The Journal of Systems and Software 226 (2025) 112438

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A proposal and assessment of an improved heuristic for the Eager Test smell detection☆

Huynh Khanh Vi Tran *, Nauman bin Ali, Michael Unterkalmsteiner, Jürgen Börstler

Blekinge Institute of Technology, Department of Software Engineering, SE-37179, Karlskrona, Sweden

ARTICLE INFO

Keywords:
Software testing
Test case quality
Test suite quality
Quality assurance
Test smells
Unit testing
Eager test
Java
JUnit

ABSTRACT

Context: The evidence for the prevalence of test smells at the unit testing level has relied on the accuracy of detection tools, which have seen intense research in the last two decades. The Eager Test smell, one of the most prevalent, is often identified using simplified detection rules that practitioners find inadequate.
Objective: We aim to improve the rules for detecting the Eager Test smell.
Methods: We reviewed the literature on test smells to analyze the definitions and detection rules of the Eager Test smell. We proposed a novel, unambiguous definition of the test smell and a heuristic to address the limitations of the existing rules. We evaluated our heuristic against existing detection rules by manually applying it to 300 unit test cases in Java.
Results: Our review identified 56 relevant studies. We found that inadequate interpretations of original definitions of the Eager Test smell led to imprecise detection rules, resulting in a high level of disagreement in detection outcomes. Also, our heuristic detected patterns of eager and non-eager tests that existing rules missed.
Conclusion: Our heuristic captures the essence of the Eager Test smell more precisely; hence, it may address practitioners' concerns regarding the adequacy of existing detection rules.

1. Introduction

Automated testing has received more attention in recent years as it enables continuous software engineering (Jabbari et al., 2018). Test code serves as the backbone of automated software testing. Therefore, sufficient test code quality is required to maintain confidence in automated testing. However, developing and maintaining high-quality test code has proven to be non-trivial (Garousi and Küçük, 2018).

Several guidelines and practices have been proposed to achieve high-quality test code at the unit testing level. Like the smells of the production code, improper testing practices lead to *test smells*, "indicating trouble in test code", as defined by Van Deursen et al. (2001). A common test smell that has been intensively studied in the literature is the Eager Test smell, which is commonly referred as a test method checking several methods of the object under test. Many detection tools such as TestQ (Breugelmans and Van Rompaey, 2008), Test Smell Detector (Bavota et al., 2015), TASTE (Palomba et al., 2018), SOCRATES (De Bleser et al., 2019b), DARTS (Lambiase et al., 2020), tsDetect (Peruma et al., 2020), and JNose Test (Virgínio et al., 2019)) were developed to detect test smells, including the Eager Test smell. Several studies (Bavota et al., 2015; Tufano et al., 2016a; Van Rompaey et al., 2006; Martins et al., 2023; Damasceno et al., 2023; Bavota et al., 2012; Spadini2018On et al., 2018) used these tools to analyze the prevalence of the Eager Test smell in practice and to investigate the impact of the test smell on multiple quality-related aspects of the test code such as coupling, cohesion, complexity, comprehensibility, maintainability, and change and defect proneness of the test code.

Nevertheless, the reported detection tools rely on imprecise detection rules (the number of method invocations and assertion statements in a unit test) due to the ambiguous definitions of the Eager Test smell. This leads to inconsistent and questionable detection results (Panichella et al., 2022, 2020). Consequently, the assessments of the prevalence and impact of the Eager Test smell in practice may have been inaccurate. Meanwhile, practitioners highly agree that a good test case should

☆ Editor: Tao Zhang.

* Corresponding author.
E-mail addresses: huynh.khanh.vi.tran@bth.se (H.K.V. Tran), nauman.ali@bth.se (N.b. Ali), michael.unterkalmsteiner@bth.se (M. Unterkalmsteiner), jurgen.borstler@bth.se (J. Börstler).

https://doi.org/10.1016/j.jss.2025.112438
Received 6 October 2024; Received in revised form 4 March 2025; Accepted 12 March 2025
Available online 24 March 2025

be atomic, i.e., one test case should be testing one aspect of a requirement or one functionality (Kochhar et al., 2019), which is the exact opposite of eager tests. Therefore, it is important to avoid, detect, and resolve this test smell. However, practitioners have expressed concerns about how the Eager Test smell has been defined and detected (Spadini et al., 2020; Schvarcbacher et al., 2019).

In this study, we first reassess the definitions and detection rules of the Eager Test smell through a literature review. Second, we propose a new, unambiguous definition of the Eager Test smell based on the definitions of the test smell found in the literature. Third, we provide a novel detection heuristic (with pseudocode) that captures the proposed definition to overcome the weaknesses of existing detection rules and approaches. Consistent with most discussions of test smells in the literature, the scope of the proposed heuristic is limited to unit tests and source code written in Java. However, we also recognize the value of extending the Eager Test smell definition to non-object-oriented programming languages, such as C, to broaden the applicability of our approach across diverse programming paradigms Lastly, to assess the proposed heuristic, we apply it to a set of 300 test cases to detect eager tests. We compare the results of the proposed heuristic with the existing detection rules. Our assessment identify a high level of disagreement between the detection rules and highlights common patterns of eager tests and non-eager tests that can be detected by our heuristic but not by the existing rules.

The remainder of the paper is structured as follows: Section 2 presents the basic concepts in unit testing and the related work. Section 3 details the design and procedure of the systematic literature review, the evaluation of the proposed heuristic, and the primary threats to validity. Section 4, Section 5, and Section 6 present the results and analysis for each of the research questions posed in the study. In Section 7, we discuss the selected findings related to the differences between our heuristic and the selected rules in detecting eager tests. Finally, Section 8 concludes our paper.

2. Background and related work

In this section, we summarize the background notions with respect to unit testing, as well as the main related work discussing (1) test smells, (2) the quality of test cases and test suites, and (3) issues in the existing approaches for detecting eager tests.

2.1. Unit testing

The primary objective of unit testing is to "enable the sustainable growth of a software project" (Khorikov, 2020). While the scope of a test generally covers a collection of executable software components, the ***scope of a unit test*** is focused on a relatively small executable (Binder, 2000). In object-oriented programming, the smallest executable unit is a class object. However, since test messages are sent to a method, it is actually a method scope testing (Binder, 2000). Accordingly, we refer to the method being tested as the *method under test* (MUT). The class containing the MUT is called the *class under test* (CUT), and the instance of this class, created in the test case, is termed the *object under test* (OUT). In this study, we adopt this terminology when defining the Eager Test smell and describing our heuristic for detecting it.

As with many terms in software development, the definition of a unit test comes with several nuances. However, despite the variations, a unit test should consistently include three key elements as follows (Khorikov, 2020; Martin, 2014):

- Verifying a small piece of code;
- Running significantly faster than other kinds of tests;
- Running in an isolated manner.

While the first two attributes are generally accepted, the third – what constitutes testing in an *isolated manner* – often sparks debate in both practice and academia. This issue of isolation forms the basis of the distinction between the *classical* (Beck, 2002) and *London* schools of unit testing (Freeman and Pryce, 2009), also known as the Detroit and mockist schools, respectively. The London school advocates for isolating the unit under test from its collaborators. A **test unit**, in this case, is a *unit of code*, which typically refers to a *class*. If a class depends on other classes, all such dependencies should be replaced with test doubles (Meszaros, 2007). In contrast, the classical approach focuses on ensuring that unit tests themselves are isolated from one another. In this approach, a **test unit** is a *unit of behavior*, which can be implemented by one or many classes.

Besides the three common elements, a unit test case also follows a common pattern, which is called the setup-stimulate-verify-teardown (S-S-V-T) cycle, as follows:

- Acquire the necessary resources,
- Send one or more stimuli to the unit under test,
- Verify that the unit responds properly, and
- Release the acquired resources.

The importance of testing one method at a time in unit testing has been emphasized in the literature (Khorikov, 2020; Hamill, 2004; Tudose, 2021). There are several reasons for this approach. Such tests are easier to understand and remain unaffected when other parts of the code change. Additionally, they allow practitioners to refactor production code with confidence, as these tests provide clear messages that precisely identify the source of any breakage introduced by refactoring. This recommendation aligns with what has been advised by researchers in order to avoid the Eager Test smell, which we explain further in detail in Section 4.1.

2.2. Test smells

The concept of test smells originates from Fowler's book on code smells (Fowler, 2018), where he explained that while code smells may not represent actual faults, they can pose quality issues that hinder maintenance or lead to bugs during later stages of development (Tufano et al., 2015). This idea was later expanded to test code by Van Deursen et al. (2001), who, in the context of unit testing for eXtreme Programming, described test smells as signs of problems in test code. Meszaros (2007), building on the work of Van Deursen et al., further developed the concept by identifying additional test smells specific to the xUnit testing framework. The most recent secondary study on test smell definitions was carried out by Garousi and Küçük (2018), which compiled a list of 139 test smells and provided an overview of approaches and tools to address them. Additionally, Aljedaani et al. (2021) offered a comprehensive review of 22 test smell detection tools developed by the research community.

