Automatic Refactoring Method to Remove Eager Test Smell

Adriano Pizzini
Graduate Program in Informatics – PPGIa - PUCPR - Curitiba, Paraná
Brazil
adriano.pizzini@ppgia.pucpr.br

Sheila Reinehr
Graduate Program in Informatics – PPGIa - PUCPR - Curitiba, Paraná
Brazil
sheila.reinehr@pucpr.br

Andreia Malucelli
Graduate Program in Informatics – PPGIa - PUCPR - Curitiba, Paraná
Brazil
malu@ppgia.pucpr.br

ABSTRACT

Unit tests are artifacts generated during the development process to identify software errors early. Unit tests can be affected by Test Smells (TS), which are defined as poorly designed tests due to bad programming practices implemented in unit test code. Problems caused by TS negatively impact the efficiency of developers and the understanding and maintenance of unit tests. Reducing the problems caused by TS can occur through refactoring activities. Despite the existence of studies that seek to identify the occurrence of TS in unit tests and proposals for semi-automatic refactoring, there are no studies related to the automatic removal of TS from the test repository. In this context, this study presents a method for automatic TS removal called Eager Test, which occurs when a test verifies more than one method of the code. The proposed method was evaluated through experiments, comparing the original version of a unit tests repository with its automatically refactored version. The results are promising, showing an Eager Test removal rate of 99,4% of unit tests in the repository without causing test errors or test fails. However, there was an increase in test execution time and of lines of code, resulting from the quantity of tests that were extracted. The proposed method supports improving the quality of unit tests and reduces the effort required by developers to remove Eager Test.

KEYWORDS

Test smell, Eager Test, Refactoring, Identification

ACM Reference Format:
Adriano Pizzini, Sheila Reinehr, and Andreia Malucelli. 2022. Automatic Refactoring Method to Remove Eager Test Smell. In *XXI Brazilian Symposium on Software Quality (SBQS '22), November 07–10, 2022, Curitiba, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3571473.3571478

1 INTRODUCTION

Unit tests (UT) are artifacts generated during the development process to act as the first line of defense in identifying errors in the system [1]. The UTs are maintained by the development team, evolve with the software, and may present design problems, known as Test Smells (TS), since the creation of the test classes [2].

SBQS '22, November 07–10, 2022, Curitiba, Brazil

ACM ISBN 978-1-4503-9999-9/22/11…$15.00
https://doi.org/10.1145/3571473.3571478

TS are indicators or symptoms of more serious design problems [3, 4] created from bad solutions [6, 7] and that violate good development practices in a given domain [8]. The origins of TS are associated with bad decisions by developers, which harm the quality of test code [9, 10].

Quality issues caused by TS negatively impact developers' efficiency [9, 10] and the understanding and maintenance of unit tests [11]. These problems increase the tendency to have errors in the test code and in the code of the system being tested (SUT) [10, 12].

The reduction of the problems caused by TS can be accomplished through refactoring operations [13]. In the case of manual refactoring's, the removal of problems caused by TS is a secondary effect of software maintenance and evolution activities [14–16]. Manual and semi-automatic refactoring's are human-dependent, and their execution can be hampered due to lack of knowledge of developers [17, 18], development team negligence [19], and development schedule pressure [20], among others. TS are long-lived [2], whose quantity increases as the system ages [12], suggesting that the development team is unaware of their existence, or does not have time to remove TS from unit tests. Furthermore, there is a possibility that the removal of TS, through human-dependent refactoring's, could include TS in the unit tests [13, 21].

Among the existing TS, the Eager Test (ET) is one of the three most widespread [17, 22–24]. Although there are studies on the identification of TS and studies related to semiautomatic refactoring, no studies were found related to the automatic removal of ET, which, unlike other types of refactoring, does not depend on the developer's approval to refactor the test. In this context, this article presents an automatic refactoring method for removing ET from the test class.

In addition to the introduction, this article is structured in 7 other sections. Section 2 presents the theoretical background. In Section 3, the related works are presented. In Section 4, an approach to the identification of ET is presented. Section 5 presents the proposed automatic refactoring method for ET removal. Section 6 presents the results obtained, and Section 7 discusses the relevance of the findings. Section 8 concludes the work with perspectives for future work.

2 THEORETICAL FOUNDATION

This section presents the theoretical background related to unit tests and their development. Furthermore, it describes test smells, their causes, and consequences, followed by the Eager Test (ET) smell.

2.1 Unit tests

Software users may be affected by flaws not identified during the development process. To identify flaws, the development team can

run manual or automated tests on the software. A test is defined by [25] as "an activity in which a system or component is performed under specified conditions, the results are observed or recorded, and an evaluation of some aspect of the system or component is made".

The purpose of test execution is to verify the software, to identify that it was developed correctly, and its validation, to identify that the correct software was developed [26]. Tests can be run at various levels of granularity. At the highest level of granularity, acceptance tests can be highlighted, which involve the use of software features from the end-user's point of view [26]. At an intermediate level, integration tests can be highlighted, which verify the functioning of a group of system modules [26]. At the lowest level of granularity, unit tests can be performed to verify software elements that can be tested separately [26].

Running unit tests (UT) increases test coverage and aims to reduce the chances that unidentified bugs will affect end users. In UT development, the team defines test cases in a test suite to identify bugs in the software. A test case is defined by [25] as "a set of test inputs, execution conditions and expected outputs, developed for a specific objective, such as exercising a certain path in a program or verifying a certain requirement".

The execution of the UTs follows the four steps described by [27]:

- Setup: The resources needed to perform a test are acquired.
- Stimulus: A stimulus from the test code is sent to the unit being tested.
- Verification: The test code queries the unit under test and fetches the stimulus result and reports the test result to the tester.
- Teardown: all used resources, such as created objects, are released from memory.

The setup step brings the SUT to an initial state by creating one or more objects to run a UT on a test case. After creating the objects, one or more SUT methods can be invoked to change the state of the SUT. This can be seen in Figure 1 in lines 73 and 74 for the "testCloning()" method and 86 and 87 for the "testEquals()" method, both from the AxisTest class of JFreeChart 1.5.3.

The SUT stimulus can be performed with the execution of one or more methods, as can be seen in line 75, with the execution of the "clone()" method, and in lines 91 and 93, with the execution of the "setVisible(" method).)". However, in the "testEquals()" method, the realization of stimuli did not occur on lines 79 to 81, only an equality check of the created objects (setup) on line 88.

