

<http://www.diva-portal.org>

Postprint

This is the accepted version of a paper presented at *8th Workshop on Validation, Analysis and Evolution of Software Tests, Montréal, Canada, March 04, 2025*.

Citation for the original published paper:

Tran, H K. (2025)

Towards Reliable Eager Test Detection: Practitioner Validation and a Tool Prototype

In:

N.B. When citing this work, cite the original published paper.

Permanent link to this version:

<http://urn.kb.se/resolve?urn=urn:nbn:se:bth-27674>

Towards Reliable Eager Test Detection: Practitioner Validation and a Tool Prototype

Huynh Khanh Vi Tran

Blekinge Institute of Technology, Sweden

huynh.khanh.vi.tran@bth.se

Abstract—Context: Existing tools for detecting eager tests produce many false positives, rendering them unreliable for practitioners. To address this, our previous work introduced a novel definition of the Eager Test smell and a heuristic for more effective identification. Comparing the heuristic’s results with existing detection rules revealed eight test patterns where the rules misclassified the presence or absence of eager tests.

Objective: We aim to gather practitioners’ feedback on our heuristic’s assessment of these eight test patterns and operationalize the heuristic in a tool we named EagerID.

Method: We conducted a survey to collect practitioners’ feedback on the eight identified test patterns and developed EagerID to detect eager tests in Java unit test cases using JUnit. We also preliminarily evaluated EagerID on 300 test cases, which were manually analyzed in our previous study.

Results: Our survey received 23 responses from practitioners with a wide range of experience. We found that most practitioners agreed with the assessment of our heuristic. Furthermore, the preliminary evaluation of EagerID returned high precision (100%), recall (91.76%), and F-Score (95.70%).

Conclusion: Our survey findings highlight the practical relevance of the heuristic. The preliminary evaluation of the EagerID tool confirmed the heuristic’s potential for automation. These findings suggest that the heuristic provides a solid foundation for both manual and automated detection.

Index Terms—Software testing, Test case quality, Test suite quality, Quality assurance, Test smells, Unit testing, Eager Test, Detection tool, Java, JUnit

I. INTRODUCTION

Test smells are signs of problems in test code [1]. The Eager Test smell has been intensively studied in the literature [2]–[7] and has been originally defined as “when a test method checks several methods of the object to be tested” [1] or when “a test verifies too much functionality” [8].

The original definitions of the Eager Test smell are rather generic, leaving them open to varying interpretations. Our literature review [9] revealed that researchers have developed differing interpretations of these definitions, leading to the creation of various detection rules. Most of these interpreted definitions focus on the number of method invocations, deviating from the essence of the original definitions, which emphasize the verification of multiple methods or functionalities. Despite this misalignment, the method-invocation count has become a guiding principle for establishing most detection rules and validating their outcomes.

To address the ambiguity of the original definitions of the Eager Test smell, in our previous work [9], we proposed a novel, more precise definition, that is “a test case is NOT an

eager test when all of its assertions assess the outcome of a single method call of the class under test (CUT)”. We then introduced a heuristic to operationalize the proposed definition. By manually assessing 300 unit test cases written in Java, we identified a set of eager and non-eager test patterns that can be detected by our heuristic but not by the existing rules. In other words, these patterns characterize how our heuristic differs in the detection of Eager Tests from the existing rules.

Through a survey, we collect practitioners’ feedback on these patterns to assess the validity of our heuristic’s verdicts. Furthermore, their feedback helps us to understand the extent to which the heuristic can address practitioners’ concerns regarding the accuracy of the Eager Test detection in practice. We also develop a prototype tool to demonstrate the feasibility of operationalizing the heuristic in real-world settings.

Overall, the contributions of this study include the following:

- The validation of the heuristic through practitioner feedback;
- An early-stage prototype of an automated detection tool.

II. RELATED WORK

Tufano et al. [4] interviewed 19 practitioners and found that most were unaware of test smells, highlighting the need for automated detection tools. Schvarcbacher et al. [10] used TsDetect [11] and surveyed practitioners about the severity and refactoring of the identified test smell instances. They found practitioners hesitated to refactor even severe smells. Silva Junior et al. [12] identified that improper practices lead to test smell introduction during test creation and execution. Interestingly, experienced professionals were not less prone to introducing test smells than novices.

Campos et al. [13] used JNose [14] to collect test smells and asked developers about refactoring them. Although severity ratings were low, developers acknowledged test smells as harmful. Damasceno et al. [15] also used JNose, trained practitioners on test smell definitions, and found Eager Test to be among the most critical. Santana et al. [16] surveyed practitioners and identified the most frequently introduced test smells (e.g., Assertion Roulette) and the most severe ones (e.g., Conditional Logic Test).

Spadini et al. [17] conducted a study most related to ours, using TsDetect [11] and introducing severity thresholds for test smells. They found detection rules for some smells too strict but their thresholds aligned with practitioner perceptions.