2.3. Quality of test cases and test suites

After summarizing the main related work on test smells in the previous section, we now turn to the quality of test cases and test suites. Test smells, as indicators of potential issues in test design, directly impact the overall quality of test cases and test suites. Hence, it is also essential to present the related work on the quality of test cases and test suites.

One of the pioneering studies in this area was conducted by Goodenough and Gerhart (1975), who garnered significant attention from the research community by exploring test adequacy and their proposed definition of reliable test data. Years later, Zhu et al. (1997) performed a comprehensive literature review on test adequacy criteria, which inspired further research into the structural quality of tests within various application domains and programming paradigms (Kapfhammer and

Soffa, 2003; Lemos et al., 2007; Pei et al., 2017). Additionally, some researchers began focusing on adapting software quality models to assess test case and test suite quality (Neukirchen et al., 2008; Athanasiou et al., 2014). More recently, there has been an increased emphasis on incorporating feedback from practitioners to develop effective quality models for test cases and test suites (Bowes et al., 2017; Grano et al., 2020; Tran et al., 2019).

A recent literature review, conducted by the authors of this study, offers a comprehensive overview of the current state of test artifact quality in software testing (Tran et al., 2021). This review introduces a quality model for test cases and test suites, containing 30 quality attributes, with measurement details provided for 11 of these attributes. The model also incorporates quality attributes adapted from ISO/IEC 25010:2011, offering a broader perspective on test case and test suite quality. Additionally, the review identifies 11 context dimensions used to characterize the software-testing environments in which quality has been evaluated. This quality model could prove valuable in developing guidelines or templates for designing test cases and test suites, as well as assessment tools for evaluating existing test artifacts.

2.4. Issues in the existing approaches for detecting eager tests

Panichella et al. (2022, 2020) conducted a manual analysis of 200 test suites generated by EVOSUITE and JTEXPERT and 49 manually written test suites to identify six types of test smells, including Eager Test. The authors used the outcome of their manual analysis to assess the accuracy of the two detection tools (Bavota et al., 2012; Peruma et al., 2020) and to examine how the detected test smells reflect real problems in practice. With respect to the Eager Test smell, the authors concluded that there is a need for a more precise approach to detecting the test smell as the current detection tools rely on highly inaccurate rules.

Panichella et al. (2022, 2020) also proposed a new criterion called ‘semantic coherence’ to characterize non-eager tests. More precisely, they stated that “a test is semantically coherent if it asserts only (transitive) properties and attributes of the class under test that all relate to a single testing scenario”. Their approach to identifying tested scenario(s) in a test case is to compare the asserted properties with the test’s purpose, via its nomenclature and comments. Nonetheless, it is unclear what the authors mean by asserted properties and how feasible it is to identify a test’s purpose via its nomenclature and comments. Hence, while we agree with the authors that a non-eager test should focus on only one testing scenario, the main limitation is that their approach to identifying the scenario depends on the readability of the code (how understandable the function names and comments are). Also, it seems to require subjective assessment that would require human involvement and perhaps would be difficult to automate.

Compared to Panichella et al. (2022, 2020), we propose an unambiguous definition of the Eager Test smell and a concrete heuristic (with pseudocode) that operationalizes the definition and can be deterministically applied to detect eager tests. Furthermore, our heuristic works at the test case level, which provides a better understanding of the presence of eager tests than the annotation at the test suite level by Panichella et al. They also acknowledged that their analysis could lead to overestimating the presence of test smells at a test case level.

The second related work that we want to discuss is from Pizzini et al. (2023). They summarized different definitions of the Eager Test smell and studied the existing detection approaches. They argued that the existing approaches are not good enough to capture eager tests as they produce a high number of false positives, i.e., wrongly label tests as having the Eager Test smell. Based on this argument, they proposed a new solution for identifying the test smell. They further implemented a refactoring method to remove eager tests in practice automatically. One of the potential threats to their study’s validity is that their study of the definitions and existing detection approaches was not systematic. For this reason, their study contained an imprecise reference to the original definitions and an incomplete list of existing approaches.

Compared to the studies of Panichella et al. (2022, 2020) and Pizzini et al. (2023), we argue that our study provides a more thorough review of how the Eager Test smell has been defined and detected. In addition, our manual analysis showed that our heuristic could provide a better detection accuracy than the existing detection approaches when assessing eager tests at a fine-grained level, i.e., the test case level.

3. Methodology

The goals of this study are (1) to understand how the Eager Test smell has been defined and detected in the literature and (2) to address the limitations of existing rules and approaches to detecting eager tests. To achieve these goals, our study aims to answer the following research questions:

- RQ1. How has the Eager Test smell been defined in the literature?
- RQ2. What rules and approaches have been proposed in the literature to detect eager tests?
- RQ3. What are the limitations of the existing detection rules and approaches?
- RQ4. How can the identified limitations be addressed?
- RQ5. To what degree does the level of recognized eager tests change when these limitations are addressed?

We conducted a literature review to answer RQ1, RQ2, and RQ3. With the review, we wanted to understand how the Eager Test smell was originally defined and any shortcomings of the current detection rules and approaches. To answer RQ4, we proposed a novel, more precise definition of the Eager Test smell and constructed a heuristic that can operationalize the proposed definition in order to detect eager tests for unit test cases with source code written in Java. The proposed definition and heuristic were built based on the information obtained by RQ1-3. Finally, we addressed RQ5 by performing a manual eager test classification of 300 unit test cases (both manually written and automatically generated) written in Java. The assessment was based on the proposed heuristic and the existing detection rules and approaches identified by RQ2.

3.1. Literature review — Search process and data extraction

The literature review on definitions and detection rules of the Eager Test smell was conducted through two searches. First, we looked for existing studies that had synthesized definitions of the Eager Test smell. This first search was performed on March 17, 2023, by searching for reviews on test smells on Scopus using the search string “(TITLE-ABS-KEY(“review” OR “map” OR “mapping”) AND TITLE-ABS-KEY (“test smells”))”. The first search returned 15 papers. By scanning the abstracts, we (three of the co-authors) selected six studies that are literature reviews on test smells. The three co-authors read the full text of these six papers to check if the papers present definitions of the Eager Test smell they have found or synthesized. We found two such studies (Garousi and Küçük, 2018; Rwemalika et al., 2023). From these two studies, we found 28 papers (cited in Garousi and Küçük, 2018) and 15 papers (cited in Rwemalika et al., 2023) that included the definitions of the Eager Test smell.

Since the first relevant review (Garousi and Küçük, 2018) was not recent (2016), while the second relevant review (Rwemalika et al., 2023) focused on a specific type of test cases (system user interactive tests), we decided to perform a second search to identify more recent work. Although we were interested in Eager Test only, for the search, we used the more general keyword of “test smell” instead to be more inclusive. The second search was conducted on Scopus using this search string: “TITLE-ABS-KEY (“test smell”) AND (LIMIT-TO (SUBJAREA, “COMP”)) AND (LIMIT-TO (DOCTYPE, “cp”) OR LIMIT-TO (DOCTYPE, “ar”))”. The search returned 93 papers. In total, we collected 136 papers (93 from the second search + 28 from the first relevant review Garousi and Küçük, 2018 + 15 from the second relevant review Rwemalika et al., 2023). After removing the duplicates, non-peer-reviewed literature (based on expert opinion), and studies that only mentioned the test smells without descriptions, we had a final list of 56 papers (available in our supplementary document, which can be accessed via the following link: https://figshare.com/s/34c20027fb4e5c2344d6) for data extraction.

During data extraction, we collected information about both the Eager Test and Assertion Roulette test smells, as Assertion Roulette was reported as identical to the Eager Test smell by Garousi and Küçük (2018). The following information was extracted from the selected 56 papers: (1) definition or description of the test smell (Eager Test or Assertion Roulette), (2) source of definition (references to other papers or the authors' interpretation), (3) detection rules, (4) source of detection rules (references to other papers or the authors' interpretation). All authors reviewed the data extraction form. After that, the first author completed the extraction process. One co-author reviewed the extraction result of 10% of the selected papers (randomly selected). Since there was a 100% agreement among two co-authors, we decided not to pursue further validation of the extraction.

3.2. Evaluation of proposed heuristic

To evaluate our heuristic for detecting eager tests, we manually assessed a set of 300 test cases, both manually written and automatically generated, using the heuristic and the existing detection rules found in the literature. To compare the level of agreement in the eager test classification outcomes among our heuristic and the detection rules, we used Cohen's kappa, which can be used to measure the degree of agreement in nominal scales between two raters (Cohen, 1960). We then followed Landis and Kock's guideline (Landis and Koch, 1977) to interpret the kappa values. The raw data (the number of test cases classified as eager tests or not by each detection rule and the heuristic) is included in Appendix. Note that we call our heuristic *EagerID* in Appendix for the sake of simplification.

3.2.1. Test case selection

To compare our heuristic with existing eager test detection rules, we manually analyzed 300 Java unit test cases used in recent studies by Panichella et al. (2022) and Sharma et al. (2023). These studies were selected because they focus on assessing eager tests in unit testing, aligning with our goals, and their selection of test cases was well-motivated and from well-known sources of open-source software systems.