Based on the unit tests presented in Figure 1, the possible interactions between the test steps defined by [27] are presented in Figure 2. From the execution of the initial setup step, the stimulus or verification steps can be performed, which can: a) return to the setup step, to create new objects; b) be followed by a new execution of the stage itself; c) be followed by the stimulus or verification step when the other was performed previously; d) be followed by the tearDown step that ends the test.

Despite being unlikely, test methods may contain only setup, stimulus, and teardown steps, without performing any verification, a situation described by the Empty Test smell.

```
71      @Test
72      public void testCloning() throws CloneNotSupportedException {
73          CategoryAxis a1 = new CategoryAxis("Test");
74          a1.setAxisLinePaint(Color.RED);
75          CategoryAxis a2 = (CategoryAxis) a1.clone();
76          assertNotSame(a1, a2);
77          assertSame(a1.getClass(), a2.getClass());
78          assertEquals(a1, a2);
79      }
80
81      /** Confirm that the equals method can distinguish all the re
84      @Test
85      public void testEquals() {
86          Axis a1 = new CategoryAxis("Test");
87          Axis a2 = new CategoryAxis("Test");
88          assertEquals(a1, a2);
89
90          // visible flag...
91          a1.setVisible(false);
92          assertNotEquals(a1, a2);
93          a2.setVisible(false);
94          assertEquals(a1, a2);
```

Figure 1: Example of unit tests with setup, stimulus, and verification steps.

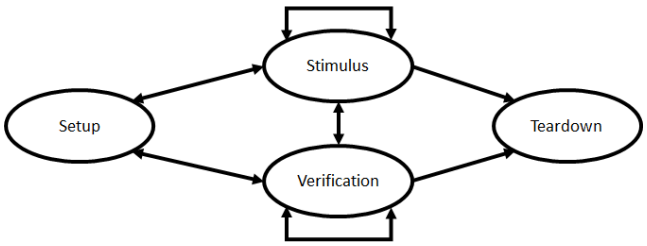

Figure 2: Interactions between UT steps [27]

2.2 Test smells

TS is defined as poorly designed unit tests [9], resulting from sub-optimal design choices applied by developers when implementing test cases [28]. TS is indicative of violations of good software development practices or symptoms of more serious design problems [3, 4].

The first description of TS was presented by [5], in an article in which eleven TS were described. There are other TS described in the literature, such as the rotten green test, described by [29], and a list of twelve TS described by [30]. The TS described in the literature have specific characteristics, identifiable causes, and their consequences affect software development.

The causes of TS are related to several factors, such as: lack of rules for UT development, as these rules are potentially less strict than those applied to the rest of the source code [31]; lack of coding standards for automated tests [20], which could serve as a guide for developers; and test generation tools [1, 23], as they seek to increase unit test coverage at the expense of quality.

In addition to these causes, the lack of knowledge about TS makes developers unable to identify TS, allowing a new TS to be inserted into the test in a maintenance activity [13, 21]. In addition, there is confirmation bias, which predisposes developers to confirm a desired behavior of the SUT, rather than trying to confirm the desired behaviors and identifying whether the SUT reacts appropriately to the unwanted behaviors [32]. Another cause of TS may be associated with the quality of the SUT code, indicating that the presence of smells in the SUT can influence the existence of TS, such as ET [2], described in the next section.

```java
public class XYTextAnnotationTest {

    /**
     * Confirm that the equals method can distinguish all the required
       fields.
     */
    @Test
    public void testEquals() {
        XYTextAnnotation a1 = new XYTextAnnotation("Text", 10.0, 20.0);
        XYTextAnnotation a2 = new XYTextAnnotation("Text", 10.0, 20.0);
        assertTrue(a1.equals(a2));

        // text
        a1 = new XYTextAnnotation("ABC", 10.0, 20.0);
        assertFalse(a1.equals(a2));
        a2 = new XYTextAnnotation("ABC", 10.0, 20.0);
        assertTrue(a1.equals(a2));

        // x
        a1 = new XYTextAnnotation("ABC", 11.0, 20.0);
        assertFalse(a1.equals(a2));
        a2 = new XYTextAnnotation("ABC", 11.0, 20.0);
        assertTrue(a1.equals(a2));
```

Figure 3: Test method affected by Eager Test [9]

The consequences of TS are associated with difficulties in understanding and maintaining the tests [11]. These difficulties can contribute to the introduction of errors in the UT that affect the SUT. Furthermore, the presence of TS increases the tendency of errors in the SUT, such as memory leaks and behavior change [10, 12].

2.3 Eager Test smell

ET occurs when a test method checks several methods of the object being tested [4]. In unit tests, checks are performed using specific instructions known as assertions. Thus, an ET-affected method must contain more than one check.

The effect of the ET's presence in a test method is to increase its responsibilities, resulting in larger and more complex test methods. Furthermore, failure to perform a check can make it difficult to understand the cause of the failure [9]. To reduce the quantity of responsibilities of a test method, one can split the test method to remove the ET and specialize its responsibilities [14, 22]. This division decreases the quantity of instructions executed for each method.

An example of a test method affected by ET can be seen in the example given by [9] and shown in Figure 3. In this test method, a sequence of verifications can be identified, represented by the instructions "assertTrue" and "assertFalse," performed after creating objects.

From the initial definition of ET, variations of the original definition emerged, which are described below:

- A test method that executes several methods of the object to be tested [33].
- A test method that attempts to test various behaviors of the tested object [9].
- A test case that checks or uses more than one method of the class being tested [17].
- A test method contains several calls (calls) to several production methods [14].

The definitions of [33], [17] and [14] incorporate the concept of the count of methods invoked, differing from the original ET definition that emphasizes the need to verify the stimuli of several SUT methods. Identifying the presence of ET by counting of invoked methods allows different interpretations of which SUT method invocations are considered. For example, all the methods invoked in the class under test can be counted, only the methods invoked in the same instance under test, or only the methods whose return value is eventually used within an assertion [17] can be counted. Furthermore, a test method can invoke methods declared in the ancestor class of the class under test, increasing the number of interpretations.

3 RELATED WORKS

The related works are divided into two categories that will be presented below: those related to the identification of ET and those related to the refactoring of ET.

3.1 Eager Test smell identification

This section presents, in chronological order, the works related to the identification of TS, restricting the relationship to those that identify the ET.

