universität
innsbruck

University of Innsbruck
Innsbruck, Austria

Department of Computer Science

Deep Test Code Analysis

Master Thesis

Florian Maier

Florian.Maier@student.uibk.ac.at

Innsbruck, 28 October 2022

Supervisor: assoz. Prof. Dr. Michael Felderer

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum

Unterschrift

Acknowledgements

I would like to thank my supervisor Michael Felderer, who took the time to read through my Master's thesis piece by piece again and again to give me tips, and who helped me to take a direction.
I would also like to thank Mauricio Aniche, Tech Academy Lead at Adyen and Assistant Professor in Software Engineering at TU Delft, for several Brainstorm sessions that helped me a lot.

Abstract

In the field of software engineering, the term ”smell” refers to a state of code in which it appears necessary to improve it using refactoring methods. In this way, for example, long, confusing code is made readable and easy to understand again. This makes it easier for the code to be maintained and reduces the occurrence of bugs. Such smells occur not only in production code, but also in test code. In software projects, tests are of great importance, and therefore it is also important to write smell-free tests.

This master thesis is dedicated to the creation of a test smell detection tool that detects a total of five test smells, namely the *anonymous test*, *assertion roulette*, *long test*, *rotten green test* and *conditional test logic* smell, in JUnit tests. This not only adds our tool to the list of existing detection tools, but also expands the list of recognizable smells, as it is the first tool to recognize the anonymous test.

Furthermore, in the course of this master thesis, a dataset of JUnit tests was created to measure the accuracy of our tool. The tool proved to be highly accurate with a precision of about 87-100%.

Contents

List of Figures

List of Tables

1 Introduction

Motivation

Code smells are known to every programmer. Even though there is no precise definition of code smells, they are always understood to be the causes of sloppily written code, which usually leads to refactoring, i.e. the reworking of already existing code [15]. As early as 1999, when the term Code Smells was coined by Kent Beck, people were looking for structures that were responsible for bad code [8][16]. There are many of these structures, for example duplicated code and long methods, which makes the code more difficult to comprehend. Detecting these code smells and fixing them (i.e. refactoring) go hand in hand in code development. So it is not surprising that over time various tools have been developed to automate the detection of code smells. Some of these tools focus on different code smells and their detection algorithms also work differently [15].

Where there is code, there are usually tests. In order to be able to determine the correctness of code, tests are needed. These are either carried out manually (i.e. a person executes the programme to be checked and observes its behaviour for correctness), or automatically using testing frameworks [5][20]. The more precise these tests are, the easier and more precise it is to locate bugs in the production code. This in turn highlights the importance of having well-functioning test code. Just as in production code (i.e. code that is necessary for the application to function, as opposed to test code), there are also sloppy designs in test code, called Test Smells [43]. Although there are numerous methods and practices for writing high quality test code [18], they are often not applied, which means that, as with production code, procedures are needed to detect test smells.

There are a large number of types of Test Smells and depending on the definition, new Test Smells are also being introduced all the time [20]. There are also a large number of tools that aim to detect test smells [3]. Some of these use different approaches to detect test smells (e.g. information retrieval or dynamic staining) and also focus on different programming languages (e.g. Java or Python).

The need for good test code or test smell detection tools can be quickly seen in real world examples. The lack of quality in test code usually leads to undetected bugs in production code. And bugs not only cost money, but also a lot of time to find. An extreme example of this is the Note 7 smartphone from Samsung. A software problem concerning battery management cost Samsung about 17 billion dollars [1].

In addition, the timing of the detection of a bug plays a major role. The more advanced

[1] ”Note 7 fiasco could burn a $17 billion hole in Samsung accounts”, Se Young Lee. Reuters, 2016

a software project is, i.e. in which phase of the software development process it is, the more expensive it will be to fix the bug. For example, in the last phase (”production / post-release”) it is supposedly 6 times more expensive than in the ”coding” phase [2].

Contribution

In the course of this master thesis, a Test Smell Detection Tool was developed. This tool fits into the list of already existing tools and thus contributes to the overall selection of possible test smell detection tools.
The tool is used to detect selected test smells, namely ”Anonymous Test”, ”Conditional Test Logic”, ”Long Test”, ”Rotten Green Test” and ”Assertion Roulette”. These five test smells were selected together with independent experts, namely assoz. Prof. Dr. Michael Felderer and Mauricio Aniche. We believe that these five smells are very widespread and can have a great impact on production. There is also no published tool for the anonymous test smell, so we felt the need to add this.
An attempt is made to use other algorithms and procedures than those of existing tools. For example, the number of statements in a test method is not determined by means of an abstract syntax tree, but semicolons and keywords are counted. In addition, no other tool was found that detects the Anonymous Test Smell. The tool is written in Java and focuses on Java JUnit tests.
Furthermore, a dataset with 1000 Java JUnit test methods was created, which were also manually checked for test smells. These test methods come from a total of eight open source GitHub projects.
Both the tool and the dataset will be made open source available to the public[3].
In summary, this Master’s thesis makes the following contribution:

- A new Test Smell Detection Tool for JUnit tests.
- New approaches to detecting test smells.
- The first tool that addresses the detection of the Anonymous Test Smell.
- A manually validated dataset with 854 Java JUnit test methods and 146 test helper methods

Structure

This thesis is structured into the following sections:
The Background chapter provides a comprehensive look at the topic of Test Smells. We present selected test smells and prevailing methods for detecting them. Furthermore, the importance of test smells in real life is discussed, i.e. whether the detection or avoidance of test smells has a high priority in companies. It is also determined what actual impact the existence of test smells has on the quality of test code and also production code.

[2] ”The exponential cost of fixing bugs”. Deepsource, 2019

[3] https://github.com/MaierFlorian/MasterThesisTestSmells

In the Related Work chapter, existing tools are presented that are used to detect test smells. In addition, studies are discussed that deal with the detection rates and accuracy of these tools, among other things. This serves as a starting point for the direct comparison with our tool. There are a number of such tools, each using different methods to approach the detection of test smells. Moreover, the focus of these tools is mostly on different types of test smells. This chapter aims to show the difference between these tools and the tool developed in the course of this master thesis.

In the Implementation chapter, the tool that was developed in the course of this master's thesis is presented. This tool is used to detect multiple test smells in JUnit test code and is provided as an open source project. The test smells detected by the tool are explained in detail, as well as the algorithms used to detect them.

In the chapter Evaluation, all steps for the assessment of the developed tool are explained. This includes the creation process of a dataset of 1000 JUnit test methods (includes helper methods) extracted from a total of eight open source GitHub projects. Each of these 1000 test methods was also manually checked for the test smells that are recognised by the tool. The results of this manual check were added to the dataset as a csv file. This dataset is made freely available, as is the tool that was created.
With the help of the developed tool and the created and reviewed dataset, the accuracy of the tool was then determined. Thus, this chapter also includes the results of the measurements on the accuracy of the tool. The results regarding the accuracy of detection are treated separately for each test smell.

In the Discussion chapter it is determined how well the developed tool recognises the respective test smells. In addition, the measured accuracies are compared with the accuracies of other tools. This should provide the reader with a comparison to help in the selection of a tool.

In the Conclusion chapter, everything is summed up again and an outlook on future work is given.

2 Background

This chapter introduces the topic of "Test Smells" and discusses relevant topics related to it. Not only is the topic itself introduced, but various types of test smells are presented as well as different methods of detecting them. Furthermore, the importance of recognising test smells is explained on the basis of their impact on the development of software.

2.1 Test Code Engineering

Writing test code is a large part of software development. The term "automated testing" refers to the automated testing of a system under test (SUT) using software tools and frameworks [47]. Usually, test code is written in a regular programming language with the aim of observing and monitoring the behaviour of a SUT under certain conditions. When this test code is written depends on the software development model used: often the test code is written after the production code has been developed (for example in the Waterfall model). There are also models in which test code is written first (e.g. Test-Driven Development). However, the following always applies: if the production code is changed, the test code must also be adapted. In this step, quality assessment is an important part.
Garousi et al. have illustrated the software test code engineering process (Figure 2.1).

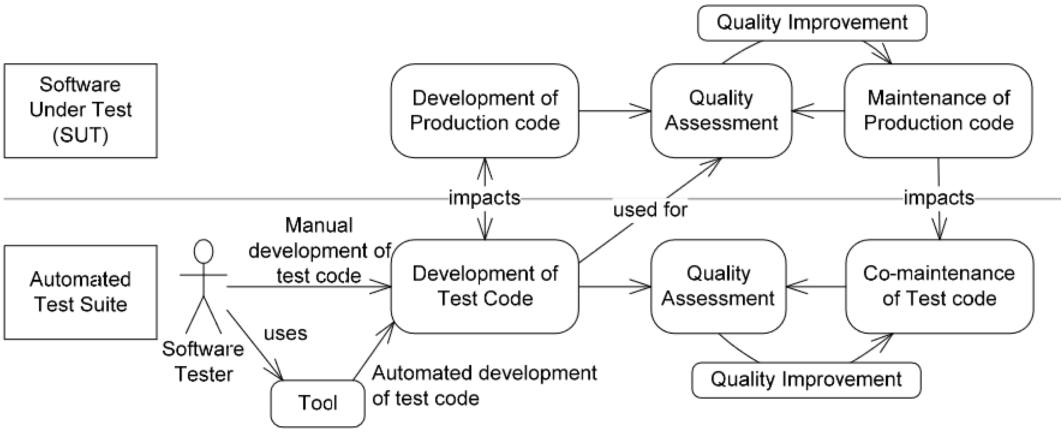

Quality Improvement
Software Under Test (SUT)
Development of Production code
Quality Assessment
Maintenance of Production code
impacts
used for
impacts
Manual development of test code
Automated Test Suite
Development of Test Code
Quality Assessment
Co-maintenance of Test code
Software Tester
uses
Tool
Automated development of test code
Quality Improvement

Figure 2.1: Software Test Code Engineering Process [47]

Garousi et al. examined studies that focused on the quality assessment part of the process. The majority of these studies addressed the question of how to recognise test smells.

2.2 The Smell Concept In Software Engineering

Before we come to the definition of "Test Smell", two more definitions are pending.

The first definition is that of "refactoring". Martin Fowler[1], a specialist in the design of software systems and co-founder of the term "code smell", describes the term "refactoring" as "a change in the internal structure of software with the aim of making it easier to understand and cheaper to modify without changing its behaviour" [16].
According to T. Mens and T. Tourwé, the first step in the refactoring process is to identify where the software should be refactored [29]. This is also the starting point for the next definition.

The second definition is that of "code smell". K. Beck and M. Fowler coined this term as an answer to the question "when should one refactor" [8]. They emphasised that they cannot give precise criteria for when refactoring is necessary. It is more a matter of human intuition to know when to refactor. The term "code smell" is only meant to indicate structures in the code that "smell", i.e. structures that could lead to the need for refactoring. They also give some examples of such structures, for example duplicated code or methods that are too long.
Similar definitions and mentions of "code smells" include "manifestations of design flaws that can degrade code maintainability"[46], and "poor implementation choices"[23].
As Beck and Fowler noted, the identification of code smells is subjective and requires human intuition. Mantyla et al. wondered to what extent developers' opinions on code smells differ and whether their assessments of bad code match those of source code metrics [27]. Their research found that subjective opinions about bad code often differ depending on the experience, role and knowledge of the developers interviewed. Moreover, their assessments did not match those of the source code metrics.

Now that we have dealt with the terms "refactoring" and "code smell", we come to the definition of "test smell". Moonen and van Deursen introduced the term "test smell" in 2001, which - like code smells in production code - describes bad design in test code [43]. Like code smells, test smells have a bad influence on the maintainability and clarity of code. The difference between these two smells is only the code to which they are limited: code smells are intended for production code, i.e. code that is necessary for the functioning of the end product. Test code, on the other hand, was written to perform developer tests, i.e. to test production code [44]. This distinction between the two types of code is important. According to van Daursen et al., refactoring both types of code differs in two ways: first, they have different types of smells, and second, improving test code requires further test-specific refactoring [43].

Table 2.1 provides a summary of the definitions and includes examples.

[1] https://martinfowler.com/aboutMe.html

Concept	Description	Examples
Refactoring	Rewrite parts of the code to make it easier to understand. The aim is to eliminate smells and thus facilitate maintainability and the detection of bugs.	Split a method into two methods so that each method has exactly one task or is not too long and therefore more readable.
Code Smell	Code structures that lead to the use of refactoring as a necessary consequence.	(1) Duplicated code (2) Method with too many statements
Test Smell	Structures in test code that lead to the use of refactoring as a necessary consequence. Differs from code smells in terms of the types of smells and the refactoring practice.	Assertions within conditional statements, which could lead to them not being executed.

