University of Brasília

Institute of Exact Sciences
Department of Computer Sciences

TestAXE: Detecting and Refactoring Test Smells in Test Suites That Use JUnit

Estevan Alexander de Paula

Monography presented as a partial requirement for
the conclusion of the Computer Engineering Course

Supervisor
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2022

University of Brasília

Institute of Exact Sciences
Department of Computer Sciences

TestAXE: Detecting and Refactoring Test Smells in Test Suites That Use JUnit

Estevan Alexander de Paula

Monography presented as a partial requirement for
the conclusion of the Computer Engineering Course

Prof. Dr. Rodrigo Bonifácio de Almeida (Supervisor)
CIC/UnB

Prof.a Dr.a Edna Dias Canedo
CIC/UnB

Walter Lucas Monteiro de Mendonça
PPGI/UnB

Prof. Dr. João Luiz Azevedo de Carvalho
Coordinator of the Computer Engineering Course

Brasília, 27th October 2022

Dedication

I dedicate this work to my wife, Giuliana, and my parents, Veronice and Renato, for helping me keep focused and motivated to pursue the conclusion of this journey until the end.

Acknowledgements

I acknowledge my supervisor, Prof. Dr. Rodrigo Bonifácio de Almeida for all the guidance and orientation. I acknowledge my wife and parents, for giving me the strength I needed when the lack of energy and time seemed too hard to manage, and my parents again for providing me with the material conditions and mental attitude that enabled me to accomplish what I have so far.

This work was conducted with support from the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES)*, by providing access to the *Portal de Periódicos*.

Abstract

The presence of *test smells* in software project have been proven to deteriorate not only its test suite's maintainability, but also to hide problems in the software's implementation. Based on the work conducted by Soares et Al, this essay presents TestAXE, a tool capable of refactoring *test smells* by using features released on the latest version of the Java testing framework, JUnit 5. By using software reengineering concepts, the tool was developed in Rascal, a language targeted to the development of metaprograms. To assess the efficiency and to understand the limitations of TestAXE, an empirical study has been conducted using a series of *Pull Requests* made by Soares et al.

Keywords: Software Reenginering, Software Testing, Software Refactoring, Metaprogramming, Code quality

Resumo

A presença de *Test Smells* em projetos de software causa a degradação da qualidade tanto da *suite* de testes desses projetos de software, quanto pode mascarar problemas presentes na implementação desses projetos. Tomando como base o trabalho desenvolvido por Soares et Al, este trabalho consiste em apresentar TestAXE, uma ferramenta capaz de aplicar refatorações em testes que contém *test smells* utilizando *features* apresentadas na última versão da bibloteca de teste para software Java, JUnit 5. A ferramenta utiliza conceitos de re-engenharia de software e foi desenvolvida em Rascal, uma linguagem voltada para a produção de meta-programas. Utilizando como referência uma série de *Pull Requests* feitos por Soares et Al, um estudo empírico foi conduzido para analisar e entender a eficiência e as limitações dessa ferramenta.

Palavras-chave: Re-engenharia de software, Testes de software, Refatoração de software, Metaprogramação, Qualidade de Código

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The development of software tests is one of the main activities of a software developer. Guaranteeing stability, correctness and code quality are usually the main concerns when developing a piece of software, aspects which can be verified, to varying degress, by automated tests. Although, as it also happens with software code development, the development of test code can suffer from suboptimal design and implementation, leading to the creation of test smells.

Test smells have been proven to deteriorate the quality of software code [1], while also affecting the maintainability of a software's test suite [2]. Nonetheless, test smells have been extensively studied in the last 20 years, in such a way that they're catalogued and several different solution have been proposed [3].

By detecting the code patterns of a smell, a piece of software may be able not only of detecting these test smells, but also of building refactored code that avoids these problems. This kind of software has been developed with different scopes and solution proposals, such as just detecting and leaving the task of actually refactoring the code to the software's maintainer, or detecting and programatically refactoring the problematic code [4, 5, 6, 7, 8, 9].

In order to solve a specific set of smells using features from the latest version of the JUnit testing framework[1], based on the work of Soares et al. [3], TestAXE was developed. TestAXE is a test smell refactoring tool built using Rascal MPL [10], a meta-programming language that allows for the easy creation and traversal of a *Parse Tree* from the definition of a grammar. TestAXE also has a thin python driver that receives CLI parameters and starts the execution of the Rascal metraprogram, responsible for detecting and refactoring smells inside a local *git* repository.

These transformations, as they will be explained in more detail later, are classified as *endogenous transformations* [11], since it is the transformation of java code to java code,

[1] `https://junit.org/junit5/`

and are also *horizontal* [11], since both the input and the output of the transformation functions are java test classes and test methods, maintaining both the source language and abstraction level. Ultimately, TestAXE is a translator software [12] that enfocus the refactoring of test smells in Java test suites by migrating source code that, although using the latest version of JUnit, is not taking advantage of its newest features.

1.1 Published Paper

With the efforts of building TestAXE done, a paper has been written, co-authored by Prof. Dr. Rodrigo Bonifácio de Almeida, and submitted to *CTIC-ES 3nd Undergraduate Research on Software Engineering Competition*[2]. This monography presents this paper and complements it, given the size limitation of its format. The paper is presented in full on chapter 2, while the design and implementation details of TestAXE and its transformations are discussed in more detail on chapter 3, as well as the smells that each one of them is capable of refactoring. Finally, chapter 4 presents an analysis of the tool's efficiency in solving test smells, as well of its limitations, and chapter 5 a wrap up of the conducted work.

[2] `https://cbsoft2022.facom.ufu.br/sbes-ctic.php`

Chapter 2

The Submitted Paper

As stated on chapter 1, a paper presenting TestAXE was published on *CTIC-ES 3nd Undergraduate Research on Software Engineering Competition*[1]. The paper starts by delivering a short discussion on the current state of metaprograms that transform, or simply software that detect, test smells. Then TestAXE is presented, with an overview of its design principles and a description of the transformation pipeline, briefly referencing the tools Rascal makes availabe to do so. Towards the end, an analysis of the software's efficiency in refactoring these smells based on the same dataset used by Soares et al[2] is presented, as well as a short discussion on the tool's limitations by choice or by lack of implementation.

The full paper is reproduced below. It has been selected as one of the three best papers submitted to *CTIC-ES*. The next chapters will present more comprehensive explanation of how TestAXE and its transformations work and how the empirical study was conducted.

[1] `https://cbsoft2022.facom.ufu.br/sbes-ctic.php`

[2] `https://github.com/easy-software-ufal/refactoring-test-smells-with-junit5`

TestAXE: Automatically Refactoring Test Smells Using JUnit 5 Features

Estevan Alexander de Paula[1], **Rodrigo Bonifácio**[1]

[1]Computer Science Department
University of Brasília (UnB)
Brasília – Brazil

estevan.paula@aluno.unb.br,rbonifacio@unb.br

Abstract. *Test smells have been proven to deteriorate the quality of the test suite of a system, to the point where several different tools have been devised with the objective of detecting or sometimes even fixing these smells. Have been extensively studied in more recent years, these smells have been cataloged and researchers have proposed a series of source code transformations capable of eliminating these smells. Our goal in this paper is to present* TESTAXE*, a tool to refactor test smells using the latest features of JUnit 5. We present an empirical assessment of* TESTAXE *accuracy and highlight its current limitations.*

1. Introduction

The design and implementation of a software system might evolve gradually. However, existing reports [Izurieta and Bieman 2007, Eick et al. 2001] show that during such evolutionary efforts, wrong design decisions might happen, eventually leading to code smells and increasing the technical debt of a system. Indeed, the accumulation of bad design decisions might cause the software's design to decay [Parnas 1994, Eick et al. 2001, Izurieta and Bieman 2007, de Silva and Balasubramaniam 2012], reflecting not only on the application code but also on the testing assets. The impact of code smells in the application code has been extensively studied in the last 20 years (e.g., [Sjoberg et al. 2013]), though recent research has specifically investigated the negative impact of test smells—not only on the comprehension and maintenance of test suites [Bavota et al. 2015], but also on the quality of the testing and application code [Spadini et al. 2018, Tufano et al. 2016].

Existing tools have been designed to identify [Palomba et al. 2018, Peruma et al. 2020] and refactor test smells [Lambiase et al. 2020, Santana et al. 2020, Pizzini 2022]. For instance, the ORACLEPOLISH tool identifies the smells *Brittle Assertions* and *Unused Input* in JUnit test cases [Huo and Clause 2014]; while the TASTE tool [Palomba et al. 2018] leverages information retrieval techniques on textual and structural features of test cases to identify test smells. More recently, Lambiase et al. presented DARTS (Detection And Refactoring of Test Smells), an IntelliJ plugin for detecting and refactoring the test smells *General Fixture*, *Eager Test*, and *Lack of Cohesion of Test Methods* [Lambiase et al. 2020]. Examples of tools able to detect a more comprehensive number of test smells include (a) TSDETECTOR [Peruma et al. 2020] and JNOSE [Virgínio et al. 2020]. These tools use pattern matching on the abstract-syntax trees of test cases and identify 19 and 21 types of test smells, respectively.

Aljedaani et al. present a systematic mapping study on the field of **test smell detection** [Aljedaani et al. 2021], reporting a total of 22 tools available in the literature. Differently, there are not so many tools targeting **test smell refactoring**. As already mentioned, the DARTS tool identifies and refactors three types of test smells [Lambiase et al. 2020]; while the research tool of Santana et al. [Santana et al. 2020] refactor the test smells *Assertion Roulette* and *Duplicate Assert*. Although these tools show evidence of the importance of automatic JUnit test smell refactoring, they do not consider the recent catalog of refactoring recommendations that benefit from the new JUnit 5 features [Soares et al. 2022].

The goal of this paper is to present the design and evaluation of TESTAXE, a tool that (a) supports developers in the task of migrating JUnit test cases to use the new features of the JUnit 5 test framework and (b) identifies and refactors five test smells using the new features of JUnit 5. We present some background in the next section. Section 3 presents the design and implementation of TESTAXE. We present details of an empirical assessment of TESTAXE in Section 4. Finally, in Section 5 we present some final remarks. TESTAXE is available at https://github.com/PAMunb/JUnit5Migration/

2. Background and related work

According to Spadini et al. [Spadini et al. 2018], "test smells are sub-optimal design choices in the implementation of test code" and several studies bring evidence that test smells might compromise not only the quality of the test suites [Bavota et al. 2015, Virgínio et al. 2019] but also the general quality of software systems [Tufano et al. 2016, Spadini et al. 2018, Kim et al. 2021, Wu et al. 2022]. For instance, Spadini et al. mined the source code history of ten open source projects and observed a correlation between the smells *Indirect Testing* and *Eager Testing* and the error-proneness of production code [Spadini et al. 2018]. Kim et al. also report that test smells make the code more error-prone [Kim et al. 2021].