In 2007, [34] proposed a set of metrics for the identification of TS General Fixture and ET. In the case of ET, the identification of the TS was performed using the Production-Type Method Invocations (PTMI) metric, which represents the count of invocations of SUT methods by the test which, if greater than three, indicates that ET was identified. The authors validated the defined metric through manual inspection in the ArgoUML project.

In 2012, [33] carried out a study to analyze the distribution of TS in public repositories and developed a tool to assist in the identification of TS. The rule applied to the ET identification was the existence of a test that uses (i.e., invokes) more than one method of the SUT class being tested. The authors carried out two empirical studies to assess the impact of TS on maintainability, the first to assess the distribution of TS in software repositories and the second to assess the impact of TS on maintenance activities.

In 2018, [28] made available the Textual AnalySis for Test smEll detection (Taste) tool that uses information retrieval techniques to identify three types of TS instead of using metrics related to the count of SUT method invocations by the test. The identification of the ET is performed through the similarity analysis of the methods invoked by the test. The authors carried out an empirical study to assess the effectiveness of the tool. The evaluation results indicated a 44% increase in effectiveness in the identification of STs.

In 2020, [14] presented the open-source tool TsDetect that uses a set of detection rules to identify the presence of TS in a test class. In the authors' proposal, ET occurs when a test method invokes multiple methods of the class being tested. To make the identification, the proposal associates the test class with the production class to identify which production class methods are being invoked. The authors performed an empirical study to identify the distribution of nineteen TS in a randomly generated dataset from 656 open-source Android applications.

In 2020, [23] presented JNose, a tool to identify TS in a test class that uses the rules defined by [14]. The JNose tool presents details about the identified TS, such as the class of tests and the lines affected by the identified TS. The authors used the tool to analyze the commons-io project repository and compared the presented tool with those described in the literature.

Table 1: Comparison of ET identification proposals

Authors	ET identification proposal
Rompaey et al., 2007 [34]	Metrics, count of methods invoked
Bavota et al., 2012 [33]	Count of methods invoked
Palomba et al., 2016 [28]	Text similarity of invoked methods
Peruma et al., 2020 [14]	Rules, invoked methods
Virgínio et al., 2020 [23]	Rules, invoked methods
This proposal	Cycles of stimulus-checks

The proposals described above use the methods invoked and/or the respective count of invocations of SUT methods by the test method as ET identification criteria. This type of approach does not consider which methods can be invoked as part of the test setup step, in which the resources for test execution are being obtained. In this case, a test method may be mistakenly identified as affected by ET, as no verification is being performed in this situation. Furthermore, it is not possible to determine based solely on the invoked SUT methods, which parts of the test method code should be extracted. Another point to be mentioned is the fact that the proposals identify the production class related to the test by removing the term "Test" at the beginning or end of the test file. In these proposals, the identification of the test-related production class is necessary to identify whether an invoked method is from the production class associated with the test class or from another. However, in cases where unit tests are created with different names to represent specific test scenarios, these proposals fail to associate the production class with the test class. This is a case that can be observed, for example, with the test files AreaChartTest and BarChartTest in JFreeChart, which test an object of class JFreeChart created through the object factory of the ChartFactory file.

Unlike the proposals described, this article proposes the identification of the presence of ET based on the existence of stimulus-verification cycles within a test method, as described in section 4.1. The existence of stimulus-check cycles within a test method is in line with the definition of ET given by [4] and the cycles can be used to delimit the instructions that need to be extracted for new methods. Table 1 presents the works related to the respective ET identification proposal.

3.2 Eager Test smell refactoring

When searching the scientific databases, only one ET refactoring proposal was found, the proposal by [35], which describes a plugin for the semiautomatic refactoring of three TS: General Fixture, Eager Test, and Lack of Cohesion of Test Methods. The proposed plugin acts at the time of commit, which is when the developer submits code to the repository. At the time of submission, the developer is alerted to the presence of the ET and evaluates and approves or rejects the refactoring proposal elaborated by the approach. The proposal of [35] is human-dependent; that is, the application of the approach can be harmed by the factors described in section 2.1, and it was not evaluated, which makes the analysis and comparison with new proposals difficult and does not consider that the removal automatic should keep unchanged the order of execution of the extracted code for other tests.

The ET refactoring proposal presented in this article is based on automatic refactoring, that is, without human dependence, of all unit tests existing in a software project. Furthermore, it considers that the test methods created from the refactoring must be executed in the same order as the original test method so that the interactions that existed in the method before the refactoring can be preserved.

4 IDENTIFYING THE BEHAVIOR OF UNIT TESTS AND THE PRESENCE OF EAGER TEST SMELL

To be able to automatically refactor a test class, it is necessary to identify the behavior of the unit tests and identify the presence of ET in each of them. In this work, the identification of the unit tests behavior was performed using the public repository of the JFreeChart framework, version 1.5.3, known to be affected by ET [8]. The behavior analysis of the tests was used to devise an ET identification algorithm to be used with the refactoring method, described in section 5.

4.1 Identification of test behavior

The identification of the behavior of the UTs of a repository was performed by inspecting the repository's test classes. As the repository presents a considerable quantity of test classes, a manual inspection would be unfeasible. To facilitate inspection, a tool was developed to instrument the tests, so that it was possible to identify the type and execution order of each of the instructions contained in the unit tests.

In the test classes, two methods were added to log the start of the test (setup) and the end of the test (tearDown) to standardize the display of test execution. This inclusion allowed to aggregate all the tests of a class with the delimitation of the beginning and the end of the execution of the instructions of a class of tests. In addition, the tests were instrumented to record the execution of all instructions contained in the test method. After instrumentation, all instrumented tests were executed, and the execution sequence of test instructions was stored in a database.

To analyze the behavior of the tests, a file was generated in CSV format, in a format compatible with process analysis tools, in which each instruction executed by the test method is considered an activity. The actions performed are described in Table 2. The actions performed by 348 test classes were exported to a file.