In particular, these 300 test cases came from two widely-used datasets: the SF110 dataset (Fraser and Arcuri, 2014) (containing 110 projects from SourceForge.net) and the RepoReapers dataset (Munaiah et al., 2017) (generated from 1,857,423 GitHub repositories). Specifically, among the 300 selected test cases, 100 automatically generated test cases came from 17 different projects (from the SF110 dataset), while 200 manually written tests were from eight different projects (from the SF110 and the RepoReapers datasets).

Another reason for using the same test cases as Panichella et al. and Sharma et al. is that they shared their manual assessment of eager tests. Hence, by analyzing the same data, we could cross-validate our results with theirs, as explained in detail below. It is also worth noting that we did not assess all test cases analyzed in these two studies, as our heuristic requires a clear understanding of the production code, which can consume a considerable amount of time. Given our study's limited human resources, we randomly selected 300 test cases for our manual classification.

In Sharma et al.'s study, the authors used one of the common detection rules found in the literature ("a test method calls multiple production methods") to manually classify 427 test cases (manually written). Note that the authors did not specify the threshold used in their manual classification and whether they counted any method calls from the production code or from the class under test only. However, based on the test cases classified as eager tests in their published assessment data, we could clearly see that the counted method calls were from the production code, and the number of such calls was always more than four. With this information, we argued that their detection rule was identical to the detection rule DR2.3 in Table 10. As their manual assessment is accessible, we were able to compare our eager test classification with theirs and explored the similarities and differences in interpreting the same detection rule.

With Panichella et al.'s study, the authors used their own proposal ("At least 2 assertions in a test case and at least one assertion is not on the result of a get met" — DR4 in Table 10) to manually identify eager tests in their set of 149 test suites (100 automatically generated + 49 manually written ones). Their assessment was conducted at the test suite level, while we assessed eager tests at a finer-grain level, i.e., the test case level. For this reason, we could not study the similarities and differences in interpreting the detection rule as we did with Sharma et al.'s study. Nevertheless, one of their findings drew our attention. The authors concluded that many test suites were classified as eager according to their detection rule DR4 (20% of the automatically generated test suites and 80% of the manually written test suites). Since we wanted to verify if we could observe the same pattern with our heuristic at the test case level, it was still worth it for us to work on the same data set.

3.3. Validity threats

In the following, we discuss potential threats to the validity of the literature review, and the heuristic and its validation.

3.3.1. Threats to construct validity

One of the main challenges in conducting this study was related to the literature review. We followed the standard procedure when conducting the review. Nonetheless, there were two steps, including the search validation, and the quality assessment of the selected studies that we did not perform, which might affect the review's outcome. Also, since our search was on Scopus only, we might have missed other relevant studies that could be found via other search engines. We mitigated this problem by including studies cited by two recent literature reviews on eager tests to expand our selected studies collection. Other common threats to the validity of secondary studies are bias in paper selection and data extraction. To address these problems, the paper selection result and the data extraction form were reviewed and discussed by all authors. Also, two authors extracted data for a subset of seven papers. There was a high consistency in the data extracted by the two authors, and the final data extraction result was also reviewed by all authors.

Another threat was related to how we understood the original definitions of the Eager Test smell. As mentioned earlier, the definitions are rather informal, which gives room for interpretation. Therefore, there could be details or ideas in the definitions that we might not interpret the same way as the authors of the definitions. However, we argue that our heuristic captures the essence of the Eager Test smell as we thoroughly studied the original definitions. On top of that, we presented concrete patterns of both eager and non-eager test cases that could be highlighted by our heuristic only and not by the other detection rules in the literature.

Connected to the threat above, another challenge was our interpretation of the detection rules. To make sure that we understood the detection rules sufficiently, we looked at both the original references and the source code of the tools that implemented the detection rules. Nevertheless, there could still be details that we missed, which can lead to a different understanding, as shown by the different assessment results between us and Sharma et al. regarding the detection rule DR2.3 (details in Section 6.4).

Another threat was how we conducted our manual assessment. Even though the assessment was mainly done by the first author, the outcomes were discussed in iterations with two other co-authors via face-to-face discussions and workshops. Also, for each assessed test case in our manual assessment, we attached the production code, from which we collected the necessary information to execute our heuristic. As such, we are confident that our assessment based on the heuristic is highly traceable and internally consistent.

3.3.2. Threats to external validity

In this study, we assessed the presence of eager tests in a set of 300 test cases. The choice of the tests used in this study might potentially impact the generalizability of our findings. As the sample set was rather small, we might potentially miss different root causes of the differences between our heuristic and the existing detection rules. Nonetheless, our specific findings regarding what our heuristic can contribute above the existing rules remain valid despite the small sample set of test cases.

4. Eager test definitions, detection rules and limitations (RQ1–RQ3)

4.1. Eager test definitions (RQ1)

4.1.1. Sources of definitions

The first important finding from the literature review was that there are actually two primary sources of definitions for Eager Test and Assertion Roulette test smells: Van Deursen et al. (2001) and Meszaros (2007) (Table 1). However, not all studies cited these two primary sources for their test smells' definitions. Among the 56 selected studies, 47 studies provided definitions of the Eager Test smell, but only 38 out of the 47 studies cited the primary sources. The other 9 (47 − 38) studies either did not mention the source of their test smells' definitions or cited different studies instead. Similarly, among 48 studies that provided definitions of the Assertion Roulette smell, only 41 studies referred to the primary sources. It is also worth noting that the definitions given by Van Deursen et al. (2001) were more often cited than the definitions given by Meszaros (2007) (as shown in Table 1).

Between the two primary sources, Van Deursen et al. (2001) were the first ones who introduced the concept of test smells to convey poorly designed unit tests. Meszaros (2007) acknowledged the introduced test smells and offered a broader point of view on the same topic by classifying test smells into three levels: project, behavior, and code-level smells. A test smell at a higher level contains lower-level test smells. A common factor between the two sources of definitions is that the authors defined the test smells rather informally, giving room for interpretation, which is further discussed in the subsequent section.

With respect to the association between Eager Test and Assertion Roulette, Meszaros noted that Eager Test can be the cause of Assertion Roulette. However, neither Meszaros nor van Deursen et al. discussed their equivalence. For the Eager Test smell, its definitions from both sources focus on either the number of methods or functionalities to be tested. Meanwhile, for the Assertion Roulette smell, both sources emphasize the problem with assertions having no explanation to assist the test failure analysis. The claimed equivalence was found only in a later study (Garousi and Küçük, 2018), where the authors interpreted the original definitions.

4.1.2. Interpretations of the original definitions

Since the definitions of the Eager Test and Assertion Roulette smells given by Van Deursen et al. (2001) and Meszaros (2007) are rather ambiguous, researchers studying these two types of test smells provided their own interpretations. Tables 2 and 3 categorize those interpreted definitions of Eager Test and Assertion Roulette, respectively. By comparing the two tables, we can see the overlap in interpreted definitions between Eager Test and Assertion Roulette. More specifically, a common definition, namely *having multiple assertions*, was used to characterize both types of test smells in several studies (Garousi and Küçük, 2018; Tahir et al., 2016; Camara et al., 2021).

When looking more closely at each type of test smell, we can see that the Eager Test has more definition variations than the Assertion Roulette. With Assertion Roulette, the interpreted definitions mainly

Table 1
Definitions of Eager Test and Assertion Roulette given by Van Deursen et al. (2001) and Meszaros (2007).

Test smell	Van Deursen et al. (2001)	Meszaros (2007)
Eager Test[a]	When a test method checks several methods of the object to be tested, it is hard to read and understand, and therefore more difficult to use as documentation. Moreover, it makes tests more dependent on each other and harder to maintain. (cited by 32 studies)	The test verifies too much functionality in a single test method. (cited by 6 studies)
Assertion Roulette[b]	"Guess what's wrong?" This smell comes from having a number of assertions in a test method that have no explanation. If one of the assertions fails, you do not know which one it is. (cited by 35 studies)	It is hard to tell which of several assertions within the same test method caused a test failure. (cited by 6 studies)

[a] The definition was provided in 47 studies.
[b] The definition was provided in 48 studies.

Table 2
Interpreted definitions of Eager Test.

Interpreted definition	Number of studies[a]		
	Based on DS1[b]	Based on DS2[c]	NA[d]
Checking multiple methods of the object under test	11	1	1
Checking multiple methods of the class under test	5	1	0
Checking or using multiple methods of the class under test	2	0	0
Checking multiple different functionalities	2	2	3
Having multiple methods of the object under test	4	0	1
Having multiple methods of the class under test	2	0	1
Having multiple calls to multiple production methods	4	2	2
Exercising multiple non-related features of the system under test	1	0	0
Having multiple assertions	0	0	1

[a] The studies' references are available via https://figshare.com/s/34c20027fb4e5c2344d6.
[b] DS1 — Based on the definition of Eager Test given by Van Deursen et al. (2001).
[c] DS2 — Based on the definition of Eager Test given by Meszaros (2007).
[d] NA — The authors did not specify the source(s) of definition.

Table 3
Interpreted definitions of Assertion Roulette.