Table 2.1: The Smell Concept in Software Engineering

2.3 Unit Testing And JUnit

Since test methods (more precisely JUnit test methods) are the focus of this Master's thesis, important test-specific terms are explained and illustrated below.

Unit testing describes the activity of testing units of code in isolation and comparing the result with an expected result [30]. In Java, such a unit is usually a class. A unit test then usually calls one or more methods of this class and checks their output for correctness.
JUnit is a unit testing framework for Java [1]. All test methods used in this master thesis are JUnit test methods. In *code snippet 1* an example of a JUnit test is shown.

Code snippet 1: JUnit test example

```java
public class TestSomeClass {

    @Test
    public void testAddWithPositiveNumbers(){
        SomeClass sc = new SomeClass();
        Assert.assertEquals(8, sc.add(4, 4));
    }

    @Test
    public void testAddWithHelperMethod(){
        helperAdd(2, 2, 4);
    }

    public void helperAdd(int a, int b, int expected){
        SomeClass sc = new SomeClass();
        Assert.assertEquals(expected, sc.add(a, b));
    }

}
```

JUnit uses annotations to mark tests or to define a sequence[2]. An annotation is always written before a method. The most important annotation for us is "@Test", which defines a JUnit test method (see *code snippet 1*). Other annotations worth mentioning are "@BeforeEach" (formerly "@Before") and "@AfterEach" (formerly "@After"), which define a method that is executed before each test. This allows, for example, variables to be initialised before the tests.

When testing a method, the expected output is compared with the actual output [26]. Instead of comparing the results with an if-statement, for example, JUnit uses so-called assertions[3]. Assertions can pass or fail. If all assertions in a test method pass, the test passes. If at least one of the assertions fails, the entire test method fails. If a test method has no assertions, the test usually passes. Assertions always begin with the string "assert". The only exception is the assertion "fail".
Selected assertions are presented below.

- *assertEquals(int expected, int actual)*: This assertion exists in many variations for different data types. In this case, it is checked whether two integers are equal. See *code snippet 1.*
- *assertTrue(boolean condition)*: This assertion checks if a condition is true.
- *fail()*: This assertion ensures that the test fails immediately. This assertion can be called, for example, when a certain exception is thrown.

Regardless of the framework used, unit testing often uses helper methods [14]. Assertions are stored in these helper methods and called with variable input. The method "helperAdd" in *code snippet 1* serves as an example of such a helper method.

2.4 List Of Selected Test Smells

Selected test smells are described below. Some of them are detected by our tool and described in detail in the chapter "Implementation".

- Anonymous Test: Anonymous tests are tests whose names poorly describe the task of the test [36]. If a test has a good name, it is the name alone that tells you what the test does and you do not have to go through the test line by line. This test smell reduces the reader's understanding. *Code snippet 2* shows an example of an anonymous test.
- Assertion Roulette: This describes the situation when a test method contains several assertion-statements without explanation [7]. This smell is a bad programming decision, which results in the fact that if the test fails, one does not know which of the assertions was the reason. *Code snippet 2* shows an example of an assertion roulette.

[2] https://junit.org/junit5/docs/current/user-guide/writing-tests-annotations
[3] https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html

- Conditional Test Logic: A test method includes conditional test logic if it includes one or more control statements, for example ”if”, ”switch”, and ”for” statements [33]. A test should always run the same way and therefore the use of conditional statements is not advisable. *Code snippet 2* shows an example of conditional test logic.

- Eager Test: Several methods of the object to be tested are tested [7]. This has a particular impact on maintainability.

- Lazy Test: Several test methods check the same method under the same conditions [7].

- Long Test / Verbose Test: Test methods that contain too many statements are called long tests [36]. Verbose Tests [9], also called ”Obscure Tests” [36] are hard to understand due to their complexity. These tests are very similar. They reduce readability.

- Rotten Green Test: This refers to a test method that includes assertions but never executes them. This happens, for example, because the assertions are bound to a condition that does not occur [14]. This can cause the test to pass when executed, even though some of the assertions it contains might not have passed. A similar type of test is the Smoke test, which does not contain any assertions and therefore usually passes.

- Mystery Guest: If a test method accesses external resources, there is a risk that the values will change and the result will vary [7].

- Test Pollution: Test Pollution occurs when a test uses a shared resource, such as writing to a file or changing a global variable [21].

- Duplicated Code: Test methods with redundant code [9]. This test smell increases the maintenance cost.

Code snippet 2: Example of different test smells

```
public class TestSomeClass {

   @Test
   public void testAdd(){ // Assertion Roulette smell
      SomeClass sc = new SomeClass();
      Assert.assertEquals(8, sc.add(4, 4));
      Assert.assertEquals(2, sc.add(4, -2));
      Assert.assertEquals(10, sc.add(9, 2));
   }

   @Test
   public void testAddWithCondition(){ // Conditional test logic smell
      SomeClass sc = new SomeClass();
      if(sc.add(4, 4) > 0) {
            Assert.assertEquals(8, sc.add(4, 4));
        }
    }

    @Test
    public void test(){ // Anonymous test smell
        SomeClass sc = new SomeClass();
        Assert.assertEquals(6, sc.add(2, 4));
    }

}
```

The first method has the Assertion Roulette smell. When the test fails, it is not clear which assert-statement was to blame for the failure. The second method includes conditional test logic. If the condition does not occur, the assertion is never carried out. The third method does not have a meaningful name, thus it is an anonymous test.

2.5 Test Smell Detection Techniques

There are four predominant techniques that test smell detection tools use to detect test smells in code [3]. These methods are briefly presented below.

- Metrics: Using a metrics-based approach, the smell symptoms are measured in numbers and compared to a certain threshold. Often the source code is parsed and converted into an abstract syntax tree (AST), which facilitates the further procedure. Our tool, on the other hand, avoids the use of an AST. For example, our tool detects a long test by counting certain keywords and semicolons in the code.

- Rules/Heuristic: This is an extension of the metrics-based technique. Here, the code is searched using patterns. For example, our tool uses many regular expressions to search for patterns. For the detection of anonymous tests, the method name is searched for patterns using natural language processing methods.

- Information Retrieval: Information is extracted from the code and normalised using natural language processing methods (e.g. stemming, keyword removal, ...). The normalised text is then weighted and evaluated using machine learning algorithms.

- Dynamic Tainting: In this method, the code is executed and evaluated using runtime information.

In this context, the term ”static analysis” must also be explained. This describes the method of inspecting code without any input data and without executing the code [6]. With static analysis, it does not matter what the programme does. Rather, one searches for specific patterns. Therefore, for example, you will not find bugs that you are not looking for [10].

2.6 Distribution And Impact Of Test Smells

Now that we have explained all the relevant terms and introduced some test smells, it is time to talk about the impact of test smells. More specifically, we focus on the actual frequency of test smells in software and their impact on programmers.

Bavota et al. examined 16 open source and two industrial software systems and analysed the distribution of various test smells [7]. Each of these software systems was implemented in Java and the JUnit framework was used. All software systems examined contained a total of 637 JUnit tests. Of these, 82% contained at least one test smell. Furthermore, they found that the existence of a test smell in a test often entails the existence of other test smells. For example, in about two thirds of the cases, other test smells coexisted with the Assertion Roulette smell. The Lazy test and eager test also often occurred in combination. Similar results were also obtained by Peruma et al. [34].

In another study, Bavota et al. investigated to what extent the existence of test smells influences the understanding of source code during its maintenance [7]. Twenty master's students participated in the study. It turned out that the test smells had a negative impact on the students' understanding of the source code during maintenance, more specifically on accuracy as well as time.

2.7 How Developers Deal With Test Smells

This raises the question of the extent to which developers are familiar with test smells, whether they actively seek not to implement test smells, and how they deal with test smells.

Junior et al. created a survey with the aim of finding out whether professional developers unintentionally implement test smells [22]. More specifically, they asked themselves three research questions: (1) Do professionals use poor design practices that lead to test smells? (2) Do experienced developers implement fewer test smells than inexperienced ones? (3) What daily practices of these developers lead to test smells?
They found that companies often set poor standards which are adopted by developers and lead to test smells. Furthermore, experience does not necessarily lead to a developer introducing fewer test smells. Finally, they found that developers often use conditional structures or repetitions and generic configuration data, which eventually leads to test smells.

Kim et al. addressed the question of how test smells in software systems evolve over time and what the motivation was for developers to resolve existing test smells [25]. They conducted a study in which they monitored the test smells in selected software systems for about three years. They observed that more test smells were introduced into the code over time. However, the test smell density remained about the same compared to the lines of code. Furthermore, they checked all commits that had to do with the removal of test smells. In only 17% of these commits were test smells deliberately addressed. In the remaining cases, the reason for removing test smells was a by-product of other refactoring and deleting tests.

To improve the above statistics and to suggest better standards for testing to companies, there are various collections of existing test smells and guidelines to eliminate or prevent them [19][20]. Based on the literature found, Garousi et al. developed a lifecycle for the manifestation and elimination of test smells (see figure 2.2).

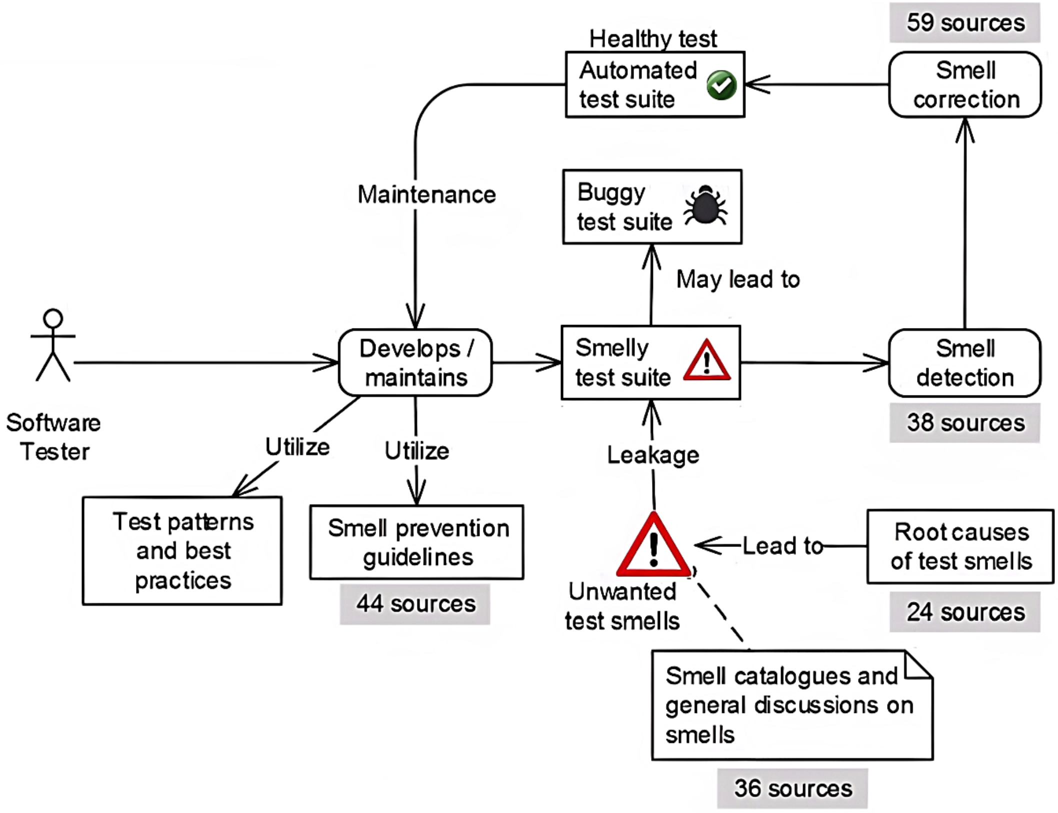


Figure 2.2: Test Smell Life Cycle by Garousi et al. [20]

Developers use guidelines and patterns to create test code, which will usually contain test smells despite preventive measures. These are then eliminated by means of smell detection and correction measures.

2.8 Automated Test Generating Tools

In order to reduce the time and cost of creating tests, automatic test generating tools have been developed. These tools are usually developed for specific frameworks and are used to automatically generate test cases, usually with the aim of having the largest possible test coverage.
These tools are sometimes tailored to certain aspects of testing (for example, only to test boundary values as input). Some of these tools require additional information from the user, others do without any input and are therefore completely autonomous.
Some of these tools are presented below.