Unlike their work, we study multiple detection rules and investigate misalignment among them.

Overall, we have seen several studies that explored test smells by leveraging detection rules and tools, analyzing the test smells’ severity, and surveying practitioners about their impact. However, we found no studies investigating whether practitioners agreed with the assessments of these rules and tools or whether the interpreted definitions and corresponding detection rules align with practitioner perceptions.

III. METHODOLOGY

The study aims to (1) gather practitioners’ feedback on our heuristic’s results for eight patterns of eager and non-eager tests it detects but existing rules miss, and (2) assess its automation potential. These patterns and rules were identified in our previous study [9], which analyzed 300 test cases to assess the agreement between our heuristic and existing rules. In this paper, *test pattern* refers to structural characteristics of unit tests for distinguishing eager and non-eager tests, differing from Binder’s Testing Object-Oriented Systems [18], where it denotes a test design technique.

To achieve these main goals, we aim to answer the following research questions:

RQ 1. To what extent do practitioners agree with the outcome of our heuristic on the eight patterns of eager and non-eager tests?

RQ 2. To what extent can the proposed heuristic be automated?

We conducted a survey to answer RQ1 and implemented an eager test detection tool, namely EagerID based on the proposed heuristic to answer RQ2.

A. Survey design

In this section, we first explain how we selected the existing detection rules, test patterns, and test cases to build the survey questionnaire, followed by describing how we conducted the survey. The survey instrument is available at <https://figshare.com/s/c7fcc8ac4d607c87da11>.

1) *Detection rules and test patterns selection:* Table I and Table II present the selected rules for comparison with our heuristic and the patterns of eager and non-eager tests that our heuristic does not agree with the selected rules, respectively. These rules and patterns were identified in our previous study [9].

It is worth emphasizing that the test patterns are presented to highlight the disagreement between our heuristic and existing rules. Therefore, Table II does not present rules when they align with our heuristic. Also, certain test patterns may require further details so that their test cases can be classified by rules other than those listed in the table. For instance, under rule DR3, whether a test case following test pattern TP2 is considered eager depends on how the non-assert and assert statements are grouped within the test case.

TABLE I
OUR EAGER TEST DEFINITION AND DETECTION RULES FROM THE LITERATURE TO COMPARE WITH OUR HEURISTIC

| ID | Description | Ref |
|---------------------------|---|------|
| Our eager test definition | A test case is NOT an eager test when all of its assertions assess the outcome of a single method call of the CUT | [9] |
| DR1 | At least 2 method calls from CUT in a test case (<i>excluding constructor</i>). | [19] |
| DR2.1 | At least 3 PTMI in a test case (<i>excluding constructor, considering only methods from CUT</i>). | [20] |
| DR2.2 | At least 5 PTMI in a test case (<i>excluding constructor, considering only methods from CUT</i>). | [11] |
| DR2.3 | At least 5 PTMI in a test case (<i>excluding constructor, considering only methods from production code (code developed by the project team that will end up in the released product)</i>). | [2] |
| DR3 | At least two cycles of non-verification instructions followed by verification instructions | [21] |
| DR4 | At least 2 assertions in a test case and at least one assertion is not on the result of a get method (<i>no definition of get methods</i>). | [22] |

CUT: Class under test, PTMI: Production type method invocations

2) *Example test case selection:* We selected eight real test cases from the set of 300 test cases that we analyzed in earlier work [9], each of which illustrates a test pattern described in Table II. The selection criterion is that test cases should be simple enough to represent the test patterns, i.e., helping the survey participants to understand the test patterns without a need to study the relevant production code.

3) *Survey questionnaire:* We designed our survey as a structured cross-sectional study following the guidelines of Kitchenham and Pfleeger [23]. The survey began with an introduction presenting the prevalent definition of an eager test: “An eager test is a test that assesses too much functionality in a single test method” [8], followed by two main sections.

In the first section, we outlined eight patterns of eager and non-eager tests in which our heuristic disagrees with the selected detection rules. For each pattern, we provided a description and an example test case. For instance, Figure 1 illustrates the survey question designed for test pattern TP3.

The description highlighted the characteristics of the pattern that make a test case under this pattern an eager test or not according to the existing rules and our heuristic. Under the example, we explicitly presented reasons why the test case would be considered an eager test and why it would not. Practitioners were then asked to select the reason that best aligned with their opinion. This approach was intended to assess the extent to which practitioners agree with the verdicts produced by the detection rules in the literature and with the verdicts produced by our heuristic.