Garousi and Küçük present a comprehensive survey on test smells, contributing with a taxonomy and a catalog of test smells [Garousi and Küçük 2018]. Their taxonomy groups test smells into six categories, including Test Execution, Test Logic, and Test Dependencies. Listing 1 shows an example of the Conditional Test Logic smell, which might lead the test execution to not run specific assertions [Soares et al. 2022]. Since test smells compromise the quality of the systems, it is fundamental to provide guidelines, idioms, and patterns that might help developers to avoid taking bad design decisions as well as design and implement tools for detecting and removing test smells [Palomba et al. 2018, Peruma et al. 2020, Lambiase et al. 2020, Santana et al. 2020, Pizzini 2022].

The work of Soares et al. has shown promising results of using new features from JUnit 5 to remove test smells [Soares et al. 2022]. More specifically, their paper describes seven features of JUnit 5 that can aid developers to remove 13 test smells, including Conditional Test Logic and Assertion Roulette [Soares et al. 2022] smells. The authors also define new refactorings in terms of templates, which we use as the basis for our TESTAXE implementation. Similarly to previous works [Lambiase et al. 2020, Santana et al. 2020], we use pattern matching on abstract syntax trees to implement TESTAXE—the first tool that implements four (of seven) refactorings from the Soares et al. catalog [Soares et al. 2022]

```
1 @Test
2 void first_test() {
3   if (lastContainerId == null) {
4     lastContainerId = genericContainer.getContainerId();
5   } else {
6      assertNotEquals(lastContainerId,
7      genericContainer.getContainerId());
8   }
9 }
```

Listing 1. Example of the Conditional Test Logic smell [Soares et al. 2022]

3. TestAXE

TESTAXE was devised as a tool to automatically detect and remove code smells present in Java software tests, especially those software that uses the JUnit 5 test framework without taking advantage of its newest features. TESTAXE is composed of two separate parts: a Python CLI application to prepare the environment, and a program transformation tool (hereafter `transformer`)—implemented in the Rascal meta-programming language [Klint et al. 2009]- that is responsible for transforming the test code.

3.1. The CLI Component

TESTAXE makes available a CLI application with a thin Python "shell" script to perform a few basic steps before calling the actual Rascal implementation—that transforms Java test code. This Python script recognizes two CLI options: the path of the repository to be transformed, and the number of maximum files to which the transformations will be applied. The repository is assumed to use the git versioning system, as the application creates, if it does not already exist, a `junit5-migration` branch and checks out to it. After checking out the new branch, the application finally calls the Rascal transformer meta-program implementation. As the transformation finishes, the CLI gets the modified file list from git and applies an external code formatting tool from Google.[1]

3.2. The Rascal Transformer Component

The second and main component of TESTAXE is a meta-program that leverages Rascal's powerful parse tree generator and its traversal functions to detect and refactor test smells, especially a set of smells whose refactor was proposed on [Soares et al. 2022]. These refactoring proposals are based mainly on new JUnit 5 features, such as new test annotations, assertion methods, and helper methods. This component also implements transformations for helping developers to migrate from legacy JUnit code to adopt new features of JUnit 5.

The transformer collects the files of interest by traversing the directory structure of the repository recursively. It then parses the file contents generating parse trees and executes a pipeline of transformations. Transformations are functions that comprise two steps: (a) a verifying step that checks preconditions and (b) a transformation step that refactors a test smell. As the outermost grammatical element is a `CompilationUnit`, transformations are essentially functions that take in a `CompilationUnit` as an argument

[1] https://github.com/google/google-java-format

and also return a `CompilationUnit`. As the transformer applies the pipeline functions, it collects metrics to determine which and how many transformations effectively modified a test case.

3.3. The Transformation Pipeline

As mentioned before, the transformation pipeline is comprised of a collection of functions. Indeed, we assign names to these functions in a hash map, so that we can collect metrics during the pipeline execution. The transformation pipeline is applied by iterating over an associative mapping of names and functions, while also aggregating a map of names to integers, representing the number of times a transformation has been applied.

As transformations may interfere with each other, the order in which the pipeline is assembled may yield different results. For instance, two of the transformations that were implemented deals with a sequence of assertion statements grouped together. These are the `AssertAll` and `ParameterizedTest` transformations, which fix, respectively, the `Assertion Roulette` and the `Test Code Duplication` smells.

Supposing there is a `Calculator` class with a `diff` static method, which returns the difference of the two numbers received as parameters. A test case for this method could look like Listing 2. As there is a collection of assertions being made and the intent of the test is that all of them be executed, if the first one fails, the two last would not even run, making it difficult to test the class as intended.

```
1 @Test
2 public void testCalculatorDiff() {
3   assertEquals(Calculator.diff(5, 1), 4);
4   assertEquals(Calculator.diff(10, 3), 7);
5   assertEquals(Calculator.diff(3, 6), -3);
6 }
```

Listing 2. A test method for a calculator class.

One possible solution would be to apply the `AssertAll` transformation. JUnit 5 offers a method that receives multiple lambda functions and runs all of them, disregarding individual assertion failures while the test method is running. After it finishes, it provides an adequate report on failed assertions, if any. Applying this transformation to Listing 2 results in the code in Listing 3.

```
1 @Test
2 public void testCalculatorDiff() {
3   Assert.assertAll(
4          () -> assertEquals(Calculator.diff(5, 1), 4),
5          () -> assertEquals(Calculator.diff(10, 3), 7),
6          () -> assertEquals(Calculator.diff(3, 6), -3)
7   );
8 }
```

Listing 3. A test method for a calculator class refactored with the AssertAll transformation.

Nonetheless, there is another, arguably more adequate, solution transformation for this case: the `ParameterizedTest` refactoring. A parameterized test receives the test

data as parameters, whose values can come from different sources. It is the most adequate refactoring considering repeated assertions of idempotent methods using different argument values. Applying this refactoring to the code in Listing 2 results in the code in Listing 4.

```
1 @ParameterizedTest
2 @CsvSource({ "5, 1, 4", "10, 3, 7", "3, 6, -3" })
3 public void testCalculatorDiff(int a, int b, int c)
4 {
5   assertEquals(Calculator.diff(a, b), c);
6 }
```

Listing 4. A test method for a calculator class refactored with the ParmeterizedTest transformation.

With listings 4 and 3, it becomes clear that both transformations are mutually exclusive, and therefore the order of the transformations in the pipeline is decisive for the end result. Table 1 lists the current set of TESTAXE transformations, as well as the execution order of the pipeline.

Order	Smell	Transformation	Description
T1	Exception Handling	`ExpectedException`	Transforms tests with exception parameters to `assertThrows` assertions
T2	-	`ExpectedTimeout`	Transforms tests with timeout parameters to `assertTimeout` assertions
T3	Assertion Roulette	`AssertAll`	Groups sequential assertions inside an `assertAll` call, guaranteeing every assertion will be verified
T4	Conditional Test	`ConditionalAssertion`	transforms tests that run their assertions conditionally, with an if statement wrapping their body, to tests that are conditionally run, by using the `@EnableIf("methodName")` annotation
T5	Test Code Duplication	`RepeatedTest`	Transforms tests that are wrapped within a for loop, to tests that have the `@RepeatedTest(iterationCount)` annotation
T6	Mystery Guest	`TempDir`	Adds a test parameter annotated by `@TempDir`, which is resolved into a temporary directory, to tests that use temporary files
T7	-	`SimpleAnnotations`	Migrates JUnit 4 annotations, including `@Before`, `@BeforeClass...`, into their JUnit 5 counterparts, as well as adding the necessary imports for the other transformations

Table 1. Set of TESTAXE transformations

These transformations are implemented in a modular fashion, each within its own module. In total, TESTAXE is comprised of 24 files, of which 22 are Rascal source code files, each one is a module, one is the Python driver, and the last one is the google source code formatter. TESTAXE has over 3750 lines of code.

3.4. How transformation works

Transformations are functions that receive a CompilationUnit and return a CompilationUnit that may or not have been modified. A CompilationUnit is a syntax definition reflecting the abstract syntax of a Java source code, which involve many structures (e.g., package declaration, imports, class definitions). These structures are traversed in order to get to the test method declarations, so detecting and fixing smells is possible. The structures are traversed from CompilationUnit to MethodDeclaration.

This is the underlying path that is traversed when it is necessary to modify or extract data from test declarations, but Rascal enables skipping several intermediate steps in this path through its `visit` expression. In order to, from a root node, modify or extract information from another node nested in its hierarchy, one can use the `visit` expression. For instance, Listing 5 shows an example of a method that searches for the first method declaration within a class declaration, returning a maybe structure that may or not contain this method declaration.

```
1 public Maybe[MethodDeclaration] getFirstMethod(ClassDeclaration classDeclaration) {
2     top-down visit(classDeclaration) {
3         case MethodDeclaration methodDeclaration: return just(methodDeclaration);
4     }
5     return nothing();
6 }
```

Listing 5. Visit expression to access deep nested values inside a node.

Listing 5 shows how, despite how nested a node may be in the parse tree, accessing it is a concise task by using the Rascal `visit` expression and pattern matching features. When a node matches the pattern, there are two different kinds of executions that may take place, depending on how the case is written: a direct replacement of the matched node with the $=>$ operator, or arbitrary code execution, with the ":" operator, which may resolve into a replacement as well (when using the `insert` statement). Both of these approaches appear on TESTAXE code.

Listing 6 shows a simplified version of the `SimpleAnnotation` transformation that helps developers to migrate test code to JUnit 5. This transformation uses the first kind of pattern matching, meaning that there is the replacement of the matched node by another one of the same type. It is also common that a more complex control flow may be necessary for some transformations, though. For instance, the `TempDir` transformation requires two changes in the test code: the replacement of all of the `createTempFile` method invocations directly from the File class, to a File instance received as a method parameter on the test method and the addition of a parameter to the test method with this instance as its value. For this, a boolean variable may be used to dictate whether `File.createTempFile` method invocations were found so that the addition of the test parameter may be done later. Listing 7 shows the segment of the code that does exactly this.

Another important aspect is that the pattern matching can be done in the AST form or in the concrete syntax form, which is extensively used by TESTAXE. The concrete syntax form allows for the extraction of node elements within a form that resembles the actual parsed source and can be seen throughout all of the listings.

```
1 public CompilationUnit executeSimpleAnnotationsTransformation(CompilationUnit unit) {
2   if(verifySimpleAnnotations(unit)) {
3     unit = top-down visit(unit) {
4      case (MethodModifier)`@BeforeClass` => (MethodModifier)`@BeforeAll`
5      case (MethodModifier)`@Before` => (MethodModifier)`@BeforeEach`
6     }
7   }
8   return unit;
9 }
```

Listing 6. Direct replacement example in the SimpleAnnotation transformation

```
private MethodDeclaration replaceTempFilesWithTempDir(MethodDeclaration method) {
  bool tempDirUsed = false;
  method = top-down visit(method) {
    case (MethodInvocation) `File.createTempFile(<ArgumentList args>)`: {
      tempDirUsed = true;
      insert((MethodInvocation) `tempDir.createTempFile(<ArgumentList args>)`);
    }
  }
  if(tempDirUsed)
    method = addMethodParameter(method, (FormalParameter) `@TempDir File tempDir`);
  return method;
}
```

Listing 7. Arbitrary code execution in the pattern match in the TempDir transformation.