Tufano et al. developed a tool to generate assert statements. On the first run, this tool produced assert statements that were exactly the same as those written by developers 62% of the time [42].

JCrasher focuses on type information of methods and tries to cause runtime exceptions in these methods with random data [11]. By using random data, JCrasher runs completely autonomously.

TestGen4J is a tool for creating JUnits test cases that focus specifically on boundary values [45]. However, the user has to write the necessary values in an XML file.

JUB[4] is a JUnit test case generator framework which, unlike other tools that can only be used with the command line, is available as an extension for numerous IDEs. JUB is ahead of other tools in some aspects, for example, tests can also be created for protected methods.

Another example of an automatic test generating tool is Devmate[5], which uses AI to generate unit tests from a requirements perspective and user scenarios.
On the other hand, Daka et al. note that automatically generated tests usually do not cover real-world scenarios, but focus on covering as much code as possible [13]. For this reason, it is usually difficult to choose meaningful method names for these tools. Thus, test methods are often named only ”test0” or similar.
Automatically generated tests are therefore no guarantee that no test smells will occur, as in this case with the anonymous test.

[4] http://jub.sourceforge.net/

[5] https://www.devmate.software/

3 Related Work

This chapter presents existing test smell detecting tools. Among other things, the methods they use for detection, the test smells they focus on and their target language are discussed. A study has also been published on some of these tools, discussing their detection rates and accuracy, among other things. This chapter is intended to serve as a starting point for a later comparison with our tool.

Aljedaani et al. compiled a catalogue of all known peer-reviewed test smell detection tools [3]. Among other things, they also drew up a list of characteristics that can be used to classify the test smell detection tools. These are:

- Programming Language: In which language is the tool written and on which programming language is the detection aimed? Our tool is written in Java and detects test smells in java test methods.
- Supported Test Framework: Which testing frameworks are supported by the tool? For example, our tool focuses on JUnit tests.
- Correctness: How accurate is the tool in detecting test smells?
- Detection Technique: Which of the detection techniques presented in section 2.5 is used to detect test smells?
- Interface: How can the user interact with the tool? For example, is there a graphical interface or a command line interface?
- Usages Guide Availability: Is it explained to the user how to use the tool?
- Adoption in Research Studies: Is the tool mentioned in studies and other literature?
- Tool Website: Is there a website for the tool, for example including source code.

In addition to these characteristics, we would like to add another characteristic which we consider important: Customisability. Certain test smells are not clearly defined and require subjective assessment. An example of this would be the Long test. This occurs when a test method becomes too long and difficult to read. The question arises as to how many statements are needed to make a method a long test. Another example is the anonymous test. Although there are numerous guidelines that specify the structure of good method names (for example, 'test' + method name), some companies have their own guidelines. The characteristic *customisability* should therefore describe the extent to which one can influence the detection algorithms of the tools.

Since our tool focuses on JUnit tests, we will limit ourselves in the following to tools that concentrate on them. In addition, we only present those tools that detect the same or similar test smells as our tool, or make some other relevant contribution. Tools about which little is known (e.g. measured accuracies or applied methods) were also excluded. The presented test smell detection tools consist of selected tools from the catalogue by Aljedaani et al. published in 2021 [3]. A search on google-scholar did not reveal any further relevant findings.

TsDETECT is an open source test smell detection tool and compared to the following tools, rather one of the better known tools [33][34]. It was mentioned in the following studies, among others: [17], [24], [32], [34], [35], [37], [38], and [39]. TsDETECT uses different detection rules to detect test smells. It has a high average precision score of 96% and an average recall score of 97%. It can be used via the command line and detailed documentation can be found on its website[1]. Due to the open source publication on github[2] and the relatively simple detection rules, TsDetect offers an easy way to modify existing rules and add new ones. Currently, the tool includes rules for 21 different test smells, including Assertion Roulette, Eager Test, Lazy Test, and Verbose Test. For each smell, a separate detection rule is implemented. For example, for assertion roulette, the system checks whether there is more than one assertion without an explanation message (additional parameter in the assertion method) in a test method. This is done with the help of an Abstract Syntax Tree (AST).

In the course of their study on the impact of test smells on software maintenance, Bavota et al. developed a rule-based test smell detection tool (unnamed) [7]. This uses very simple rules to achieve the highest possible recall. For example, to detect the mystery guest smell, it only looks at whether the JUnit class uses an external resource. The tool can be operated via the command-line, but there is no usage-guide. It is unknown in which programming language the tool is implemented.

TASTE (Textual AnalySis for Test smEll detection) is a textual-based test smell detector which uses textual analysis to detect three types of test smells: general fixture, eager test, and Lack of Cohesion of Methods [31]. The method used falls into the category of information retrieval. For example, to detect the eager test, they assumed that two methods affected by this smell have a high textual similarity. Based on this assumption, they replaced the method calls with their source code. If two test methods have called the same methods, they now have a high textual similarity. Based on the textual similarity of two test methods, the probability of an eager test smell was then calculated. TASTE achieves a precision of 57%-75%, a recall of 60%-80%, and a F-Score of 62%-76%. It is not known in which language the tool is implemented and how to

[1] https://testsmells.org/

[2] https://github.com/TestSmells/TestSmellDetector

interact with the tool. There is no usage guide.

Martinez et al. developed RTj, a tool to detect rotten green tests [28]. The tool mixes two techniques: rule-based detection and dynamic tainting. In the first step, the tool performs static analyses on the parsed code. In the second step, the tool checks during the execution of the tests whether the previously selected elements are executed. A special feature of RTj is that it makes suggestions to the user for refactoring. The tool is operated via the command-line and there is also a usage-guide. In [28] there is no mention of the accuracy of the tool.

To provide an accurate comparison to our results, see Table 3.1 for a detailed summary of the results of each tool. Only the test smells that are also recognised by our tool are included. It should be noted that we have not found a test smell detection tool that deals with the anonymous test.

Smell Type	Detection Tool								
	tsDetect			[7]			RTj		
	P	R	F	P	R	F	P	R	F
AR	94.74%	90.00%	92.31%	100%	100%	?	-	-	-
CTL	100%	100%	100%	-	-	-	-	-	-
LT	?	?	?	-	-	-	-	-	-
RGT	-	-	-	-	-	-	?	?	?
AT	-	-	-	-	-	-	-	-	-

Table 3.1: Results of other test smell detection tools

P: Precision, R: Recall, F: F-Score, AR: Assertion Roulette, CTL: Conditional Test Logic, LT: Long/Verbose Test, RGT: Rotten Green Test, AT: Anonymous Test

4 Implementation

This chapter deals with the implementation of our test smell detection tool. On the one hand, the features of the tool are discussed, on the other hand, the detected test smells are discussed in more detail and the applied algorithms are explained.

4.1 General Information

The test smell detection tool, which was developed in the course of this master thesis, currently focuses on the following test smells: (1) Anonymous Test, (2) Long Test, (3) Conditional Test Logic, (4) Rotten Green Test, and (5) Assertion Roulette.

The detection concentrates on smells in JUnit test code. The tool only analyses Java test files. The source code is not included in the analysis. The aim is to achieve the highest possible accuracy with the shortest possible runtime.
The tool itself is programmed in Java.

The detection of the smells is based on a rules/heuristic approach, whereby the source code is searched for patterns using regular expressions. Natural language processing methods, more precisely part-of-speech (POS) tagging, are also used.

Interaction with the tool takes place via a graphical user interface (GUI). The main reason for choosing interaction via a GUI instead of a command-line was the easier conditions for configuration. Our tool offers various possibilities for customisation. In addition, a GUI facilitates the selection of Java test files from various sources. If one were to use the tool via the command line, specifying several file paths would result in an unnecessarily complex command. However, consideration is being given to incorporating interaction via the command line to enable automated use. Integration into an IDE such as IntelliJ would also be possible in the future.

Some types of test smell often require further adjustment. In the following, the customisable parameters of the recognised test smells are presented. Figure 4.1 shows a screenshot of the configuration window of our GUI.

- Anonymous Test: Various suggested names for test methods can be found in grey literature, which should provide a good insight into the function of test methods. Of these, five have been selected that we found most useful. By default, the test methods are checked against each of these suggestions. However, if the user wants a single name pattern to be used consistently, he or she can set this.

- Long Test: There are various opinions on when a test contains too many statements and is therefore too difficult to read. By default, we set a threshold of 13 statements per test method. This value is based on a study on severity thresholds for test smells [39]. However, the user can change this value as desired.

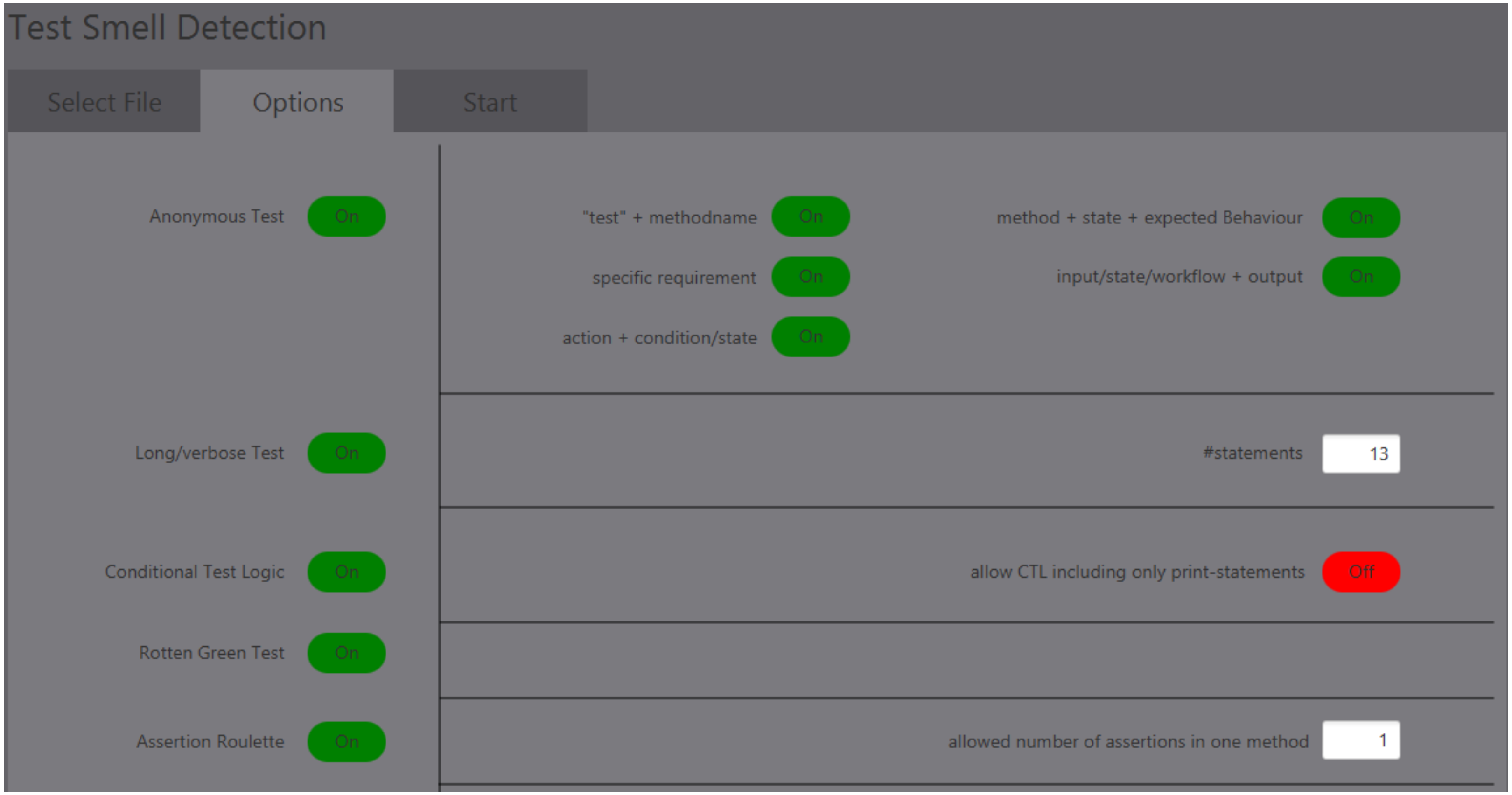
- Conditional Test Logic: The use of conditional test logic in tests not only degrades readability, but can also be the cause of other smells, such as Rotten Green Test, and produce unexpected results. However, conditional statements can be useful in certain circumstances, such as for manual debugging. Therefore, the user of our tool has the option to allow conditional statements that only contain print statements.