Here, the reasons provided were detailed and directly tied to the example test case, yet they were crafted to encapsulate the essence of the corresponding detection rules and our heuristic. Hence, we argue that practitioners’ responses reflected not just their views on the example test case but also on the broader test patterns the test case represented. To maintain neutrality, we avoided referencing either the existing rules or our heuristic

TABLE II
TEST PATTERNS AND THE DISAGREEMENT BETWEEN OUR HEURISTIC (EAGERID) AND EXISTING RULES (CF. TABLE I)

| ID | Pattern description | Is an eager test? | |
|-----|---|-----------------------------------|-----------------------------------|
| | | Yes, based on | No, based on |
| | A test case of this pattern has | | |
| TP1 | (a) Multiple method calls from the class under test in its Set-up and/or Tear-down, AND (b) Only one assert statement verifying the outcome of a single method call. | DR1, DR2.1, DR2.2, DR2.3 | EagerID |
| TP2 | (a) Multiple method calls from the class under test, AND (b) All assert statements invoking only helper methods (get methods), AND (c) All assert statements verifying the outcome of a single method call. | DR1, DR2.1, DR2.2, DR2.3 | EagerID |
| TP3 | (a) All assert statements invoking helper methods different from get methods (such as <code>size()</code> , <code>contains()</code>), AND (b) All assert statements verifying the outcome of a single method call. | DR4 | EagerID |
| TP4 | (a) Two or more sets of non-assert statements followed by assert statements, AND (b) All assert statements verifying the outcome of a single method call. | DR3 | EagerID |
| TP5 | (a) Assert statement(s) verifying the outcome of one method call from the class under test, AND (b) One or more additional assert statements in the Set-up. | EagerID | DR1, DR2.1, DR2.2, DR2.3 |
| TP6 | (a) Multiple assert statements verifying different constructors. | EagerID | DR1, DR2.1, DR2.2, DR2.3 |
| TP7 | (a) All assert statements invoking only helper methods (get methods), AND (b) Assert statements verifying the outcomes of different method calls. | EagerID | DR4 |
| TP8 | (a) Only one set of non-assert statements followed by assert statements, AND (b) Assert statements verifying different method calls. | EagerID | DR3 |

in the provided reasons. We also included free-text fields for practitioners to share additional opinions or comments.

The second section of the survey collects (1) demographic information about the practitioners (their experience with unit testing and their familiarity with test smells and Eager Test in particular) and (2) any other comments they might want to share with us.

4) *Survey instrument evaluation*: We refined the questionnaire through discussions with researchers on the question logic, wording, and understandability, followed by a pilot study with six external assessors (three researchers and three practitioners). Feedback on presenting example test cases and clarifying test patterns guided further improvements.

5) *Survey distribution*: Our survey targeted practitioners with experience in writing and executing unit test cases. Participants were recruited through convenience sampling via

Test pattern #3. Such a test case has

- (a) All assert statements invoking helper methods different from **get** methods (such as `size()`, `contains()`), AND
- (b) All assert statements verifying the outcome of a single method call.

An example of this pattern is as follows:

```

1 public void testFindImplementations() throws Exception {
2     Resolver resolve = new Resolver();
3
4     List<Class> list = resolve.findImplementations(Base.class);
5
6     assertTrue(list.contains(ImplOne.class));
7     assertTrue(list.contains(ImplTwo.class));
8     assertEquals(2, list.size());
9 }

```

Which of the following statements aligns most with your opinion?

It is an eager test, because all of its assert statements are NOT based on **get** methods' results (they are based on `size()` and `contains()` instead).

It is NOT an eager test, because the additional methods invoked in the assert statements are still helper methods (even though they are not **get** methods) used to verify the outcome (`list`) of a single method call (`resolve.findImplementations(Base.class)` [line #4]).

If you do not agree with either of the statements or have any further comments, please write them here.

Fig. 1. Our survey question for test pattern TP3 (cf Table II)

LinkedIn, and personal networks.

Data collection ran from October 23 to November 18, 2024, yielding 23 responses, 21 of which were complete (answering all questions). One skipped test pattern TP6, and the other omitted TP8. Also, two respondents did provide information regarding their working experience.

6) *Survey data analysis*: To analyze the comments given by the survey respondents, we used a thematic coding approach [24]. This approach helped us to collect and group concerns, and feedback from the respondents regarding the assessment of our proposed heuristic.

B. Evaluation of detection tool

We used the same set of 300 test cases (including automatically generated and manually written test cases) from our manual assessment in our previous study [9] to evaluate the performance of the detection tool EagerID. Among these 300 test cases, 200 were sourced from Panichella et al.'s study [25], consisting of 100 automatically generated and 100 manually written test cases. The remaining 100 test cases, all manually written, were drawn from Sharma et al.'s study [26]. The 100 automatically generated test cases, created using EVOSUITE¹, originated from 17 different projects, whereas the 200 manually written test cases were collected from eight distinct projects.

We randomly split the test cases into two sets, 150 test cases each. One set was a training set to support the development of the tool. We calculated the precision, recall, and F-Score when running the tool on the other set to evaluate its accuracy. Note that the ground truth was established based on our manual assessment with the heuristic.