Interpreted definition	Number of studies[a]		
	Based on DS1[b]	Based on DS2[c]	NA[d]
Having multiple assertion statements without explanation	27	3	6
Having multiple assertion statements that do not provide any description of why they failed	2	0	1
Having multiple assertion statements with at least one that does not provide any description of its failure	3	0	0
Having multiple assertions	2	0	0
Having an assertion statement that does not contain an explanation	0	0	1
Being hard to tell which of several assertions failed	1	3	1

[a] The studies' references are available via https://figshare.com/s/34c20027fb4e5c2344d6.
[b] DS1 — Based on the definition of Eager Test given by Van Deursen et al. (2001).
[c] DS2 — Based on the definition of Eager Test given by Meszaros (2007).
[d] NA — The authors did not specify the source(s) of definition.

Table 4
Proposed detection rules for Eager Test.

Proposed rule	Source of definition[a]	Number of studies[b]
Number of production type method invocations in a test case	S1, S2	7
JUnit classes having at least one method that uses more than one method of the tested class	S1	4
More than one method call from class under test in a test case	S1	1
The test method instructions contain at least two cycles of non-verification instructions followed by verification instructions	S1	1
The low textual similarity among the method calls in a test case	S1	1
The test must have more than one assertion and at least one assertion is not on the result of a get method	S1	1

[a] S1 — Based on the definition of Eager Test given by Van Deursen et al. (2001), S2 — Based on the definition of Eager Test given by Meszaros (2007), NA — The authors did not specify the source(s).
[b] The studies' references are available via https://figshare.com/s/34c20027fb4e5c2344d6.

revolve around one aspect: assertion(s) without explanation. For Eager Test, the interpreted definitions scatter from a unit test having multiple method calls of the object or class under test to a unit test checking multiple methods or functionalities. Note that having multiple method calls is not the same as checking/verifying several methods in a test case. Ultimately, the findings show diverse interpretations of the Eager Test smell in the literature, even though they are mainly based on the same two definitions (see Table 1).

4.2. Eager test detection rules (RQ2)

Since each type of test smells, especially Eager Test, has been interpreted differently, their detection rules are also divergent, as summarized in Tables 4 and 5. On the one hand, with Assertion Roulette, most of the proposed detection rules are about the assertion statements having no explanation except one study (Junior et al., 2020), which stated that an Assertion Roulette test is the case of "putting together tests that could be run separately". This particular detection rule of Assertion Roulette is broad enough to cover the Eager Test smell as well. On the other hand, while the proposed detection rules for Eager Test are more divergent, they can be grouped into three main categories: (1) the number of invoked methods, (2) the number of cycles of non-verification instructions followed by verification instructions, (3) the textual similarity among the tested methods. The first category of detection rules is the major one (12 out of 14 studies), while the other two categories each came from two other studies.

It is worth reemphasizing that the reason for including Assertion Roulette in the review was because Assertion Roulette and Eager Test were claimed to be the same (Garousi and Küçük, 2018) and we did not want to miss finding definitions or detection rules of the Eager Test smell. Nevertheless, the analysis presented earlier shows that the original definition of Assertion Roulette is very different from Eager Test, and the overlap in the interpreted definitions and rules does not cover more relevant aspects than those targeted explicitly for Eager Test. Therefore, our following discussion focuses on the definitions and detection rules for Eager Test only (as presented in Tables 1, 2, and 4).

Table 5
Proposed detection rules for Assertion Roulette.

Proposed rule	Source of definition[a]	Number of studies[b]
Some tests fail and it is not possible to identify the failure cause	S1, S2	1
Putting together tests that could be run separately	S1, S2	1
The number assertions without description	S1, NA	13 (3 from NA)
The number of assertions without additional/optional argument	S1	2

[a] S1 — Based on the definition of Eager Test given by Van Deursen et al. (2001), S2 — Based on the definition of Eager Test given by Meszaros (2007), NA — The authors did not specify the source(s).
[b] The studies' references are available via https://figshare.com/s/34c20027fb4e5c2344d6.

4.3. Limitations of the eager test detection rules (RQ3)

At first sight, the definitions of the Eager Test smell given by Van Deursen et al. (2001) and Meszaros (2007) seem quite different as the former focused on methods of the object under test while the latter stressed the functionalities under test. To further investigate the difference, we looked into the extra details provided by Meszarous for the Eager Test smell. Meszarous explained that the root cause of the Eager Test smell is that practitioners verify multiple "test conditions" in a single test. The author further described a test condition as follows:

"A test condition is a particular behavior of the system under test (SUT) that we need to verify. It can be described as the following collection of points:

If the SUT is in some state **S1**, and

I exercise the SUT in some way **X**, then

the SUT should respond with **R** and
the SUT should be in state **S2**".

In the context of unit testing, it is reasonable to consider **X** as a method under test. Hence, this definition of the Eager Test smell given by Meszarous is also oriented around verifying multiple methods, which is ultimately aligned with the definition from Van Deursen et al. (2001), i.e., "a test method checks several methods of the object to be tested" (details in Table 1).

Nevertheless, as we can see from the literature review, not all interpreted definitions of the Eager Test smell (Table 2) focus on the same issue as the original definitions. The interpreted definitions mainly focused on the number of methods invocations, which is not the same as verifying multiple methods. One or more invocations of methods other than the method under test might be required as part of the test setup or to acquire intermediate results in order to verify the outcome(s) of the method under test. This issue with the interpreted definitions has also been raised by other researchers (Panichella et al., 2022; White and Krinke, 2022). Therefore, focusing on the number of method invocations does not adequately capture the original definitions of the Eager Test smell. However, counting the number of method invocations has been used as a lodestar to establish most detection rules (Table 4) and verify the detection results.

Furthermore, our observation of the current state of the art is that many studies (Breugelmans and Van Rompaey, 2008; Bavota et al., 2012; Tufano et al., 2016b; De Bleser et al., 2019a; Palomba et al., 2018; Peruma et al., 2020; Santana et al., 2022) have used these imprecise interpreted definitions and detection rules to build their detection tools which are in turn used to further analyze the prevalence of eager tests and their impact on the quality of test code (in terms of Comprehensibility and Maintainability). The problem is that this introduces false positives, as reported in the recent studies of Panichella et al. (2022, 2020).

On top of that, the assessment is not agreed upon by practitioners (Spadini et al., 2020; Schvarcbacher et al., 2019) even though practitioners highly agree that a good test case should be atomic, i.e., one test case should be testing one aspect of a requirement or one functionality (Kochhar et al., 2019), which is the exact opposite of eager tests. Hence, the current assessment of the prevalence and impact of eager tests requires re-evaluation as it might not be adequate.

5. Consolidated eager test definition and heuristic (RQ4)

To address the issues with the existing detection rules, we propose a more precise definition of the Eager Test smell and a novel heuristic that can operationalize the proposed definition to detect eager tests. In this section, we describe how we derived our definition of the Eager Test smell from the original definitions given by Van Deursen et al. (2001) and Meszaros (2007) and well as how we constructed our heuristic.

5.1. Proposed definition of the Eager Test smell

According to the original definition of the Eager Test smell given by Van Deursen et al. (2001) ("a test method checks several methods of the object to be tested"), it is unclear whether checking *multiple method calls* of *the same method* is acceptable, i.e., classified as not eager. Likewise, Meszarous' definition emphasizes "verifying multiple "test conditions" in a single test", in which we argued that a "test condition" also means a method to be tested (more details in Section 4.3). Hence, with both of these broad definitions, it is unclear whether a test case is classified as an eager test when it verifies the outcomes of *multiple method calls* of *the same method*.

Nevertheless, we argue that it is important to be clear about the case of verifying multiple method calls of the same method. For example, a test case can invoke multiple method calls of a single method to check corner cases (negative value, zero, positive value). Since the object under test (OUT) behaves differently in each corner case, each method call presents a different behavior of OUT. In other words, the test case actually tests different behaviors of OUT even though there is only one method under test. Hence, our interpretation of these definitions is that a test case is an eager test if it verifies the outcomes of multiple method calls, disregarding whether these calls are from the same method or not. Based on our interpretation, our novel, more precise definition of the Eager Test smell is as follows:

> A test case is NOT an eager test when all of its assertions assess the outcome(s) of a single method call of the class under test (CUT).

The definition is based on the analysis of what a method does (presented by its outcome(s)) and what is actually assessed by assertion statements. The outcome(s) of a method call can be: the return value/object(s) AND/OR the state change of an object caused by the method call. The object with the changed state can either be from the class under test (CUT) or from another class. The second case (object from a class different from CUT) occurs when the object is passed by parameter to the method under test (MUT).

5.2. Description of the heuristic

In this section, we describe our heuristic which can operationalize our proposed definition of the Eager Test smell in the context of unit test cases written in Java. The heuristic pseudocode is in Pseudocode 1.

5.2.1. Method classification

Since our proposed definition of the Eager Test smell is based on the analysis of what a method does (presented by its outcome(s)), we need to have a systematic classification of method types. In this study, we relied on existing taxonomies (Dragan et al., 2006, 2009; Moreno and Marcus, 2012) of method stereotypes in the Object-oriented (OO) paradigm. These taxonomies categorize methods into four main types: creational, mutator, collaborator, and accessor. Table 6 presents these main types and their sub types. The table also shows how we adapt the method stereotypes proposed by the taxonomies in our heuristic.