- Rotten Green Test: There are currently no possibilities for adjustment for this.

- Assertion Roulette: Even though the assertion roulette test smell already occurs as soon as a test method contains more than one assert-statement without additional explanation, this value may be too low for some developers. Therefore, the user can set the maximum number of assert-statements per test method.

After selecting the Java files to be checked and adjusting the detection algorithms, the user can initialise the search process. While the tool is running, the user is continuously shown information about what is being done. At the end of the execution, the user is informed about how long the tool took for the individual processes. Finally, the user has the option of downloading all results as a .csv file. An example of this is shown in Figure 4.2. The UML Activity Diagram 4.3 shows in a simplified way how the tool works.

In addition to this master thesis, which among other things serves as documentation of the tool, the GUI contains a section that briefly introduces the test smells and explains how to use the tool.



Anonymous Test On

"test" + methodname On

specific requirement On

action + condition/state On

method + state + expected Behaviour On

input/state/workflow + output On

Long/verbose Test On

#statements 13

Conditional Test Logic On

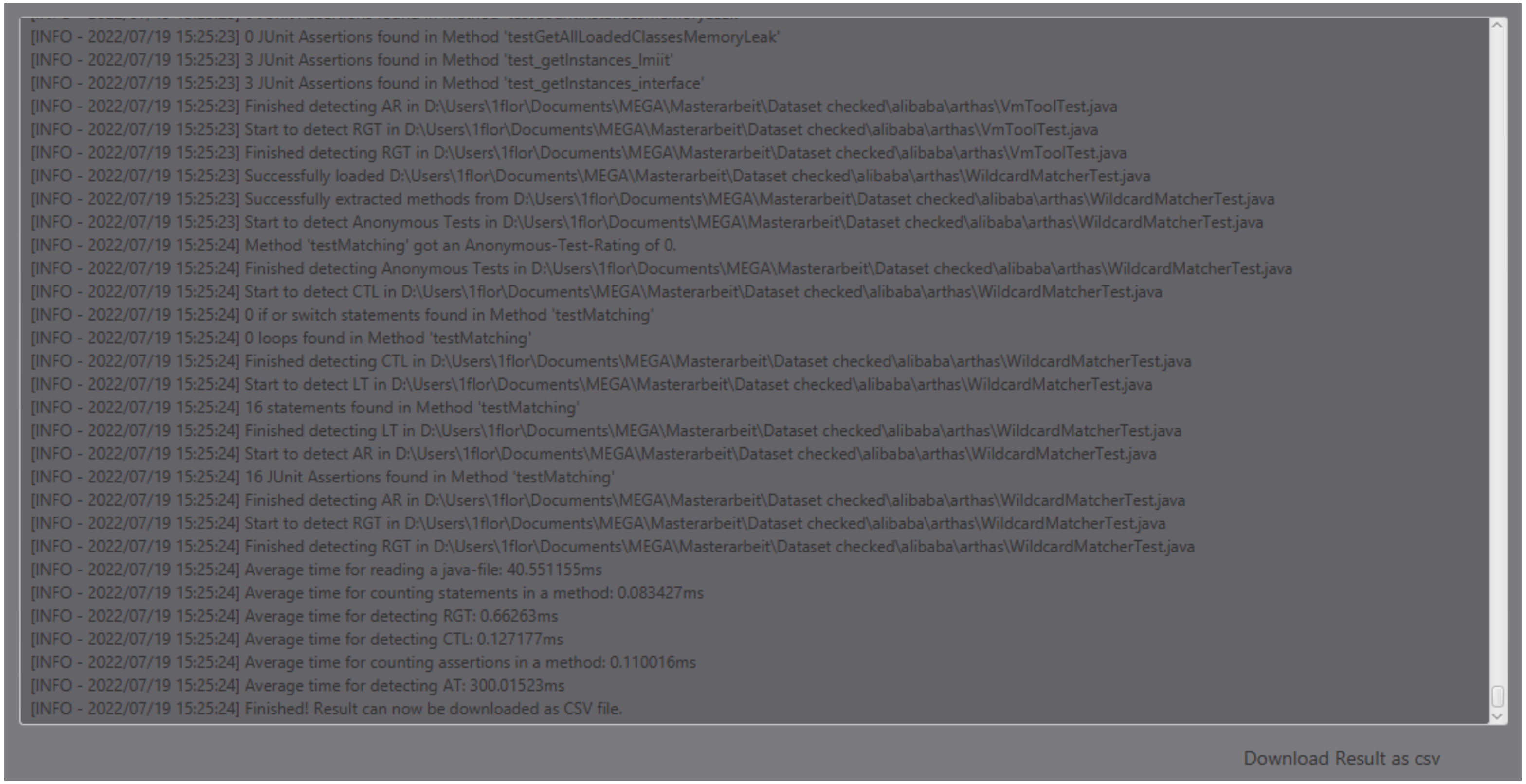
allow CTL including only print-statements Off

Rotten Green Test On

Assertion Roulette On

allowed number of assertions in one method 1

Figure 4.1: Configuration options offered by our tool



```
[INFO - 2022/07/19 15:25:23] 0 JUnit Assertions found in Method 'testGetAllLoadedClassesMemoryLeak'
[INFO - 2022/07/19 15:25:23] 3 JUnit Assertions found in Method 'test_getInstances_lmiit'
[INFO - 2022/07/19 15:25:23] 3 JUnit Assertions found in Method 'test_getInstances_interface'
[INFO - 2022/07/19 15:25:23] Finished detecting AR in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\VmToolTest.java
[INFO - 2022/07/19 15:25:23] Start to detect RGT in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\VmToolTest.java
[INFO - 2022/07/19 15:25:23] Finished detecting RGT in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\VmToolTest.java
[INFO - 2022/07/19 15:25:23] Successfully loaded D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\WildcardMatcherTest.java
[INFO - 2022/07/19 15:25:23] Successfully extracted methods from D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\WildcardMatcherTest.java
[INFO - 2022/07/19 15:25:23] Start to detect Anonymous Tests in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\WildcardMatcherTest.java
[INFO - 2022/07/19 15:25:24] Method 'testMatching' got an Anonymous-Test-Rating of 0.
[INFO - 2022/07/19 15:25:24] Finished detecting Anonymous Tests in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\WildcardMatcherTest.java
[INFO - 2022/07/19 15:25:24] Start to detect CTL in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\WildcardMatcherTest.java
[INFO - 2022/07/19 15:25:24] 0 if or switch statements found in Method 'testMatching'
[INFO - 2022/07/19 15:25:24] 0 loops found in Method 'testMatching'
[INFO - 2022/07/19 15:25:24] Finished detecting CTL in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\WildcardMatcherTest.java
[INFO - 2022/07/19 15:25:24] Start to detect LT in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\WildcardMatcherTest.java
[INFO - 2022/07/19 15:25:24] 16 statements found in Method 'testMatching'
[INFO - 2022/07/19 15:25:24] Finished detecting LT in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\WildcardMatcherTest.java
[INFO - 2022/07/19 15:25:24] Start to detect AR in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\WildcardMatcherTest.java
[INFO - 2022/07/19 15:25:24] 16 JUnit Assertions found in Method 'testMatching'
[INFO - 2022/07/19 15:25:24] Finished detecting AR in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\WildcardMatcherTest.java
[INFO - 2022/07/19 15:25:24] Start to detect RGT in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\WildcardMatcherTest.java
[INFO - 2022/07/19 15:25:24] Finished detecting RGT in D:\Users\1flor\Documents\MEGA\Masterarbeit\Dataset checked\alibaba\arthas\WildcardMatcherTest.java
[INFO - 2022/07/19 15:25:24] Average time for reading a java-file: 40.551155ms
[INFO - 2022/07/19 15:25:24] Average time for counting statements in a method: 0.083427ms
[INFO - 2022/07/19 15:25:24] Average time for detecting RGT: 0.66263ms
[INFO - 2022/07/19 15:25:24] Average time for detecting CTL: 0.127177ms
[INFO - 2022/07/19 15:25:24] Average time for counting assertions in a method: 0.110016ms
[INFO - 2022/07/19 15:25:24] Average time for detecting AT: 300.01523ms
[INFO - 2022/07/19 15:25:24] Finished! Result can now be downloaded as CSV file.
```

Figure 4.2: Information shown by our tool

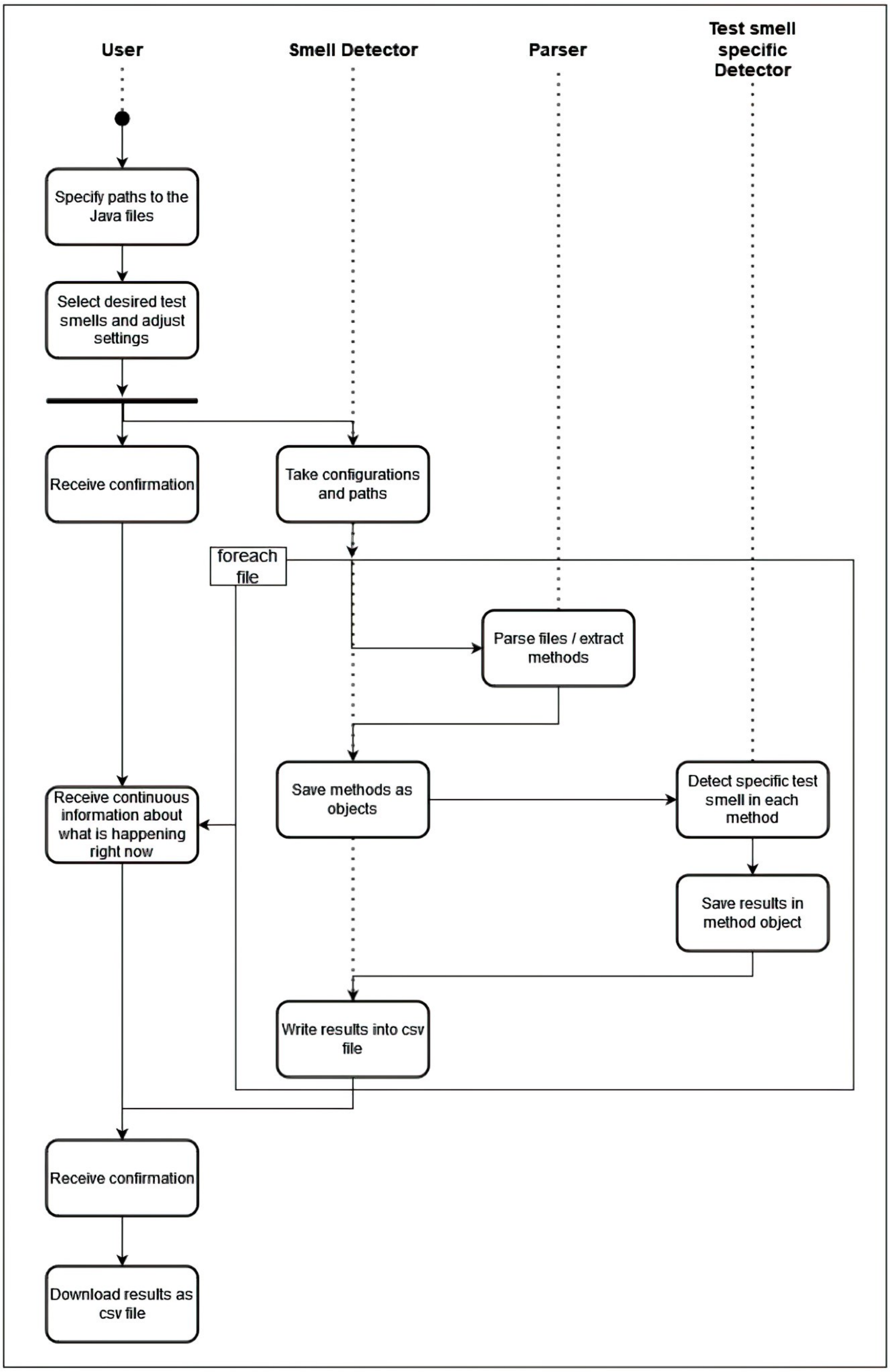

User
Smell Detector
Parser
Test smell specific Detector
Specify paths to the Java files
Select desired test smells and adjust settings
Receive confirmation
Take configurations and paths
foreach file
Parse files / extract methods
Save methods as objects
Detect specific test smell in each method
Receive continuous information about what is happening right now
Save results in method object
Write results into csv file
Receive confirmation
Download results as csv file

Figure 4.3: UML Activity Diagram of our tool

4.2 File Parsing And Method Extraction

Regardless of the settings of the tool, a Java file is always parsed at the beginning and the methods contained are then extracted. The exact order is as follows:

1. One of the files specified by the user is selected (in order).

2. All methods in this file are extracted and stored in the form of Java objects.

3. Depending on the settings, the methods are checked one after the other for the individual test smells.

The Java files are parsed in the classic way using Java IO libraries.
Java methods are recognised by means of regular expressions (regex). Other test smell detection tools such as TsDetect usually use an AST for this purpose. The reason for using regular expressions is the possible performance comparison with other tools.
The following regex was used to extract the method declarations in a Java file. If a method contains the annotation '@Test', this is included in the method declaration. This information is later used to distinguish between test methods and helper methods.

```
"((@Test)?((\\n)|(\\s)|(\\t))*(public|protected|private|static)
    +[\\w\\<\\>\\[\\]]+\\s+(\\w+) *\\([^\\)]*\\) *(\\{?|[^;]))"
```

To obtain the content of each method, the text between one and the next method declaration was used. Even though it is rather unlikely, lines of code that do not belong to the method could be included. To prevent this, the curly brackets are also counted, and thus only the code that is within the relevant brackets is considered to be the method body.
To extract the method name from the method declaration, the following regex was used:

```
"\s([a-zA-z\\d_]*)(?=[(])"
```

Each extracted test method is stored in a Java object 'Method', which contains, among other things, the information about the method declaration, method body, and method name. Later, the results of the test smell detection are also stored in this object.