¹<https://github.com/EvoSuite/evosuite>

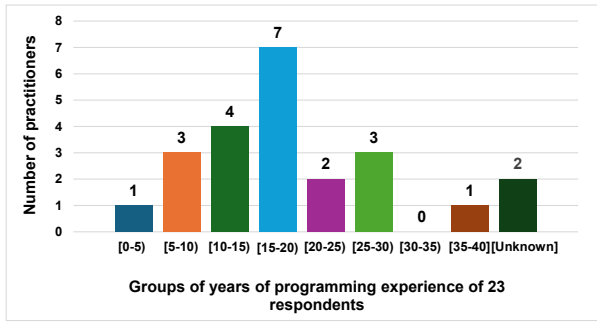


Fig. 2. Responses distribution based on their programming experience²

C. Validity threats

1) *Construct validity*: A major threat to our study is whether respondents interpreted terms like method calls, helper methods, and class under test as we intended. While aware of this risk, we balanced it against keeping the survey concise. Another issue arose from logical or technical flaws in the example test cases, which made it harder for respondents to evaluate them as eager tests. Free-text fields partially mitigated these threats by allowing respondents to share doubts and opinions.

2) *Internal validity*: Our survey primarily recruited participants via LinkedIn, excluding other platforms whose members might have different perspectives. Self-selection bias also posed a challenge, as non-respondents, likely less interested in the topic [27], may hold differing views. Since we distributed survey links in LinkedIn groups without tracking non-responses, sending reminders was not possible.

3) *External validity*: Our survey results are based on a small convenience sample, which may not fully represent practitioner opinions. However, as an exploratory study, it aimed to gather preliminary feedback on our heuristic rather than provide definitive validation. Additionally, our evaluation of EagerID on 300 test cases is limited, potentially missing some tool issues or limitations.

IV. PRACTITIONERS' FEEDBACK ON THE TEST PATTERNS (RQ1)

A. Demographics

Overall, the experience of the respondents (21 out of 23, as 2 did not provide this information) ranged from 4 to 40 years, with a median of 15 years and a mean of 16.2 years. Figure 2 shows the distribution, grouping respondents into 5-year experience bins.

Among all respondents (23 in total), most of them (20 out of 23) were familiar with unit test cases written in Java. The majority (16 out of 23) were aware of the concept of test smells before taking the survey, while only 8 were familiar with the concept of the Eager Test smell prior to the survey.

²Two respondents did not provide their working experience.

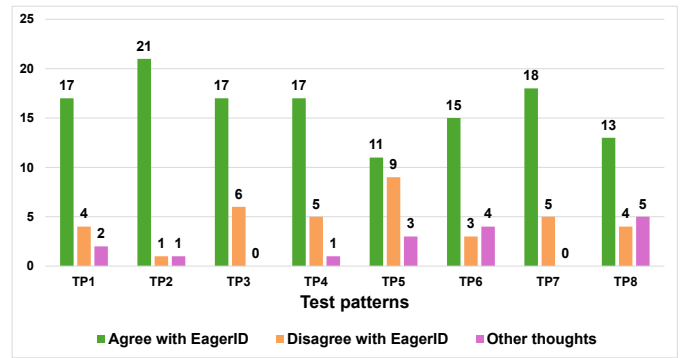


Fig. 3. Practitioners' agreement with the eager test assessment of the eight test patterns given by our heuristic - EagerID³

B. Practitioners' agreement level with the heuristic on the eight test patterns

Figure 3 gives an overview of the extent to which the respondents agreed with our heuristic's assessment of the eight test patterns.

For each test pattern, we present three numbers. The first (green) represents respondents who agreed with both our heuristic's eager test assessment and its reasons. The second (orange) shows those who agreed with the existing rules' assessment and reasons, indicating disagreement with our heuristic. The third (purple) represents respondents who disagreed with all provided reasons but shared the same eager test classification as either our heuristic or the existing rules.

Overall, for most of the test patterns, the majority of the respondents agreed with our heuristic. The assessment of our heuristic for test pattern TP2 had the highest agreement with the practitioners, followed by TP7, TP1, TP3, and TP4, which are all at least 74% (17 out of 23 respondents). For test patterns TP5, TP8, and TP6, the practitioners tended to agree less with our heuristic (48%, 59%, and 68%, respectively).

We studied their comments to understand the disagreement in the eager test assessment between our heuristic and practitioners. Note that there were no comments given by respondents who disagreed with our heuristic on the assessment of test patterns TP3 and TP7.

We first examined the test patterns (TP5–8) that our heuristic classifies as eager tests but the existing rules do not (Table II). In this group, the assessment of our heuristic of TP5 and TP8 had the least agreement with practitioners. For pattern TP5 (see Figure 4), a test case under this pattern is described as having

- (a) Assert statement(s) verifying the outcome of one method call from the class under test, AND
- (b) One or more additional assert statements in the set-up.