The CSV file was imported into the Disco process analysis program (http://www.fluxicon.com) to display the sequence of execution of the test instructions, resulting in the map shown in. Each

Table 2: Description of actions recorded through test and SUT instrumentation

Authors	ET identification proposal
Rompaey et al., 2007 [34]	Metrics, count of methods invoked
Bavota et al., 2012 [33]	Count of methods invoked
Palomba et al., 2016 [28]	Text similarity of invoked methods
Peruma et al., 2020 [14]	Rules, invoked methods
Virgínio et al., 2020 [23]	Rules, invoked methods
This proposal	Cycles of stimulus-checks

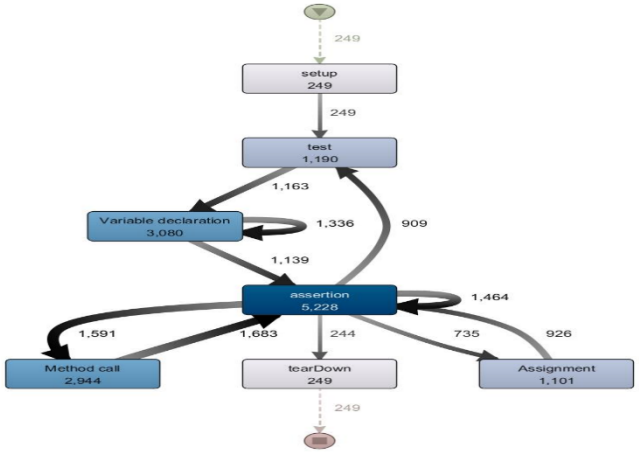

Figure 4: Map of JFreeChart (v1.5.3) unit tests execution before refactoring

action performed is shown in Figure 4 as a node, containing the action's name and its count of occurrences. The arrows indicate the sequence of activities execution and have the associated count of occurrences. To facilitate the understanding of the map, the low expression paths were filtered, showing the instructions of 249 test classes.

In Figure 4 from the execution of a test method, the instructions executed are the creation of objects ("Variable declaration"), followed by the execution of a verification (assertion), indicating which verifications are performed after setup.

After the first scan performed, four behaviors were identified:

1. The end of the test: which can be observed on the path from "assertion" to "tearDown", with 244 occurrences.

2. The execution of other checks: which can be observed in the path from "assertion" to "assertion", with 1,464 occurrences.

3. The execution of assignments: can be observed through the path from "assertion" to "Assignment", with 735 occurrences.

4. The execution of SUT methods: which can be observed through the path from "assertion" to "Method call", with 1,591 occurrences.

The first behavior described indicates that the check was the last statement executed in the test class. The second behavior indicates that the developer performed one or more checks in sequence to check the setup, the execution of a SUT method, or assignments, because the check runs after one of these three situations. However, the third and fourth behaviors identified show that the developer has included one or more statements within the same test method to perform operations that need at least one re-verification.

Therefore, within the same test method, the existence of a sequence of instructions, followed by one or more checks that precede another type of instruction (which are not checks), proves that the developer can use the same test method to do numerous checks. Within this context, it is assumed that the developer stimulates the SUT and makes the necessary checks to identify whether the stimulus resulted in the expected behavior of the SUT.

The third and fourth behavior patterns described above suggest that developers, after the setup verification: stimulate SUT methods and carry out verifications in sequence; or they stimulate the SUT, store the stimulus result in local variables and perform the necessary checks.

4.2 Eager Test smell identification

Based on the analysis of the behavior of the tests and the SUT described in section 3.1, an algorithm was constructed to identify whether ET affects a test method. This algorithm was created to identify if the test method instructions contain at least two cycles of non-verification instructions followed by verification instructions. The reason for the creation is the fact that current approaches to ET identification consider the count of SUT methods executed by a test as an indicator of ET presence. Thus, they cannot be incorporated into the implementation of the proposed method since a SUT method can be invoked before the stimulus, be the stimulus, or be invoked after the SUT stimulus. The identification of the method invoked to stimulate the SUT to verify its behavior is not a trivial task, as it depends on the analysis of the test scenario.

Even considering that a method represents the behavior of an object, taking the count of the quantity of methods executed as an indicator of the existence of ET can produce incorrect results.

As an example, in the "testCloning()" method shown in Figure 1, the test method stimulates the "clone()" method and performs three (3) checks. However, the previous instruction is the execution of the "setAxisLinePaint" method. In this case, the execution of two SUT methods is an indicator of ET presence according to some ET identification approaches. However, in this case, executing the method on line 75 is part of the object's setup and not a stimulus to be checked.

From the definition of [4] and the behavior of the unit tests presented in section 3.1, this research considers that ET affects a test method when it is formed by alternating non-verification instructions with forming verification instructions, at least two sequences or cycles of alternations.

An example of the existence of stimulus and verification cycles can be seen in the "testEquals()" method in Figure 1, which creates objects on lines 86 and 87, performs verification on line 88, stimulates the object on line 91, performs verification on line 92, and re-stimulate and check on lines 93 and 94. In this example, the test method stimulates and tests the SUT more than once.

The proposed algorithm analyzes the instructions of a method to locate the checks performed (assertions) and identify if there are executions of methods (stimuli) after the first check is found.

5 AUTOMATIC REFACTORING METHOD FOR EAGER TEST SMELL REMOVAL

The automatic refactoring method developed is based on the findings of [13], which identified that the removal of the Eager Test often occurs in conjunction with a refactoring operation of the Extract Method type, in which the parts of the test affected by the smell are extracted to compose a new test method. The development identification algorithm is executed for each test method in the repository, and if the return is that the test method is affected by ET, the refactoring is performed with the application of the proposed method.

The proposed method is composed of six steps: identifying the variables declared in the test, converting declarations of local variables into assignments, converting declared variables into attributes, updating references, extracting test(s), and updating the test execution order. The first four steps of the method are related to the extraction of local variables from a test affected by the ET smell, and the last two are associated with the extraction and organization of the tests created, as described below:

Identify the local variables declared in the test: it consists of identifying the local variables that are declared in the test and referenced by all the code snippets that will be extracted to compose new tests. As shown in Figure 5, there are variable declarations on lines 112, 114, 117, 119, 122, and 123.

Convert local variable declarations to assignments: Variable declarations that have initialization are converted to assignment statements. To avoid compilation errors, the data type of the local variable is used in the attribute declaration. For example, the code "CategoryPlot plot = (CategoryPlot) this.chart.getPlot();" shown on line 122 of Figure 5 will be converted to "plot = (CategoryPlot) this.chart.getPlot();". The initialization of the variables is kept in the original position of the method so that the sequence of execution of the instructions is not altered.

