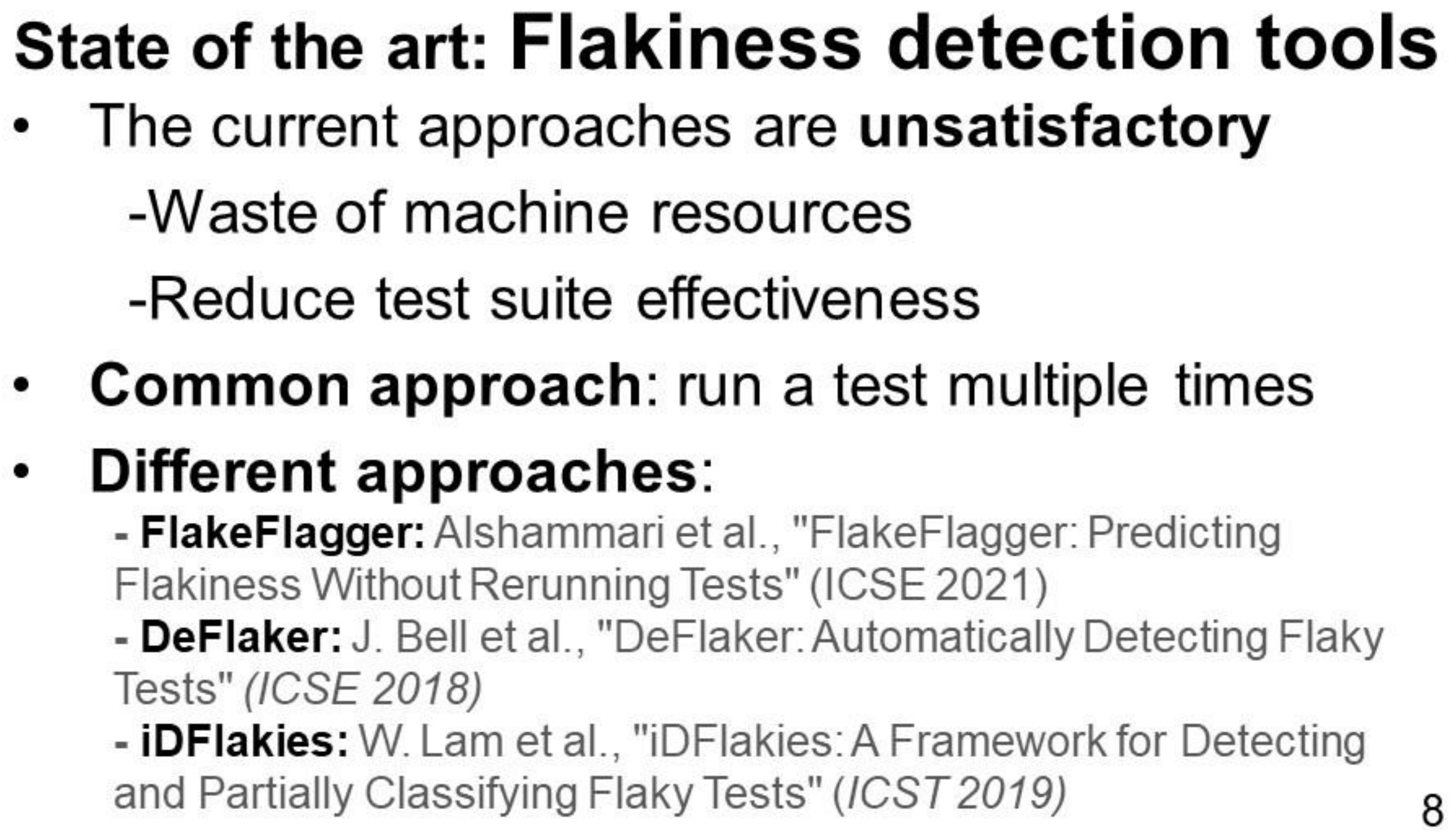


State of the art: Citations to test smells

- **TsDetect:** Anthony Peruma et al., "TsDetect: an open source test smells detection tool" (ESEC/FSE 2020)
- **JNose:** Tássio Virgínio et al., "JNose: Java Test Smell Detector" (SBES 2020)
- **DARTS:** Stefano Lambiase et al., "Just-In-Time Test Smell Detection and Refactoring: The DARTS Project" (ICPC 2020)
- **TestHound:** M. Greiler et al., "Automated Detection of Test Fixture Strategies and Smells" *(*ICST 2013)
- **RAIDE:** Railana Santana et al., "RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring" (SBES 2020)
- **TestQ:** M. Breugelmans et al., "TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites" (WASDeTT 2008)

7

Figure 99. Slide 7: State of the Art Citations to Test Smells

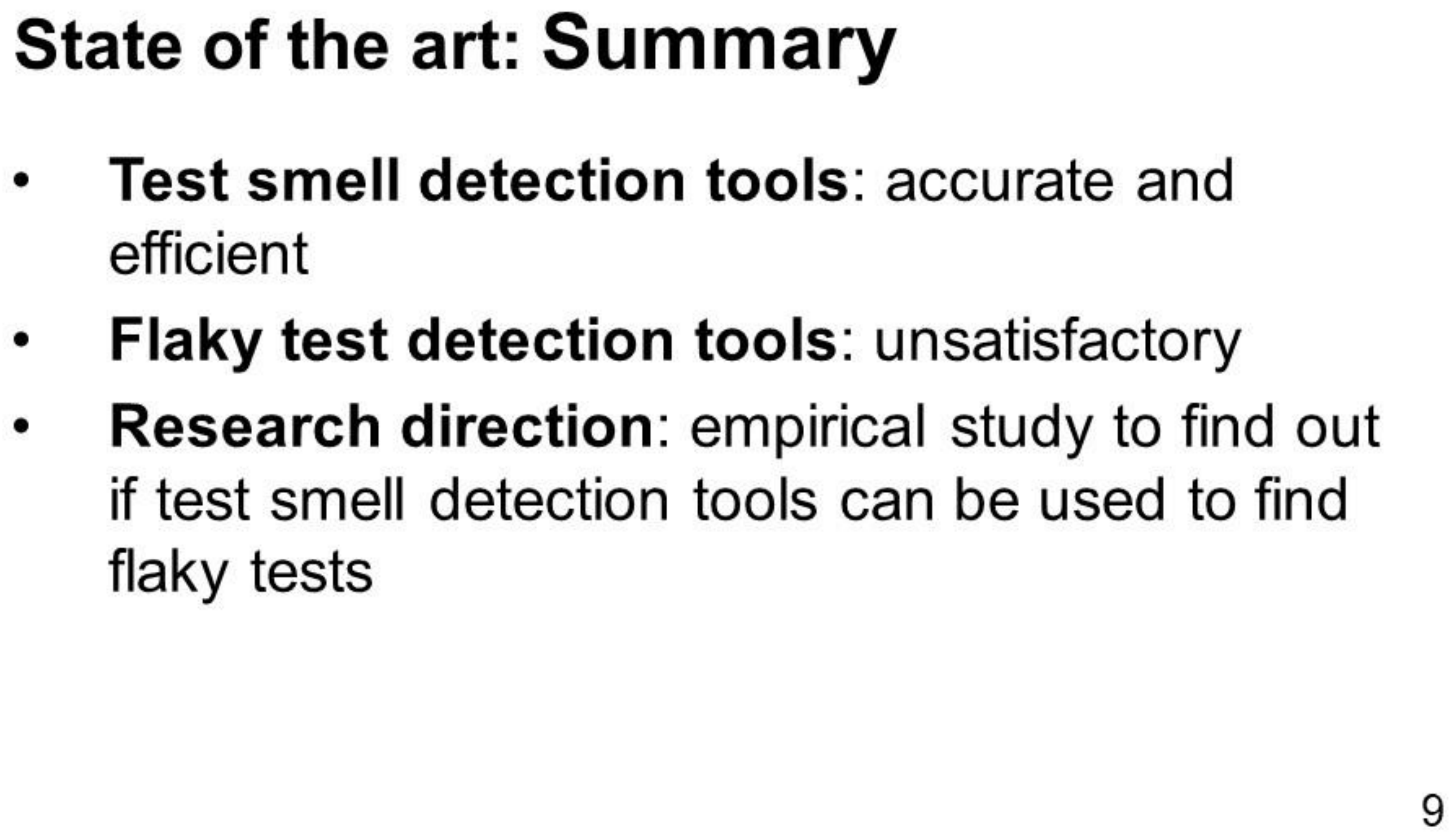


State of the art: Flakiness detection tools

- The current approaches are **unsatisfactory**
  - -Waste of machine resources
  - -Reduce test suite effectiveness
- **Common approach**: run a test multiple times
- **Different approaches**:
  - **- FlakeFlagger:** Alshammari et al., "FlakeFlagger: Predicting Flakiness Without Rerunning Tests" (ICSE 2021)
  - **- DeFlaker:** J. Bell et al., "DeFlaker: Automatically Detecting Flaky Tests" *(ICSE 2018)*
  - **- iDFlakies:** W. Lam et al., "iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests" (*ICST 2019)*