4. Empirical Assessment

The goal of this empirical study is to assess the accuracy of TESTAXE transformations. Overall, we pose the following research questions:

1. Out of all the transformations that were applied, how often were they correct?
2. Were any of the transformations applied when they should not?
3. Out of all the refactoring opportunities, how often were they detected?

These questions can be answered by two metrics that were measured considering the output files TESTAXE produce: $Precision$, which measures how correctly the transformations were applied, and $Recall$, which measures how frequently were the refactoring opportunities taken. The computation of these metrics is as follows:

$$Precision = \frac{TP}{TP + FP}; Recall = \frac{TP}{TP + FN}$$

In which TP stands for true positives, FP for false positives, and FN for false negatives. For an overall performance, we calculate the F1-Score (F_1), which is computed as follows:

$$F_1 = 2\frac{Precision \times Recall}{Precision + Recall}$$

We apply the TESTAXE transformation over a curated dataset of 38 JUnit test cases. These test cases come from a study based on pull requests from Soares et al. [Soares et al. 2022]. [2]

4.1. Results

Table 2 shows the results of our empirical study. Note that our results outline a careful, in the sense of not taking any risks, but not a complete set of transformations. Since $Precision$ is 1, it means that there were no false positive cases, but the downside of this carefulness appears when looking at the measured value of $Recall$, showing room for improvements in smell detection. That is especially true for the `ParameterizedTest` transformation, which was frequently present on the false negative transformations, while not appearing once on the true positive.

[2] https://github.com/easy-software-ufal/refactoring-test-smells-with-junit5.

Smell	Transformation	TP	FP	FN	Precision	Recall	F_1
Assertion Roulette	`AssertAll`	97	0	41	1	0.70	0.83
Conditional Test Logic	`ConditionalAssertion`	1	0	2	1	0.34	0.5
Duplicate Assert	`ParameterizedTest`	0	0	21	0	0	0
Mystery Guest	`TempDir`	1	0	0	1	1	1
Test Code Duplication	`RepeatedTest`	4	0	0	1	1	1
Overall Result	-	103	0	64	1	0.62	0.76

Table 2. Smell/Transformations metrics

4.2. Limitations

The detection of smells can be rather naive. For instance, the detection of the `Conditional Test Logic` smell is a simple verification of a method body wrapped inside an if. Even if the assertions are all inside if statements, as shown in listing 8, the refactoring won't apply if there are any statements outside the if statement.

```
1 @Test
2 public void conditionalTestWithPrecedingStatements() {
3    someMethod();
4    if(true) {
5        Assert.assertEquals("something", "something");
6    }
7 }
```

Listing 8. Undetected conditional test logic due to method invocations before the if statement

That is also true for the `Test Code Duplication` smell, if there are any statements outside the for loop, the smell is not detected and therefore, the refactoring is not applied. For this same smell, for test correctness sake, the transformation is not applied if there are any method calls, or even values that are not literals (`integers`, `booleans`, `strings`, ...). Listing 9 shows an example of smell that would not be detected.

```
1 @Test
2 public void testCodeDuplicationWithMethodInvocationAssertion() {
3    Assert.assertEquals(1, 1);
4    Assert.assertEquals(multiply(5, -1), -5);
5    Assert.assertEquals(10, 10);
6 }
```

Listing 9. Unconsidered Test Code Duplication smell due to method invocation inside the assertions

5. Final Remarks and Future Work

This paper details the design and implementation of TESTAXE, a tool that refactors legacy JUnit code smells using the new features of JUnit 5. Currently, TESTAXE supports five (of seven) refactorings that Soares at al. detail in their paper [Soares et al. 2022]. We empirically evaluated TESTAXE and found that it presents a reasonable accuracy (F_1 of 0.76), though there are blind spots that lead TESTAXE to miss some opportunities for removing test smells.

As future work, we intend to complement this research in three main directions. First, we want to improve the accuracy of TESTAXE, so that it could deal with the corner cases that are harming its overall performance on fixing test smells. Second, we want to implement the remaining transformations detailed in [Soares et al. 2022]. Finally, we aim at conducting a case study with one of our industry patterns.

Declaration

Most of this work has been conducted by the first author of this paper (Estevan Alexander de Paula), during his final year project as an undergraduate student in Computer Engineering (at the University of Brasília).

References

Aljedaani, W., Peruma, A., Aljohani, A., Alotaibi, M., Mkaouer, M. W., Ouni, A., Newman, C. D., Ghallab, A., and Ludi, S. (2021). Test smell detection tools: A systematic mapping study. In *Evaluation and Assessment in Software Engineering*, EASE 2021, page 170–180, New York, NY, USA. Association for Computing Machinery.

Bavota, G., Qusef, A., Oliveto, R., Lucia, A. D., and Binkley, D. W. (2015). Are test smells really harmful? an empirical study. *Empir. Softw. Eng.*, 20(4):1052–1094.

de Silva, L. and Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151. Dynamic Analysis and Testing of Embedded Software.

Eick, S., Graves, T., Karr, A., Marron, J., and Mockus, A. (2001). Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12.

Garousi, V. and Küçük, B. (2018). Smells in software test code: A survey of knowledge in industry and academia. *J. Syst. Softw.*, 138:52–81.

Huo, C. and Clause, J. (2014). Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 621–631, New York, NY, USA. Association for Computing Machinery.

Izurieta, C. and Bieman, J. M. (2007). How software designs decay: A pilot study of pattern evolution. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 449–451.

Kim, D. J., Chen, T. P., and Yang, J. (2021). The secret life of test smells - an empirical study on test smell evolution and maintenance. *Empir. Softw. Eng.*, 26(5):100.

Klint, P., van der Storm, T., and Vinju, J. J. (2009). RASCAL: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 168–177. IEEE Computer Society.

Lambiase, S., Cupito, A., Pecorelli, F., De Lucia, A., and Palomba, F. (2020). Just-in-time test smell detection and refactoring: The darts project. In *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, page 441–445, New York, NY, USA. Association for Computing Machinery.

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

Parnas, D. (1994). Software aging. In *Proceedings of 16th International Conference on Software Engineering*, pages 279–287.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2020). 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*, ESEC/FSE 2020, page 1650–1654, New York, NY, USA. Association for Computing Machinery.

Pizzini, A. (2022). Behavior-based test smells refactoring : Toward an automatic approach to refactoring eager test and lazy test smells. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 261–263.

Santana, R., Martins, L. A., Rocha, L., Virgínio, T., Cruz, A., Costa, H. A. X., and Machado, I. (2020). RAIDE: a tool for assertion roulette and duplicate assert identification and refactoring. In *34th Brazilian Symposium on Software Engineering, SBES 2020, Natal, Brazil, October 19-23, 2020*, pages 374–379. ACM.

Sjoberg, D. I., Yamashita, A., Anda, B. C., Mockus, A., and Dybå, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156.

Soares, E., Ribeiro, M., Gheyi, R., Amaral, G., and Santos, A. M. (2022). Refactoring test smells with junit 5: Why should developers keep up-to-date. *IEEE Transactions on Software Engineering*, pages 1–1.

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., and Bacchelli, A. (2018). On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., and Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 4–15, New York, NY, USA. Association for Computing Machinery.

Virgínio, T., Martins, L., Rocha, L., Santana, R., Cruz, A., Costa, H., and Machado, I. (2020). Jnose: Java test smell detector. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, SBES '20, page 564–569, New York, NY, USA. Association for Computing Machinery.

Virgínio, T., Santana, R., Martins, L. A., Soares, L. R., Costa, H., and Machado, I. (2019). On the influence of test smells on test coverage. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, SBES 2019, page 467–471, New York, NY, USA. Association for Computing Machinery.

Wu, H., Yin, R., Gao, J., Huang, Z., and Huang, H. (2022). To what extent can code quality be improved by eliminating test smells? In *2022 International Conference on Code Quality (ICCQ)*, pages 19–26.

Chapter 3

A Detailed Look on TestAXE

TestAXE is a language processor, more specifically, a translator that performs software reenginering [12]. It has two different parts: (a) a driver written in Python[1], that is responsible for doing a basic parsing of the CLI parameters received, and (b) a Rascal[2] metraprogram containing all the transformations and test smell refactorings. Each part will be explained individually below.

3.1 The CLI Driver

The driver part of TestAXE does four basic tasks: (a) parse CLI arguments, (b) create a new *git* branch to accommodate the modification of files in a separate environment, (c) execute the translator software, written in Rascal, and finally (d) executing the google formatter[3] on modified files. The last step is important because the transformations builds code whose identation tends to be inadequate, but by keeping its execution limited to modified files, the side effects of this formatting is minimal.

3.1.1 CLI arguments

The CLI arguments received by the driver are the input directory, on flag `--input_dir`, or `-i` for the short version, and a number representing the maximum count of modified files, on flag `--max_files`, or `-m` for the short version. The input directory is required to be a directory holding *git* repository, while the maximum count of modified files is an optional parameter.

[1] `https://www.python.org/`
[2] `https://www.rascal-mpl.org/`
[3] `https://github.com/google/google-java-format`

3.1.2 Git Branch

A *git* branch named *junit5-migration* is then created at the source input directory. This allows for the modification of files without affecting the workflow of the main branch, while also making it easier for the refactorings to be transformed into a *Pull Request*[4] on *GitHub*[5], a *Merge Request*[6] on *GitLab*[7], or similar. If a branch named *junit5-migration* already exists, then the driver simply performs a checkout to it. Of course, if there are local uncommited changes, the branch creation or checkout will fail and the driver will stop its execution.

3.1.3 Executing Transformations

The design and organization of TestAXE will be described later in this same chapter, as well as the intricacies of its transformations. Although, for now it suffices to say that the driver calls a shell command executing Rascal's REPL jar file[8], as it is the recommended way of running Rascal programs outside of an IDE such as Eclipse[9]. The REPL allows for the execution of Rascal modules through an entrypoint function conveniently named *main*, while also allowing arguments to be passed to this entrypoint function. Of course, the Rascal module executed by the driver contains the code that reads, transforms and writes the files in the input directory.

3.1.4 Formatting

The output code is formatted using Google's Java formatter[10]. Only files marked as modified by *git* are formatted, as a way to minimize the impact of the opinionated formatting in the repository's files, while also eliminating extra spaces and tabs, or the lack of these characters produced by the introduction of code written in Rascal's concrete syntax form, which can differ much when compared to the code written by the software's maintainers in terms of indentaion.