Convert identified variables into attributes: it consists of creating static attributes with the names and data types used in the declaration of local variables. The creation of static attributes is necessary so that the state of objects can be maintained after running the tests. The local variable declaration described above will cause the following attribute to be defined: "static CategoryPlot plot".

Update references: if there is a need to change the name of the attributes to avoid duplication of names (which would cause syntax and semantic errors), step 4 is performed, which locates the code points in the method that references the variables that have been renamed and replace the reference to the new name used in the attribute.

```
110
111     // create a dataset...
112     Number[][] data = new Integer[][] {{-30, -20}, {-10, 10}, {20, 30}};
113
114     CategoryDataset newData = DatasetUtils.createCategoryDataset("S",
115             "C", data);
116
117     LocalListener l = new LocalListener();
118     this.chart.addChangeListener(l);
119     CategoryPlot plot = (CategoryPlot) this.chart.getPlot();
120     plot.setDataset(newData);
121     assertEquals(true, l.flag);
122     ValueAxis axis = plot.getRangeAxis();
123     Range range = axis.getRange();
124     assertTrue(range.getLowerBound() <= -30,
125             "Expecting the lower bound of the range to be around -30: "
126             + range.getLowerBound());
127     assertTrue(range.getUpperBound() >= 30,
128             "Expecting the upper bound of the range to be around 30: "
129             + range.getUpperBound());
130
131 }
```

A
B
C

Figure 5: Unit test from the BarChartTest.java test class affected by ET

Extract tests: consists of extracting the instructions from the second stimulus-verification cycle, with which test methods will be created. The C highlight in Figure 5 shows the starting point of the second code block that will be extracted into a new test method. The extraction is based on the identification of code blocks delimited by assertions that are instructions used to make verifications. In case there are two or more assertions in sequence, the extraction will be performed after the last assertion. The instructions contained in the extracted cycles are removed from the original test method, leaving only the instructions from the first cycle.

Update the test execution order: it consists of annotating the original method and the extracted methods with "Order(x)", in which x is a sequential number, incremented with each new test extracted and used to determine the execution order of the extracted test within the test class. This step is necessary to ensure that the instructions of the extracted methods are executed in the same sequence as the original method. As the JFreeChart test repository does not contain test classes in which the order has been previously defined, the initial implementation of the refactoring method annotates the test classes to execute the test methods according to the order defined in the refactoring.

6 EVALUATION AND RESULTS

The evaluation of the proposed method was conducted through experimentation, having generated a copy of the original refactored repository automatically. The following hypotheses were defined:

- Null hypothesis ($H1_0$): the quantity of failures of the refactored tests is equal to the quantity of failures of the unrefactored tests.
- Alternative hypothesis ($H1_1$): the quantity of failures of refactored tests is different from the quantity of failures of unrefactored tests.
- Null hypothesis ($H2_0$): the execution time of refactored tests is equal to the execution time of unrefactored tests.
- Alternative hypothesis ($H2_1$): the execution time of refactored tests is different from the execution time of unrefactored tests.
- Null hypothesis ($H3_0$): the branch coverage of refactored tests is smaller than branch coverage of unrefactored tests.
- Alternative hypothesis ($H3_1$): the branch coverage of the refactored tests is greater than or equal to the branch coverage of the unrefactored tests.

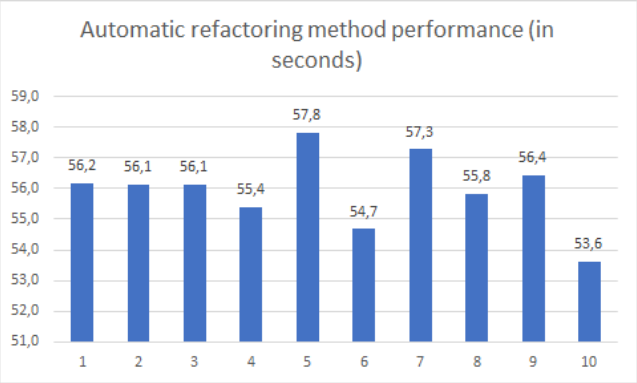

Figure 6: Execution time of the proposed refactoring method in 10 executions (in seconds)

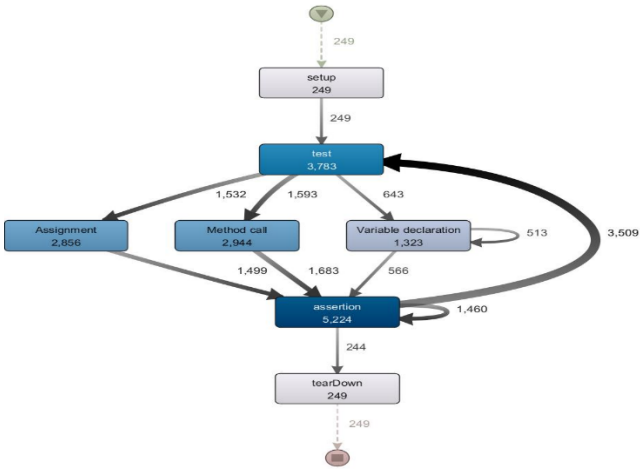

Figure 7: Behavior of unit tests after refactoring

From there, tests were performed on the refactored and non-refactored repositories. The evaluation of hypotheses H1 and H2 was performed using the Wilcoxon signed rank test with a significance level of 5%. For hypothesis H1, a value of 0 was assigned to classes executed without failures/errors and a value of 1 to classes executed with failures/errors. For hypothesis H2, the execution times of each test class were collected from the Maven log file. To evaluate the branch coverage, it was necessary to include the JaCoCo coverage analysis framework, version 0.8.8. Test classes with errors after refactoring were removed from both repositories. Then the two projects were compiled, and the coverage logs were imported. The evaluation of hypothesis H3 was performed using the Wilcoxon signed rank test, one-tailed and with a significance level of 5%.

To compute the execution time of the proposed method, all tests from each repository were executed ten times, considering the average execution time of each test class. Figure 6 shows the time for each of the ten executions of the proposed refactoring method. The refactoring performance test was performed on a computer with Windows 10, Intel(R) Core(TM) i7-7700HQ 2.80GHz CPU, 16.0 GB of RAM, and 240GB Sandisk SSD, resulting in an average of 56 seconds.

As can be seen in Table 3, 350 test classes were found, of which 38 were not affected by ET. We refactored 312 classes that contained at least one method affected by ET. Of the refactored classes, 17 presented failures/errors after refactoring. Twenty-three nested/inner classes were found and ignored.