Some other code parsing methods have also been created for later use:

- *getBodyOfOneLiners*: Some if-statements or loops are one-liner, i.e., they have no curved brackets. This method returns the body of such code.

- *getCodeBetweenCurvedBrackets*: This method returns the code between curved Brackets. This is necessary, for example, if you want to have the content of a for-loop.

- *removeStringsInsideQuotationMarks*: Eliminating strings within quotes ensures that searches for keywords do not have irrelevant results. The following regex is used. Applicable code passages are deleted.

```
"(\"(.*?)\")|('(.*?)')"
```

- *getTestMethods*: Returns all methods that have the annotation '@Test'.
- *getHelperMethods*: Returns all methods that do not have the annotation '@Test'.
- *getFileContentWithoutComments*: Used when parsing the Java files. Deletes all comments from the code. The following regex is used for one-line comments:

  ```
  "(?<=;|\s) *//.*"
  ```

  For multi-line comments (starting with '/*' and ending with '*/'), however, this regex is used:

  ```
  "((['\"])(?:(?!\\2|\\\\).|\\\\.)*\\2)|\\/\\/[^\\n]*|\\/\\*(?:[^*]
      |\\*(?!\\/))*\\*\\/"
  ```

- *removeUnwantedCharacters*: removes non-ASCII characters and other unwanted characters

4.3 Detection Algorithms

In the following, the individual test smell detection algorithms are discussed in detail.

4.3.1 Long Test

A long test occurs when a test method has reached a length that makes the test method less manageable and understandable. The measured variable is typically the number of statements.

A statement in Java is an instruction that is to be executed. According to Javapoint[1], a tutorial and information website for Java, there are three types of Java statements: Expression Statements, Declaration Statements, and Control Statements.
Expression statements are usually used to calculate or assign values. Declaration statements are those instructions that declare a new variable or constant, including data type and name. Control statements decide on the course of the program. These include conditional statements, loops, and jump statements (such as 'return' and 'break'). Below, we can see an example of the different statement types[2].

```
//declaration statement
int number;
//expression statement
```

[1] https://www.javatpoint.com/

[2] https://www.javatpoint.com/types-of-statements-in-java

```
number = 412;
//control flow statement
if (number > 10 )
{
//expression statement
System.out.println(number + " is greater than 100");
}
```

Declaration and expression statements usually end with a semicolon. Control flow statements that are at the beginning of a block, as is the case with an if-statement, for example, do not end with a semicolon.

Many detection tools, including tsDetect, use the Java AST to count statements. This contains every expression of a method in the form of a tree. By means of a visit method, each element of the tree is visited[3]. If an element is a Java statement, a counter is counted upwards.
Although using an AST is a quick way to count the statements, there is a simpler method.

Our tool counts the semicolons and certain keywords in a method. For this, the method body is used minus comments and strings within quotes.
To search for the keywords, we simply iterate over each word in the source code. The following keywords are searched for:

```
"catch", "do", "else", "finally", "for", "if", "switch", "try", "while"
```

To calculate the number of semicolons, all semicolons in the code were replaced by an empty string. The difference in the length of the code was then calculated.

```
int amountSemicolons = content.length() - content.replace(";", "").length();
```

Opinions differ on the question of when a test method is considered too long. TsDetect, for example, set a limit of 123 statements[4].
Spadini et al. conducted a study on severity thresholds of test smells [39]. They calculated three thresholds for the long (verbose) test. The lowest severity threshold is 13, which we have also set as the default value in our tool. The other two thresholds are 19 and 30 statements.

4.3.2 Conditional Test Logic

If a test method contains so-called control statements, the execution order of the method is influenced. This means that depending on a condition, the course of the execution is changed. For example, a part of the code is only executed under a certain condition and another part is executed several times. If this is the case with a test method, it is called

[3] https://www.programcreek.com/2011/07/java-count-number-of-statements-in-a-method/

[4] https://github.com/TestSmells/TestSmellDetector/blob/master/src/main/java/testsmell/smell/VerboseTest.java - accessed 20.07.2022

Conditional Test Logic.

Conditional Test Logic in test methods brings with it three problems:

- The complexity increases: it is more difficult to see what is happening in the method because you have to pay attention to certain conditions.
- The test method does not necessarily always produce the same result, but depends on a condition.
- Other test smells can be caused by this. For example, an assertion within a conditional block cannot be executed because the condition is not met. This causes the Rotten Green test smell.

Our tool recognises the conditional test logic smell using regular expressions. The following control statements are searched for in the method body:

- if / switch statements: Based on a condition, a code block is executed or not.
- while / for loops: A code block is not executed at all or is executed several times.

Even though this does not count as a control statement, we decided to also look for try-statements:

- try statement: An attempt is made to execute a code block. Even though no condition is explicitly specified, the attempt of a try-statement may fail. This means that the content of the try block is not executed.

It is noted that only try statements are searched for and not catch and finally statements. The reason for this is that the existence of a catch/finally statement is always preceded by a try-statement.

The following regex were used for the detection:

- if / switch:

```
    "(\n)(\s|\t)*((if)|(switch))(\s)*(?=[(])"
```

- while / for:

```
    "(\n)(\s|\t)*((while)|(for))(\s)*(?=[(])"
```

- try:

```
    "(\n)(\s|\t)*((try))(\s)*([(].*)?(?=[{])"
```

Our tool offers the user the possibility to allow conditional statements, which only contain print-statements. The reason for this option is that print-statements normally only have a logging purpose, so they do not change the way the method works. For this, each line of the conditional block is checked to see if it begins with the following string:

```
"system.out.print"
```

4.3.3 Assertion Roulette

If a test method contains several assertion statements, it is called an assertion roulette. The name comes from the fact that if the test method fails, one does not know which of the assertion statements is to blame.

```
@Test
public void testCalculator() {
    Assert.assertEquals(add(2,2), 4);
    Assert.assertEquals(add(2,-1), 1);
    Assert.assertEquals(add(-1,-1), -2);
    Assert.assertEquals(sub(2, 2), 0);
    Assert.assertEquals(mul(2, 4), 10);
    Assert.assertEquals(div(8, 2), 4);
}
```

For this reason, it is good practice to use only one assertion-statement per test method. Another way to prevent this smell is to add an assertion-message. With JUnit, you can add an explanation as an additional parameter in assertions. Among other things, this explanation should help you understand which assertion failed and for what reason.

However, our tool ignores this latter possibility and focuses only on the number of assert-statements. The reason for this is that the existence of an explanation message says nothing about whether it is also meaningful. Here is an example:
A test method contains ten assertions, each with an explanation message. However, these messages do not give any meaningful explanation, but only read ”this is the first test”, ”this is the second test”, and so on. Other test smell detection tools would not recognise an assertion roulette in this case. However, we are of the opinion that refactoring would clearly be necessary here. It is more important to us that a method is manually checked again to be on the safe side than that a smell is overlooked.

At this point, however, it should be noted that developers very rarely specify such an explanation message at assertions. In our dataset, which contains 1000 test methods, there was only one test method that had assertions with an explanation message.

Our tool uses the following regex for the detection of assertion-statements:

```
"((([a|A]ssert)(\\w*))|([f|F]ail))(\\s)*(?=[(])"
```

Each JUnit assert-statement begins with the string ”assert”. The only exception is the statement ”fail”. The use of this regex brings two potential problems with it:

1. Methods are falsely recognised as assertion statements because they follow the pattern.

2. Custom assertions are recognised as well.

We do not consider problem 2 to be critical. In fact, we see it as a good thing that custom assertions are also recognised. The first problem, on the other hand, is much more critical, because it can lead to false results. However, we believe that non-assert methods do not normally start with ”assert” or ”fail”. Therefore, we expect a very low error rate. And even if this leads to a wrong result, it is quickly detected by the refactoring.

In the assertion roulette smell, it is important that helper methods are also checked. For this reason, an iterative algorithm is used to detect this smell, which also checks each helper method and helper methods of helper methods.
In order to recognise helper methods, it is checked whether the method body contains method names of other methods. The search for helper methods is only carried out within the same Java file.

4.3.4 Rotten Green Test

If a test method contains assertions which are not executed (possibly under certain conditions), this is called a Rotten Green Test. This smell provides a false sense of security for the developer. If a rotten green test is executed, it may appear to be a successful test, even though some assertions have not been executed at all.

We distinguish between the following scenarios for this smell:

- An assert-statement is located within conditional test logic (e.g. if- and try-statements or loops). Here, the execution of the assert-statement depends on a condition. In our experience, this is the most common cause of a Rotten Green test.

```
@Test
public void testAddMethod() {
    if(someCondition)
        Assert.assertEquals(add(2,2), 4);
}
```

- An assert-statement is located after a return-statement. Sometimes a method is terminated early by a return-statement. If this is the case, subsequent assertions are no longer executed.

```java
@Test
public void testAddMethod() {
    // some code
    if(someCondition)
        return;
    Assert.assertEquals(add(2,2), 4);
}
```

- The test method itself is not rotten, but it includes helper methods which are rotten.

```java
@Test
public void testAddMethod(){
    Assert.assertEquals(add(2, 2), 4);
    someHelperMethod();
}

private void someHelperMethod(){
    if(someCondition)
        Assert.assertEquals(add(2, -1), 1);
}
```

- The test method calls a helper method (which contains assertions). However, this call is made within CTL.

```java
@Test
public void testAddMethod() {
    if(someCondition){
        someHelperMethod();
    }
    Assert.assertEquals(add(2, 2), 4);
}

private void someHelperMethod(){
    Assert.assertEquals(add(2, -1), 1);
}
```

- The test method does not include assert statements. Strictly speaking, this is a so-called smoke test and not a rotten green test. However, here too the test lights up as passed, although no assertions have been carried out. Therefore, we have decided to cover the smoke test here as well.

```java
@Test
public void testAddMethod() {
    int a = 2;
    int b = 3;
    System.out.println(a + b);
}
```

Our Rotten Green Test smell detection algorithm consists of four parts:

- Count all assertions that are within CTL. Helper methods are taken into account. For this purpose, regular expressions are used to search for CTL statements in the method body and their content is saved in a list. Then, with the help of a regex, assert-statements are searched for within these conditional statements.
- It is counted how many method names of helper methods, which contain assertions, occur within CTL.
- To check whether an assertion occurs after a return statement in the test method, the following regex is used. If there is at least one hit, this means that the test method is rotten.

```
"(?:return(\\s(\\w)+)?;)(.(?<!(assert|fail))|\\n|\\s|\\t)*
    (?:((([a|A]ssert)\\w*)|([f|F]ail))(\\s)*(?=[(]))"
```

- In the same way as for the detection of the assertion roulette smell, the assertions in the test method including helper methods are counted. This is for the detection of the smoke test.