8

Figure 100. Slide 8: State of the Art Flakiness Detection Tools

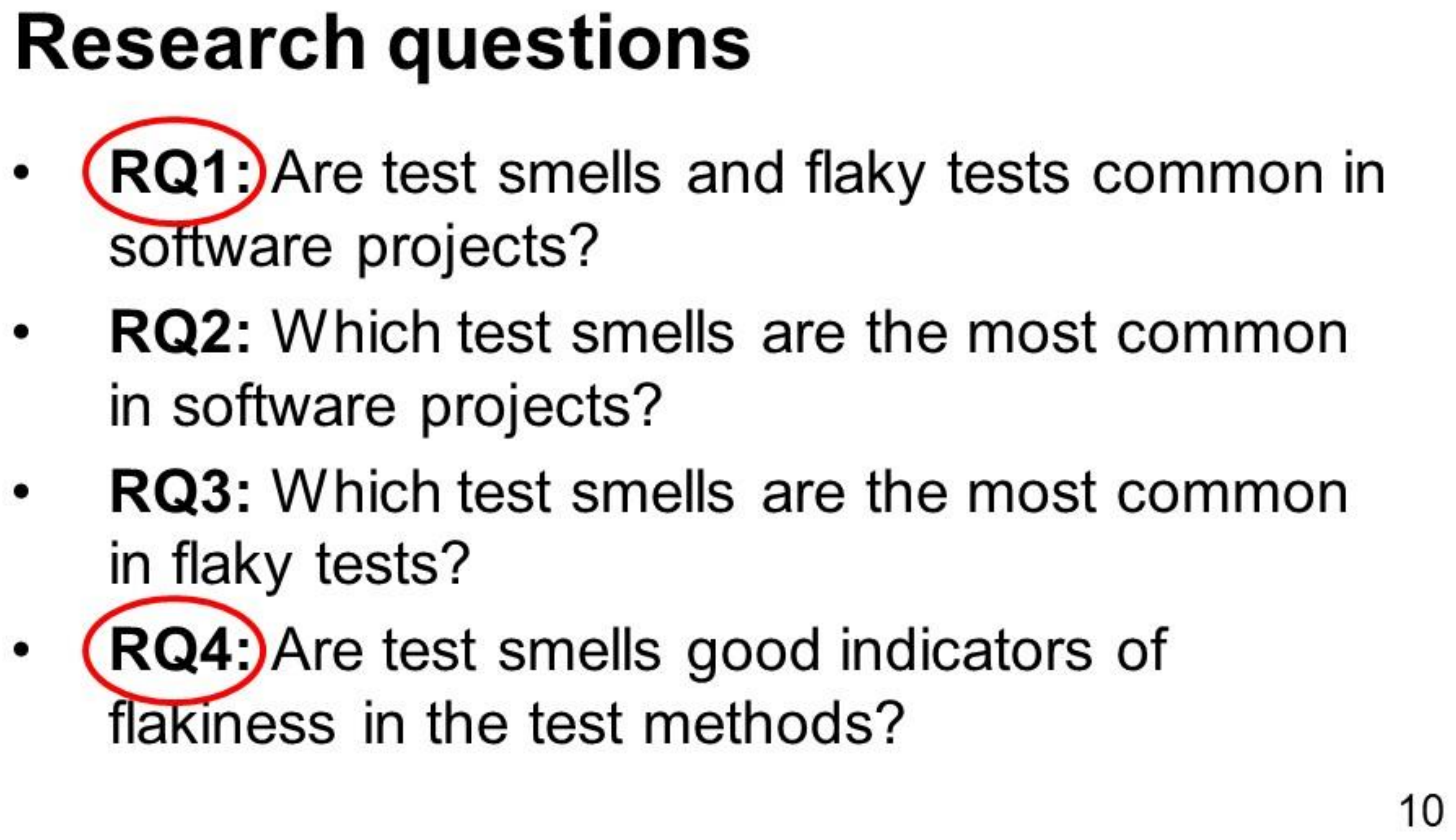


State of the art: Summary

- **Test smell detection tools**: accurate and efficient
- **Flaky test detection tools**: unsatisfactory
- **Research direction**: empirical study to find out if test smell detection tools can be used to find flaky tests

9

Figure 101. Slide 9: State of the Art Summary

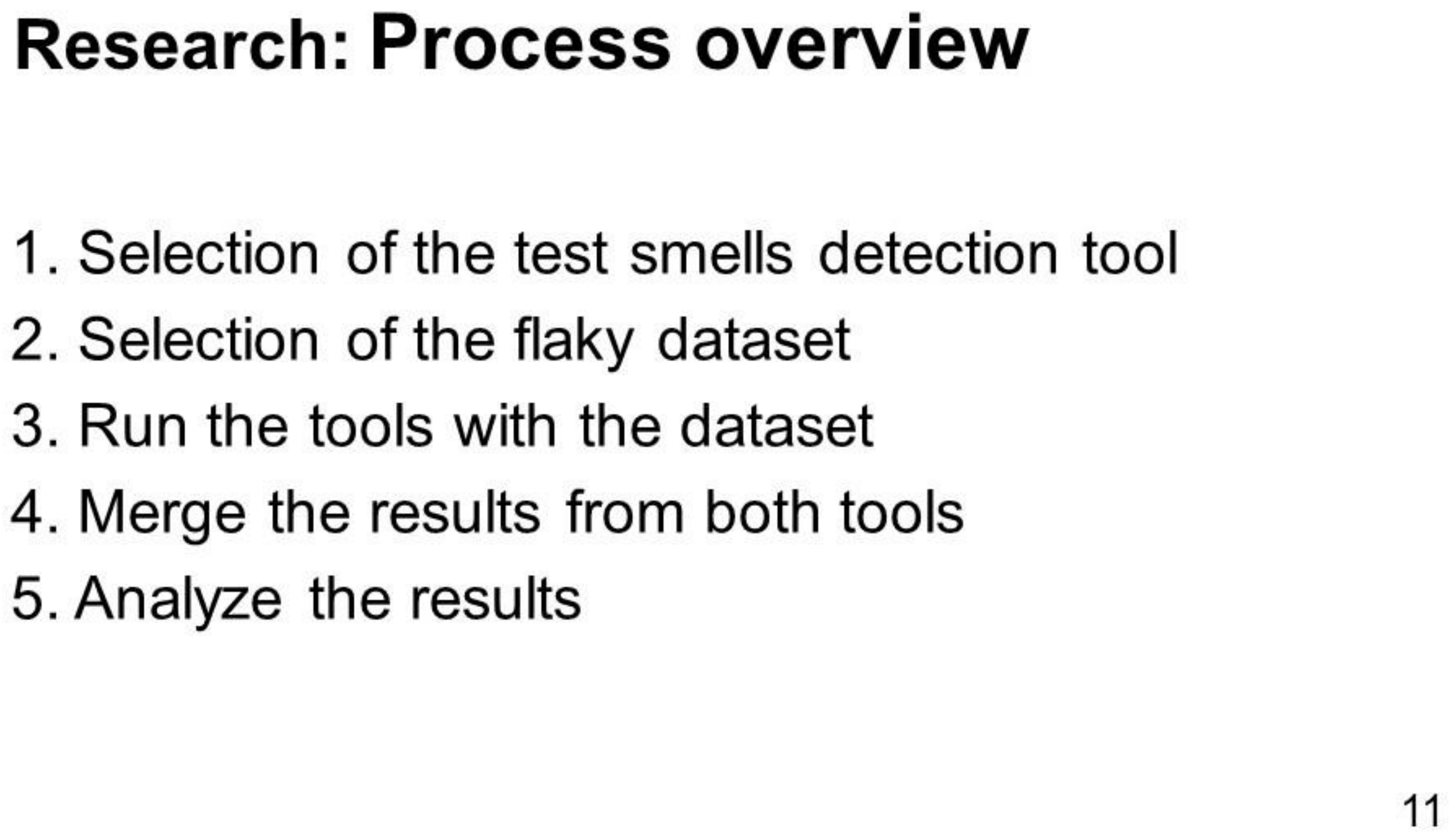


Research questions

- **RQ1:** Are test smells and flaky tests common in software projects?
- **RQ2:** Which test smells are the most common in software projects?
- **RQ3:** Which test smells are the most common in flaky tests?
- **RQ4:** Are test smells good indicators of flakiness in the test methods?

10

Figure 102. Slide 10: Research Questions

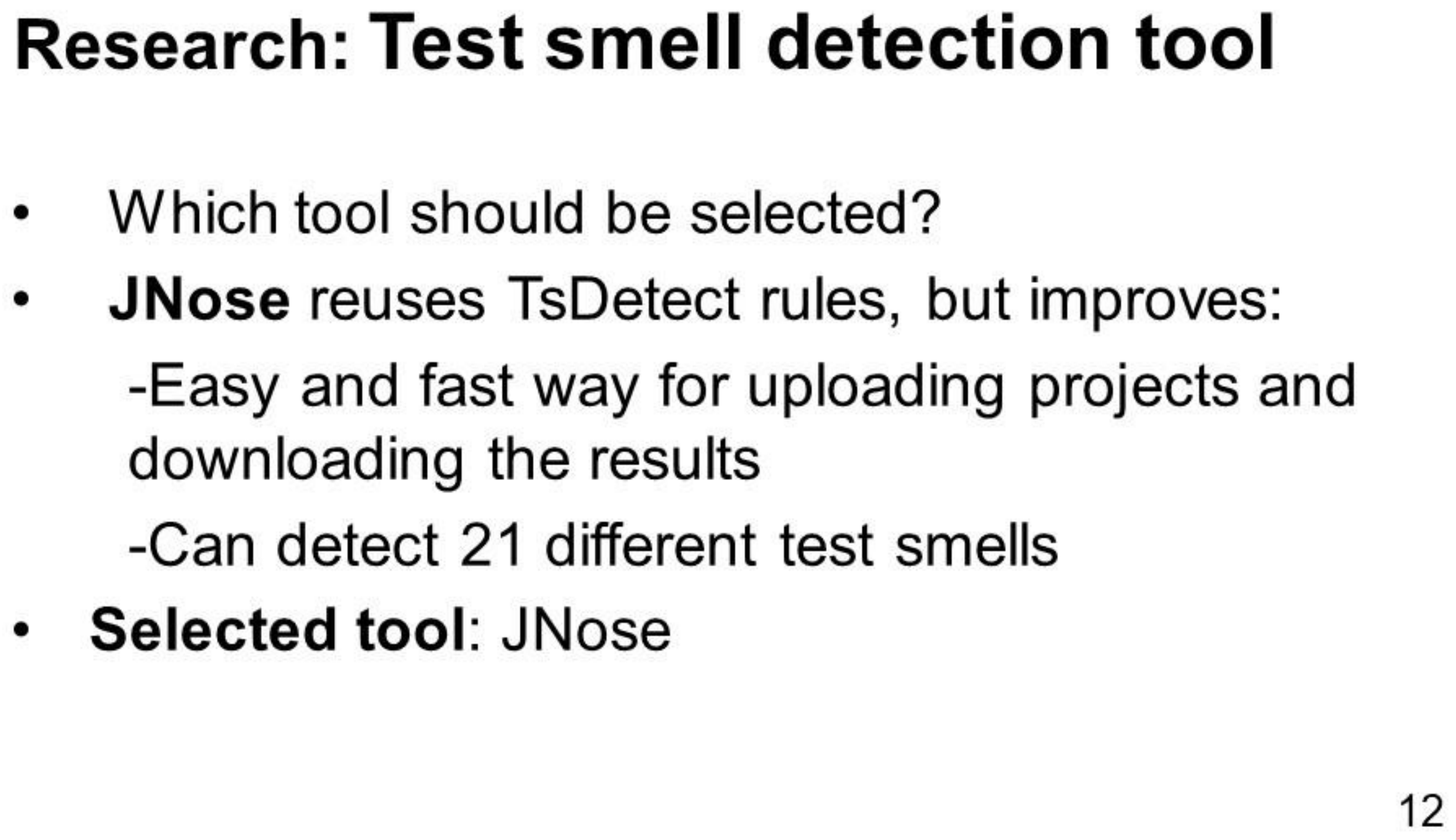


Research: Process overview

1. Selection of the test smells detection tool
2. Selection of the flaky dataset
3. Run the tools with the dataset
4. Merge the results from both tools
5. Analyze the results