[4] https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests

[5] https://github.com/

[6] https://docs.gitlab.com/ee/user/project/merge_requests/

[7] https://about.gitlab.com/

[8] https://www.rascal-mpl.org/start/

[9] https://www.eclipse.org/

[10] https://github.com/google/google-java-format

3.2 The Transformer

The transformer metaprogram, that reads and transforms java code, removing test smells and applying other necessary modifications to do so, is composed of a pipeline of transformations that receive a `CompilationUnit` and return a `CompilationUnit`, which may or may not have been modified by said function. While transformations are applied, metrics are collected regarding which and how many of these transformations produced modified output. A transformation won't be applied on every input code, as it performs validations on the input's contents to decide whether it should or not modify it.

As mentioned before, a transformation is essentially a function that receives and returns a `CompilationUnit`. A `CompilationUnit` is a generated type used to read and manipulate parsed source code according to a syntax definition, which will be later analyzed in more detail. But, in order to facilitate metric collection, a useful `Transformation` type has been defined. This type associates a name string and a transformation function in the terms mentioned previously. Listing 3.1 shows the definition of such type.

```
1 data Transformation = transformation(str name, CompilationUnit (CompilationUnit) function);
```

Listing 3.1: Transformation type and its constructor

3.2.1 The Transformation Pipeline

As mentioned, the transformations implemented by TestAXE are based on the work of Soares et Al [3]. However, from the transformations that were proposed in their work, two were left out: (a) *Resource Lock* and (b) *Repeated Tests* for code repetition between test cases.

Resource Lock was left out because applying it meant detecting resource usage that might be dependent on race conditions, and which test cases, running concurrently, would have to be refactored to eliminate race conditions, and the complexity of such verification is far from trivial, especially considering the time available for its design and implementation.

Repeated Tests, on the other hand, was left out due to the need of comparing test cases with other test cases. Aside from the increase in time complexity, since every test case would have to be compared with other test cases, it would have to be very carefully implemented in order to avoid false positives, which would eliminate tests cases.

The transformation pipeline is a list of `Transformation` variables. By checking on the difference between the input and the output of each function call, it is possible to account for which ones produced a modified output. Listing 3.2 shows how the entrypoint function mentioned on subsection 3.1.3 builds the transformation pipeline. The order

in which these transformations are applied may interfere with the final result, as one transformation may modify code that would be decisive for the conditional check of the next one. This aspect is explored in more detail on the paper presented in chapter 2.

```
list[Transformation] transformations = [
    transformation("ExpectedException", expectedExceptionTransform),
    transformation("ExpectedTimeout", expectedTimeoutTransform),
    transformation("AssertAll", executeAssertAllTransformation),
    transformation("ConditionalAssertion", executeConditionalAssertionTransformation),
    transformation("ParameterizedTest", executeParameterizedTestTransformation),
    transformation("RepeatedTest", executeRepeatedTestTransformation),
    transformation("TempDir", executeTempDirTransformation),
    transformation("SimpleAnnotations", simpleAnnotationTransform)
];
```

Listing 3.2: The transformation pipeline

The function references shown on 3.2 have all been imported from their respective transformation modules. Each transformation is contained within its own module, and a few utilities modules have also been written in order to avoid code repetition. Functions to manipulate and detect Java test methods, for instance, are commonly used throughout the transformations and have been extracted to separate modules.

Not all of the transformations shown on 3.2 refactor test smells. `SimpleAnnotations`, for instance, deals mainly with adjusting imports and updating annotations, which is, nonetheless, essential for the well functioning of the test smell refactors.

Listing 3.3 demonstrates how files are modified and metrics collected. The files contained in the `allFiles` variable are the ones contained in the `--input_dir` directory, explained earlier in this chapter. Listing 3.4 presents the implementation of the function that effectively applies the transformations and adds transformations metrics to the metrics map, which associates transformation names, (the same names used in the `Transformation` type constructor), to integers representing the amount of times that transformation has been applied. This function also tracks the total applied transformation count, allowing the pipeline to stop when the `--max_files` parameter is reached.

```
CompilationUnit transformedUnit;
for(loc f <- allFiles) {
    str content = readFile(f);

    <transformedUnit, totalTransformationCount, transformationCount> = applyTransformations(
        content,
        totalTransformationCount,
        transformationCount,
        transformations
```

```
10        );
11
12      writeFile(f, transformedUnit);
13      if((maxFiles) > 0 && (totalTransformationCount >= maxFiles)) {
14         break;
15      }
16  }
```

Listing 3.3: Implementation of the file transformation loop, as well as the update of the metrics map

```
1   public tuple[CompilationUnit, int, map[str, int]] applyTransformations(
2       str code,
3       int totalTransformationCount,
4       map[str, int] transformationCount,
5       list[Transformation] transformations
6     ) {
7     CompilationUnit unit = parse(#CompilationUnit, code);
8
9     for(Transformation transformation <- transformations) {
10      CompilationUnit transformedUnit = transformation.function(unit);
11      if(unit != transformedUnit) {
12        transformationCount[transformation.name] += 1;
13        totalTransformationCount += 1;
14      }
15      unit = transformedUnit;
16    }
17
18    return <unit, totalTransformationCount, transformationCount>;
19  }
```

Listing 3.4: The implementation of source code parsing and transformation

With a general understanding of how the transformation pipeline is built and applied, a more detailed explanation on the syntax definitions and on each of the transformations is presented below.

Syntax Definitions and Parsing

Rascal allows for the definition of context-free grammars, building parsers according to these definitions. The syntax definitions used in TestAXE were written by Prof. Dr. Rodrigo Bonifácio de Almeida. The `CompilationUnit` definition mentioned earlier is one of such definitions used in TestAXE, and is the outermost syntax definition a Java source code can have. Listing 3.5 shows the definition of a `CompilationUnit`.

```
start syntax CompilationUnit = PackageDeclaration? Imports TypeDeclaration*;
```

Listing 3.5: Syntactic definition of the `CompilationUnit`

The `start` keyword in listing 3.5 denotes a definition that is the root of any generated `ParseTree`. The listing clearly shows that the definition of a `CompilationUnit` is non-terminal, meaning it is derived from other syntactic definitions, namely `PackageDeclaration`, `Imports` and `TypeDeclaration`. In other words, a `CompilationUnit` is composed of these three "lower" syntactic definitions. There are, however, two special tokens in this definition that have a special meaning: the `?` character means the preceding definition may or not be present, while the `*` character denotes a list of the preceding definition. Therefore, a `CompilationUnit` is composed of a `PackageDeclaration` node that may or not be present, an `Imports` node, and a list of `TypeDeclaration` nodes.

Understanding the syntactic definition of the most common elements in the transformations is required to properly understand them, so the path from `CompilationUnit` to a method declaration will be presented below, which will later aid the understanding of the actual refactoring implementations. Listing 3.6 shows syntax definitions sequentially from `CompilationUnit` to `MethodDeclaration`, with emphasized text on the definitions that lead from the first to the latter. The listing also exposes some definitions that contain literals, such as `";"` and `"{"`. These characters inside quotation marks are interpreted literally by the parser.

```
1  start syntax CompilationUnit = PackageDeclaration? Imports TypeDeclaration*;
2
3  syntax TypeDeclaration = ClassDeclaration ";"*
4                         | InterfaceDeclaration ";"*
5                         ;
6
7  syntax ClassDeclaration = NormalClassDeclaration
8                          | EnumDeclaration
9                          ;
10
11 syntax NormalClassDeclaration = ClassModifier* "class" Identifier TypeParameters? Superclass?
       ↪ Superinterfaces? ClassBody;
12
13 syntax ClassBody = "{" ClassBodyDeclaration* decls "}" ";"? ;
14
15 syntax ClassBodyDeclaration = ClassMemberDeclaration
16                             | InstanceInitializer
17                             | StaticInitializer
18                             | ConstructorDeclaration
19                             ;
```

```
20
21 syntax ClassMemberDeclaration = FieldDeclaration
22                              | MethodDeclaration
23                              | ClassDeclaration
24                              | InterfaceDeclaration
25                              ;
26
27 syntax MethodDeclaration = MethodModifier* MethodHeader MethodBody;
```

Listing 3.6: All syntactic definitions from `CompilationUnit` to `MethodDeclaration` organized sequentially

Figure 3.1 shows a tree visualization of these nodes. Each node connected to its parent node denote an *or* composition, meaning that the parent node may be any of such nodes, while each node connected to its sibling node denote a single definition. Nodes highlighted in blue background show the path from `CompilationUnit` to `MethodDeclaration`.

Listings 3.8 to 3.13 show each node value in the incremental parsing of the example source code in listing 3.7. Each listing displays the value that would be contained inside the `ParseTree` node of that specific type.

```
1  package app.example;
2
3  import org.junit.jupiter.api.Test;
4
5  public class ExampleTestClass {
6      @Test
7      public void exampleTestMethod() {
8        Assertions.assertEquals(1, 1);
9      }
10 }
```