Since we use a static analysis to detect the smell, we can only detect possible causes of the Rotten Green test. To determine whether the assertions are actually not executed, one would have to follow a dynamic tainting approach. However, we believe that our approach provides fairly accurate results.

4.3.5 Anonymous Test

Usually, there are always several developers working on a software project. The code written by another developer must be understood. Tests not only serve to validate the production code, but also offer a good opportunity to understand the production code. If you understand what a test checks, i.e. what the expected result of a method is, then it is easier to understand how the method works.
If a test method is well named, this ensures good readability and a good overview of what the test method does. In the best case, the method name of the test is sufficient and there is no need to look at the code.

If the name of a test method is poorly chosen and, for example, meaningless, it is called an anonymous test.

In scientific papers and grey literature, one can find many different suggestions for good test naming. We have selected five naming patterns that we think make the most sense and are used frequently:

- '*test*' + method name [4][13][40]: A simple pattern that makes it immediately clear which production method is being tested. This naming makes sense mainly when the method to be tested is simple. If several tests are necessary for the same production method, for example because it has to be tested with different conditions, it is often wiser to choose a more precise name.

```java
public class Calculator {

    public double add(double a, double b){
        return a + b;
    }

}

public class CalculatorTest {

    @Test
    public void testAdd() { // Pattern: 'test' + method name
        Calculator cal = new Calculator();
        Assert.assertEquals(cal.add(2,2), 4);
    }

}
```

- Method under test + state under test + expected behaviour (order can vary) [2][41]: This naming pattern describes relatively well what happens in the test method. The disadvantage here is that the test method name can quickly become quite long [4]. Example[5]:

```
isAdult_AgeLessThan18_False()
```

- Express a specific requirement [40]: Tests a method under a certain requirement. This requirement is then stated in the test method name. Can be linked well with other name patterns. Example[6]:

```
testIsNotAnAdultIfAgeLessThan18()
```

- Action + condition/state [4]: With this name pattern, a method or action can be tested under several conditions or states. This pattern is relatively simple and expressive. Examples[7]:

```
executeOnDisabledControl()
executeOnInvisibleControl()
```

- input/state/workflow + output [4][40]: This naming pattern describes exactly what is being done or what the input is and what the expected output should

[5] https://medium.com/@stefanovskyi/unit-test-naming-conventions-dd9208eadbea

[6] https://medium.com/@stefanovskyi/unit-test-naming-conventions-dd9208eadbea

[7] https://www.codeaffine.com/2014/03/17/getting-junit-test-names-right/

be. Since a lot of information is transmitted with this naming, the name can also quickly become very long. Example[8]:

```
Given_UserIsAuthenticated_When_InvalidAccountNumberIsUsedToWithdrawMoney_
    Then_TransactionsWillFail()
```

In addition, our tool takes into account an antipattern that is very often used in the naming of test methods. The test method is then simply called 'test'. Often a number is added at the end, e.g. 'test1', 'test2', etc.. Such naming is very impractical, as it does not give any information about the test method.

Figure 4.4 roughly describes the procedure of our tool for the detection of the Anonymous Test Smell. At the beginning, it is checked whether the method name of the test method follows the previously described antipattern. If this is the case, the algorithm stops immediately and the method name gets a bad rating. If this is not the case, the test method name is split by camel case and the words occurring in the name are stored

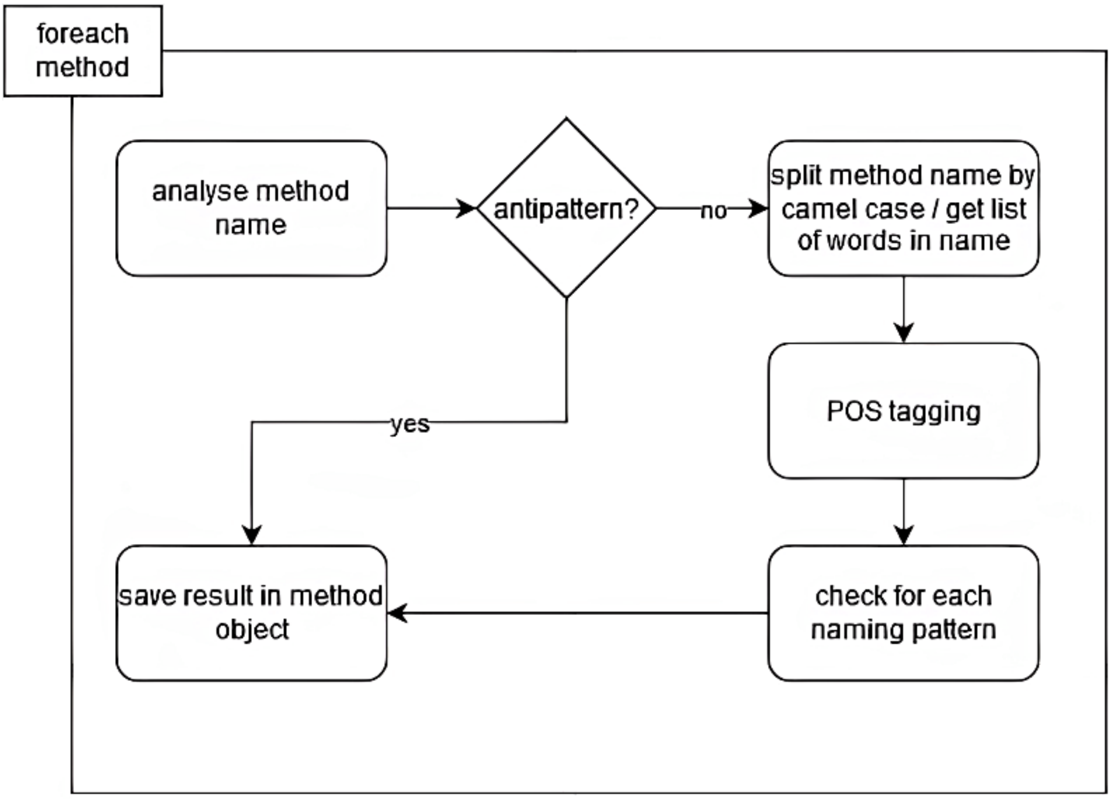

foreach method
analyse method name
antipattern?
no
split method name by camel case / get list of words in name
POS tagging
yes
check for each naming pattern
save result in method object

Figure 4.4: Procedure for the detection of the Anonymous Test smell

[8] https://medium.com/@stefanovskyi/unit-test-naming-conventions-dd9208eadbea

Role	POS tags
condition	IN, JJ, JJR, JJS, PDT, RB, RBR, RBS
half-condition	POS, PRP, PRP$
state	NN, NNS
object	NN, NNS, NNP, NNPS
result	MD
action	VB, VBG, VBD, VBN, VBP, VBZ

Table 4.1: Roles assigned to POS tags

in a list. Each word is then analysed using POS tagging. Depending on the user's configuration, the system then checks whether naming patterns apply. The result, i.e. whether the method name follows a certain pattern or not, is stored in the method object at the end.

The POS tags (for example, whether a word is a verb or a noun) are processed later. In the process, roles are assigned to some POS tags. For example, the POS tag JJ, which describes an adjective, is assigned the role "condition". These roles are later used for the detection of some naming patterns. In total, we have created 6 roles: condition, result, state, object, half-condition, and action. A half-condition are words that are not necessarily a condition, but should still be considered. An example of this are possessive pronouns (her, his, our, ...). Some POS tags are not assigned a role, because we consider them irrelevant for the detection of the naming patterns. These are then simply ignored during further processing. Examples of such POS tags are CD (cardinal digit), FW (foreign word), and UH (interjection). A list of all role assignments and POS tags are presented in table 4.1. Role assignment was based on discretion and trial and error.

A rating is assigned for each naming pattern. A rating of 0 indicates that the naming pattern matches the method name. A rating of 1 indicates some doubt: to some extent the naming pattern follows the pattern. A rating of 2 means that the pattern does not apply.

The following describes the process of checking for each naming pattern.

- '*test*' + method name: If the method name starts with 'test' and the second part of the method name is a method that is called in the method body, this naming pattern gets a rating of 0.

```
@Test
public void testSomeMethod(){ // begins with 'test', second part is a method
    that is called in the methodbody
    // some code
    someMethod();
    // some code
} // rating will be 0
```

If this is not the case, the system checks whether all words of the second part occur in the method body. If this is the case, a rating of 1 is assigned.

```java
@Test
public void testEmptyLongArray() { // begins with 'test' and words of the second part occur in the method body
    Assert.assertArrayEquals(ArrayUtils.EMPTY_LONG_ARRAY, new long[0]);
} // rating will be 1
```

If the test method name begins with a method name and ends with 'test', a rating of 1 is also assigned.

```java
@Test
public void someMethodTest(){ // begins with a method that is called in the methodbody, ends with 'test'
    // some code
    someMethod();
    // some code
} // rating will be 1
```

Otherwise, a rating of 2 is given.

```java
@Test
public void testSomeMethod1(){ // begins with 'test', second part is NOT a method that is called in the methodbody. Instead, there are three words: 'some', 'method', '1'. The first two words occur in the method body, but '1' does not.
    // some code
    someMethod();
    // some code
} // rating will be 2

@Test
public void testSomething(){ // begins with 'test', second part does not occur in method body
    // some code
    someMethod();
    // some code
} // rating will be 2
```

- Method under test + state under test + expected behaviour: Since this pattern requires that the name of a method being tested is in the test method name, it first checks if this is the case. If this is not the case, this pattern gets a rating of 2 and the algorithm terminates.

```java
@Test
public void ageLessThan18_False(){ // method 'isAdult' not in test method name
    Assert.assertFalse(isAdult(17));
} // rating will be 2
```

Otherwise, the different roles that appear in the test method name are counted. Based on the number of roles present, a rating is then assigned for this naming pattern. In addition, keywords are searched for in the test method name. These are: *input*, *output*, *result*, *in*, *out*, *true*, *false*. The calculations are as follows:

```
if((amountStates + amountObjects + amountConditions >= 2 && amountAction >= 1)
    || (!keywords.isEmpty() && amountStates + amountObjects +
    amountConditions >= 2) ||
   (amountResults >= 1 && amountStates + amountObjects + amountConditions >=
       2)) {
   rating = 0;
}
else if ((amountStates + amountObjects + amountConditions >= 1 && amountAction
    >= 1) || (!keywords.isEmpty() && amountStates + amountObjects +
    amountConditions >= 1) ||
   (amountResults >= 1 && amountStates + amountObjects + amountConditions >=
       1)) {
   if (rating >= 1) {
      rating = 1;
   }
}
else {
   if (rating >= 2) {
      rating = 2;
   }
}
```

An example is given below for better understanding:

```
isAdult_ageLessThan18_False()
/*'isAdult' is the method to be tested. The other words are: 'age', 'less',
    'than', '18', 'False'
'False' is a keyword
For each of the remaining words the POS tag is identified:
age: noun (NN), less: adjective comparative (JJR), than: preposition (IN), 18:
    number (CD)
So there are the following roles: 1x state, 2x condition
A rating of 0 is given.*/
```

- Express a specific requirement: Here, a similar algorithm is followed as for the previous pattern. The keywords here are: *if*, *unless*, *with*, *without*, *condition*, *requires*, *requirement*. If one of these keywords occurs, a rating of 0 is assigned. Otherwise, the occurring roles are looked at. If the 'condition' role occurs, a rating of 0 is assigned. If the role 'half-condition' occurs instead, a rating of 1 is assigned. Otherwise, a rating of 2 is given.

- Action + condition/state: The same algorithm is used here, but there are no keywords. Here it is looked whether the roles 'action', 'state', 'condition', and 'half-condition' occur. The rating is assigned as follows:

```
if(!hasAction){
    rating = 2;
}
else{
    if(hasCondition) {
        rating = 0;
    }
    else if (hasState) {
        rating = 0;
    }
    else if (hasHalfCondition){
        if(rating >= 1) {
            rating = 1;
        }
    }
}
```

- input/state/workflow + output: The same procedure is followed here as well. First, keywords are searched for. The keywords *input* and *in* indicate that the input is described in the method name. The keywords *output*, *out*, *result*, *results*, *produces*, *yields*, *return*, *returns*, *false*, and *true* indicate an output. The keywords *and*, *then*, *first*, *after*, and *before* could describe a workflow. We assume that the description of a workflow also contains a verb. The system then checks whether an action or a state occurs in the method name. The calculation of the rating runs as follows:

```
if(!hasOutput){
    rating = 2;
}
else{
    if(hasInput) {
        rating = 0;
    }
    if(hasAction && maybeWorkflow) {
        rating = 0;
    }
    if(hasState) {
        rating = 0;
    }
}
```

4.4 Output

After running our tool, the user can download the results as a .csv file. Depending on the previously selected settings, this file contains detailed results for each examined issue.