Table 4 shows the quantity of test methods affected by ET in the repository, followed by the quantity of test methods before refactoring, that is, from the original repository, and the quantity of test methods after refactoring.

Table 5 shows the quantity of logical lines of code, that is, disregarding comments and blank lines, before and after refactoring.

For the evaluation of the defined hypotheses, all classes of tests were considered (N=350). The quantity of errors and failures in the execution of the refactored tests was higher than the quantity of errors and failures in the execution of the unrefactored tests. Wilcoxon's sign-of-rank test indicates that the quantity of errors and failures is statistically significant (V = 153, p-value = 4.201e-05), resulting in the rejection of the null hypothesis $H1_0$.

The comparison of the execution time of the unrefactored tests with the refactored ones indicated that the execution time of the last ones was superior to the first ones. Wilcoxon's sign-of-rank test indicated that the difference in execution time is statistically significant (V = 14672, p-value = 1.157e-15), resulting in the rejection of the null hypothesis $H2_0$.

In the evaluation of hypothesis H3, the application of the Wilcoxon signed rank test (V = 0, p-value = 1) suggests that the null hypothesis H30 cannot be rejected. The analysis of the data collected from the two repositories showed that the mean and median coverage of branches are, respectively, 17.59 and 7 in both.

Figure 7 shows the sequence of instructions executed for each test after executing the refactoring method. The quantity of occurrences is different from the unrefactored version due to the change in the behavior of tests after automatic refactoring and the filters applied to avoid displaying low expression paths. The change in the behavior of the tests can be observed in the existence of three execution paths after the beginning of the test ("Test" node): the execution of assignments, invocation of methods, or declaration of variables. In the three possible paths, the test method is terminated after performing one or more checks.

Table 3: Detail of existing test classes

Quantity of test classes			
Found	**Not affected by ET**	**Refactored**	**With errors after refactoring**
350	38	312	17

Table 4: Comparison of the quantity of methods affected by ET and the total quantity of methods before and after refactoring

Quantity of active test methods		
Before refactoring and affected by ET	**Before refactoring (all)**	**After refactoring**
1047	2516	6516

Table 5: Quantity of logical lines of code before and after refactoring

Quantity of logical lines of code	
Before refactoring	**After refactoring**
32.745	43.827

```
86      static Number[][] data;
87      static CategoryDataset newData;
88      static LocalListener l;
89      static CategoryPlot plot;
90      static ValueAxis axis;
91      static Range range;
92
93      @Test
94      @Order(1)
95      public void testReplaceDataset() {
96          data = new Integer[][] { { -30, -20 }, { -10, 10 }, { 20, 30 } };
97          newData = DatasetUtils.createCategoryDataset("S", "C", data);
98          l = new LocalListener();
99          this.chart.addChangeListener(l);
100         plot = (CategoryPlot) this.chart.getPlot();
101         plot.setDataset(newData);
102         assertEquals(true, l.flag);
103     }
104
105     @Test()
106     @Order(2)
107     void testReplaceDataset_1() {
108         axis = plot.getRangeAxis();
109         range = axis.getRange();
110         assertTrue(range.getLowerBound() <= -30, "Expecting the lower bour
111         assertTrue(range.getUpperBound() >= 30, "Expecting the upper bounc
112     }
```

Figure 8: Refactored test method

Figure 8 shows the test method shown in Figure 5 after refactoring. As can be seen in this figure, the local variables were converted into static attributes (lines 86 to 91) and the second stimulus-verification cycle was extracted to compose the "testReplaceDataset_1()" method.

7 DISCUSSION OF RESULTS

Test execution time in the original repository went from 3.9 seconds to 4.8 seconds in the refactored repository, an increase of approximately 23%. The execution time in the original repository is higher than the 2.752s identified by [36]. The fact that the execution time in this experiment is different from that found by [36] can be attributed to the evolution of the system. Considering only the times of this experiment, the increase in the execution time in the refactored repository corroborates the warning of [4] that the increase in the quantity of test methods results in a greater quantity of executions of the setup/tearDown methods.

However, this increase is not exclusively due to the increase in the quantity of executions of setup methods, as only 24 test classes have a specific setup method (methods annotated with "BeforeEach"). Using attributes instead of local variables changes the way memory is allocated and managed. In addition, the creation of test methods with the code snippets extracted from the original method increases the number of times that the test framework performs the context switch, that is, the execution of instructions to pass a test method for another.

An expected result of applying the method was an increase in the quantity of test methods and logical lines of code. This increase is due to the creation of test methods with more specific responsibilities, that is, that check smaller parts of the SUT code when compared to the unrefactored test method. Reducing the size of test methods by extracting ET-affected chunks may improve test readability and maintainability. This possibility of improvement is related to the fact that the size of a code is a significant predictor in calculating readability [37] and that code that is easy to read tends to be easier to understand [38].

Regarding the proposed refactoring method, even if the order of execution of the instructions of the unrefactored test is maintained after refactoring, it is not possible to guarantee that the tests continue to be executed without errors or test failures.

The analysis of the 17 test classes that presented errors after applying the method made it possible to identify the cause of the test failure: they are test classes that act as listeners, that is, they are communicated by the class being tested that there was an event that needs to be checked by the test class. For example, the CategoryMarkerTest test class implements the MarkerChangeListener interface, overriding the necessary methods and declaring a non-static attribute called "lastEvent" of type "MarkerChangeEvent". In the "testGetSetKey" method of this class, shown in Figure 9, there is a check on line 176 to identify whether the last event corresponds to what was expected after the execution of a stimulus by executing the following instruction: "assertEquals(m, this.lastEvent.getMarker());".

After refactoring, the value of the "lastEvent" attribute on line 176 is "null", differing from the value of the unrefactored version. The behavior change of the tests causes an exception of type NullPointerException to be thrown because of the context switching of the tests performed by the JUnit framework that resets the value of the attributes.

The generalization of the described case suggests that test methods that somehow depend on non-static attributes declared in the test class itself cannot be automatically refactored.

Thus, identifying situations in which automatic refactoring cannot be performed is important to avoid errors in refactored tests that impair test coverage. Based on the coverage data collected, the proposed method can maintain the test coverage level.