Listing 3.7: Original source code

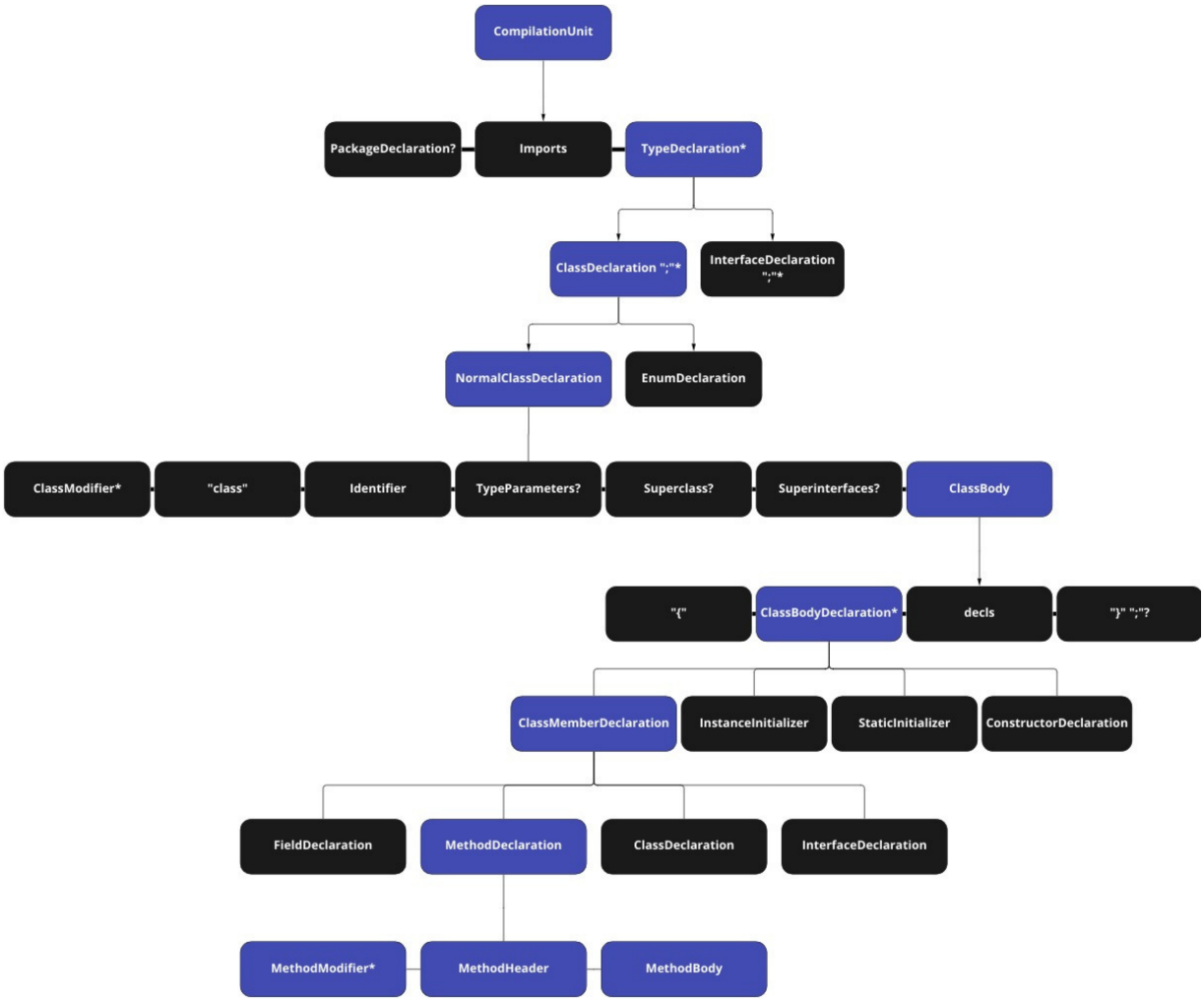

Figure 3.1: Syntax definition tree with highlighted nodes to show the path from `CompilationUnit` to `MethodDeclaration`

```java
package app.example;

import org.junit.jupiter.api.Test;

public class ExampleTestClass {
    @Test
    public void exampleTestMethod() {
      Assertions.assertEquals(1, 1);
    }
}
```

Listing 3.8: `CompilationUnit` node

```java
public class ExampleTestClass {
    @Test
    public void exampleTestMethod() {
      Assertions.assertEquals(1, 1);
    }
}
```

Listing 3.9: `ClassDeclaration` node

```java
{
    @Test
    public void exampleTestMethod() {
      Assertions.assertEquals(1, 1);
    }
}
```

Listing 3.10: `ClassBody` node

```java
@Test
public void exampleTestMethod() {
  Assertions.assertEquals(1, 1);
}
```

Listing 3.11: `ClassBodyDeclaration` node

```java
@Test
public void exampleTestMethod() {
  Assertions.assertEquals(1, 1);
}
```

Listing 3.12: `ClassMemberDeclaration` node

```java
@Test
public void exampleTestMethod() {
  Assertions.assertEquals(1, 1);
}
```

Listing 3.13: `MethodDeclaration` node

Furthermore, breaking down a `MethodDeclaration` node makes it possible for the detection and the manipulation of a test method. Some transformations manipulate method modifiers, method names, or even write new methods as a whole, so the definitions inside a `MethodDeclaration` are important and frequently used throughout TestAXE. Listing 3.14 exposes the syntax definitions that comprise a method declaration.

```
syntax MethodDeclaration = MethodModifier* MethodHeader MethodBody;

syntax MethodModifier = Annotation
                      | "public"
                      | "protected"
                      | "private"
                      | "abstract"
                      | "static"
                      | "final"
                      | "synchronized"
                      | "native"
```

```
12                      | "strictfp"
13                      ;
14
15 syntax MethodHeader = Result MethodDeclarator Throws?
16                      |  TypeParameters Annotation* Result MethodDeclarator Throws?
17                      ;
18
19 syntax MethodBody = Block ";"*
20                   | ";"
21                   ;
```

Listing 3.14: Syntactic definitions contained within a `MethodDeclaration`

Table 3.1 shows what breaking down the `MethodDeclaration` node on listing 3.13 into the more specialized syntactic definitions on listing 3.14 would result in. The table is organized according the token's appearance in listing 3.13. Functions that detect test methods, for instance, do so by checking if any of the `Annotation` nodes contained within the `MethodDeclaration` has `@Test` as its value. Similarly, transformations that modify this default `@Test` annotation do so by rebuilding the whole `MethodDeclaration` node with another `Annotaion` node in its place, such as `@ParameterizedTest` or `@RepeatedTest(i)`.

Node Value	Outer Node Type	Inner Node Type
@Test	MethodModifier	Annotation
public	MethodModifier	-
void	Result	-
exampleTestMethod	MethodDeclarator	-
{ Assertions.assertEquals(1, 1); }	MethodBody	Block

Table 3.1: Break down of nodes contained in a `MethodDeclaration` (outer nodes) into more specialized nodes (inner nodes)

`ParseTree` Traversal

As well as providing the tools for `ParseTree` generation, Rascal offers ways to traverse, inspect and modify it. The `visit` expression traverses the `ParseTree` received as argument and allows for the definition of patterns that, when matched, perform a specific action. This pattern is known as *Tree Pattern Matcher*[13]. The action to be performed can be in one of two forms: (a) replacement or (b) code block execution.

A replacement can be identified as by the `=>` operator, and its effect is the replacement of the matched pattern by whatever is on the right-hand side of it, given that both share the same type. Listing 3.15 shows an example of a replacement: transforming a private method into a public method.

A code block execution, on the other hand, is preceded by `:`, and allows for the execution of arbitrary code, that may or may not replace the left-hand side. Listing 3.16 shows an example of such pattern, which is the function used in TestAXE to detect `MethodDeclaration` nodes that contain a test method. There are two special statements inside a code block in such circumstances: an `insert` statement will replace the matched pattern with the node passed as parameter, while a `fail` statement will skip the current match.

```
top-down visit(methodDeclaration) {
    case (MethodModifier) `private` => (MethodModifier) `public`
}
```

Listing 3.15: Replacement inside a `visit` expression

```
public bool isMethodATest(MethodDeclaration method) {
  top-down visit(method) {
    case (Annotation) `@Test`: return true;
    case (Annotation) `@ParameterizedTest`: return true;
    case (Annotation) `@RepeatedTest(<ElementValue _>)`: return true;
  }

  return false;
}
```

Listing 3.16: Code block execution inside a `visit` expression

With this introduction on how the `ParseTree` is generated, traversed and manipulated, the next subsections will go into more detail on how each transformation works. `SimpleAnnotations`, `ExpectedException` and `ExpectedTimeout` transformations were all written by Prof. Dr. Rodrigo Bonifácio de Almeida, and while the first has had some minor modifications for the current version of TestAXE, the last two remained unchanged.

3.2.2 SimpleAnnotations

While the `SimpleAnnotations` transformation does not refactor any test smell by itself, none of the transformations that do so would work without it. It acts by updating special annotations used by JUnit from version 4 to version 5, as well as by adding the correct import of the newest testing API in the framework. Listing 3.17 shows the most important part of the transformation.

```
public CompilationUnit executeSimpleAnnotationsTransformation(CompilationUnit unit) {
        if(verifySimpleAnnotations(unit)) {
                unit = top-down visit(unit) {
                        case (Imports)`<ImportDeclaration* imports>` => updateImports(imports)
```

```
5                       case (MethodModifier)‘@BeforeClass‘ => (MethodModifier)‘@BeforeAll‘
6                       case (MethodModifier)‘@Before‘ => (MethodModifier)‘@BeforeEach‘
7                       case (MethodModifier)‘@After‘ => (MethodModifier)‘@AfterEach‘
8                       case (MethodModifier)‘@AfterClass‘ => (MethodModifier)‘@AfterAll‘
9                       case (MethodModifier)‘@Ignore‘ => (MethodModifier)‘@Disabled‘
10              }
11      }
12      return unit;
13  }
```

Listing 3.17: The main part of the `SimpleAnnotations` transformation

3.2.3 ExpectedException

This transformation, as well as `SimpleAnnotation`, does not refactor any test smell and aims to update testing code to use the latest definitions of JUnit 5, in this case, the capturing of exception throwing on test methods. Listing 3.18 presents this transformation.

```
1   public CompilationUnit executeExpectedExceptionTransformation(CompilationUnit unit) {
2       if(verifyExpectedException(unit)) {
3               unit = top-down visit(unit) {
4                       case (Imports)‘<ImportDeclaration* imports>‘ => updateImports(imports)
5
6                       case (MethodDeclaration)‘@Test(expected = <TypeName exception>.class)
                            ↪ public void <Identifier name>() <Throws t> { <BlockStatements
                            ↪ stmts> }‘ =>
7                            (MethodDeclaration)‘@Test
8                                           ’public void <Identifier name>() <Throws t> {
9                                           ’      Assertions.assertThrows(<TypeName
                                                ↪ exception>.class, () -\> {
10                                          ’         <BlockStatements stmts>  });
11                                          ’    }
12                                          ’
13                                          ’    ‘
14
15                      case (MethodDeclaration)‘@Test(expected = <TypeName exception>.class)
                            ↪ public void <Identifier name>()  { <BlockStatements stmts> }‘
                            ↪ =>
16                          (MethodDeclaration)‘@Test
17                                          ’  public void <Identifier name>() {
18                                          ’      Assertions.assertThrows(<TypeName
                                                ↪ exception>.class, () -\> {
19                                          ’         <BlockStatements stmts>  });
20                                          ’    }
21                                          ’
```

```
22                                                  ’  ‘
23                  }
24          }
25          return unit;
26  }
```

Listing 3.18: The main part of the `ExpectedException` transformation

3.2.4 ExpectedTimeout

The `ExpectedTimeout` transformation, as well as the two previous ones, does not target any test smell but rather the update of code that does not use the latest APIs offered by JUnit. The transformation is shown on Listing 3.19.

```
1   public CompilationUnit executeExpectedTimeoutTransformation(CompilationUnit unit) {
2           if(verifyTimeOut(unit)) {
3                   unit = top-down visit(unit) {
4                           case (Imports)‘<ImportDeclaration* imports>‘ => updateImports(imports)
5
6                           case (MethodDeclaration)‘@Test(timeout = <ConditionalExpression exp>)
7                                                   ’public void <Identifier name>() <Throws t> {
8                                                   ’  <BlockStatements stmts>
9                                                   ’}‘ =>
10
11                                  (MethodDeclaration)‘@Test
12                                                      ’public void <Identifier name>() <Throws t> {
13                                                      ’      Assertions.assertTimeout(Duration.
                                                            ↪ ofMillis(<ConditionalExpression exp>),
                                                            ↪ () -\> {
14                                                      ’         <BlockStatements stmts>   });
15                                                      ’    }
16                                                      ’
17                                                      ’    ‘
18
19                          case (MethodDeclaration)‘@Test(timeout = <ConditionalExpression exp>)
20                                                  ’public void <Identifier name>()  {
21                                                  ’  <BlockStatements stmts>
22                                                  ’}‘ =>
23
24                                  (MethodDeclaration)‘@Test
25                                                      ’public void <Identifier name>() {
26                                                      ’      Assertions.assertTimeout(Duration.
                                                            ↪ ofMillis(<ConditionalExpression exp>),
                                                            ↪ () -\> {
27                                                      ’         <BlockStatements stmts>   });
```

```
28                                                  '    }
29                                                  '
30                                                  '    '
31                  }
32          }
33          return unit;
34  }
```