Four respondents disagreed with the provided reasons why our heuristic and the existing rules classify the example test case as an eager test or not. Their concerns focused on the technical details, arguing that a constructor always returns an

³One respondent skipped test pattern TP6 and one omitted test pattern TP8.

```

1 public void test3() throws Throwable {
2     Home home0 = new Home();
3     PhotoController photoController0 = new
4         PhotoController(home0, (UserPreferences) null);
5     assertNotNull(photoController0);
6
7     long long0 = photoController0.calculateTime();
8
9     assertEquals(1372766400496L, long0);
10 }

```

Fig. 4. Example test case of test pattern TP5

```

1 public void testEquals() throws Exception {
2     Category a = new Category();
3     a.setName("giraffes");
4     Category b = new Category();
5     b.setName("camels");
6     Category c = new Category("giraffes");
7
8     assertEquals(a, c);
9     assertTrue(a.equals(b));
10    assertTrue(b.equals(new Object()));
11 }

```

Fig. 5. Example test case of test pattern TP8

instance unless it throws an exception, making the first assert statement (line 5) unnecessary. Hence, their criticism was more about the use of *assertNotNull* for the constructor’s outcome rather than for the heuristic’s assessment. Indeed, two respondents commented that “assuming that ‘photoController0’ can be null because ‘new’ fails in some way, that should definitely be its own test case”, and “this would be an eager test because it would be testing two different behaviors, but right now, the first assert does not do anything.” Similarly, the other respondent reasoned that “line 5 is useless, and if line 5 is removed, then the test is not eager.” In other words, without the logical concern regarding the first assert statement, these four respondents would have agreed with our heuristic, bringing the total agreement to 15 respondents (65%).

There is only one respondent who disagreed with our heuristic and commented that having assertions as preconditions is sometimes necessary and acceptable to ensure that the state of the object under test is correct. While we agree on the importance of testing data construction, we contend this should be a separate test, as failures in data construction render dependent tests meaningless.

Regarding test pattern TP8, a test case under this pattern is described as having

- (a) Only one set of non-assert statements followed by assert statements, AND
- (b) Assert statements verifying different method calls.

Similar to test pattern TP5, a test case under test pattern TP8 is an eager test according to our heuristic and not an eager test based on the existing detection rule (Table II). Its example test case is shown in Figure 5.

Among the respondents who agreed with the reasons that the test case is not an eager test, none commented on their selection. Five practitioners, however, disagreed with all the provided reasons. Two of them found the example test case

```

1 public void testGetConnectionUserPassSettings() throws Exception {
2     String username = "user";
3     String password = "_secret";
4     String url = "jdbc:h2:mem:ormlite-up;USER=" + username +
5         ";PASSWORD=" + password;
6     JdbcConnectionSource sds = new JdbcConnectionSource(url);
7     sds.setUsername(username);
8     sds.setPassword(password);
9
10    assertNotNull(sds.getReadOnlyConnection(null));
11
12    sds.close();
}

```

Fig. 6. Example test case of test pattern TP1

did not make sense, while two others classified it as an eager test because it tested different behaviors (via different inputs) of the same method, *equal*. This reasoning actually aligns with our heuristic’s assessment, which stated that the assert statements verified different method calls of the same method, inherently involving different inputs. Including these two respondents brings the total agreement to 68% (15 out of 22 respondents who answered TP8).

One practitioner stated the test case was not an eager test because it tested a single method, *equals*. However, for TP7, the same practitioner agreed with our heuristic that the test case was eager since it verified multiple calls to the same method. This contradicts their stance on TP8, where the test case also verifies multiple calls to *equals*. Thus, we argue that we found no valid arguments against our heuristic for this pattern TP8.

The test pattern in which practitioners had the third lowest level of agreement with our heuristic is TP6, of which a test case has multiple assert statements verifying different constructors. Respondents who disagreed with the provided reasons highlighted the same issue as with TP5, noting that checking if a created object is null is unrealistic. They suggested assert statements should actually focus on checking whether the created object’s fields are initialized correctly. Based on their feedback, we conclude that agreement with our heuristic would likely increase if the assert statements were more meaningful such as comparing the created object with an actually valid object (i.e., different from *null*).

Our next investigation is the group of test patterns (TP1–4) that our heuristic classifies as non-eager tests. For test pattern TP1, such a test case has multiple method calls in Set-up and/or Teardown while having one assert statement verifying one single method call. Its example test case is shown in Figure 6.

For TP1, respondents who disagreed with our heuristic argued that the teardown method (line 11) should have been in a separate TearDown or AfterTest method. They viewed the test case as invoking too much functionality, thus classifying it as an eager test. While we agree the teardown should be separate, an eager test concerns *testing* too much functionality, not merely *invoking* too much functionality.