In addition to the adapted method types, we propose two sub types for a **producer**. Note that these subtypes make sense only in the context of unit testing.

- **Internal producer**: the producer method is implemented (or overridden) in the method's class, i.e., the class under test (CUT).
- **External producer**: the producer method is implemented in a class different from the class under test. This producer method is called to compute some result based on the object under test's attribute(s). For instance, this external class can be the parent class and any other classes in the external libraries.

Note that we do not distinguish *internal* and *external* for **get**, **creational**, and **mutator** methods. Since a **get** method is simply used to retrieve an object's attribute(s), it does not matter whether the get method is implemented in the CUT or not. Likewise, if a method like **creational** and **mutator** is implemented outside CUT, there is no reason to test this method in CUT. Testing such a method should be handled by a separate test case at the class the method belongs to so that there is no mix of the two units (CUT) in one single test case. Hence, **creational** and **mutator** methods we refer to in this heuristic are the methods implemented in CUT. Table 7 presents the final list of method types used in our heuristic.

5.2.2. Steps of the heuristic

Our heuristic contains three steps as explained below.

Table 6
Method stereotypes as defined by Moreno and Marcus (Moreno and Marcus, 2012) and our adaptations.

Category	Stereotype	Description (Moreno and Marcus, 2012)	Action	Reasons	Our description
Accessor	Get	Returns a local field directly	Included	Needed for the heuristic	Returns an object's attribute without modifying the attribute.
Accessor	Predicate	Returns a Boolean value that is not a local field	Merged with Property	We are interested in only the difference between a **get** and the others (**predicate** and **property**). Therefore, we group **predicate** and **property** into one type, called **producer**	A **producer** computes a result based on an object's attribute(s). It means that a **producer** does not change the object's state, and its return type is not of interest. A **producer** can return a Boolean result (as a **predicate**) or have other return types (as a **property**).
Accessor	Property	Returns information about local fields	Merged with Predicate		
Accessor	Void-accessor	Returns information about local fields through the parameters	Excluded	It does not make sense in the context of Java programming.	N/A
Mutator	Set	Changes only one local field	Merged with Command and Non-void command	We are not interested in different stereotypes under this category as the number of attributes changed by a mutator and its return type is not needed for the heuristic.	A **mutator** is a method that changes the state of the object to which it belongs, i.e., must change one or more attributes of the object and optionally have a return value/object.
Mutator	Command	Changes more than one local fields	Merged with Set and Non-void command		
Mutator	Non-void command	Command whose return type is not void or Boolean	Merged with Set and Command		
Creational	Constructor	Invoked when creating an object	Merged with Copy-constructor and Factory	How an object is created is not relevant to the heuristic. Hence, we merged Constructor, Copy-constructor and Factory into one called Creational	A **creational** method is responsible for creating an object.
Creational	Copy-constructor	Creates a new object as a copy of the existing one	Merged with Constructor and Factory		
Creational	Factory	Instantiates an object and returns it	Merged with Constructor and Copy-constructor		
Creational	Destructor	Performs any necessary cleanups before the object is destroyed	Excluded	It does not make sense in the context of Java programming.	N/A
Collaborational	Collaborator	Connects one object with another type of objects	Excluded	A Collaborational, according to the authors, works on objects of classes different from itself and can also be either an accessor or mutator. Hence, the main difference is which object the method works on. Since this differentiation on the target object is irrelevant to our heuristic, we only care if a method is a mutator or an accessor instead of further classifying it as a collaborator-mutator or collaborator-accessor.	N/A
Collaborational	Controller	Provides control logic by invoking only external methods	Excluded		
Collaborational	Local controller	Provides control logic by invoking only local methods	Excluded		

Table 7
Method types used in our heuristic.

Method type	Sub type	CUT[a]	Effect				
			Initialize attribute	Retrieve attribute	Change attribute	Return attribute	Return value/object
Creational		✓	✓	–	–	–	[Optional]
Mutator		✓	–	✓	✓	–	[Optional]
Accessor	Get	✓[b]	–	✓	–	✓	–
	Internal Producer	✓	–	✓	–	–	✓
	External Producer	–	–	✓	–	–	✓

[a] CUT: (implemented in) Class under test.
[b] GET method can be implemented in CUT or outside CUT.

Step 1. For each method call in a test case, step 1 of the heuristic is to collect information that should be assessed in order to verify the outcome(s) of the method call. Overall, the information collected by step 1 depends on the method type (Section 5.2.1) as explained further below, and summarized in Table 8.

- A **creational** or **mutator** method is for initializing or modifying an object's attributes. The method might also return some value. Hence, to verify if a creational and mutator method behaves correctly, one needs to check if the attributes are modified correctly, and if the return value/object(s) (if any) is produced as expected. Hence, the information to be collected is the modified attribute(s) and the return value/objects (if any).
- For a **get** (a sub type of **accessor**), we do not collect any information as we argue that it is a helper method in the context of unit testing, i.e., it is called to verify the outcome(s) of another method.
- An **internal producer** (a sub type of **accessor**) computes the return value/object based on an object's attribute(s) without modifying the object's attributes. Since the method's implementation (to produce the return value/object) is in the class under test, we argue that the purpose of invoking such a method is to verify if the implementation is correct. Hence, the information to be collected is the return value/object.
- An **external producer** (a sub type of **accessor**) is similar to an **internal producer** in the sense that the external producer also does not modify an object's state and the return value is computed based on the object's attribute(s). However, the method's implementation (to produce the return value/object) is located outside of the class under test. Therefore, we argue that this method is invoked to check whether the object's attribute(s) is modified (or initialized) correctly by a mutator (or creational). In other words, we regard an **external producer** as a helper method in the context of unit testing, i.e., they are called to verify the outcome(s) of another method (similar to a **get** method). Hence, no information needs to be collected.

Table 8
Information to collect in Step 1 of the heuristic.

Method type	Retrieved attribute	Modified (or initialized) attribute	Return value/object
Creational		✓	[if any]
Mutator		✓	[if any]
Get			
Internal Producer			✓
External Producer			

Step 2. Step 2 of the heuristic is to collect the information that **all** assert statements verify. To acquire such information, we first collect the information that each assert statement verifies, then combine them together.

Similar to the logic behind Step 1, the information collected from each assert statement depends on the types of methods (Section 5.2.1) invoked in the assert statement as explained further below, and summarized in Table 9.

Table 9
Information to collect in Step 2 of the heuristic.

Method type	Retrieved attribute	Modified (or initialized) attribute	Return value/object
Creational		✓	[if any]
Mutator		✓	[if any]
Get	✓		
Internal Producer			✓
External Producer	✓		

- If the assert statement invokes a **creational** method (or contains the object(s) created by a **creational** method call), what the assert statement tries to verify is whether the object is created correctly by the **creational** method. Hence, the information to be collected is the created object and its attributes initialized by the **creational** method.
- If the assert statement invokes a **mutator** method (or contains return value/object(s) of a **mutator** method call), what the assert statement tries to verify is whether the attributes of the target object are modified correctly by the **mutator** method, and if the return value/object(s) (if any) is produced as expected. Hence, the information to be collected is the attributes modified by the **mutator** method AND the value/object(s) returned by the **mutator** method.
- If the assert statement invokes a **get** method (or contains return values/objects of a **get** method call), we argue that developers do not try to check whether the **get** method is implemented

correctly. Instead, the invocation of the **get** method is to verify the outcome(s) of another method. One typical example is that the assert statement assesses a get method's return value, which is an object's attribute, in order to verify if the attribute is modified correctly by a precedent mutator method. Hence, the information to be collected is the attribute(s) or object(s) returned by the **get** method.
- If the assert statement invokes an **external producer** method call (or contains return values/objects of an **external producer** method call), we argue that the purpose of invoking the **external producer** method is the same as invoking a get method. Hence, the information to be collected is the attribute(s) used by the **external producer** method to produce the return values or objects.
- If the assert statement invokes an **internal producer** method (or contains return values/objects of an **internal producer** method call), we argue that what the assert statement tries to verify is the behavior of the **internal producer** method only. Hence, the information to be collected is the return values/objects of the **internal producer** method.

At this step (Step 2), since we need to combine the information verified by every assert statement together, it is important to distinguish different changes to an attribute due to different method calls (creational, mutator methods). In other words, the collected information should maintain the traceability between changes made to a modified (or initialized) attribute and the corresponding mutator (or creational) method calls. To identify such a connection, we argue that if an attribute assessed by an assert statement is modified by multiple method calls, then among these method calls, the method call right before the assert statement is the one that the change of the attribute associates with. For example:

```
public void test1() throws Exception{
    mutator1[attri1, attr2]
    mutator2[attr1]
    assertion1[attr1]
    mutator3[attr1, attr3]
    assertion2[attr1]
}
```

In this example, *assertion1* assesses attribute *attr1*, which is modified by mutator method *mutator2*, and assertion2 assesses *attr1*, which is modified by mutator method *mutator3*. To distinguish between the two connections: (1) *mutator2* and *attr1* assessed by *assertion1*, (2) *mutator3* and *attr1* assessed by *assertion2*, we update the attribute's name as follows before collecting the required information:

```
public void test1() throws Exception{
    mutator1[attr1, attr2]
    mutator2[attr1a]
    assertion1[attr1a]
    mutator3[attr1b]
    assertion3[attr1b]
}
```

Step 3. Based on the information collected by Step 1 and Step 2, at Step 3 of the heuristic, we want to check if the information verified by all of the assert statements (Step 2) is the outcome(s) 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.