Regarding the instruction execution pattern, a meaningful change was identified after the refactoring. For example, from the beginning of the test, there was an increase in the quantity of assignments, from 1,101 to 2,856, because of the substitution of declarations of local variables for declarations of attributes.

```
168     @Test
        public void testGetSetKey() {
170         CategoryMarker m = new CategoryMarker("X");
171         m.addChangeListener(this);
172         this.lastEvent = null;
173         assertEquals("X", m.getKey());
174         m.setKey("Y");
175         assertEquals("Y", m.getKey());
176         assertEquals(m, this.lastEvent.getMarker());
177
178         // check null argument...
179         try {
180             m.setKey(null);
181             fail("Expected an IllegalArgumentException for null.");
182         }
183         catch (IllegalArgumentException e) {
184             assertTrue(true);
185         }
186     }
```

Figure 9: Example of a method that checks from attributes declared in the tests class

```
108     @Test
        public void testCloning() throws CloneNotSupportedException {
110         CategoryTextAnnotation a1 = new CategoryTextAnnotation(
111                 "Test", "Category", 1.0);
112         CategoryTextAnnotation a2 = (CategoryTextAnnotation) a1.clone();
113         assertTrue(a1 != a2);
114         assertTrue(a1.getClass() == a2.getClass());
115         assertTrue(a1.equals(a2));
116     }
```

Figure 10: Example of a not refactored test method

In the refactored repository, the quantity of SUT methods invoked ("Method call") after an assertion ("assertion") was reduced from 1,591 to 0. This reduction was due to the segmentation of stimulus-verification cycles, that is, from the extraction of instructions from each cycle in test methods, indicating that the test method is finished after executing a verification or a sequence of verifications.

In addition, the quantity of local variables declared in test methods has been reduced from 3080 to 1323. The permanence, after refactoring, of test methods that declare variables associated with the SUT object being tested is due to the existence of unrefactored test methods, as they are not composed of stimulus-verification cycles.

This situation can be seen in Figure 10, which shows the "testCloning()" method of the "CategoryTextAnnotationTest" class of the refactored repository, which does not present two or more stimulus-verification cycles. As shown in this figure, the first instruction is executed to create an object (i.e., setup), followed by a stimulus and three checks.

Finally, after running the proposed refactoring method, running a check on the refactored test methods results in two conditions: performing another check or terminating the test. The first condition refers to the fact that the test method can perform N checks in sequence, which, according to the proposed method, must remain unchanged. The second condition refers to the definition of [4], that a unit test should not verify more than one SUT method, that is, after performing the set of verifications necessary after performing a stimulus, the method ends of test.

8 FINAL CONSIDERATIONS AND FUTURE WORKS

This article introduced an automatic refactoring method to remove the Eager Test (ET) smell in a unit tests class. The method was evaluated with an open-source repository known to be affected by ET, showing promising results.

The main finding of this research is the confirmation of the possibility of automatic removal of ET from a test class to produce smaller and with more specific responsibilities. The refactored tests tend to be more readable and maintainable than the original ones. In addition, our findings suggests that it is possible to maintain the coverage after the refactoring. However, specialization of test methods leads to increased test execution time due to test context switching and memory allocation mode change.

The results described are relevant to support the development of automatic refactoring tools that contribute to improving the quality of unit tests without the need for intervention by the development team. Identifying cases in which refactoring cannot be performed automatically serves as a source of reference for future proposals for automatic refactoring that use the extraction of code snippets from the test method. In addition, manual refactoring operations can use the proposed method as a guide for the refactoring process.

As perspectives for future work, we intend to investigate the effects of applying the proposed method in other repositories and the impact of ET's automatic refactoring on the development team.

ACKNOWLEDGMENTS

The authors thank for the financial support Pontifícia Universidade Católica do Paraná (PUCPR), Instituto Federal Catarinense (IFC) via grant number 057/IFC/REITORIA/2019 and CAPES.

REFERENCES

[1] Grano, Giovanni. 2019. A New Dimension of Test Quality: Assessing and Generating Higher Quality Unit Test Cases. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China. ACM, New York, NY, USA, 5 pages.

[2] Tufano, Michele; Palomba, Fabio; Bavota, Gabriele; Di Penta, Massimiliano; Oliveto, Rocco; De Lucia, Andrea; Poshyvanyk, Denys. An Empirical Investigation into the Nature of Test Smells. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016). Association for Computing Machinery, New York, NY, USA, 4–15. https://doi.org/10.1145/2970276.2970340

[3] Yamashita, Aiko. Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. Empirical Software Engineering (2014) 19:1111–1143.

[4] Sousa, Leonardo da Silva. Spotting Design Problems with Smell Agglomerations. In: IEEE/ACM 38th IEEE International Conference on Software Engineering Companion, 2016.

[5] Van Deursen, A.V., Moonen, L., Bergh, A.V.D., Kok, G., 2001. Refactoring test code. In: Marchesi, M. (Ed.), Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001), University of Cagliari, pp. 92–95.

[6] Khomh, Foutse; Vaucher, Stephane; Guéhéneuc, Yann-Gaël; Sahraoui, Houari. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. The Journal of Systems and Software 84 (2011) 559–572.

[7] Arnaoudova, Venera; Di Penta, Massimiliano; Antonio, Giuliano; Gueheneuc, Yann-Gael. 2013. A New Family of Software Anti-Patterns: Linguistic Anti-Patterns. In: Proceedings of 17th European Conference on Software Maintenance and Reengineering.

[8] Sharma, Tushar; Fragkoulis, Marios; Spinellis, Diomidis. Does Your Configuration Code Smell? 2016. In: IEEE/ACM 13th Working Conference on Mining Software Repositories , Austin, TX, 2016 pp. 189-200.