For TP2 (Figure 7), only one respondent disagreed with the heuristic, commenting that the test name was too vague to determine the test purpose. Hence, the respondent could not

```

1 public void testAccessors() throws Exception {
2     final String proxyHost = "foo";
3     final int proxyPort = 1030;
4
5     HTTPRequestInfo info = populate(uri, proxyHost, proxyPort);
6
7     assertEquals(proxyHost, info.getProxyHost());
8     assertEquals(proxyPort, info.getProxyPort());
9 }

```

Fig. 7. Example test case of test pattern TP2

```

1 public void testBasic() throws Exception {
2     BeanBinDAO dao = getDAO();
3     TestEntity entity = new TestEntity();
4     entity.setString("a sample string");
5     Query q = new Query(new Criteria("string", "a sample string",
6         SearchType.EQUALS));
7
8     List<Object> result = dao.search(TestEntity.class, q);
9
10    assertEquals(1, result.size());
11
12    TestEntity saved = (TestEntity) result.get(0);
13    assertEquals("a sample string", saved.getString());
14 }

```

Fig. 8. Example test case of test pattern TP4

judge the developer’s intention, whether he or she aimed to verify multiple methods. Similarly, for TP4 (Figure 8), only one respondent disagreed with our heuristic because of the unclear test purpose. As a result, neither could assess if the test case was eager.

C. Implications

A closer look at the respondents’ comments indicated that some disagreements with our heuristic might originate from the actual example test cases we provided; not from our heuristic itself. Some respondents, despite expressing their views differently, reached the same assessment as our heuristic. Factoring in these observations, agreement with our heuristic was higher (minimum 65% for all test patterns) than indicated by raw data (minimum 48% for all test patterns).

Some disagreements with our heuristic arose from confusion between *invoking* too much functionality and *verifying* too much functionality, with the latter being the basis of the Eager Test smell definitions. This confusion may result from respondents’ unfamiliarity with eager tests, as noted in their background answers. A similar misunderstanding exists in academia, where eager test detection rules often focus on the number of method invocations in a test case, reflecting researchers’ interpretations of the original smell definitions [9].

Meanwhile, other respondents highlighted the importance of understanding a test’s purpose, such as through its name, in determining whether it is an eager test. This is a valid point, which was also suggested by Panichella et al. [25]. However, this approach depends on code readability and often requires subjective human judgment, making automation challenging.

Overall, our analysis found no substantial concerns with the heuristic. Although the survey involved a small practitioner sample, it represents a strong initial step in demonstrating the

heuristic’s alignment with practitioners in detecting eager tests in practice.

V. EAGERID - PROTOTYPE TOOL FOR AUTOMATED DETECTION (RQ2)

To answer RQ2, we developed EagerID⁴, a command-line detection tool for identifying eager tests in Java JUnit unit tests. It runs as a stand-alone .jar file, without requiring an IDE. The tool takes two inputs: the project’s Java source code folder and test folder. Its output is a list of test cases with their eager test classifications.

A. Heuristic for eager tests detection

Our tool is based on the heuristic that we introduced in our previous study [9]. In summary, the heuristic consists of three following steps, which rely on the classification of method types (Table III).

- Step 1: For each method call in a test case, collect information that should be assessed in order to verify the outcome of the method call.
 - If the method type is creational or mutator, collect initialized or modified fields and return value/object (if any);
 - If the method type is internal producer, collect return value/object.
- Step 2: Collect the information that all assert statements verify.
 - If the assert statement invokes a creational (or contains the object created by a creational), collect the created object and its initialized fields;
 - If the assert statement invokes a mutator (or contains the return value/object of a mutator), collect the modified fields and return value/object (if any);
 - If the assert statement invokes a get or external producer (or contains the return value/object of a such a method), collect the retrieved fields.
 - If the assert statement invokes an internal producer (or contains the return value/object of a mutator), collect the return value/object.
- Step 3: Check if the information verified by all of the assert statements (Step 2) is the outcome of one single method call (Step 1) in the test case. If that is the case, then the test case is NOT an eager test; otherwise, the test case is classified as an eager test by the heuristic.

B. Tool architecture

In the previous section, we explained what data needs to be collected and how it can be used in our heuristic to detect eager tests. Here, we describe how our tool can collect such data in order to operationalize the heuristic.

We used JavaParser⁵ to parse the production code and test code files in order to perform static code analysis. JavaParser builds an AST for each unit test file that is under analysis and the relevant production code.