11

Figure 103. Slide 11: Research Process Overview



Research: Test smell detection tool

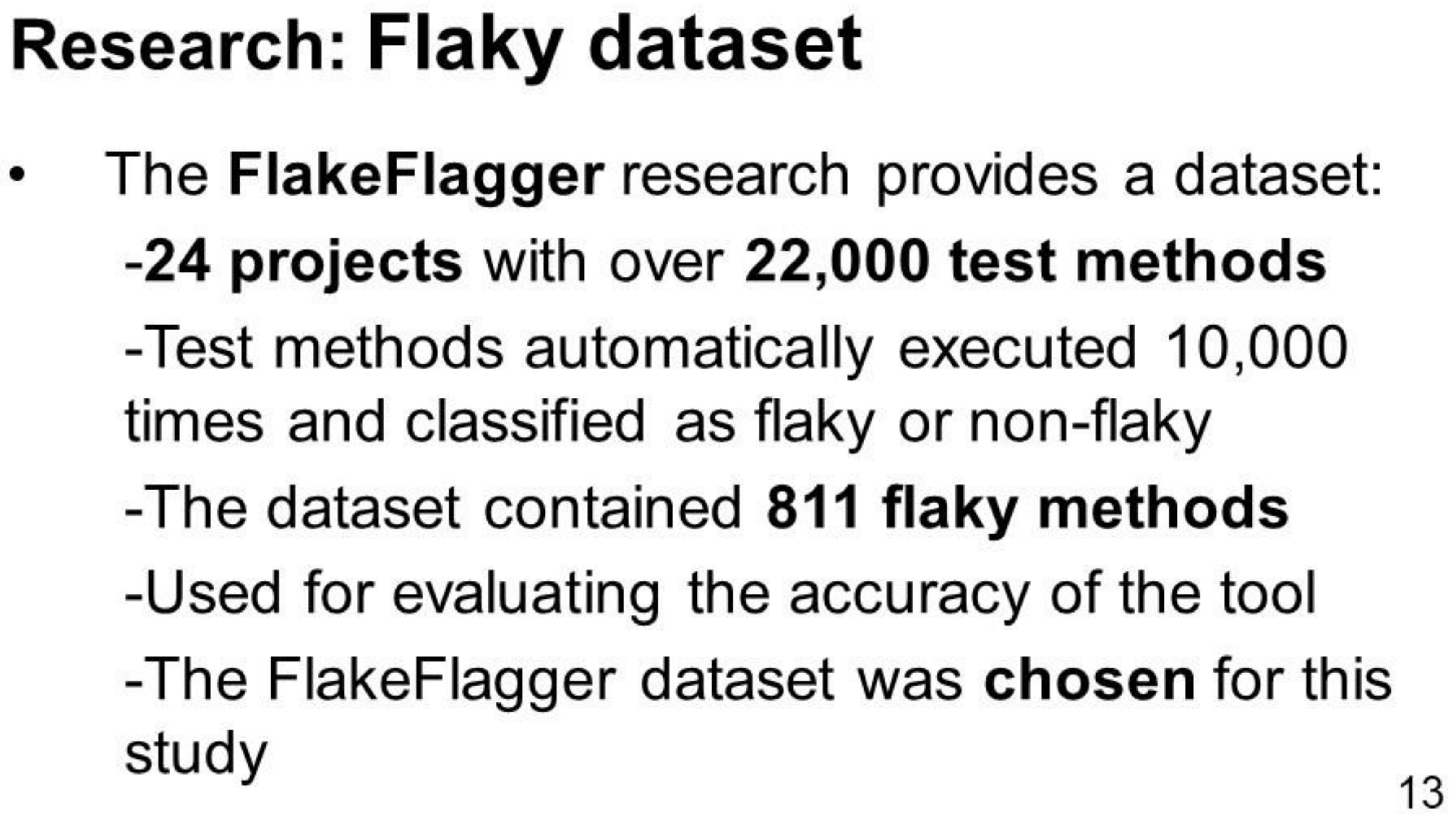
- Which tool should be selected?
- **JNose** reuses TsDetect rules, but improves:

  -Easy and fast way for uploading projects and downloading the results

  -Can detect 21 different test smells
- **Selected tool**: JNose

12

Figure 104. Slide 12: Research Test Smell Detection Tool



Research: Flaky dataset

- The **FlakeFlagger** research provides a dataset:

  -**24 projects** with over **22,000 test methods**

  -Test methods automatically executed 10,000 times and classified as flaky or non-flaky

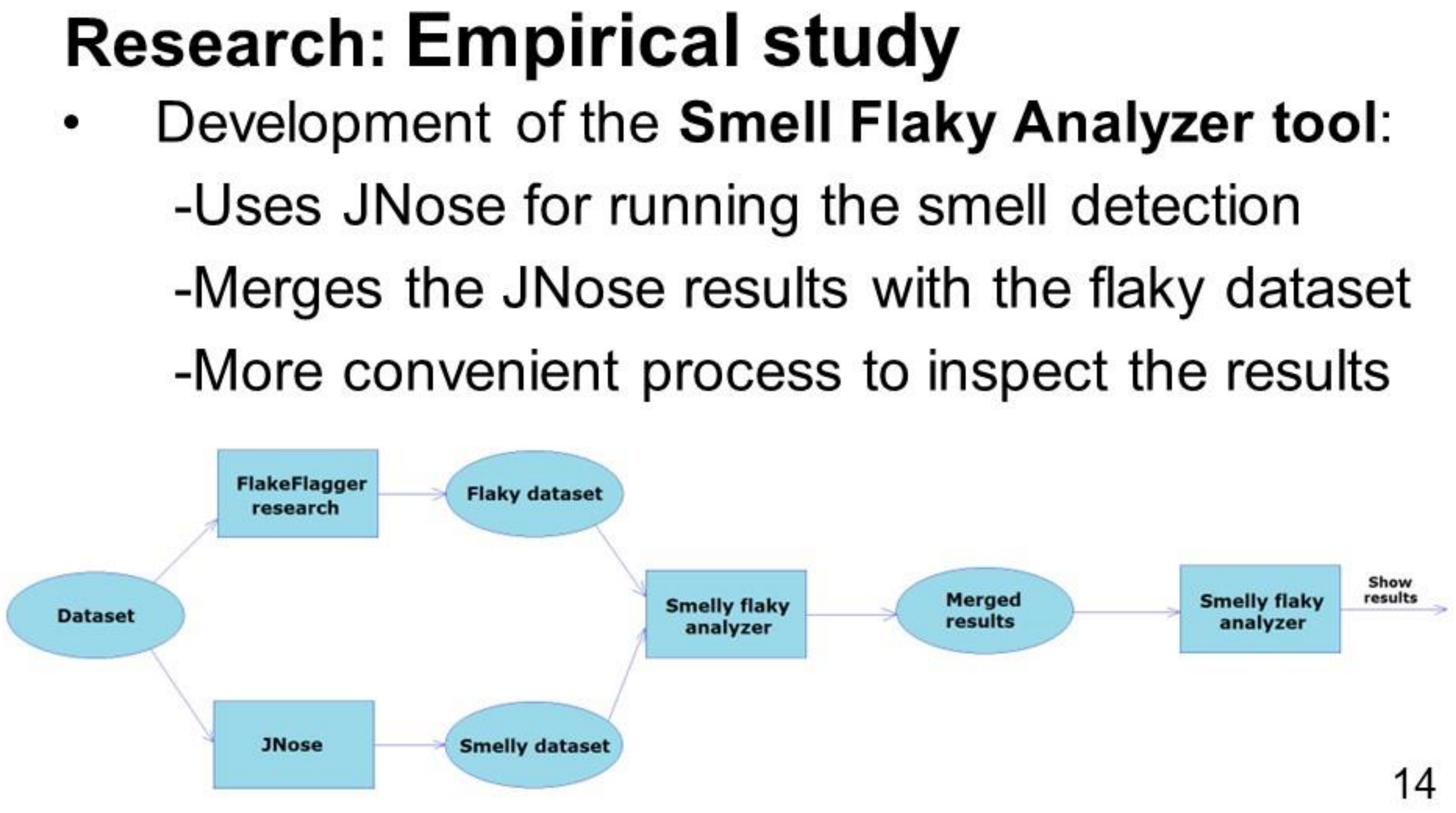
  -The dataset contained **811 flaky methods**

  -Used for evaluating the accuracy of the tool

  -The FlakeFlagger dataset was **chosen** for this study

13

Figure 105. Slide 13: Research Flaky Dataset



Research: Empirical study

- Development of the **Smell Flaky Analyzer tool**:

  -Uses JNose for running the smell detection

  -Merges the JNose results with the flaky dataset

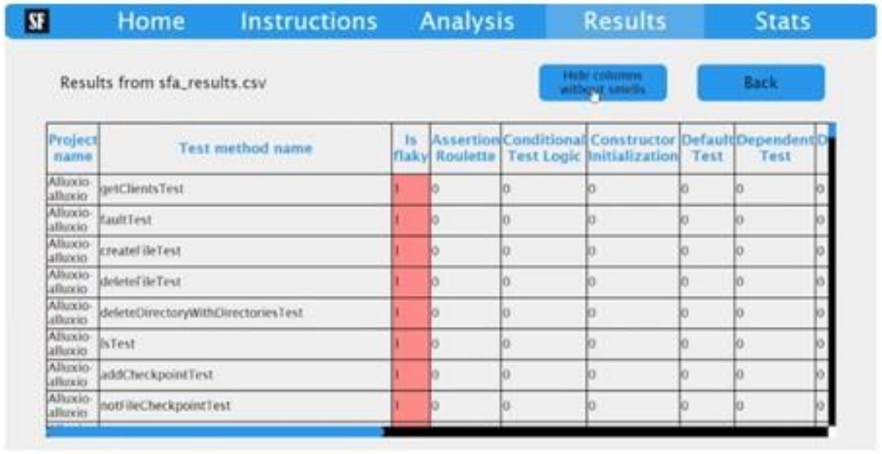
  -More convenient process to inspect the results

Dataset
FlakeFlagger research
JNose
Flaky dataset
Smelly dataset
Smelly flaky analyzer
Merged results
Smelly flaky analyzer
Show results
14