- Anonymous Test: Each naming pattern gets a rating of 0 (pattern is fulfilled), 1 (pattern is partially fulfilled / uncertain), or 2 (pattern is not fulfilled). The best rating will be applied.

- Conditional Test Logic: If a test method does not contain conditional statements, a rating of 0 is assigned. If 'if', 'switch' or 'try' statements occur, a rating of 1 is given. If loops occur, a rating of 2 is given. If both the former and the latter occur, a rating of 3 is assigned. In addition, the respective numbers are given.

- If a test method contains less or as many statements as specified in the settings, a rating of 0 is given. Otherwise, a rating of 1 is given. In addition, the number of statements is specified.

- If no assertions are present in a test method (i.e. if a smoke test occurs), a rating of -1 is assigned. If the number of assertions matches the limit specified in the settings, the rating is 0. If more assertions occur, a rating of 1 is assigned. In addition, the number of assert-statements that occur is given.

- Rotten Green Test: If no Rotten Green test is detected, a rating of 0 is assigned. In case of a smoke test, a rating of 1 is assigned. If assertions or helper methods are within CTL, a rating of 2 is assigned. This is also the case if an assertion occurs after a return statement. In this case, however, it is additionally indicated that the assertion occurs after a return.

5 Evaluation

This chapter focuses on the creation of the dataset, the process of manual evaluation, and the calculation of the accuracy of our tool. Furthermore, the performance of our tool is discussed.

5.1 Data Collection

A dataset was created with the primary use of measuring the accuracy of our test smell detection tool. It consists of the following parts:

- 854 JUnit test methods, which come from eight different GitHub repositories (see table 5.1 for the exact breakdown)
- 146 helper methods, which are located in the same files as the corresponding test methods
- One contents.txt file per GitHub repository, listing the link to the repository and all Java files that contain JUnit methods
- One checklist.csv file per GitHub repository, which contains the manual evaluation of the test methods

The dataset was created using a python script. It searched for open source GitHub repositories with the keyword *java* and various licences. The exact search query is as follows:

```
+java+in:readme+in:description+license:mit+license:bsd-3-clause-clear+license:bsd-3-
    clause+license:gpl-3.0+license:cc-by-sa-4.0
```

The results were sorted by stars (descending). Each repository found (until a sufficient amount of data was reached) was then further analysed. There were two criteria for the repository to be included in the dataset:

1. The repository contains files ending with *test.java*. We have assumed that Java test files end with *Test.java*, which is an often used naming pattern.
2. If such files exist, the files are searched for two keywords: *assert* and *junit.*

Subsequently, all repositories found by the script were checked again manually to see if they met the criteria. The dataset contains only JUnit test files. All other files that were in the repositories (e.g. production code files) were ignored.

Once the files were collected, work began on manually checking all the methods they contained for test smells.
It should be noted that not only the JUnit test methods were analysed, but also the helper methods. Our tool, however, focuses only on the test methods. For example, our tool does not check the names of helper methods.
It should also be noted that the eager and lazy test smells were also searched for manually. At the moment, our tool does not include the possibility to detect these smells.

The checklist.csv files, which contain the manual evaluation, include the following information:

- repository (e.g., *alibaba/arthas*)
- filename (e.g., *VmToolTest*)
- methodname (e.g., *testIsSnapshot*)
- rating for the anonymous test smell:
  - 0 = follows a guideline
  - 1 = unsure if the method name follows a guideline
  - 2 = follows no guidelines
- amount of statements (long test)
- rating for the conditional test logic smell:
  - 0 = contains no conditional test logic
  - 1 = contains if-statement (including switch and try/catch)
  - 2 = contains loop
  - 3 = contains if-statement and loop
- rating for the rotten green test smell:
  - 0 = all assertions should be executed
  - 1 = there are no assertions (including helper methods) (= smoke test)
  - 2 = assertion inside conditional test logic (including helper methods) or after return statement
- rating for the assertion roulette smell, including the amount of assertions. The information is in the following format: x/y, where x is the rating for the smell and y is the number of assertions.
  - 0 = only one assert-statement
  - 1 = more than one assert-statement, none have an explanation message

  - 2 = more than one assert-statement, some have an explanation message
  - 3 = more than one assert-statement, all have an explanation message

- rating for the eager test smell
  - 0 = calls only one method of the object to be tested
  - x = calls x methods of the object to be tested
- rating for the lazy test smell
  - 0 = only one test method tests a certain production method using the same fixture
  - 1 = multiple test methods test a certain production method using the same fixture

After everything was evaluated manually to the best of our knowledge, the test methods were evaluated using our tool. The results were then compared. If the results differed, we checked where the problem was. It was often recognised that the manual evaluation was flawed. This was then corrected. Due to this procedure, we are convinced that the manual evaluation no longer contains mistakes. One exception is the review of the Anonymous test. It was sometimes quite difficult to check some names on the basis of a pattern. But we believe that the error rate should be relatively low. Another exception is the eager and lazy test. Since we cannot check our manual evaluation with the tool, there is a possibility of mistakes.

5.2 Accuracy Evaluation

To measure the accuracy of our tool, the results of the tool were compared with those of the manual evaluation. Table 5.1 shows the results.
In the long test smell and assertion roulette smell, not the ratings but the number of statements or assertions were compared.
In the Anonymous test smell (AT), the results were considered equal if both ratings were either 0 or 1, or if both were 2. For example, if we gave a test method name a rating of 1 while the tool gave a rating of 0, we still let that pass as a match.

Our tool achieves an accuracy of about 92% to 100%. The detection of the Anonymous test smell is the most error-prone. The detection of the number of statements in a method and the detection of conditional test logic achieved an accuracy of 100% in each repository.

In the Discussion chapter, possible causes that led to an accuracy below 100% are discussed.

In addition to the measurements for the accuracies, we also measured the precision (see table 5.2), recall (see table 5.3) and f1-score (see table 5.4).

Accuracy Results

Accuracy measures how often numbers have been calculated correctly. TP here stands for True Positive, TN stands for True Negative, FP for False Positive and FN for False Negative [12].

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

source	#testMethods	AT	#statements	CTL	#assertions	RGT
github.com/alibaba/arthas.git	147	0.850	1.0	1.0	1.0	1.0
github.com/crossoverJie/JCSprout.git	24	0.917	1.0	1.0	1.0	1.0
github.com/dianping/cat.git	67	0.910	1.0	1.0	0.940	1.0
github.com/dromara/Sa-Token.git	68	1.0	1.0	1.0	1.0	1.0
github.com/redis/jedis.git	166	0.928	1.0	1.0	1.0	1.0
github.com/TheAlgorithms/Java.git	104	0.981	1.0	1.0	1.0	1.0
github.com/vipshop/vjtools.git	228	0.947	1.0	1.0	1.0	1.0
github.com/zxing/zxing.git	50	0.860	1.0	1.0	0.980	0.980
	854	0.924	1.0	1.0	0.990	0.998

Table 5.1: Accuracy of our Detection Tool

Precision Results

The precision tells what proportion of positive identifications were actually correct [12].

$$Precision = \frac{TP}{TP + FP} \tag{5.2}$$

source	#testMethods	AT	LT	CTL	AR	RGT
github.com/alibaba/arthas.git	147	0.653	1.0	1.0	1.0	1.0
github.com/crossoverJie/JCSprout.git	24	0.875	1.0	1.0	1.0	1.0
github.com/dianping/cat.git	67	0.840	1.0	1.0	0.971	1.0
github.com/dromara/Sa-Token.git	68	1.0	1.0	1.0	1.0	1.0
github.com/redis/jedis.git	166	0.861	1.0	1.0	1.0	1.0
github.com/TheAlgorithms/Java.git	104	0.875	1.0	1.0	1.0	1.0
github.com/vipshop/vjtools.git	228	0.932	1.0	1.0	1.0	1.0
github.com/zxing/zxing.git	50	1.0	1.0	1.0	1.0	0.800
	854	0.879	1.0	1.0	0.996	0.975

Table 5.2: Precision of our Detection Tool

Recall Results

The recall tells what proportion of actual positives were correctly identified [12].

$$Recall = \frac{TP}{TP + FN} \tag{5.3}$$

source	#testMethods	AT	LT	CTL	AR	RGT
github.com/alibaba/arthas.git	147	0.865	1.0	1.0	1.0	1.0
github.com/crossoverJie/JCSprout.git	24	1.0	1.0	1.0	1.0	1.0
github.com/dianping/cat.git	67	0.913	1.0	1.0	1.0	1.0
github.com/dromara/Sa-Token.git	68	1.0	1.0	1.0	1.0	1.0
github.com/redis/jedis.git	166	0.949	1.0	1.0	1.0	1.0
github.com/TheAlgorithms/Java.git	104	1.0	1.0	1.0	1.0	1.0
github.com/vipshop/vjtools.git	228	0.976	1.0	1.0	1.0	1.0
github.com/zxing/zxing.git	50	0.708	1.0	1.0	0.961	1.0
	854	0.926	1.0	1.0	0.995	1.0

Table 5.3: Recall of our Detection Tool

F1-score Results

The F1-score describes the harmonic mean of precision and recall [48].

$$F1 = \frac{2 * (Precision * Recall)}{(Precision + Recall)} \tag{5.4}$$

source	#testMethods	AT	LT	CTL	AR	RGT
github.com/alibaba/arthas.git	147	0.744	1.0	1.0	1.0	1.0
github.com/crossoverJie/JCSprout.git	24	0.933	1.0	1.0	1.0	1.0
github.com/dianping/cat.git	67	0.875	1.0	1.0	0.986	1.0
github.com/dromara/Sa-Token.git	68	1.0	1.0	1.0	1.0	1.0
github.com/redis/jedis.git	166	0.903	1.0	1.0	1.0	1.0
github.com/TheAlgorithms/Java.git	104	0.933	1.0	1.0	1.0	1.0
github.com/vipshop/vjtools.git	228	0.953	1.0	1.0	1.0	1.0
github.com/zxing/zxing.git	50	0.829	1.0	1.0	0.980	0.889
	854	0.896	1.0	1.0	0.996	0.986

Table 5.4: F1-score of our Detection Tool

5.3 Performance Evaluation

In addition to measuring the accuracy of our tool, performance was also monitored. The time needed to detect a smell in a test method was recorded. The time taken to parse the files was also captured. The performance test was carried out on a laptop with Intel(R) Core(TM) i7-6700HQ processor and 16GB RAM.

Table 5.5 shows the average time required for one test method. The detection of the smells usually takes less than 0.1ms. The only exception here is the anonymous test. Here it takes about 0.3 seconds to analyse a test method. The reason for this is probably the use of part-of-speech tagging. The parsing of the files (which includes the recognition of the test and helper methods) took about 75ms per file.

source	#testMethods	parsing	AT	#statements	CTL	#assertions	RGT
github.com/alibaba/arthas.git	147	55.866	311.197	0.097	0.079	0.100	0.376
github.com/crossoverJie/JCSprout.git	24	69.186	351.372	0.195	0.181	0.154	0.700
github.com/dianping/cat.git	67	51.855	295.072	0.129	0.097	0.651	0.810
github.com/dromara/Sa-Token.git	68	93.073	298.535	0.110	0.087	0.363	0.512
github.com/redis/jedis.git	166	92.329	326.199	0.082	0.072	0.250	0.369
github.com/TheAlgorithms/Java.git	104	77.444	329.794	0.076	0.066	0.200	0.313
github.com/vipshop/vjtools.git	228	75.204	331.871	0.072	0.102	0.164	0.391
github.com/zxing/zxing.git	50	84.946	330.786	0.070	0.098	0.162	0.422
	854	74.987	321.853	0.103	0.097	0.255	0.486

Table 5.5: Average execution time for one test method (in ms)

6 Discussion

In this chapter, assumptions are made about how the results of the accuracy evaluation came about. Furthermore, comparisons are made with other test smell detection tools. We discuss to what extent the results presented in chapter 5 reflect the quality of our tool. Finally, the advantages and disadvantages of our tool are highlighted.

6.1 Discussion About Accuracy

In the following, assumptions are made as to why 100% accuracy could not be achieved in some cases. Cases where 100% accuracy was achieved are not discussed.