Listing 3.19: The main part of the `ExpectedTimeout` transformation

3.2.5 AssertAll

`AssertAll` is the first transformation of the pipeline that aims test smell refactoring. *Assertion Roulette* is a test smell that happens when there are many assertions being made inside a single test method. The usual behavior of assertions is to stop the execution of a test as soon as the first one fails, which is exactly what JUnit's assertions do. One way to avoid such problem would be to refactor the test method into multiple test cases, which would also improve readability since their scope would be reduced and better defined. However, this is not always convenient, or maybe not even an option, so a better approach to address a sequence of assertion statements would be to group all of these assertions in such a way that every single assertion is run, and this is exactly what `assertAll` from JUnit 5 does. Listings 3.20 shows what a code presenting such smell would look like and listing 3.21 shows what the refactored output would look like.

```
1  @Test
2  public void exampleTestMethod() {
3      assertionA;
4      assertionB;
5      assertionC;
6  }
```

Listing 3.20: Test method containing the *Assertion Roulette smell*

```
1  @Test
2  public void exampleTestMethod() {
3      Assertions.assertAll(
4          () -> assertionA,
5          () -> assertionB,
6          () -> assertionC
7      );
8  }
```

Listing 3.21: Refactored test method using the `assertAll` expression

The `assertAll` assertion receives multiple *lambda* arguments, preferably with a single assertion being run inside each. As the test finishes, all of the lambdas have been run and a proper report on which assertions failed have failed is shown. To apply this refactoring, the `AssertAll` transformation searches for sequential assertions inside test methods and wraps them inside *lambdas* in a `assertAll` statement. Listing 3.22 shows the entrypoint function of this transformation.

```
public CompilationUnit executeAssertAllTransformation(CompilationUnit unit) {
  unit = top-down visit(unit) {
    case MethodDeclaration method => declareTestWithAssertAll(method)
                            when isMethodATest(method) && hasSequentialAssertions(method)
  }

  return unit;
}
```

Listing 3.22: Entrypoint of the `AssertAll` transformation

For readability's sake, the logic has been split on two functions, namely `hasSequentialAssertions` ↪ and `declareTestWithAssertAll`. The `when` statement expects a boolean value to determine whether the execution of the replacement action should take place. The presence of sequential assertion statements inside the method's body is verified by counting sequences of these statements. Whenever a sequence hits a length of 2, the method returns `true`, otherwise, it returns `false`. If the current `MethodDeclaration` node is a test method and contains at least one sequence of assertion statement, the transformation is applied.

Listing 3.23 shows what the transformation process looks like. The transformation keeps track of what statements will be present in the output code through the `refactoredStatements` variable. While it traverses the method body, it adds assertino statements an assertion group. Although, if the current statement is not an assertion, the assertion group is reset and its size is checked. If the assertion group has the minumum statement count, the group is rewritten as an `assertAll` expression.

```
int MINIMUM_ASSERTION_GROUP_SIZE = 2;

public CompilationUnit executeAssertAllTransformation(CompilationUnit unit) {
  unit = top-down visit(unit) {
    case MethodDeclaration method => declareTestWithAssertAll(method)
                            when isMethodATest(method) && hasSequentialAssertions(method)
  }

  return unit;
}

public MethodDeclaration declareTestWithAssertAll(MethodDeclaration method) {
  list[BlockStatement] refactoredStatements = [];
  list[BlockStatement] assertionGroup = [];
  Maybe[BlockStatement] previousStmt = nothing();

  top-down-break visit(extractMethodBody(method)) {
    case BlockStatement s : {
```

```
19        if(isStatementAnAssertion(s)) {
20          assertionGroup += s;
21        } else {
22          if(isSomething(previousStmt) && isStatementAnAssertion(unwrap(previousStmt))) {
23            if(size(assertionGroup) >= MINIMUM_ASSERTION_GROUP_SIZE) {
24              refactoredStatements += buildAssertAll(assertionGroup);
25            } else if (size(assertionGroup) != 0) {
26              refactoredStatements += assertionGroup;
27            }
28            assertionGroup = [];
29          }
30          refactoredStatements += s;
31        }
32        previousStmt = just(s);
33      }
34    }
35
36    if(size(assertionGroup) >= MINIMUM_ASSERTION_GROUP_SIZE) {
37      refactoredStatements += buildAssertAll(assertionGroup);
38    } else if (size(assertionGroup) != 0) {
39      refactoredStatements += assertionGroup;
40    }
41
42    str methodBody = ("{\n" | it + unparse(s) + "\n" | BlockStatement s <- refactoredStatements)
          ↪  + "}";
43
44    return replaceMethodBody(method, parse(#MethodBody, methodBody));
45  }
```

Listing 3.23: Transformation logic for the `AssertAll` transformation

3.2.6 ConditionalAssertion

The *Conditional Test Logic* smell happens when a test method contains control-flow structures inside the test code, such as an `if` statement. By adding conditional test logic to a test method, the test suite is prone to false positive tests, because conditional tests will be reported as being successfully ran when they actually haven't run any assertions at all. In order to avoid such behavior, one can use the `@EnableIf` annotation from JUnit 5. Listings 3.24 and 3.25 show what a test method with this smell looks like and what a refactored test method looks like, respectively.

```
@Test
public void conditionalTestLogic() {
    if(System.getProperty("os.name") == "
        ↪ Linux") {
        assertion;
    }
}
```

Listing 3.24: Test method with the *Conditional Test Logic* smell

```
@Test
@EnableIf("isOSLinux")
public void unconditionalTestLogic() {
    assertion;
}

public bool isOSLinux() {
    return System.getProperty("os.name")
        ↪ == "Linux";
}
```

Listing 3.25: Refactored code to eliminate the *Conditional Test Logic* smell

The `@EnableIf` annotation conditionally runs the whole test method, depending only on the return value of the method whose name is received as a string parameter. This transformation works by keeping a list of conditional statements that will be used to generate the *boolean* functions used to conditionally enable tests. But it only does this to tests that can be refactored. In order to avoid missing variable definitions problems, test setup code separation from the test body, etc...the only test methods that are refactored by this transformation are those whose body contains exclusively an **if** with no **else** statements. Listing 3.26 present the transformation's code.

```
public CompilationUnit executeConditionalAssertionTransformation(CompilationUnit unit) {
  list[tuple[Identifier name, Expression condition]] conditionalMethods = [];

  unit = top-down visit(unit) {
    case MethodDeclaration method: {
      if(isMethodATest(method)) {
        switch(applyTransformation(method)) {
          case just(transformedMethodData): {
            conditionalMethods += <transformedMethodData[1], transformedMethodData[2]>;
            insert(transformedMethodData[0]);
          }
          case nothing(): fail;
        }
      }
    }
  }

  return (unit |
            declareNewMethod(t.condition, t.name, it) |
            tuple[Identifier name, Expression condition] t <- conditionalMethods);
}
```

```
22
23  private Maybe[tuple[MethodDeclaration declaration, Identifier enablerName, Expression
        ↪ condition]] applyTransformation(MethodDeclaration method) {
24    Identifier testName = extractMethodName(method);
25
26    top-down-break visit(extractMethodBody(method)) {
27      case (MethodBody) `{
28                        '  if(<Expression condition>) <Statement statement>
29                        '}` : {
30        if(!isStatementABlock(statement)) fail;
31        str conditionalMethodName = "<unparse(testName)>Condition";
32        StringLiteral conditionalMethodNameLiteral = parse(#StringLiteral, "\"<
              ↪ conditionalMethodName>\"");
33        Annotation enablerAnnotation = (Annotation) `@EnableIf(<StringLiteral
              ↪ conditionalMethodNameLiteral>)`;
34        MethodBody refactoredBody = parse(#MethodBody, unparse(statement));
35        MethodDeclaration transformedMethod = replaceMethodBody(
36                                                addMethodAnnotation(method, enablerAnnotation),
37                                                refactoredBody
38                                              );
39        return just(<transformedMethod, parse(#Identifier, conditionalMethodName), condition>);
40      }
41      case MethodBody _: return nothing();
42    }
43
44    return nothing();
45  }
```

Listing 3.26: Transformation code for the `ConditionalAssertion` transformation

3.2.7 ParameterizedTest

The `ParameterizedTest` transformation fixes the *Duplicate Assert* and *Test Code Repetition* smells. A series of repetitive assertions which differ only by their parameters is the cause the first smell, while the repetition of setup code that not necessarily contains assertions, for example, causes the latter. In order to avoid repetitiveness in these cases, JUnit 5 offers the `@ParameterizedTest` annotation, which must be accompanied by at least one other annotation that provides source values to be received as parameters. Example of sources are `@NullSource`, `@EmptySource`, `@EnumSource` and the one used by the `ParameterizedTest` transformation, `@CsvSource`. These sources can be combined in order to provide a collection of source values to a test method. Listings 3.27 shows what a test method porting the

Duplicate Assert smell looks like, while 3.28 shows its refactored counterpart using the `@ParameterizedTest` annotation.

```
@Test
public void unparemeterizedTest() {
    Assertions.assertEquals(1, 1);
    Assertions.assertEquals(2, 2);
    Assertions.assertEquals(5, 5);
    Assertions.assertEquals(0, 0);
}
```

Listing 3.27: Test method presenting the *Duplicate Assert* smell

```
@ParameterizedTest
@CsvSource({
    "1, 1", "2, 2", "5, 5", "0, 0"
})
public void parameterizedTest(int a, int b
    ↪ ) {
    Assertions.assertEquals(a, b);
}
```