Figure 106. Slide 14: Research Empirical Study

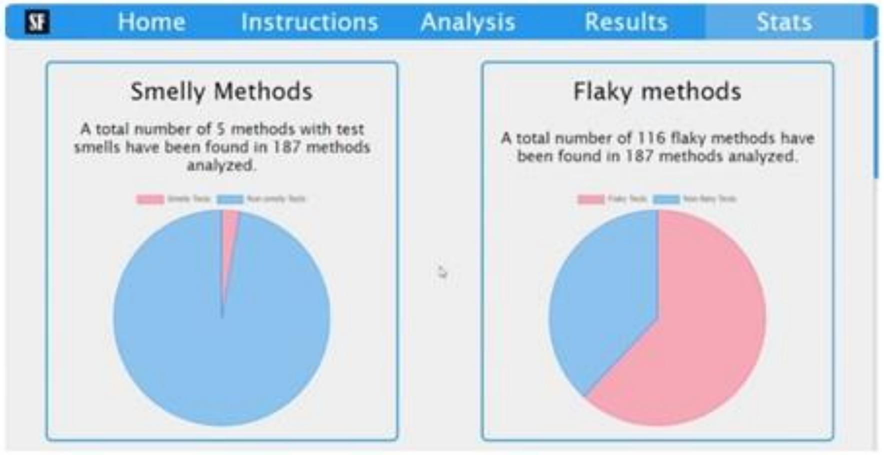
Research: Analysis of results

- CSV file uploaded to the tool
- The results can be seen in a table and some graphs provide additional information



Results from sfa_results.csv

Project name	Test method name	Is flaky	Assertion Roulette	Conditional Test Logic	Constructor Initialization	Default Test	Dependent Test	D
Alluxio-alluxio	getClientsTest	1	0	0	0	0	0	0
Alluxio-alluxio	faultTest	1	0	0	0	0	0	0
Alluxio-alluxio	createFileTest	1	0	0	0	0	0	0
Alluxio-alluxio	deleteFileTest	1	0	0	0	0	0	0
Alluxio-alluxio	deleteDirectoryWithDirectoriesTest	1	0	0	0	0	0	0
Alluxio-alluxio	isTest	1	0	0	0	0	0	0
Alluxio-alluxio	addCheckpointTest	1	0	0	0	0	0	0
Alluxio-alluxio	notFileCheckpointTest	1	0	0	0	0	0	0



A total number of 5 methods with test smells have been found in 187 methods analyzed.

A total number of 116 flaky methods have been found in 187 methods analyzed.

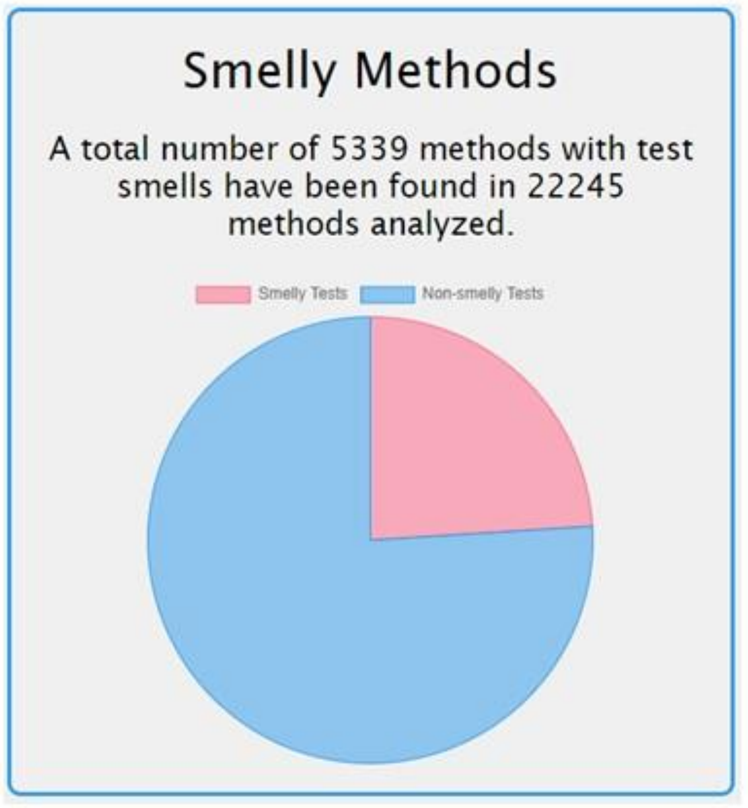
15

Figure 107. Slide 15: Research Analysis of Results

Analysis of results

- **RQ1: Are test smells and flaky tests common in software projects?**

  For test smells yes, 24.00% of the test methods had at least one test smell

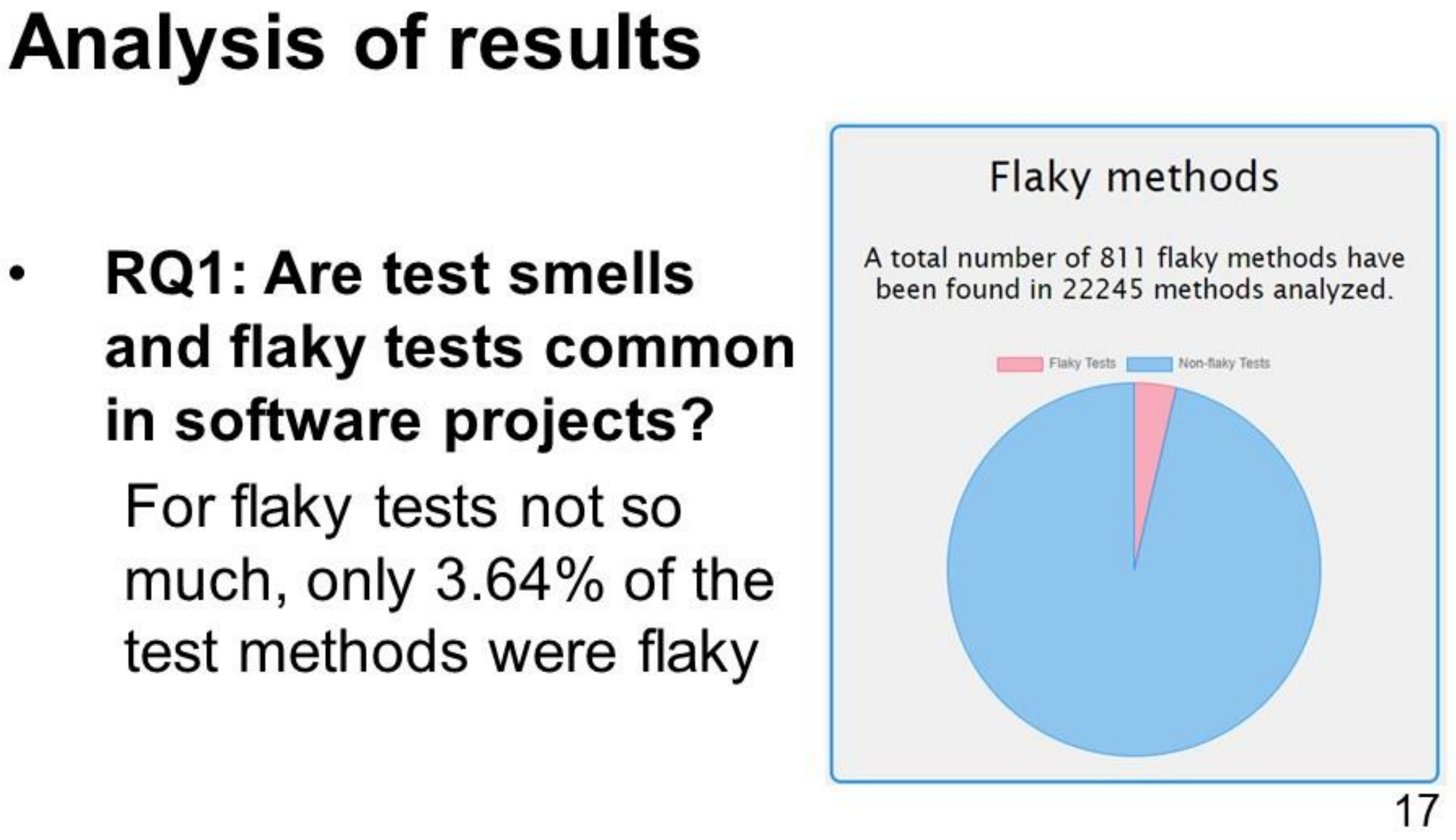


Smelly Methods

A total number of 5339 methods with test smells have been found in 22245 methods analyzed.

16

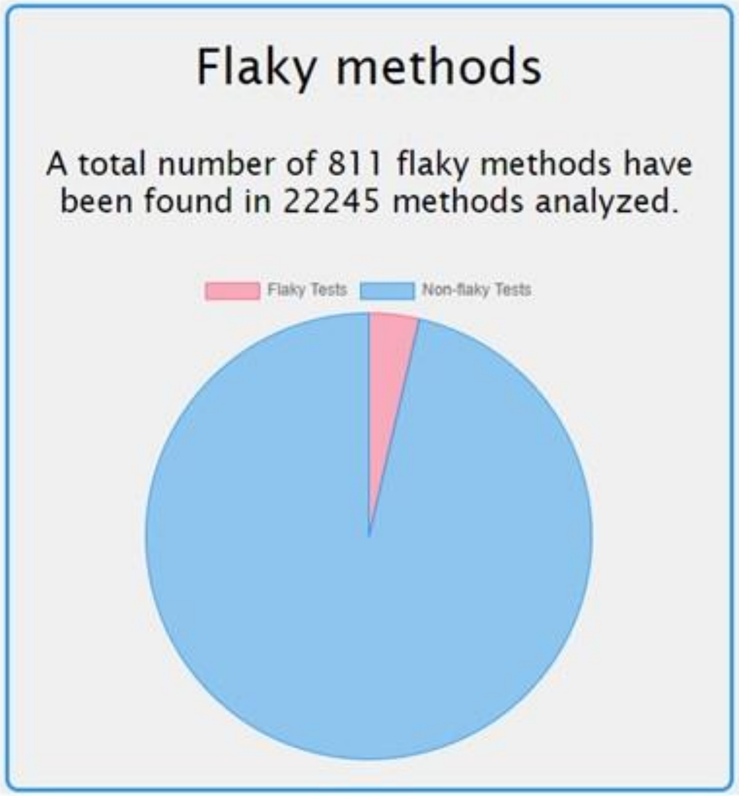
Figure 108. Slide 16: Analysis of Results