⁴https://github.com/vi-tran1987/ET_heuristic

⁵<https://javaparser.org/>

TABLE III
METHOD TYPES USED IN OUR HEURISTIC

| Method Type | Sub type | Description | CUT [*] | | Effect | | | Return value/object |
|-------------|-------------------|--|------------------|----------------|--------------|--------------|---------------------|---------------------|
| | | | Initialize field | Retrieve field | Change field | Return field | Return value/object | |
| Creational | | Is responsible for creating an object. | ✓ | ✓ | - | - | - | [Optional] |
| Mutator | | Changes the state of the object to which it belongs, i.e., must change one or more fields of the object and optionally have a return value/object. | ✓ | - | ✓ | ✓ | - | [Optional] |
| Accessor | Get | Returns an object's field without modifying the field. | ✓ ^{**} | - | ✓ | - | ✓ | - |
| | Internal Producer | Computes a result based on an object's field(s) AND is implemented (or overridden) in the method's class (CUT) | ✓ | - | ✓ | - | - | ✓ |
| | External Producer | Computes a result based on an object's field(s) BUT is implemented in a class different from the CUT. | - | - | ✓ | - | - | ✓ |

* CUT: (implemented in) Class under test

** GET method can be implemented in CUT or outside CUT

The tool analyzes only methods annotated with “Test”, i.e., a test case, in each unit test file. Within each test case, it identifies method calls by searching JavaParser expressions including VariableDeclarationExpr, MethodCallExpr, AssignExpr, and ObjectCreationExpr, treating constructors as a type of method (Table III).

For each method call in a test case, the tool extracts the method declaration and body from the production code, then collects fields that are initialized, modified, returned, or retrieved. A field is *initialized* if initialized by a constructor, *retrieved* if it appears in the method body. Meanwhile, a field is classified as *modified* if it is a target of a unary expression or assignment expression (e.g., “i++” or “i=0”, respectively). A field is marked as *returned* if directly returned by a return statement (e.g., “return i;”) or when the return statement contains a unary expression of which the field is the target (e.g., “return i++”). A field can have multiple types, such as being retrieved, modified, and returned.

A method can invoke one or more methods in its body, which we call inner method calls. We use recursion to perform the analysis described above to collect all fields that are initialized, modified, or just retrieved in each inner method body. To avoid an infinite recursion loop, we set a recursion depth to 20, meaning that the tool analyzes at most 20 layers of inner method calls.

The tool classifies each method call based on its declaration type and collected fields. The method type is *Creational* or *External Producer* if the method declaration is JavaParserConstructorDeclaration or ReflectionMethodDeclaration, respectively. To be a *Mutator*, *Get*, or *Internal Producer*, the method declaration must be JavaParserMethodDeclaration. A method is a *Mutator* if it modifies one or more fields; a *Get* if it returns a field but modifies none; and an *Internal Producer* if it computes information based on retrieved fields but neither modifies nor returns any fields.

With JavaParser, JUnit assert statements are treated as any normal method call from an external library, i.e., not part of the

production code. Hence, trying to resolve the method declaration corresponding to the invocation of JUnit assert statements will fail and JavaParser will throw an UnsolvedSymbolException. We identify JUnit assert statements when the name of the UnsolvedSymbolException is “org.junit.Assert”. Since each assert statement can invoke a method or contains the return value/object of a method, we perform the same analysis described above (collecting fields and identifying method type) for the method call associated with each assert statement.

C. Limitations

First, the tool needs to access the method declaration and method body in order to collect the accessed fields (retrieved, initialized, modified, returned). It is not feasible when the method is part of source code that was not developed by the organization responsible for the code under test. Hence, with external methods (methods outside of the production code), our tool assumes that they do not change the state of any object of the class under test. Second, since our tool relies on static code analysis, it is not possible to work with metaprogramming, a programming technique in which a computer program has the ability to treat other programs as their data [28]. In metaprogramming, we need a dynamic code analysis in order to identify the method types and collect the required fields.

D. Tool preliminary evaluation

To evaluate EagerID's accuracy, we ran it on 150 unit test cases (manually written and automatically generated) randomly selected from the 300 test cases analyzed in our manual assessment of the proposed heuristic [9]. Our manual assessment identified 85 eager tests in this set. EagerID classified 78 tests as eager, resulting in a precision of 100% (78/78), recall of 91.76% (78/85), and an F-Score of 95.70%. We then investigated the seven eager tests (false negatives) that EagerID missed. We found that detecting these cases required access to external libraries, which were not included

in the production code. This limitation, which we anticipated, is discussed in the previous section.

VI. CONCLUSIONS AND FUTURE WORK

In our previous study, we proposed a heuristic to detect eager tests in unit testing and identified eight patterns of eager tests and non-eager tests that can be detected by our heuristic but not by the existing detection rules. In this paper, we conducted a survey to collect practitioners' feedback on the assessment of the eight patterns by our heuristic. We also performed a preliminary evaluation of EagerID, the detection tool developed based on the heuristic.