5.2.3. Example

In this section, we use a simple automatically generated test case extracted from Panichella et al.'s study (Panichella et al., 2022) as shown below to demonstrate how the heuristic works.

```
@Test
  public void test2() throws Throwable {
      // MethCall_1 - type : creational
      DirEntry dirEntry0 = new DirEntry();

      // MethCall_2 - type : mutator
      dirEntry0.setSize((-1053L));

      // MethCall_3 - type : creational
      DirEntry dirEntry1 = new DirEntry();

      // MethCall_4 - type : internalproducer
      boolean boolean0 = dirEntry0.equals((Object)
          dirEntry1);

      // Assert_1 contains MethCall_5 - type : get
      assertEquals((-1053L), dirEntry0.getSize());

      // Assert_2
      assertEquals(false, boolean0);
  }
```

Based on the heuristic, for each method call in a test case, step 1 of the heuristic is to collect information that should be assessed in order to verify the outcome(s) of the method call. Hence, the information collected in Step 1 is as follows:

$MethOutcome_1 = \{dirEntry0\}$

$MethOutcome_2 = \{dirEntry0.size\}$

$MethOutcome_3 = \{dirEntry1\}$

$MethOutcome_4 = \{boolean0\}$

$MethOutcome_5 = \{dirEntry0.size\}$

Likewise, Step 2 of the heuristic is to collect the information that all assert statements verify. To acquire such information, we first collect the information that each assert statement verifies, then combine them together. Hence, the information collected in Step 2 is as follows:

$VerifiedInfo_1 = \{dirEntry0.size\}$

$VerifiedInfo_2 = \{boolean0\}$

$VerifiedInfo = \{dirEntry0.size,\ boolean0\}$

According to Step 3 of the heuristic, what needs to be checked is whether the information verified by all of the assert statements (Step 2) is the outcome(s) of one single method call (Step 1) in the test case. Based on the information collected above, we can see that

$\forall i, VerifiedInfo \not\subseteq \ MethOutcome_i$

This means that the test case **does not** verify the outcome(s) of a **single** method call; hence, it is an eager test.

5.2.4. Limitations of the heuristic implementation

This section presents the limitations when implementing our heuristic.

***Metaprogramming*.**

Since our heuristic 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 (Lilis and Savidis, 2019). In metaprogramming, we need a dynamic code analysis in order to identify the method types and collect the required data for the heuristic. For example, the method below needs a dynamic analysis to know its type.

```
protected Object getBlobObject(String getterName)
    throws BeanBinException {
  try {
    Method getter = getGetter(getterName);
    return getter.invoke(entity, new Object[0]);
  } catch (Exception e) {
    throw new BeanBinException
    ("BlobClassTemplate.getBlobObject: " +
        e.getMessage(), e);
  }
}
```

With the static code analysis, it is not possible to know what happens to the object "entity" (the object of the class under test *BlobClassTemplate*) when the method *invoke* is called. Consequently, the method type of *invoke* cannot be identified. It might still be feasible with human help, but it becomes difficult when automating the heuristic.

External methods and objects outside class under test (CUT).
With external methods (methods outside of the production code), our heuristic assumes that they do not change the state of any object of CUT. That is because it is not always possible to have access to the source code of such external methods. Nonetheless, it is worth emphasizing that with our heuristic, it does not matter whether a method in the production code affects objects of CUT or outside of CUT. As long as the affected object can be identified, it should be possible to collect the related information required by the heuristic.

5.2.5. Scalability of the heuristic

The heuristic relies on the static analysis of the source code, which might slow down its performance. Not being able to deal with metaprogramming also restricts its scalability. While the current scope of the heuristic is limited to Java, the heuristic can be extended to other object-oriented programming languages. In our recent work, we are also developing a tool that operationalizes the proposed heuristic to further boost scalability by reducing the manual effort.

6. Comparison with existing rules (RQ5)

In this section, we first present the detection rules selected for comparison. Then, we present our findings regarding the levels of agreement between each pair of detection rules, followed by our findings regarding the agreement levels between the heuristic and each detection rule.

6.1. Existing detection rules selection outcome

From the literature review, we identified in total six rules to detect eager tests. Details of the rules are in Section 4.2 and Table 4. Among these six rules, we could not apply two of them in our manual analysis. The first one, proposed by Bavota et al. (2012), works at the test suite level, while we wanted to detect eager tests at the test case level instead. The second rule, proposed by Palomba et al. (2018), relies on the textual similarity among the tested methods in a test case to classify if the test case is an eager test. The rule was excluded since calculating such similarity could not be done manually.

Consequently, we had four detection rules from the literature to compare with our heuristic. The selected rules were implemented in four (Breugelmans and Van Rompaey, 2008; De Bleser et al., 2019b; Peruma et al., 2020; Virgínio et al., 2020) out of the seven existing tools (Aljedaani et al., 2021), capable of detecting eager tests, while the excluded rules formed the basis for the remaining three tools (Lambiase et al., 2020; Palomba et al., 2018; Bavota et al., 2012). Thus, we believe our study has comprehensively considered all relevant detection rules and tools available in the literature.

Table 10 presents the selected rules for comparison with our heuristic. Note that one of these four rules comes with three different thresholds. Hence, we decided to consider each threshold as an independent rule to achieve a thorough comparison. It is also worth emphasizing that we studied the primary studies proposing these rules as well as the source code of the detection tools that implemented the rules to make sure that we understood the rules correctly. The information in column **Notes** of Table 10 summarizes the extra details we learned from this extra step.

Fig. 1 presents Cohen's kappa calculated for every pair of the detection rules (DR1–DR4) and between our heuristic (EagerID) and each detection rule. Note that we call our heuristic *EagerID* in Fig. 1 for the sake of simplification.

Our manual assessment involved three separated data sets: (a) 100 manually written test cases from Panichella et al.'s study, (b) 100 manually written test cases from Sharma et al.'s study, and (c) 100 automatically generated test cases from Panichella et al.'s study (details in Section 3.2.1). Hence, the left side of Fig. 1 indicates which data set the kappa values were calculated from. For example, for the data set (a), the first row of Fig. 1 shows the kappa values between two detection rules, DR1 and DR2.1, as 0.4751.

Also note that in Fig. 1, we presented the kappa values according to each detection rule and the heuristic, instead of showing them in three tables, one for each data set. The purpose is to provide a better overview of how each detection rule and the heuristic agree with each other in terms of identifying eager tests from all data sets. For example, the first three rows in Fig. 1 present the kappa values between the detection rule DR1 and each of the following: DR2.1, DR2.2, DR2.3, DR4, and the heuristic, from each data set. Based on Landis and Kock's guideline (Landis and Koch, 1977), we observed five levels of agreement ranging from "no agreement" (in red) to "substantial agreement" (in green) as shown in Fig. 1.

6.2. Agreement between the selected detection rules

Overall, Cohen's kappa for each pair of the selected detection rules varied from "no agreement" to "substantial agreement" (Fig. 1). It indicates a significant misalignment between the selected rules in detecting eager test cases. Nevertheless, looking more closely at the kappa values, we could see that some detection rules tended to have higher levels of agreement with certain detection rules than with the other rules.

With the detection rule DR1, its agreement with the detection rule DR2.1 was much higher than with the other rules. It is reasonable as both DR1 and DR2.1 share a common principle, i.e., counting the number of method calls from the class under test, and their thresholds are not so different from each other (2 method calls for DR1 and 3 method calls for DR2.1). Meanwhile, although DR2.2 and DR2.3 are also based on the number of method calls, their thresholds (5 method calls) were much higher than DR1 and DR2.1. Therefore, it makes sense that DR1 has less agreement with DR2.2 and DR2.3.

One interesting observation is that even though the detection rule DR3 is not based on the same principle as DR1, the levels of agreement between DR1 and DR3 for all data sets were not distinctively different from the agreement levels between DR1 and DR2.1 or between DR1 and DR2.2. One possible reason is that DR3 requires at least two cycles of non-verification instructions followed by verification instructions in a test case. A cycle of non-verification instructions is likely to involve at least one method call. Hence, an eager test based on DR3 is more likely to have at least two method calls, which fulfills the requirement of DR1.