[9] Garousi, Vahid; Küçük, Barıs. Smells in software test code: A survey of knowledge in industry and academia. The Journal of Systems and Software 138 (2018). pp52–81. https://doi.org/10.1016/j.jss.2017.12.013

[10] Spadini, Davide; Schvarcbacher, Martin; Oprescu, AnaMaria; Bruntink, AnaMaria; Bacchelli, Alberto. 2020. Investigating Severity Thresholds for Test Smells. In: 17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387453

[11] Bavota, Gabriele; Qusef, Abdallah; Oliveto, Rocco; De Lucia, Andrea; Binkley, Dave. Are test smells harmful? An empirical study. Empirical Software Engineering, pg1052–1094 (2015)

[12] Qusef, Abdallah, Elish, Mahmoud O.; Binkley, David. Exploratory Study of the Relationship Between Software Test Smells and Fault Proneness. IEEE Access. vol 7, 2019. https://doi.org/10.1109/ACCESS.2019.2943488

[13] Palomba, F.; Zaidman, A. Does Refactoring of Test Smells Induce Fixing Flaky Tests? 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 1-12, https://doi.org/10.1109/ICSME.2017.12

[14] Peruma, Anthony; Almalki, Khalid; Newman, Christian D.; Mkaouer, Mohamed Wiem; Ouni, Ali; Palomba, Fabio. 2020. tsDetect: An Open Source Test Smells Detection Tool. In: Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)

[15] Alomar, Eman Abdullah; Peruma, Anthony; Mkaouer, Mohamed Wiem; Newman, Christian; Ouni, Ali; Kessentini, Marouane. How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation. Expert Systems with Applications, volume 167, 2021, https://doi.org/10.1016/j.eswa.2020.114176

[16] Habchi, Sarra; Moha, Naouel; Rouvoy, Romain. Android code smells: From introduction to refactoring. The Journal of Systems & Software 177 (2021) 110964. https://doi.org/10.1016/j.jss.2021.110964

[17] De Blesser, Jonas; Di Nucci, Dario; De Roover, Coen. Assessing Diffusion and Perception of Test Smells in Scala Projects. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019, pp. 457-467

[18] Santana, Railana; Martins, Luana; Rocha, Larissa; Virgínio, Tássio; Cruz, Adriana; Costa, Heitor; Machado, Ivan. 2020. RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring. In Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES '20). Association for Computing Machinery, New York, NY, USA, 374–379.

[19] Daka, Ermira; Fraser, Gordon. A Survey on Unit Testing Practices and Problems. In: IEEE 25th International Symposium on Software Reliability Engineering, 2014, pp. 201-211. https://doi.org/10.1109/ISSRE.2014.11

[20] Samarthyam G.; Muralidharan M.; Anna R.K. 2017. Understanding Test Debt. In: Mohanty H., Mohanty J., Balakrishnan A. (eds) Trends in Software Testing. Springer, Singapore.

[21] Campos, Denivan; Rocha, Larissa; Machado, Ivan. 2021. Developer's perception on the severity of test smells: an empirical study. In: Proceedings of XXIV Congresso Ibero-Americano em Engenharia de Software (CibSE'2021).

[22] Palomba, Fabio; Nucci, Dario Di; Panichella, Annibale; Oliveto, Rocco; De Lucia, Andrea. On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study. 2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST), 2016, pp. 5-14.

[23] Tássio Virgínio, Luana Martins, Larissa Rocha, Railana Santana, Adriana Cruz, Heitor Costa, and Ivan Machado. 2020. JNose: Java Test Smell Detector. In Proceedings of the 34th Brazilian Symposium on Software Engineering. 564–569.

[24] Aljedaani, Wajdi; Peruma, Anthony; Aljohani, Ahmed; Alotaibi, Mazen; Mkaouer, Mohamed Wiem; Ouni, Ali; Newman, Christian D.; GHALLAB, Abdullatif; LUDI, Stephanie. 2021. Test Smell Detection Tools: A Systematic Mapping Study. In: Proceedings of Evaluation and Assessment in Software Engineering (EASE 2021). Association for Computing Machinery, New York, NY, USA, 170–180. https://doi.org/10.1145/3463274.3463335

[25] International Organization for Standardization/ International Electrotechnical Comission. ISO/IEC 24765 - Systems and software engineering — Vocabulary. Geneve, 2010, 410p.

[26] Ammann, Paul; Offutt, Jeff. Introduction to Software Testing. Cambridge University Press; 2nd edition (December 13, 2016) 364 pages. ISBN-10 : 9781107172012

[27] Meszaros, Gerard. xUnit Test Patterns: Refactoring Test Code. Addison Wesley, 2007

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

[29] Delplanque, Julien; Ducasse, Stéphane; Polito, Guillermo; Black, Andrew P.; Etien, Anne. Rotten Green Tests. 2019 In:45 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 500-511,

[30] Peruma, Anthony; Almalki, Khalid; Newman, Christian D.; Mkaouer, Mohamed Wiem; Ouni, Ali; Palomba, Fabio. 2019. On the Distribution of Test Smells in Open-Source Android Applications: An Exploratory Study. In: 29th Annual International Conference on Computer Science and Software Engineering, November 4–6, 2019.

[31] Wiklund, Kristian; Eldh, Sigrid; Sundmark, Daniel; Lundqvist, Kristina. Technical Debt in Test Automation. In: Proceedings of IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012. https://doi.org/10.1109/ICST.2012.192

[32] Calikli, Gul; Bener, Ayse. Empirical analysis of factors affecting confirmation bias levels of software engineers. Software Quality Journal 23, 695–722 (2015). https://doi.org/10.1007/s11219-014-9250-6

[33] Bavota, Gabriele; Qusef, Abdallah; Oliveto, Rocco; De Lucia, Andrea; BINKLEY, David. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. 2012 In: Proceedings of 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 56-65,

[34] Rompaey, B., Du Bois, B., Demeyer, S., Rieger, M., 2007. On the detection of test smells: A Metrics-Based approach for general fixture and eager test. IEEE Transaction on Software Engineering. 33 (12), pp.800–817.

[35] Lambiase, Stefano; Cupito, Andrea; Pecorelli, Fabiano; De Lucia, Andrea; Palomba, Fabio. 2020. Just-In-Time Test Smell Detection and Refactoring: The DARTS Project. In Proceedings of the 28th International Conference on Program Comprehension (ICPC '20). Association for Computing Machinery, New York, NY, USA, 441–445.

[36] Vahabzadeh, Arash; Stocco, Andrea; Mesbah, Ali. 2018. Fine-Grained Test Minimization. In Proceedings of 40th International Conference on Software Engineering (ICSE '18). ACM, New York, NY, USA, 12 pages.

[37] Posnett, Daryl; Hindle, Abram; Devanbu, Premkumar. 2011. A simpler model of software readability. In: Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11). Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/1985441.1985454

[38] Setiani, Novi; Ferdiana, Ridi; Hartanto, Rudy. Test Case Understandability Model. in IEEE Access, vol. 8, pp. 169036-169046, 2020, https://doi.org/10.1109/ACCESS.2020.3022876