Listing 3.28: Refactored code to eliminate the *Conditional Test Logic* smell

This transformation is only applied to test methods that repeat the same assertion for several different argument values. By detecting that every assertion is the same, the test can be refactored to a single assertion invocation that receives each argument value at a time. After making all assertions are the same method, extracting each assertion argument and its type suffices, as this allows for the addition of the test parameters with correct types. Then, a single assertion is built with the parameters received by the test. Listing 3.29 shows the main logic for this transformation.

```
data Argument = argument(str argType, Expression expression);

public CompilationUnit executeParameterizedTestTransformation(CompilationUnit unit) {
  unit = top-down visit(unit) {
    case MethodDeclaration method => parameterizeTest(method)
                                     when (isMethodATest(method) &&
                                             testHasMultipleStatements(extractMethodBody(
                                                 ↪ method)) &&
                                             allStatementsAreTheSameAssertion(
                                                 ↪ extractMethodBody(method)))
  }

  return unit;
}

private MethodDeclaration parameterizeTest(MethodDeclaration method) {
  list[ArgumentList] args = [];

  switch(extractAssertionsArguments(extractMethodBody(method))) {
    case just(argList): args = argList;
    case nothing(): return method;
```

```
20    }
21
22    list[list[Argument]] invocationArgs = [];
23    switch(extractArguments(args)) {
24      case just(arguments): invocationArgs = arguments;
25      case nothing(): return method;
26    }
27
28    if(!allArgumentsHaveSameType(invocationArgs)) return method;
29
30    return refactorTest(method, invocationArgs);
31  }
32
33  private MethodDeclaration refactorTest(MethodDeclaration method, list[list[Argument]]
        ↪ invocationArgs) {
34    list[FormalParameter] methodParams = [];
35    str assertionArgs = "";
36    int i = 0;
37    for(Argument arg <- head(invocationArgs)) {
38      str argName = "arg<i>";
39      methodParams += parse(#FormalParameter, "<arg.argType> <argName>");
40      assertionArgs += "<argName>, ";
41      i += 1;
42    }
43    method = (method | addMethodParameter(it, param) | FormalParameter param <- methodParams);
44
45    ArgumentList assertionArgList = parse(#ArgumentList, assertionArgs[..-2]);
46    MethodInvocation assertionWithParameterizedArgs = top-down-break visit(
47        extractFirstMethodInvocation(extractMethodBody(method))
48      ) {
49      case ArgumentList _ => assertionArgList
50    };
51
52    MethodBody refactoredBody = (MethodBody) `{
53                                              '  <MethodInvocation assertionWithParameterizedArgs
                                                    ↪ >;
54                                              '}`;
55
56    return addParameterizedTestAnnotation(replaceMethodBody(method, refactoredBody),
        ↪ invocationArgs);;
57  }
```

Listing 3.29: Transformation logic for the `ParameterizedTest` transformation

3.2.8 RepeatedTest

Out of all the refactorings implemented, `RepeatedTest` is probably the most complex. The purpose of this transformation is to refactor test methods with defined finite loops inside them to use the `@RepeatedTest(i)` annotation instead, in which `i` is the amount of times the test method should be repeated. This avoids that the failure of one execution stops all the others, which brings the *Conditional Test Logic* smell, although not as explicitly as in the case of the `ConditionalAssertion` transformation. Listing 3.30 shows what a unrefactored test looks like, while 3.31 shows what the refactored output looks like.

```
1 @Test
2 public void repeatedTest() {
3     for(i = 0; i < 5; i++) {
4         Assertions.assertEquals(1, 1);
5     }
6 }
```

Listing 3.30: Test method containing a repetition structure

```
1 @RepeatedTest(5)
2 public void repeatedTest() {
3     Assertions.assertEquals(1, 1);
4 }
```

Listing 3.31: Refactored test method using the `@RepeatedTest` annotation

This transformation is complex because there are several conditions that have to be checked before making sure applying it is safe. Whether the variable used for the iteration condition is used inside the test, or how was the iteration build, from initialization to updating, how many stop conditions there is, are all aspects to be taken into account when applying this transformation.

The transformation is accomplished by extracting the *for* loop's statements and its information: initialization values, conditional expressions, identifiers (variable names) used and update expressions. In order to simplify the manipulation and transport of this data, a `ForStatementData` type has been created, aggregating them together. After the *for* statement's information has been extracted, the verifications mentioned above are carried out. Then, the iteration count is inferred from the *for* loop. Only *for* loops with a single condition statement and updated variable are transformed, and the update statement can only be a simple increment of the counter variable. The condition, however, may be any of the basic numeric comparisons. Listing 3.32 shows the `RepeatedTest` transformation logic.

```
1 data ForStatementData = forStatementData(
2     map[Identifier, int] forInitValues,
3     StatementExpressionList forUpdateExpression,
4     list[Identifier] forUpdateIdentifiers,
5     list[tuple[Identifier id, str op, IntegerLiteral vl]] forConditionParts,
6     Statement statement
7 );
```

```
8
9  public CompilationUnit executeRepeatedTestTransformation(CompilationUnit unit) {
10   unit = top-down visit(unit) {
11     case MethodDeclaration method : {
12                                 switch(extractForStatementData(extractMethodBody(method))) {
13                                   case just(forStmtData): {
14                                     if(isTransformationApplyable(forStmtData) && isSomething(
                                           ↪ resolveIterationCount(forStmtData))) {
15                                       insert(declareTestWithRepeatedTest(method, forStmtData));
16                                     }
17                                   }
18                                   case nothing(): fail;
19                                 }
20                               }
21   }
22
23   return unit;
24 }
25
26 public MethodDeclaration declareTestWithRepeatedTest(MethodDeclaration method,
       ↪ ForStatementData f) {
27   IntegerLiteral iterationCount = parse(#IntegerLiteral, toString(unwrap(resolveIterationCount
         ↪ (f))));
28   MethodBody newBody = parse(#MethodBody, unparse(f.statement));
29
30   Annotation repeatedTestAnnotation = (Annotation) `@RepeatedTest(<IntegerLiteral
         ↪ iterationCount>)`;
31   Annotation regularTestAnnotation = (Annotation) `@Test`;
32
33   return replaceMethodBody(
34     unwrap(replaceMethodAnnotation(method, regularTestAnnotation, repeatedTestAnnotation)),
35     newBody
36   );
37 }
```

Listing 3.32: `RepeatedTest` transformation

3.2.9 TempDir

Tests that involve external resources are often a problem, as they depend on external, and usually uncontrollable and unreliable, state. Depending on this uncontrollable external state is what constitutes the *Mystery Guest* smell. In the specific case of the usage of files inside tests, JUnit offers an extension to provide temporary directories, providing a safe alternative do deal with files while being certain one test case will not interfere

with other test cases. The TempDirectory extension provides the `@TempDir` annotation, which can be used to provide a file in a temporary directory, received as a test method parameter. Listing 3.33 displays a test method containing the *Mystery Guest* smell by using files, while listing 3.34 shows what the refactoring of this test smell using the `@TempDir` annotation would look like.

```
1 @Test
2 public void mysteryGuestTest() {
3     File outputFile = File.createTempFile(
      ↪ "report", null);
4     writeSomethingToFile(outputFile);
5     Assertions.assertTrue(outputFile.
      ↪ length() > 0);
6 }
```

Listing 3.33: Test method containing the *Mystery Guest* smell by using files

```
1 @Test
2 public void refactoredTest(@TempDir File
  ↪ tempDir) {
3     File outputFile = tempDir.
      ↪ createTempFile("report", null);
4     writeSomethingToFile(outputFile);
5     Assertions.assertTrue(outputFile.
      ↪ length() > 0);
6 }
```

Listing 3.34: Refactored test method using the `@TempDir` annotation

The implementation for this transformation is very simple, as it simply searches for invocations of the `File.createTempFile()` method and replaces them for a `tempDir.createTempFile` ↪ `()` invocation. If any of such invocations have been made, and consequently replaced, a `@TempDir File tempDir` annotated parameter is added to the test method. Listing 3.35 contains the whole transformation code.

```
1  public CompilationUnit executeTempDirTransformation(CompilationUnit unit) {
2    unit = top-down visit(unit) {
3      case MethodDeclaration method => replaceTempFilesWithTempDir(method)
4                                           when isMethodATest(method)
5    }
6
7    return unit;
8  }
9
10 private MethodDeclaration replaceTempFilesWithTempDir(MethodDeclaration method) {
11   bool tempDirUsed = false;
12   method = top-down visit(method) {
13     case (MethodInvocation) `File.createTempFile(<ArgumentList args>)`: {
14       tempDirUsed = true;
15       insert((MethodInvocation) `tempDir.createTempFile(<ArgumentList args>)`);
16     }
17   }
18
19   if(tempDirUsed) method = addMethodParameter(method, (FormalParameter) `@TempDir File tempDir
       ↪ `);
```

```
20
21   return method;
22 }
```

Listing 3.35: `TempDir` transformation code

Chapter 4

TestAXE's Efficiency and Limitations

In order to assess the efficiency of TestAXE, an empirical study has been conducted. By using the same data set built by Soares et Al. [3][1] and reverting to the commits immediately before their commits have been made, it was possible to reuse their knowledge of where the smells could be found on the selected code bases.

The empirical study consisted in applying TestAXE to the software projects analyzed by Soares et Al. in the aforementioned data set. In order to speed up the process, three Python scripts were written, and they can be verified on annexes I and II. The files containing known test smells were aggregated in a separate directory, which would be supplied as the `--input_dir` CLI parameter. This way, the analysis was made simpler as every file was contained in the same directory. After analyzing the changes made by TestAXE to each file, it was moved back to its original location on the original software project and the project's test suite was run, in order to assert that no errors were introduced, nor the behavior of the test method changed aside from the refactoring goals mentioned on chapter 3.

4.1 Empirical Study Results

The analysis of the study resulted in the confection of two tables, one associating each project name to the efficiency metrics that were computed, as well as some important information such as which transformations were applied successfully applied (true positives), which transformations were mistakenly applied (false positive), which transformations were not applied when they should (false negatives) and the git commits these

[1] https://github.com/easy-software-ufal/refactoring-test-smells-with-junit5/blob/main/Pull%20Requests/Pull%20Requests.csv

files were in. With these metrics, three measurements were computed to help visualize the efficiency of TestAXE: (a) *Precision*, which measures how often the transformations were correctly applied; (b) *Recall*, measuring how often were refactoring opportunities missed; and finally (c) *F1-Score* (F_1), for an overall performance. These measurements are computed as shown below:

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

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

$$F_1 = 2\frac{Precision \times Recall}{Precision + Recall}$$

Table 4.1, also shown on the paper presented in chapter 2, aggregates results by test smell and transformation. A reduced version of the table aggregating data by project is shown on table 4.2, showing only the numeric measurements. The full version can be found on github[2].

Smell	Transformation	TP	FP	FN	Precision	Recall	F_1
Assertion Roulette	`AssertAll`	97	0	41	1	0.70	0.83
Conditional Test Logic	`ConditionalAssertion`	1	0	2	1	0.34	0.5
Duplicate Assert	`ParameterizedTest`	0	0	21	0	0	0
Mystery Guest	`TempDir`	1	0	0	1	1	1
Test Code Duplication	`RepeatedTest`	4	0	0	1	1	1
Overall Result	-	103	0	64	1	0.62	0.76

Table 4.1: Efficiency metrics by smell and transformation

Results have been discussed on the paper reproduced in chapter 2.

4.2 Limitations and Possible Improvements

There are several improvement opportunities and limitations on TestAXE. Some of them will be discussed below grouped by the transformation that they belong.

4.2.1 Conditional Assertion

Limitation: detection of conditional test logic

`Conditional Assertion` currently only refactors test that contain nothing more than an `if` statement inside its body definition. Therefore, if there are any statements outside of that

[2] `https://github.com/alexander-p30/testaxe-empirical-study`

`if` statement, the transformation won't be applied. Listing 4.1 shows an example of a test method clearly contains conditional test logic, but won't be refactored by `Conditional Assertion`.

```
1 @Test
2 public void testMethod() {
3     int x = 5;
4     if(x == 5) runAssertion();
5 }
```

Listing 4.1: Test case presenting the *Conditional Test Logic* smell

This limitation is, however, by design, since it would require to implement finer control on what these statements outside the `if` statement are doing, and whether they influence or not the return value of the boolean expression of the `if` statement, which can even be impossible in some cases.