Analysis of results

- **RQ1: Are test smells and flaky tests common in software projects?**

  For flaky tests not so much, only 3.64% of the test methods were flaky

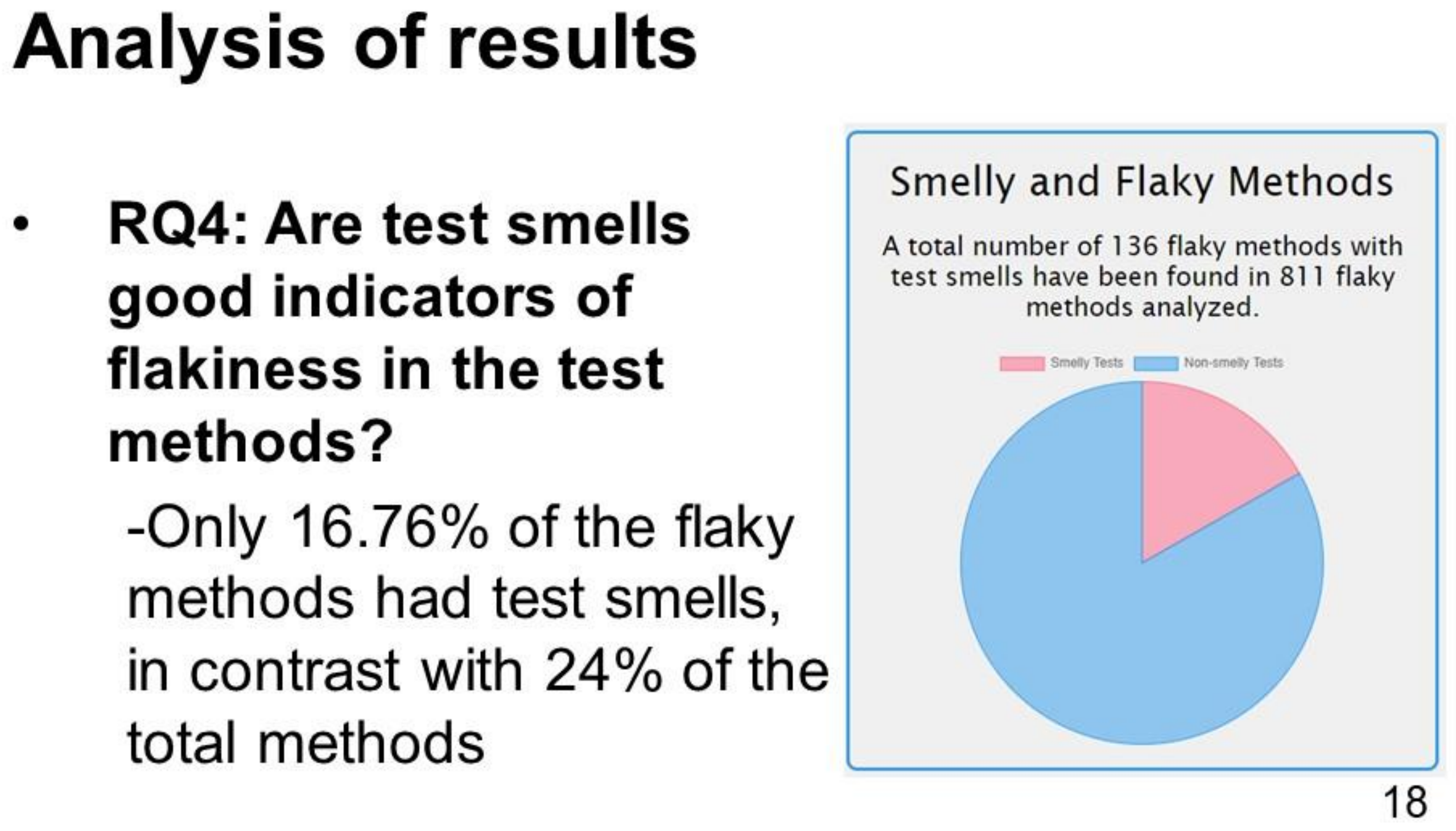


Flaky methods

A total number of 811 flaky methods have been found in 22245 methods analyzed.

17

Figure 109. Slide 17: Analysis of Results



Analysis of results

- **RQ4: Are test smells good indicators of flakiness in the test methods?**

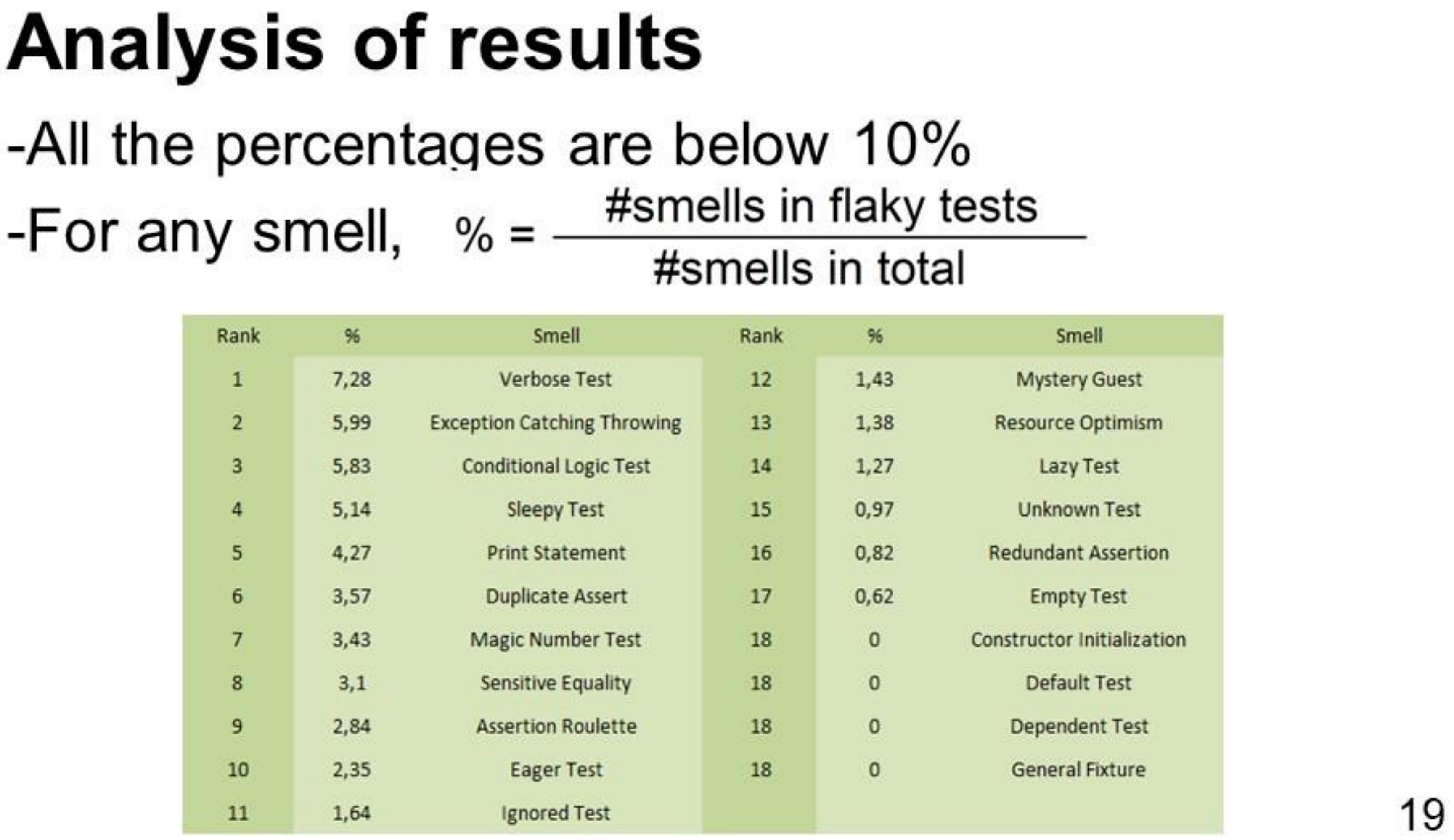
  -Only 16.76% of the flaky methods had test smells, in contrast with 24% of the total methods

Smelly and Flaky Methods

A total number of 136 flaky methods with test smells have been found in 811 flaky methods analyzed.