While DR1 shared some agreement (even though not very high) with DR2.1, DR2.2, DR2.3, and DR3, as analyzed above, DR1 had no

		DR1	DR2.1	DR2.2	DR2.3	DR3	DR4	EagerID
Data set (a)	DR1		0.4751	0.1899	0.2359	0.3353	-0.1678	0.3782
Data set (b)	DR1		0.3253	0.2343	0.3371	0.1189	-0.0008	0.2281
Data set (c)	DR1		0.4625	0.0286	0.1872	0.1268	-0.1816	0.0511
Data set (a)	DR2.1	0.4751		0.5039	0.3844	0.3256	-0.1538	0.3653
Data set (b)	DR2.1	0.3253		0.5401	0.5344	0.3066	0.1581	0.4479
Data set (c)	DR2.1	0.4625		0.0922	0.4180	0.1353	-0.1812	0.0541
Data set (a)	DR2.2	0.1899	0.5039		0.3801	0.1446	-0.0538	0.1446
Data set (b)	DR2.2	0.2343	0.5401		0.7302	0.2784	0.0796	0.2714
Data set (c)	DR2.2	0.0286	0.0922		0.2469	0.0005	-0.0315	0.0035
Data set (a)	DR2.3	0.2359	0.3844	0.3801		0.2880	-0.2890	0.1659
Data set (b)	DR2.3	0.3371	0.5344	0.7302		0.2886	0.1355	0.3083
Data set (c)	DR2.3	0.1872	0.4180	0.2469		0.3609	-0.0135	0.0569
Data set (a)	DR3	0.3353	0.3256	0.1446	0.2880		-0.0700	0.6736
Data set (b)	DR3	0.1189	0.3066	0.2784	0.2886		0.5917	0.7252
Data set (c)	DR3	0.1268	0.1353	0.0005	0.3609		0.2206	0.3017
Data set (a)	DR4	-0.1678	-0.1538	-0.0538	-0.2890	-0.0700		0.0856
Data set (b)	DR4	-0.0008	0.1581	0.0796	0.1355	0.5917		0.6524
Data set (c)	DR4	-0.1816	-0.1812	-0.0315	-0.0135	0.2206		0.4229
Data set (a)	EagerID	0.3782	0.3653	0.1446	0.1659	0.6736	0.0856	
Data set (b)	EagerID	0.2281	0.4479	0.2714	0.3083	0.7252	0.6524	
Data set (c)	EagerID	0.0511	0.0541	0.0035	0.0569	0.3017	0.4229	

< 0.01 No agreement
0.01 – 0.20 Slight agreement
0.21 – 0.40 Fair agreement
0.41 – 0.60 Moderate agreement
0.61 – 0.80 Substantial agreement

Data set (a): 100 **manually written** test cases from **Panichella et al.'s study**
Data set (b): 100 **manually written** test cases from **Sharma et al.'s study**
Data set (c): 100 **automatically generated** test cases from **Panichella et al.'s study**

Fig. 1. Degree of agreement (using Cohen's kappa coefficient) in the eager test classification between the detection rules in the literature (Table 10) and the proposed heuristic (EagerID).

Pseudocode 1 Our heuristic to detect eager test

Let $MethCalls$ be a list containing all method calls in the test case
Let $Asserts$ be a list containing all assertion statements in the test case

```
begin Step 1:
Initialize array MethOutcomes ← ∅
foreach method call MethCall_i ∈ MethCalls do
    Identify the method type of MethCall_i based on the descriptions of method types
    Initialize MethOutcome_i ← ∅

    if type of MethCall_i is Get OR External Producer then
        do nothing
    else if type of MethCall_i is Creational then
        MethOutcome_i ← all attributes initialized by MethCall_i
        if MethCall_i returns any value or object then
            MethOutcome_i ← MethOutcome_i ∪ all return values and objects of MethCall_i
        end if
    else if type of MethCall_i is Mutator then
        MethOutcome_i ← all attributes modified by MethCall_i
        if MethCall_i returns any value or object then
            MethOutcome_i ← MethOutcome_i ∪ all return values and objects of MethCall_i
        end if
    else if type of MethCall_i is Internal Producer then
        MethOutcome_i ← all return values and objects of MethCall_i
    end if
end for
Add MethOutcome_i to MethOutcomes
end Step 1

begin Step 2:
Initialize VerifiedInfo ← ∅
foreach assert statement Assert_j ∈ Asserts do
    VerifiedInfo_j ← all objects and attributes passed as arguments in Assert_j
    MethCalls_Assert_j ← all method calls in Assert_j

    foreach method call MethCall_k ∈ MethCalls_Assert_j do
        Identify the method type of MethCall_k based on the descriptions of method types

        if type of MethCall_k is Get OR External Producer then
            VerifiedInfo_j ← VerifiedInfo_j ∪ all attributes retrieved by MethCall_k
        else if type of MethCall_k is Creational then
            VerifiedInfo_j ← VerifiedInfo_j ∪ all attributes initialized by MethCall_k
            if MethCall_k returns any value or object then
                VerifiedInfo_j ← VerifiedInfo_j ∪ all return values and objects of MethCall_k
            end if
        else if type of MethCall_k is Mutator then
            VerifiedInfo_j ← VerifiedInfo_j ∪ all attributes modified by MethCall_k
            if MethCall_k returns any value or object then
                VerifiedInfo_j ← VerifiedInfo_j ∪ all return values and objects of MethCall_k
            end if
        else if type of MethCall_i is Internal Producer then
            VerifiedInfo_j ← VerifiedInfo_j ∪ all return values and objects of MethCall_k
        end if
    end for
    VerifiedInfo ← VerifiedInfo ∪ VerifiedInfo_j
end for
end Step 2
```

agreement with DR4 for all data sets. It is probably due to the fact that DR1 and DR4 are based on completely different principles. (DR4 requires at least two assertions, and at least one assertion should not be on the result of a get method.) Hence, their eager test detection outcomes were not aligned with each other.

Next, we looked at the agreement levels between the three detection rules DR2.1, DR2.2, and DR2.3. Since these rules share the same principle, that is, counting the number of method calls, it is reasonable that their agreement was relatively high compared to the rest. For example, while both DR2.1 and DR2.2 count the number of method calls from the class under test, the threshold of DR2.1 was three method

begin *Step 3:*
Initialize $isEagerTest \leftarrow false$
Initialize $NumOfVerifiedMeths \leftarrow 0$
for all $MethOutcome_l$ in $MethOutcomes$ **do**
  if $VerifiedInfo \nsubseteq MethOutcome_l$ **then**
    $isEagerTest \leftarrow true$
    Return $isEagerTest$
  end if
end for
for $l \leftarrow 1$ **to** m **do**
  if $VerifiedInfo \subseteq MethOutcome_l$ **then**
    $NumOfVerifiedMeths \leftarrow NumOfVerifiedMeths + 1$
  end if
  if $NumOfVerifiedMeths \geq 2$ **then**
    $isEagerTest \leftarrow true$
    Return $isEagerTest$
  end if
end for
return $isEagerTest$
end *Step 3*

Table 10
Eager test detection rules from the literature to compare with our heuristic.

ID	Detection rule	Notes	Ref
DR1	At least 2 method calls from **CUT** in a test case	Excluding constructor	De Bleser et al. (2019a)
DR2.1	At least 3 PTMI in a test case	Excluding constructor, considering only methods from **CUT**	Breugelmans and Van Rompaey (2008)
DR2.2	At least 5 PTMI in a test case	Excluding constructor, considering only methods from **CUT**	Peruma et al. (2020)
DR2.3	At least 5 PTMI in a test case	Excluding constructor, considering only methods from **production code** *(code developed by the project team that will end up in the released product)*	Van Rompaey et al. (2007)
DR3	At least two cycles of non-verification instructions followed by verification instructions	None	Pizzini et al. (2023)
DR4	At least 2 assertions in a test case and at least one assertion is not on the result of a get method	No definition of get methods	Panichella et al. (2020)

CUT: Class under test
PTMI: production type method invocations

calls and the threshold of DR2.2 was five method calls. Hence, if DR2.2 classified a test case as an eager test, DR2.1 would naturally give the same assessment. With the detection rule DR2.3, as it requires at least five method calls from the production code, DR2.3 covers DR2.2 which requires at least five method calls from a smaller scope, i.e., the class under test. It means that test cases classified as eager tests by DR2.2 would be covered by DR2.3 as well. We argue that their only differences in threshold and the scope of the counted methods caused the misalignment in their detection outcomes. Meanwhile, this group of detection rules had either no agreement or, at most, slight agreement with the detection rule DR4. It is probably because DR4 is based on a completely different principle than DR2.1, DR2.2, and DR2.3 as explained earlier.

While the detection rule DR4 appeared to have high levels of disagreement with the rest, it had the least conflict with DR3. We believe that is because of an implicit common factor in these two rules. More specifically, DR3 requires at least two cycles of non-verification instructions followed by verification instructions in a test case, which implies at least two assertions. Having at least two assertions is also a requirement of DR4. However, DR4 put more restrictions on the assertions by requiring that at least one assertion must be not on the result of a get method. It led to the disagreement between DR4 and DR3.

6.3. Agreement between our heuristic and the selected detection rules

In the previous section, we presented our findings regarding the agreement levels between each pair of the detection rules. Here, we look at the extent to which our heuristic agreed with each detection rule in terms of identifying eager tests from the same data sets: (a) 100 manually written test cases from Panichella et al.'s study, (b) 100 manually written test cases from Sharma et al.'s study, and (c) 100 automatically generated test cases from Panichella et al.'s study (details in Section 3.2.1).