Improvement: do not wrap boolean methods inside another boolean method

This transformation always creates a new method which returns the `if`'s conditional expression. However, this is not necessary if that expression is the invocation of a boolean method defined inside the same `CompilationUnit`, as shown on listing 4.2.

```
1 public boolean isMonday() {
2     ...;
3 }
4
5 @Test
6 public void testMethod() {
7     if(isMonday()) runAssertion();
8 }
```

Listing 4.2: Test case using a boolean method as the conditional expression

4.2.2 ParameterizedTest

Improvement: detect assertion groups

Assertion groups that repeat the same assertion method could be refactored to a single invocation of each assertion method. This would require, however, some form of verification to make sure assertions do not interfere with each other. Listing 4.3 shows a test method with a group of two different assertions being repeated that could be refactored in such fashion.

```java
@Test
public void testMethod() {
    Assertions.assertEquals(1, 1);
    Assertions.assertEquals(2, 2);
    Assertions.assertEquals(3, 3);
    Assertions.assertNotNull(1);
    Assertions.assertNotNull(2);
}
```

Listing 4.3: Test method with repated assertions for different values

4.2.3 RepeatedTest

Improvement: refactor `for` statements with more diverse update statements

Currently, `RepeatedTest` only refactors test methods that contain for statements that update their iteration counter in a unary increment (`i++; i += 1`) or unary decremenet (`i --; i -= 1`). The transformation would be more frequently applied if other forms of updates were also to be accepted.

Project	TruePositive	FalsePositive	FalseNegative	Precision	Recall	F1
Aeron	1	0	0	1	1	1
Cas	2	0	0	1	1	1
Jenkins	2	0	0	1	1	1
kafka	0	0	1	-	0	-
Seata	3	0	0	1	1	1
Camel	0	0	0	-	-	-
Tutorials	0	0	0	-	-	-
Quarkus	1	0	0	1	1	1
MyBatisGenerator	1	0	10	1	0.091	0.17
Dropwizard	6	0	0	1	1	1
Dubbo	21	0	3	1	0.88	0.93
Janusgraph	1	0	0	1	1	1
Javaparser	19	0	0	1	1	1
Jetty	0	0	4	-	0	-
Jmeter	0	0	0	-	-	-
Jodd	0	0	1	-	0	-
Cryptomator	1	0	0	1	1	1
Flowable	1	0	0	1	1	1
java_design_patterns	0	0	1	-	0	-
Mybatis_plus	4	0	3	1	0.57	0.73
Spring_boot_starter	0	0	1	-	0	-
Okhttp	6	0	0	1	1	1
Opengrok	2	0	3	1	0.4	0.57
Reactor_core	7	0	1	1	0.88	0.93
Sentinel	0	0	9	-	0	-
Simplify	1	0	6	1	0.14	0.25
Spring_framework	1	0	0	1	1	1
Spring_boot_admin	2	0	0	1	1	1
Disruptor	0	0	0	-	-	-
Checkstyle	1	0	0	1	1	1
Lettuce_core	2	0	8	1	0.2	0.33
Mybatis	2	0	0	1	1	1
Jsoup	9	0	0	1	1	1
Spring_petclinic	4	0	1	1	0.8	0.89
Halo	1	0	0	1	1	1
Mindustry	2	0	0	1	1	1
Zaproxy	0	0	1	-	0	-
Zookeeper	0	0	11	-	0	-
Overall Result	103	0	64	1	0.62	0.76

Table 4.2: Efficiency metrics by project

Chapter 5

Conclusion

Software test code is as prone to the introduction of sub-optimal design and implementation as is software code. Sub-optimally implemented tests may hide problems on the software it is testing, as well as degrading the maintainability of the test suite itself. Attempting to solve that problem, TestAXE was conceived. Based on the refactoring proposals made by Soares et Al. [3] for a set of test smells, TestAXE receives a git repository directory and applies a series of transformations, of which the majority aim to refactor a specific test smell.

In order to assess the efficiency of the transformations' implementation, an empirical study was conducted. The results of this empirical study shows that TestAXE has a *Precision* of 1, which means that every transformation that was applied, was correct. Although, a *Recall* of 0.62 shows that there is a lot of room to improve how broad the cases in which the transformations are applied is. The overall F_1 score of 0.76 highlights a reasonable overall performance, considering that there are improvements to be made and some limitations, being existent by design, might fixed with some losses on Precision.

References

[1] Spadini, Davide, Fabio Palomba, Andy Zaidman, Magiel Bruntink e Alberto Bacchelli: *On the relation of test smells to software code quality.* Em *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, páginas 1–12, 2018. 1

[2] Bavota, Gabriele, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia e David Binkley: *Are test smells really harmful? an empirical study.* 20(4):1052–1094. 1

[3] Soares, Elvys, Marcio Ribeiro, Rohit Gheyi, Guilherme Amaral e Andre Medeiros Santos: *Refactoring test smells with JUnit 5: Why should developers keep up-to-date.* página 18. 1, 16, 38, 43

[4] Pizzini, Adriano: *Behavior-based test smells refactoring : Toward an automatic approach to refactoring eager test and lazy test smells.* Em *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, páginas 261–263, 2022. 1

[5] Peruma, Anthony, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni e Fabio Palomba: *Tsdetect: An open source test smells detection tool.* Em *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, página 1650–1654, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450370431. https://doi.org/10.1145/3368089.3417921. 1

[6] Palomba, Fabio, Andy Zaidman e Andrea De Lucia: *Automatic test smell detection using information retrieval techniques.* Em *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, páginas 311–322, 2018. 1

[7] Santana, Railana, Luana Almeida Martins, Larissa Rocha, Tássio Virgínio, Adriana Cruz, Heitor A. X. Costa e Ivan Machado: *RAIDE: a tool for assertion roulette and duplicate assert identification and refactoring.* Em *34th Brazilian Symposium on Software Engineering, SBES 2020, Natal, Brazil, October 19-23, 2020*, páginas 374–379. ACM, 2020. https://doi.org/10.1145/3422392.3422510. 1

[8] Lambiase, Stefano, Andrea Cupito, Fabiano Pecorelli, Andrea De Lucia e Fabio Palomba: *Just-in-time test smell detection and refactoring: The darts project.* Em *Proceedings of the 28th International Conference on Program Comprehension*, ICPC '20, página 441–445, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450379588. https://doi.org/10.1145/3387904.3389296. 1

[9] Aljedaani, Wajdi, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D. Newman, Abdullatif Ghallab e Stephanie Ludi: *Test smell detection tools: A systematic mapping study*. Em *Evaluation and Assessment in Software Engineering*, EASE 2021, página 170–180, New York, NY, USA, 2021. Association for Computing Machinery, ISBN 9781450390538. https://doi.org/10.1145/3463274.3463335. 1

[10] *Rascal*. https://www.rascal-mpl.org/, acesso em 2022-08-09. 1

[11] Parr, Terrence: *Language applications cracked open*. Em *Language Implementation Patterns: Create Your Own Domain Specific and General Programming Languages*, páginas 20–36. Pragmatic Bookshelf, 1ª edição, ISBN : 978-1-934356-45-6. 1, 2

[12] Lämmel, Ralf: *The notion of a software language*. Em *Software Languages: Syntax, Semantics and Metaprogramming*, páginas 1–50. Springer Cham, 1ª edição, ISBN 978-3-319-90798-7. 2, 14

[13] Parr, Terrence: *Walking and rewriting trees*. Em *Language Implementation Patterns: Create Your Own Domain Specific and General Programming Languages*, páginas 116–145. Pragmatic Bookshelf, 1ª edição, ISBN : 978-1-934356-45-6. 23

Annex I

Python Script to Clone Git Repositories Used in the Empirical Study

This python script clones all the repositories inside the *Pull Request* spreadsheet written by Soares et Al. It clones each project and creates a symlink to the files containing the test smell on the project's root folder.

```
import csv
import os
import subprocess

def git(*args):
    return subprocess.check_call(['git'] + list(args))

with open('Pull Requests.csv') as csv_file:
    csv_reader = csv.reader(csv_file)
    for i, row in enumerate(csv_reader):
        if i == 0: continue

        repository_name = row[0]
        print(f'\n==========={repository_name}===========')
        repository_folder = f'{i}_{repository_name.replace(" ", "_")}'
        if not os.path.isdir(repository_folder):
            pr_url = row[3]
            remote_repository_url = '/'.join(pr_url.split("/")[:-2])
            print(git('clone', remote_repository_url, repository_folder))
            print(f'\nRepository {repository_name} cloned!\n==========')
        else:
            print(f'Repository {repository_name} already present!\n==========')

```

```
24          smell = row[2].replace(' ', '_')
25          smelly_class_symlink_path = repository_folder + '/smell_class_' + smell
26          if not os.path.islink(smelly_class_symlink_path):
27              smell_class_path = os.getcwd() + '/' + repository_folder + row[1].replace('\\', '/
                    ↪ ')
28              print(f'Path: {smell_class_path}')
29              os.symlink(smell_class_path, smelly_class_symlink_path)
30              print(f'Symlink for class with smell in repository {repository_name} ' +
31                      f'created at {smelly_class_symlink_path}')
32          else:
33              print(f'Symink {smelly_class_symlink_path} already exists!')
```

Listing I.1: Python script to clone repositories

Annex II

Python Script to Aggregate Source Code Used in the Empirical Study

This python script aggregates every project's class test containing the known smell into a separate directory, with every file put into a sub directory containing the original project's name.

```
import csv
import os
import subprocess
import shutil

def git(*args):
    return subprocess.check_call(['git'] + list(args))

def mkdir(path):
    try:
        os.mkdir(path)
    except Exception:
        pass

aggregation_folder = '0_all_files'
mkdir(aggregation_folder)

with open('Pull Requests.csv') as csv_file:
    csv_reader = csv.reader(csv_file)
    for i, row in enumerate(csv_reader):
        if i == 0: continue

        repository_name = row[0]
        print(f'\n==========={repository_name}===========')
        repository_folder = f'{i}_{repository_name.replace(" ", "_")}'
```

```
26          smell_class_path_on_csv = row[1]
27          smell_class_path = os.getcwd() + '/' + repository_folder + smell_class_path_on_csv.
                ↪ replace('\\', '/')
28          smell_class_filename = smell_class_path.split('/')[-1]
29          destination_directory = f'{aggregation_folder}/{repository_folder}'
30          mkdir(destination_directory)
31          destination_path = f'{destination_directory}/{smell_class_filename}'
32          print(f'origin: {smell_class_path}')
33          print(f'destination: {destination_path}')
34          try:
35              shutil.copyfile(smell_class_path, destination_path)
36          except FileNotFoundError:
37              print(f'File not found! {smell_class_path}')
```

Listing II.1: Python script to aggregate all test files containing known test smells into a separate directory