🟥 Smelly Tests
🟦 Non-smelly Tests
18

Figure 110. Slide 18: Analysis of Results



Analysis of results

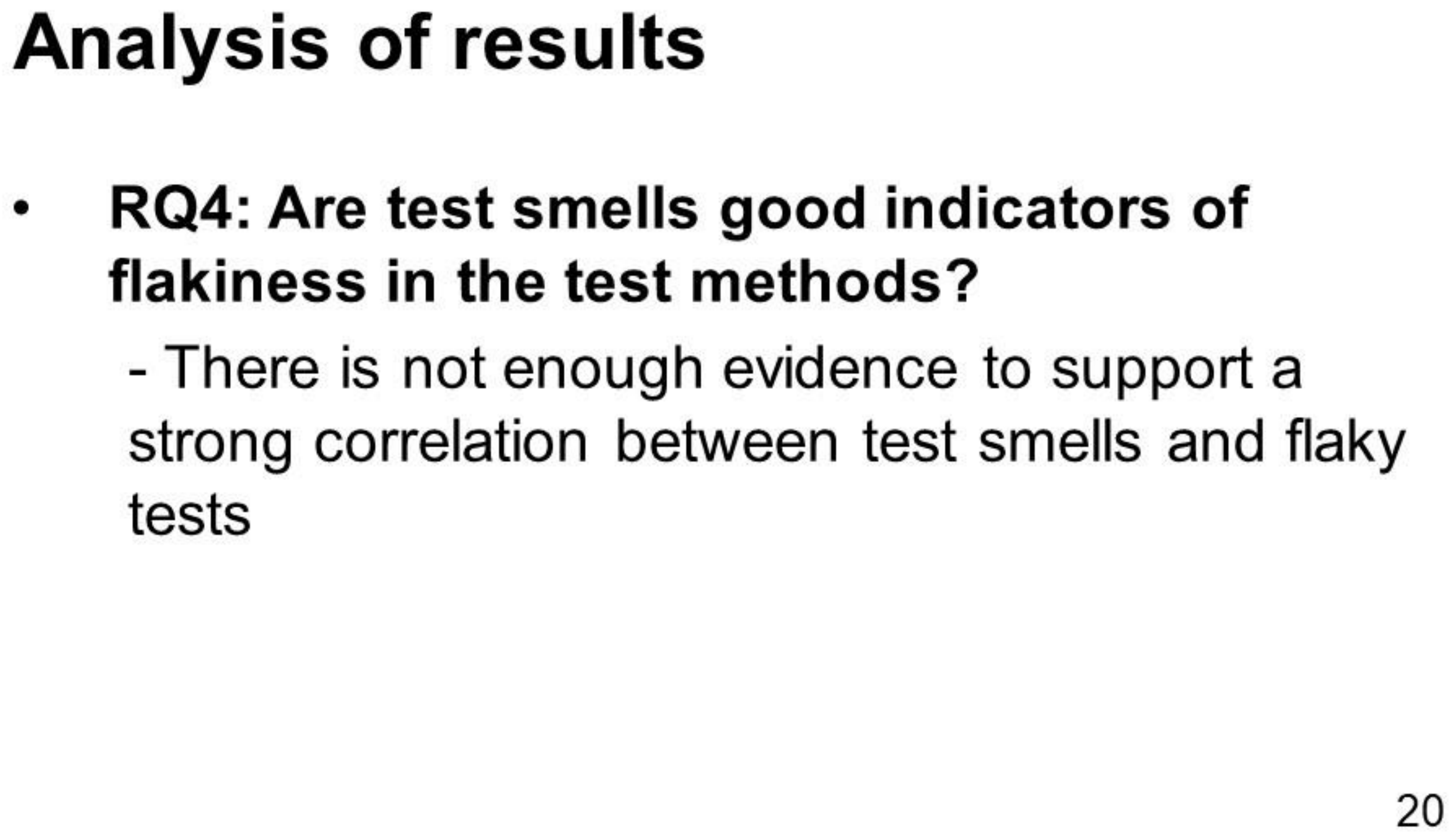
-All the percentages are below 10%

-For any smell, $\% = \frac{\#\text{smells in flaky tests}}{\#\text{smells in total}}$

Rank	%	Smell	Rank	%	Smell
1	7,28	Verbose Test	12	1,43	Mystery Guest
2	5,99	Exception Catching Throwing	13	1,38	Resource Optimism
3	5,83	Conditional Logic Test	14	1,27	Lazy Test
4	5,14	Sleepy Test	15	0,97	Unknown Test
5	4,27	Print Statement	16	0,82	Redundant Assertion
6	3,57	Duplicate Assert	17	0,62	Empty Test
7	3,43	Magic Number Test	18	0	Constructor Initialization
8	3,1	Sensitive Equality	18	0	Default Test
9	2,84	Assertion Roulette	18	0	Dependent Test
10	2,35	Eager Test	18	0	General Fixture
11	1,64	Ignored Test			

19

Figure 111. Slide 19: Analysis of Results



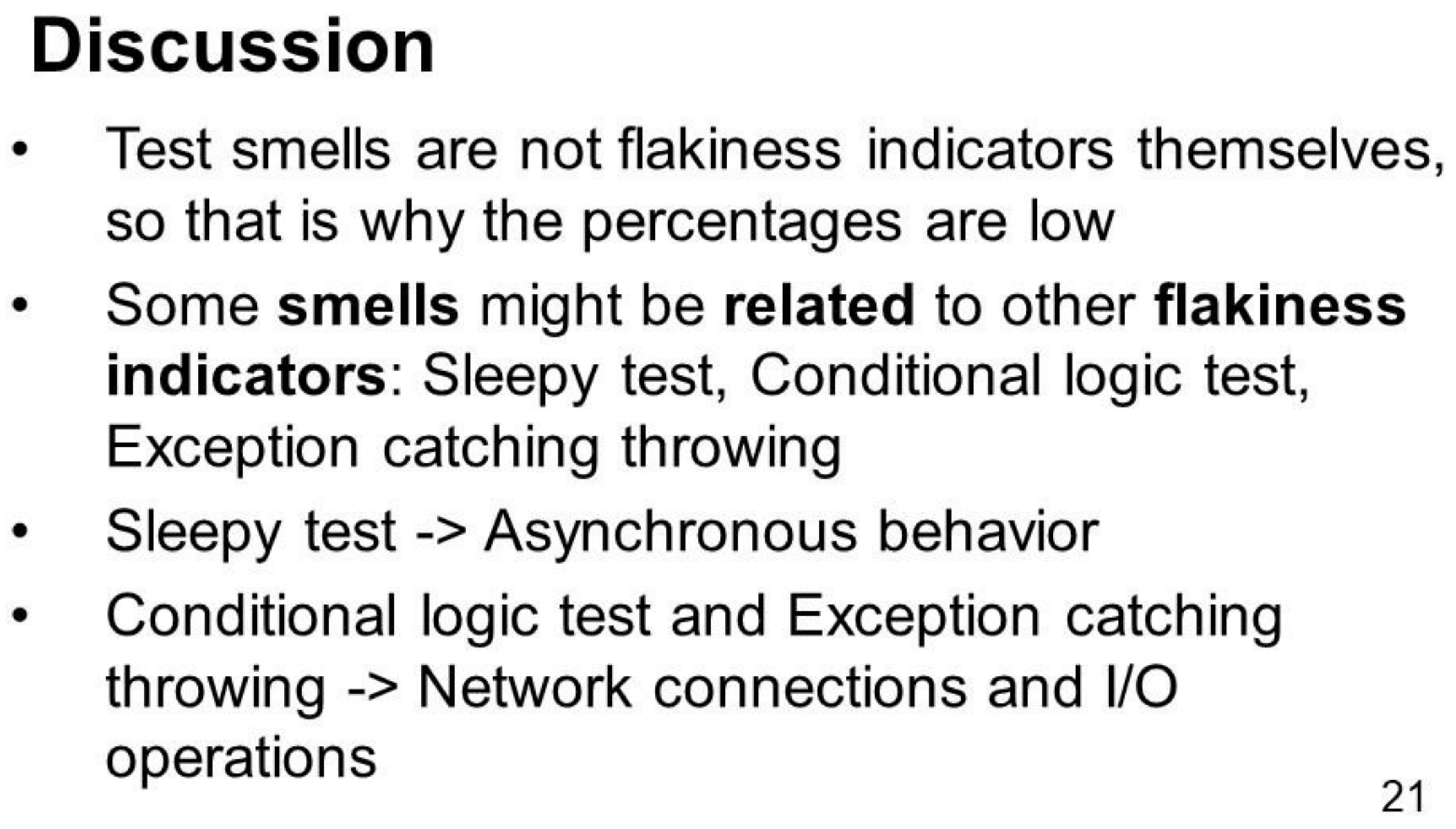
Analysis of results

- **RQ4: Are test smells good indicators of flakiness in the test methods?**

  - There is not enough evidence to support a strong correlation between test smells and flaky tests

20

Figure 112. Slide 20: Analysis of Results

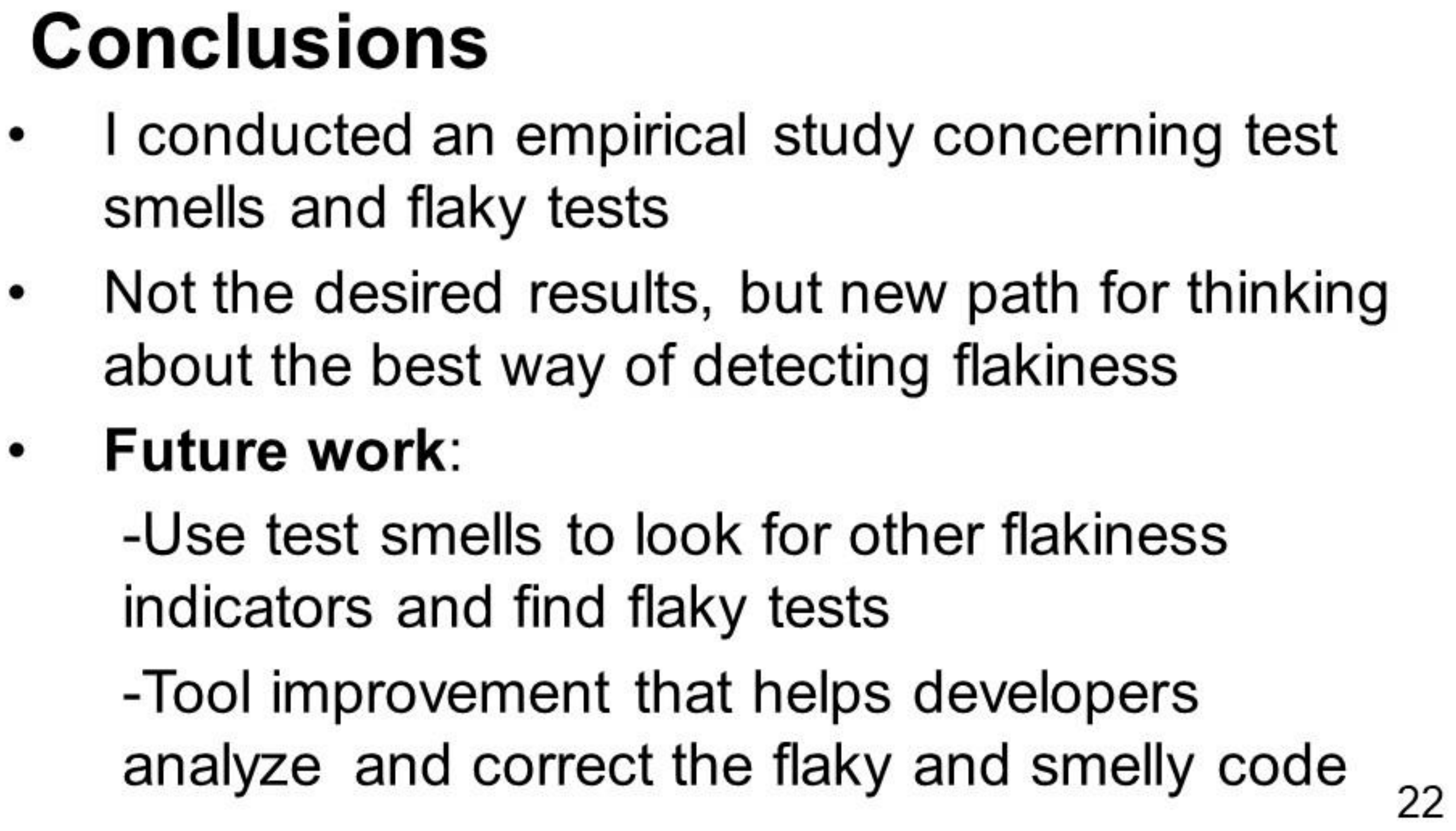


Discussion

- Test smells are not flakiness indicators themselves, so that is why the percentages are low
- Some **smells** might be **related** to other **flakiness indicators**: Sleepy test, Conditional logic test, Exception catching throwing
- Sleepy test -> Asynchronous behavior
- Conditional logic test and Exception catching throwing -> Network connections and I/O operations