Anonymous Test

Whether a test method corresponds to one of the given patterns is often a difficult question. While a programme that strictly follows an algorithm will define a method name as bad, a human who understands the meaning behind the name would still let the name pass.
An example of this is the following:

```
@Test
public void testIntArray() {
    int[] data = {1,3,4,5};
    ObjectView objectView = new ObjectView(data, 3);
    String expected = "@int[][\n" +
           "   @Integer[1],\n" +
           "   @Integer[3],\n" +
           "   @Integer[4],\n" +
           "   @Integer[5],\n" +
           "]";
    Assert.assertEquals(expected, objectView.draw());
}
```

In the manual evaluation, this method was given a rating of 1. Strictly speaking, this method does not comply with the pattern *"test" + method name*, but we know that an array in Java is represented with two square brackets and *Int* is an abbreviation for *Integer*. Our tool, however, does not have this information.

Assertion Roulette

The reason for error-proneness when counting assertions are several helper methods with the same name but different parameters:

```java
private void someHelperMethod(int value){
    Assert.assertEquals(value, someMethod.getValue());
}

private void someHelperMethod(String content){
    Assert.assertEquals(content, "true");
    Assert.assertTrue(content.length() > 2);
}

@Test
public void testSomeMethod(){
    // some code
    someHelperMethod(2);
    someHelperMethod("test");
}
```

Our tool does not check what types the parameters are. So, even though this happened very rarely in our test, it can happen that a wrong helper method is analysed and thus a wrong number of assertions is obtained. So in the example above, only two assertions could be counted: the *someHelperMethod* method was called twice, and our tool thinks this method contains only one assertion. In reality, however, the *testSomeMethod* method contains three assert-statements.

Rotten Green Test

Our detection algorithm failed only on one method, concerning the Rotten Green test. The reason was similar to the one for assertion roulette: a helper method with the same name as the test method contained all assert-statements, while the test method contained none. Since both methods had the same name, this was not recognised. The test method was thus recognised as a smoke test.

```java
@Test
public void someMethod(){
    someMethod(10);
    someMethod(15);
}

private void someMethod(int value){
    Assert.assertTrue(value >= 10);
}
```

6.2 Comparison With Other Tools

Our tool joins a list of existing test smell detection tools.
In this following, we compare our tool with *tsDetect*, a well-known test smell detection tool that uses a rule-based approach [33]. Since only the test smells *Conditional Test Logic* and *Assertion Roulette* are covered in both tools, the comparison is limited to

source	CTL			AR		
	precision	recall	f-score	precision	recall	f-score
Our tool	1.0	1.0	1.0	0.996	0.995	0.996
tsDetect	1.0	1.0	1.0	0.947	0.900	0.923

Table 6.1: Comparison with tsDetect

these two smells. As can be seen in Table 6.1, both tools have 100% exactness for the *CTL* smell. For the *AR* smell, our tool is minimally more accurate. It should be noted, however, that we are referring here to the values provided by tsDetect. The tools were not evaluated on the same dataset.

The test smells *Long Test* and *Rotten Green Test* are detected by other tools, but there is no information on the accuracies. Therefore, a comparison is not possible.
We have not found any other tool that includes the recognition of the *Anonymous Test.* Therefore, we do not have a comparison here, but we believe that we have laid a good foundation with an accuracy of over 92%.

6.3 Meaningfulness Of Our Evaluation

In chapter 5 we presented the results of the evaluation of our tool. The tool showed high accuracy in detecting the different test smells. The question now is how meaningful these measurements are and how well they measure the quality of our tool.

The evaluation results were obtained by comparing the results of the tool and the dataset previously created by hand. Consequently, the validity of the evaluation results is related to the accuracy of the dataset.
We followed a 2-step plan in creating the dataset: first, the JUnit tests were evaluated to the best of our knowledge. However, it was to be expected from the beginning that careless errors and other inaccuracies could occur. The second step was therefore to compare the results of the tool with the dataset and correct errors in the dataset. This step should not be misunderstood: the results of the tool were not simply taken over. If the tool had a different result than the dataset, the respective JUnit test method was checked again. Due to this 2-step plan, we assume that the dataset is very accurate and thus the evaluation results are also correct. However, special attention should be paid to the anonymous test. Here the evaluation was particularly difficult and in some cases had to be decided subjectively.

One could also question the quality of the test data used (i.e. the JUnit test methods) and claim that they are not representative enough to determine the accuracy of the tool.
We used a total of 854 test methods from eight active and highly rated github repositories. These have different subject areas and one can see a clear difference in the

programming of the test methods. In some cases, non-Latin characters were also used. We therefore consider this to be a good representation of JUnit tests.

6.4 Advantages Of Our Tool

Our open source test smell detection tool is available to users via a standalone Java jar file, which provides a clear GUI when executed. Because of this, users are not dependent on other tools (such as a specific IDE or command line). However, there are plans for integration as a plugin for various IDEs as well as for use via command line.

Only JUnit test files are required as input. Since no production code has to be processed, this increases the speed of detection of test smells. Incorrectly specified files (those that do not contain JUnit tests) are simply ignored by the tool.
However, because we only work with test files, there is also the possibility of a lower accuracy in the detection of test smells. For example, with our approach we can only detect symptoms for the Rotten Green test. When we created our dataset, we also focused only on the symptoms of the test smells. There is still a possibility that an assertion will not be performed due to another factor. If we took the production code into account or applied algorithms based on a dynamic tainting procedure, we could analyse the Rotten Green test more precisely.

After analysing the test files, our tool provides a csv file with detailed results. Most of the results are presented as numbers. This enables the user to process the results more easily.

7 Conclusion

In the course of this master thesis, a test smell detection tool was created that detects the test smells *Anonymous Test*, *Long Test*, *Conditional Test Logic*, *Assertion Roulette* and *Rotten Green Test*. This makes our tool the first to detect the *Anonymous Test* smell.
The tool is provided open source and offers a graphical user interface as well as comprehensive documentation, which makes the tool easy to use. Additionally, the possibility to partially customize the detection algorithms is provided.
Our tool focuses on JUnit tests and only requires JUnit test files as input and no production code. The tool uses a rules/heuristic procedure to detect the test smells, whereby natural language processing methods are also used. We aimed to use methods for the detection of test smells that other tools do not use.

Furthermore, a dataset was created that contains 854 JUnit test methods and 146 helper methods. In addition, this dataset was manually checked for seven test smells. This dataset will be made open source together with the manual evaluation of the data.

Using our dataset, the accuracy, precision, recall, and F1-score of our tool was measured. Our tool has a very high average accuracy in the detection of test smells, which makes it in no way inferior to other test smell detection tools.
In addition to measuring accuracy, we also measured the performance of our tool. The test smells are measured at a speed at which even large software projects can be checked for smells in a reasonable time.

The tool offers several places for expansion in future projects. Besides the addition of algorithms for the detection of other test smells, we consider it useful to include the use of the tool via command line and an integration into IDEs. Furthermore, the algorithms for the detection of the anonymous test smell offer room for improvement.

In summary, this master thesis contributes the following:

- A new Test Smell Detection Tool for JUnit tests, covering five test smells
- New approaches to detecting test smells
- The first tool that addresses the *Anonymous Test* smell
- An extensive dataset, which was manually checked for seven test smells

Bibliography

[1] Junit website. https://junit.org. Accessed: 2022-07-08.

[2] Unit testing best practices with .net core and .net standard. https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices. Accessed: 2022-08-02.

[3] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180, 2021.

[4] F. Appel. Getting junit test names right. https://www.codeaffine.com/2014/03/17/getting-junit-test-names-right/. Accessed: 2022-08-02.

[5] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40(11):1100–1125, 2014.

[6] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[7] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 56–65. IEEE, 2012.

[8] Kent Beck, Martin Fowler, and Grandma Beck. Bad smells in code. *Refactoring: Improving the design of existing code*, 1(1999):75–88, 1999.

[9] Manuel Breugelmans and Bart Van Rompaey. Testq: Exploring structural and maintenance characteristics of unit test suites. In *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. Citeseer, 2008.

[10] Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007.

[11] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[12] S. Dagnechaw. What is accuracy, precision, and recall? and why are they important? https://shiffdag.medium.com/what-is-accuracy-precision-and-recall-and-why-are-they-important-ebfcb5a10df2. Accessed: 2022-10-22.

[13] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 57–67, 2017.

[14] Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew P Black, and Anne Etien. Rotten green tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 500–511. IEEE, 2019.

[15] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5–1, 2012.

[16] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[17] Gordon Fraser, Alessio Gambi, and José Miguel Rojas. Teaching software testing with the code defenders testing game: Experiences and improvements. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 461–464. IEEE, 2020.

[18] Vahid Garousi and Michael Felderer. Developing, verifying, and maintaining high-quality automated test scripts. *IEEE Software*, 33(3):68–75, 2016.

[19] Vahid Garousi and Barış Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software*, 138:52–81, 2018.

[20] Vahid Garousi, Baris Kucuk, and Michael Felderer. What we know about smells in software test code. *IEEE Software*, 36(3):61–73, 2018.

[21] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 223–233, 2015.

[22] Nildo Silva Junior, Larissa Rocha, Luana Almeida Martins, and Ivan Machado. A survey on test practitioners' awareness of test smells. *arXiv preprint arXiv:2003.05613*, 2020.

[23] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84. IEEE, 2009.

[24] Dong Jae Kim. An empirical study on the evolution of test smell. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 149–151. IEEE, 2020.

[25] Dong Jae Kim, Tse-Hsun Peter Chen, and Jinqiu Yang. The secret life of test smells-an empirical study on test smell evolution and maintenance. *Empirical Software Engineering*, 26(5):1–47, 2021.

[26] Panagiotis Louridas. Junit: unit testing and coiling in tandem. *IEEE Software*, 22(4):12–15, 2005.

[27] Mika V Mantyla, Jari Vanhanen, and Casper Lassenius. Bad smells-humans as code critics. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 399–408. IEEE, 2004.

[28] Matias Martinez, Anne Etien, Stéphane Ducasse, and Christopher Fuhrman. Rtj: a java framework for detecting and refactoring rotten green test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 69–72, 2020.

[29] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[30] Michael Olan. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges*, 19(2):319–328, 2003.

[31] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. Automatic test smell detection using information retrieval techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 311–322. IEEE, 2018.

[32] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 523–533. IEEE, 2020.

[33] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1650–1654, 2020.

[34] Anthony Peruma, Khalid Saeed Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. 2019.

[35] Anthony Peruma, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. An exploratory study on the refactoring of unit test files in android applications. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 350–357, 2020.

[36] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based assessment of test quality. *J. Object Technol.*, 6(9):231–251, 2007.

[37] Martin Schvarcbacher, Davide Spadini, Magiel Bruntink, Ana Oprescu, et al. Investigating developer perception on test smells using better code hub-work in progress. In *2019 Seminar Series on Advanced Techniques and Tools for Software Evolution, SATTOSE*, volume 2019, 2019.

[38] Elvys Soares, Márcio Ribeiro, Guilherme Amaral, Rohit Gheyi, Leo Fernandes, Alessandro Garcia, Baldoino Fonseca, and André Santos. Refactoring test smells: A perspective from open-source developers. In *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*, pages 50–59, 2020.

[39] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. Investigating severity thresholds for test smells. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 311–321, 2020.

[40] O. Stefanovskyi. Unit test naming conventions. `https://medium.com/@stefanovskyi/unit-test-naming-conventions-dd9208eadbea`. Accessed: 2022-08-02.

[41] A. Trenk. Testing on the toilet: Writing descriptive test names. `https://testing.googleblog.com/2014/10/testing-on-toilet-writing-descriptive.html`. Accessed: 2022-08-02.

[42] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating accurate assert statements for unit test cases using pretrained transformers. In *2022 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 54–64. IEEE, 2022.

[43] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95. Citeseer, 2001.

[44] Bart Van Rompaey, Bart Du Bois, and Serge Demeyer. Characterizing the relative significance of a test smell. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 391–400. IEEE, 2006.

[45] Shuang Wang and Jeff Offutt. Comparison of unit-level automated test generation tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 210–219. IEEE, 2009.

[46] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 306–315. IEEE, 2012.

[47] Vahid Garousi Yusifoğlu, Yasaman Amannejad, and Aysu Betin Can. Software test-code engineering: A systematic mapping. *Information and Software Technology*, 58:123–147, 2015.

[48] Zach. What is accuracy, precision, and recall? and why are they important? `https://www.statology.org/f1-score-vs-accuracy/`. Accessed: 2022-10-22.