Overall, the result of our survey demonstrates strong practitioner agreement with the heuristic for detecting eager tests, confirming its validity and practical relevance. No concerns from practitioners about the heuristic further reinforce its soundness. Additionally, the high precision, recall, and F-Score of our tool highlight the heuristic's potential for automation. These results suggest that the heuristic is a reliable foundation for both manual and automated detection of eager tests. For future work, we will focus on scaling EagerID and validating it across more diverse datasets.

ACKNOWLEDGMENT

This work was supported by ELLIIT, the Swedish Government's Strategic Research Area in IT and Mobile Communications, as well as by research grants for the GIST (20220235) and SERT projects from the Knowledge Foundation in Sweden. Also, I would like to express my sincere gratitude to my supervisors, Nauman bin Ali, Michael Unterkalmsteiner and Jürgen Börstler, for their invaluable guidance, support, and insightful feedback throughout this research.

REFERENCES

- [1] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 2001, pp. 92–95.
- [2] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–816, 2007.
- [3] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *2018 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2018, pp. 1–12.
- [4] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 4–15.
- [5] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, pp. 1052–1094, 2015.
- [6] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE, 2012, pp. 56–65.
- [7] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empirical study," in *Proceedings of the 9th international workshop on search-based software testing*, 2016, pp. 5–14.
- [8] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional, 2007.
- [9] H. K. V. Tran, N. b. Ali, M. Unterkalmsteiner, and J. Börstler, "A proposal and assessment of an improved heuristic for the eager test smell detection," 2024, available at SSRN 4997775.
- [10] M. Schvarcbacher, D. Spadini, M. Bruntink, and A. Opreescu, "Investigating developer perception on test smells using better code hub - work in progress -," in *CEUR Workshop Proceedings*, vol. 2510, 2019.
- [11] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "Tsdetect: An open source test smells detection tool," in *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1650–1654.
- [12] N. Silva, L. Rocha, L. A. Martins, and I. Machado, "A survey on test practitioners' awareness of test smells," in *CibSE 2020 - Proceedings of the 23rd Iberoamerican Conference on Software Engineering*, 2020.
- [13] D. Campos, L. Rocha, and I. Machado, "Developers' perception on the severity of test smells: an empirical study," in *CibSE 2021 - Proceedings of the XXIV Ibero-American Conference on Software Engineering*. Ibero-American Conference on Software Engineering, 2021.
- [14] T. Virginio, L. Martins, L. Rocha, R. Santana, A. Cruz, H. Costa, and I. Machado, "Jnose: Java test smell detector," in *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, ser. SBES '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 564–569.
- [15] H. Damasceno, C. Bezerra, E. Coutinho, and I. Machado, "Analyzing test smells refactoring from a developers perspective," in *Proceedings of the XXI Brazilian Symposium on Software Quality*, ser. SBQS '22. New York, NY, USA: Association for Computing Machinery, 2023.
- [16] R. Santana, D. Fernandes, D. Campos, L. Soares, R. Maciel, and I. Machado, "Understanding practitioners' strategies to handle test smells: a multi-method study," in *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, ser. SBES '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 49–53.
- [17] D. Spadini, M. Schvarcbacher, A. Opreescu, M. Bruntink, and A. Bacchelli, "Investigating severity thresholds for test smells," in *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, 2020, pp. 311–321.
- [18] R. Binder, *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.
- [19] J. De Bleser, D. Di Nucci, and C. De Roover, "Assessing diffusion and perception of test smells in scala projects," in *IEEE International Working Conference on Mining Software Repositories*, vol. 2019-May, 2019, pp. 457–467.
- [20] M. Breugelmanns and B. Van Rompaey, "Testq: Exploring structural and maintenance characteristics of unit test suites," in *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. Citeseer, 2008, p. 11.
- [21] A. Pizzini, S. Reinehr, and A. Malucelli, "Automatic refactoring method to remove eager test smell," in *Proceedings of the XXI Brazilian Symposium on Software Quality*, ser. SBQS '22. New York, NY, USA: Association for Computing Machinery, 2023.
- [22] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 523–533.
- [23] B. A. Kitchenham and S. L. Pfleeger, "Personal opinion surveys," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. Springer, 2008, pp. 63–92.
- [24] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 International Symposium on Empirical Software Engineering and Measurement*, Sep. 2011, pp. 275–284.
- [25] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Test smells 20 years later: detectability, validity, and reliability," *Empirical Software Engineering*, vol. 27, no. 7, 2022.
- [26] T. Sharma, S. Georgiou, M. Kechagia, T. A. Ghaleb, and F. Sarro, "Investigating developers' perception on software testability and its effects," *Empirical software engineering : an international journal*, vol. 28, no. 5, p. 120, 2023.
- [27] M. D. Duda and J. L. Nobile, "The fallacy of online surveys: No data are better than bad data," *Human Dimensions of Wildlife*, vol. 15, no. 1, pp. 55–64, 2010.
- [28] Y. Lilis and A. Savidis, "A survey of metaprogramming languages," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–39, 2019.