21

Figure 113. Slide 21: Discussion



Conclusions

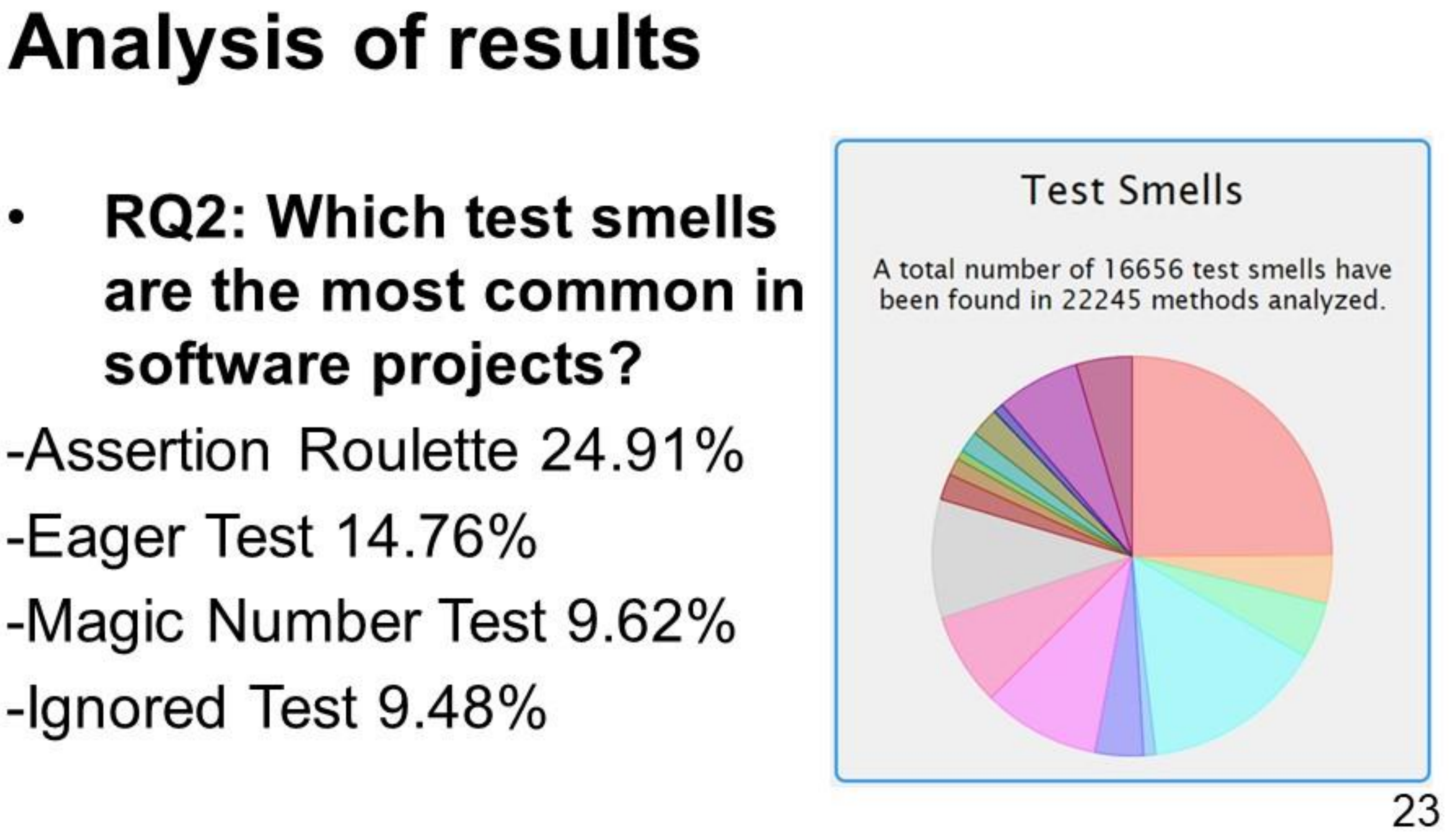
- I conducted an empirical study concerning test smells and flaky tests
- Not the desired results, but new path for thinking about the best way of detecting flakiness
- **Future work**:

  -Use test smells to look for other flakiness indicators and find flaky tests

  -Tool improvement that helps developers analyze and correct the flaky and smelly code

22

Figure 114. Slide 22: Conclusions



Analysis of results

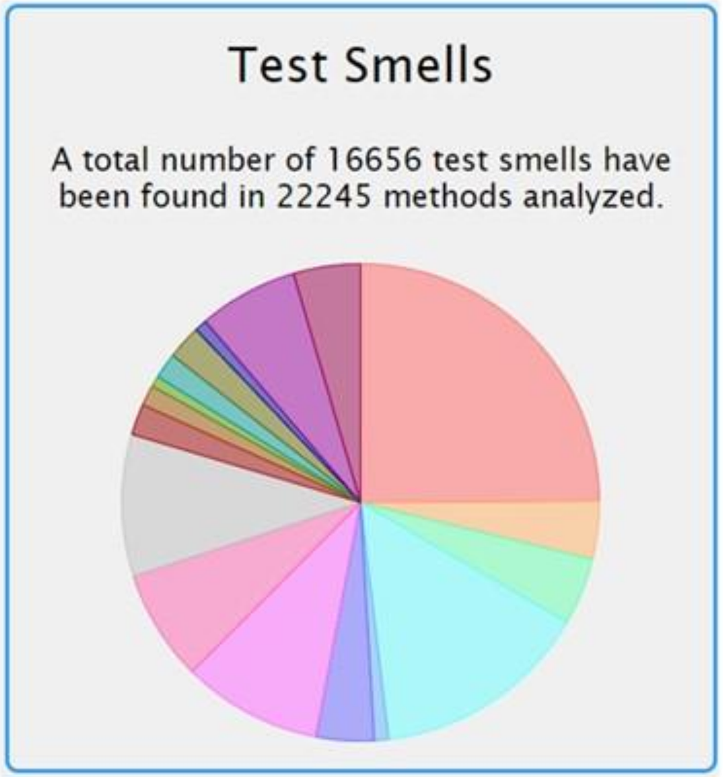
- **RQ2: Which test smells are the most common in software projects?**

-Assertion Roulette 24.91%

-Eager Test 14.76%

-Magic Number Test 9.62%

-Ignored Test 9.48%



23

Figure 115. Slide 23: Analysis of Results

Analysis of results

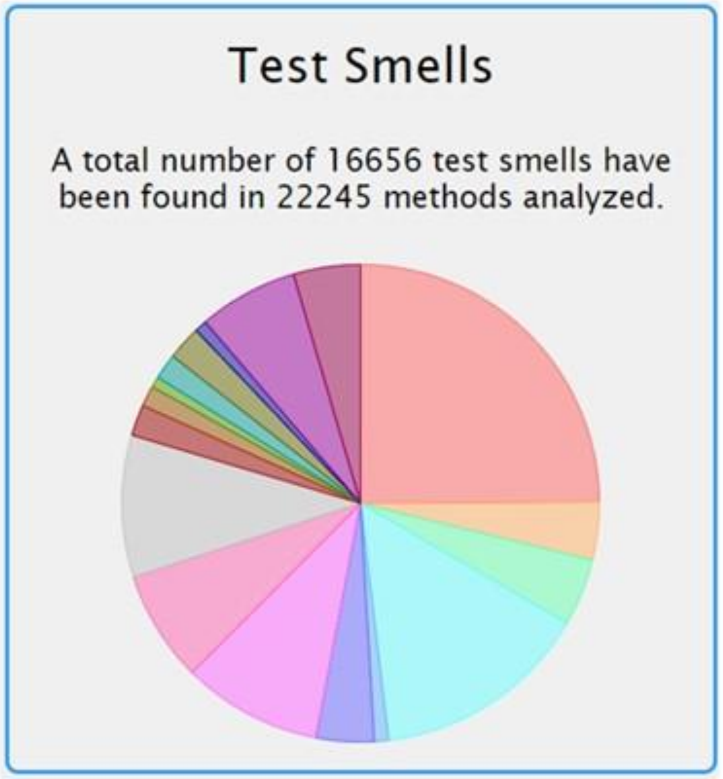
- **RQ3: Which test smells are the most common in flaky tests?**

