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]

```
@Test
void first_test() {
  if (lastContainerId == null) {
    lastContainerId = genericContainer.getContainerId();
  } else {
     assertNotEquals(lastContainerId,
     genericContainer.getContainerId());
  }
}
```

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.