-Verbose Test 7.28%

-Exception Catching Throwing 5.99%

-Conditional Logic Test 5.83%

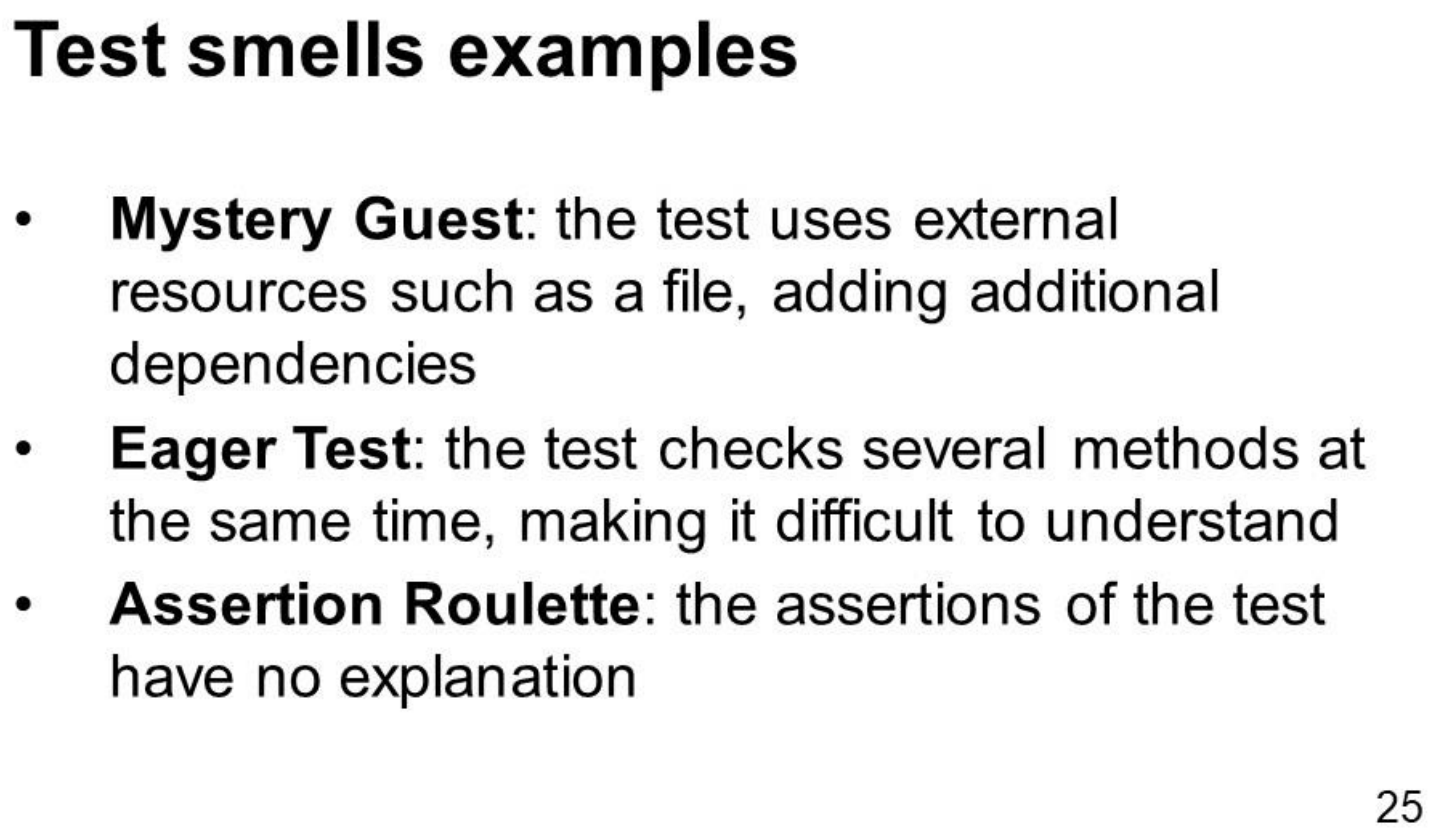
-Sleepy Test 5.14%


Test Smells

A total number of 16656 test smells have been found in 22245 methods analyzed.

24

Figure 116. Slide 24: Analysis of Results

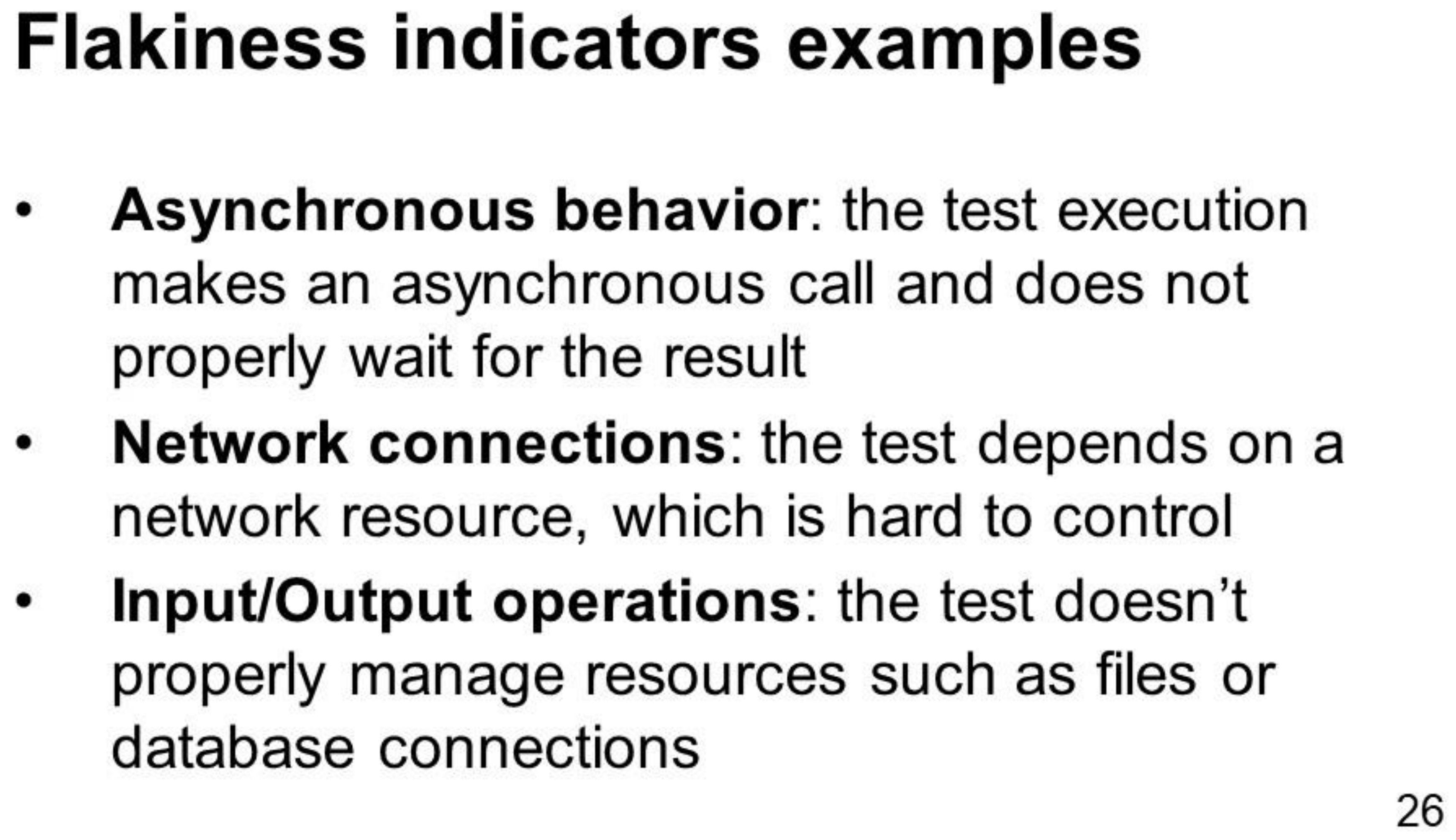


Test smells examples

- **Mystery Guest**: the test uses external resources such as a file, adding additional dependencies
- **Eager Test**: the test checks several methods at the same time, making it difficult to understand
- **Assertion Roulette**: the assertions of the test have no explanation

25

Figure 117. Slide 25: Test Smells Examples

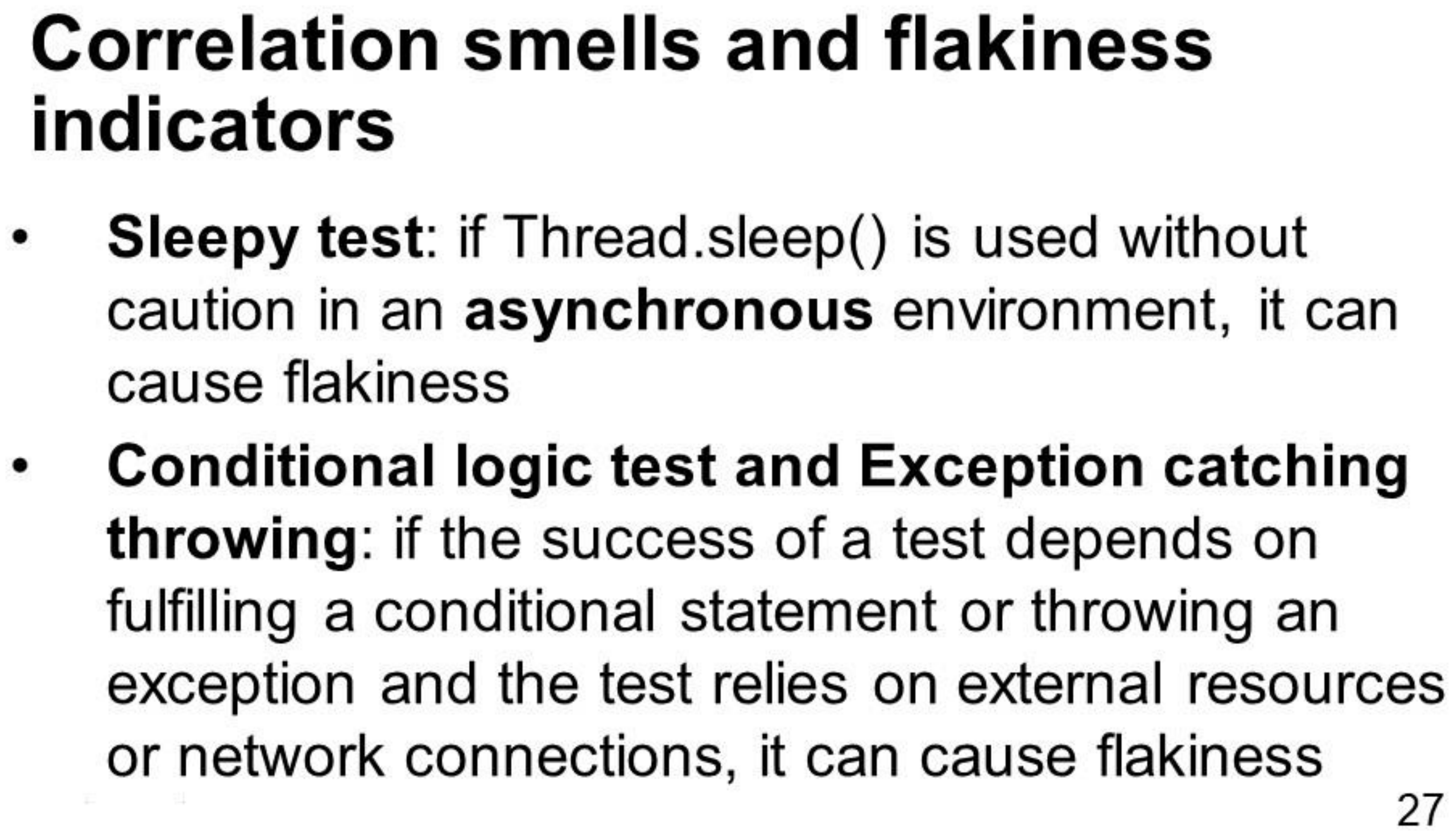


Flakiness indicators examples

- **Asynchronous behavior**: the test execution makes an asynchronous call and does not properly wait for the result
- **Network connections**: the test depends on a network resource, which is hard to control
- **Input/Output operations**: the test doesn't properly manage resources such as files or database connections

26

Figure 118. Slide 26: Flakiness Indicators Examples



Correlation smells and flakiness indicators

- **Sleepy test**: if Thread.sleep() is used without caution in an **asynchronous** environment, it can cause flakiness
- **Conditional logic test and Exception catching throwing**: if the success of a test depends on fulfilling a conditional statement or throwing an exception and the test relies on external resources or network connections, it can cause flakiness

27

Figure 119. Slide 27: Correlation Smells and Flakiness Indicators