Fig. 2 shows the percentages of test cases classified as eager tests or not based on our heuristic and each detection rule. For example, on the one hand, 23% of the manually written test cases from Panichella et al.'s study were classified as eager tests according to the detection rule DR1 (At least 2 method calls from class under test (CUT) in a test case) but not by our heuristic. On the other hand, our heuristic marked 6% of the same data set as eager tests, while the detection rule DR1 did not. To better understand the agreement levels between our heuristic and each detection rule, we present the calculated kappa between our heuristic and each detection rule in Fig. 1. Overall, based on both Figs. 1 and 2, we observed the same pattern as between the detection rules, that is, there was little agreement between our heuristic and each rule.

Fig. 2. Eager test classification based on our heuristic and the detection rules in the literature (Table 10).

First, between the two types of test cases (manually written versus automatically generated), the eager test classification for the automatically generated test cases had more disagreement. One factor that might potentially explain this observation is that the automatically generated test cases often follow one pattern as they were all created by EVOSUITE. For example, it appeared very frequently that EVOSUITE inserted an assert statement to verify the outcome of a creational method call in the Set-up of a test case. Naturally, the test case also has at least another assert statement in the Verify part to verify the outcome(s) of another method call. The test case, therefore, tries to verify the outcomes of multiple method calls, which is classified as an eager test, according to our heuristic. However, the other detection rules from the literature do not have the same conclusion. On the contrary, the manually written test cases were created by different developers with different styles or mindsets when designing test cases. Hence, their test cases are more diverse than the automatically generated ones.

Second, we also noticed that there are some detection rules having higher levels of agreement with our heuristic than others. For all data sets, there was either no agreement or, at most, fair agreement between the detection rule DR2.2 (At least 5 method calls from CUT in a test case) and our heuristic. It makes DR2.2 have the highest disagreement with our heuristic. In contrast, the detection rule DR3 (At least two cycles of non-verification instructions followed by verification instructions) was the one having the least disagreement with our heuristic, especially in the case of the manually written test cases (with substantial agreement). Meanwhile, the levels of agreement between our heuristic and two detection rules DR2.1 and DR4 fluctuated, ranging widely from slight agreement to moderate agreement (with DR2.1), or from slight agreement to substantial agreement (with DR4). We further discuss the root causes of the differences between our heuristic and the selected detection rules in Section 7 and their implications in Section 7.1.

6.4. Agreement between our assessment and the assessments from the studies of Panichella et al. (2022) and Sharma et al. (2023)

With Panichella et al.'s study, as explained earlier in Section 3.2.1, the authors proposed the detection rule DR4. However, since they analyzed the eagerness of test suites, not test cases, we could not conduct a direct comparison in the assessment results. Nevertheless, the authors did mention that 39 out of the 49 manually written test suites (approximately 80%) that they assessed were eager according to the detection rule DR4. Meanwhile, our manual assessment showed that 57% of the assessed test cases (manually written) were eager tests. In terms of automatically generated test cases, also based on DR4, Panichella and his colleagues found that 20 out of the 100 test suites were eager. On the contrary, our heuristic classified 63 out of 100 test cases as eager tests. These differences show that the eagerness of test suites alone does not reflect real, rectifiable issues in test cases. On top of that, since our assessment also demonstrates the misclassification problem of the detection rule DR4, their assessment result might not be a reliable indicator for eager tests in practice.

In Sharma et al.'s study, the authors used the detection rule DR2.3 in their manual assessment. Our assessment agreed with Sharma et al. in terms of the test cases that they marked as eager tests, which are 13 in total. However, there were 33 test cases (out of 100) that were classified as eager by our heuristic but discarded by Sharma and his colleagues. This disagreement probably stems from the fact that we had no clear description regarding whether they actually considered any method call from the production code (as we assumed) or strictly limited to the class under test. Our second assumption is that their threshold was five method calls based on the test cases they marked as eager. This second assumption was made due to no extra details provided in their paper.

7. Discussion

In this section, we discuss the main aspect of our results, that is the root causes of the differences between our heuristic and the selected detection rules.

To understand the reasons behind the misalignment in detecting eager tests between our heuristic and each detection rule, we looked into the test cases that drew different assessment outcomes. Our investigation was in a pair-wise basic, i.e., between the heuristic and each of the detection rules.

According to the detection rule DR1, as long as a test case has only one method call from the class under test (CUT), it is not an eager test. However, the assessed test cases showed that even if a test case meets the condition, it can still be an eager test when having one or more assert statements in the Set-up (besides the assert statements for verifying the method call from CUT). Generally, the Set-Up contains the instantiation of the object under test which might require the invocations of methods (also known as helper methods) to bring the object under test to the desired state for testing. Hence, we argue that when a test case has some extra assert statements to verify the outcomes of method calls in the Set-up, the test case no longer verifies the outcome(s) of one single method call, which consequently should be classified as an eager test instead. One example test case to demonstrate this scenario is as follows:

```
Example Test Case #1
@Test
public void test3() throws Throwable {
    Home home0 = new Home();
    PhotoController photoController0 = new
        PhotoController(home0, (UserPreferences)
        null, (View) null, (ViewFactory) null,
        (ContentManager) null);
    assertNotNull(photoController0);

    long long0 = photoController0.getTime();
    assertEquals(1372766400496L, long0);
}
```

Another reason that makes this detection rule problematic is that it does not count constructors in the assessment. In other words, the rule does classify a test case that contains multiple assert statements verifying multiple constructors as an eager test. Nevertheless, we argue that this scenario satisfies the original definitions of eager tests and should not be discarded. One example is as follows:

```
Example Test Case #2
@Test
public void testConstr() throws Exception {
    HttpBot bot = new
        HttpBot(getValue("wikiMW1_13_url"));
    assertNotNull(bot);
    bot = new HttpBot(new
        URL(getValue("wikiMW1_13_url")));
    assertNotNull(bot);
}
```

Our assessment also showed that the detection rule DR1 could wrongly classify a non-eager test, i.e., a test case is not an eager test even though it contains at least two method calls from CUT. This falls into two typical scenarios. The first one is when a test case has multiple method calls from CUT in its Set-up and/or Teardown while having one single assert statement verifying a single method call from CUT. These method calls in the Set-up and/or Teardown are the helper methods that should not be considered a sign of eagerness of the test case. The following test case can demonstrate our point:

```
Example Test Case #3
@Test
public void testGetConnectionUserPassSetters()
     throws Exception {
    String username = "user";
    String password = "_secret";
    String url = "jdbc:h2:mem:ormlite-up;USER=" +
        username + ";PASSWORD=" + password;
    JdbcConnectionSource sds = new
        JdbcConnectionSource(url);
    sds.setUsername(username);
    sds.setPassword(password);
    assertNotNull(sds.getReadOnlyConnection(null));
    sds.close();
}
```

The second scenario is when a test case contains multiple **get** methods (defined and discussed in Section 5.2.1). These **get** methods are from CUT but invoked to verify the outcome(s) of a single method call from CUT. Clearly, such invocations of **get** methods, which are very common in practice, do not make the test case an eager test. The test case below falls into this scenario:

```
Example Test Case #4
@Test
public void testAccessors() throws Exception {
    final String uri = "http://localhost:80";
    final String proxyHost = "foo";
    final int proxyPort = 1030;
    final String proxyUser = "bar";
    final String proxyPassword = "fly";
    final SSLProperties ssl = new SSLProperties();

    HTTPRequestInfo info = populate(uri, proxyHost,
        proxyPort, proxyUser,
                                  proxyPassword, ssl);

    assertEquals(uri, info.getURI().toString());
    assertEquals(proxyHost, info.getProxyHost());
    assertEquals(proxyPort, info.getProxyPort());
    assertEquals(proxyUser, info.getProxyUser());
    assertEquals(proxyPassword,
        info.getProxyPassword());
    assertEquals(ssl, info.getSSLProperties());
}
```

The detection rules DR2.1 and DR2.2 are not different from DR1 except for the threshold. While DR2.1 states that a test case must have at least three method calls from CUT to be classified as an eager test, DR2.2 increases the threshold to five method calls from CUT. Since the thresholds of DR2.1 and DR2.2 are much higher than DR1, the number of test cases classified as eager tests by DR2.1 and DR2.2 is much lower than by DR1 (as shown in Fig. 2). In contrast, DR2.1 and DR2.2 missed more eager tests than DR1 due to the same reason. For example, the following test case is classified as an eager test by DR1 (as well as by our heuristic) but not by DR2.1 and DR2.2 due to the differences in threshold.

```
Example Test Case #5
@Test
public final void testEntities() {
    String s = "&#039;";
    String t = "’";
    assertEquals(t, MediaWiki.decode(s));
    s = "&quot;";
    t = "\"";
    assertEquals(t, MediaWiki.decode(s));
}
```

In the same group of detection rules, DR2.3 is less restrictive than DR2.1 and DR2.2, as DR2.3 considers any method call from the production code while having a threshold of 5. However, our manual assessment showed that DR2.3 still misclassified many test cases. As the concept behind DR2.3 is actually the same as DR1, DR2.1, and DR2.2, the misclassified test cases by DR2.3 also fall into the two scenarios that we explained earlier with DR1.

The detection rule DR3 classifies eager tests from a different angle. According to this rule, an eager test must have at least two cycles of non-verification instructions followed by verification instructions. The first problem of this detection rule is that even if a test case has only one cycle of non-verification instructions followed by verification instructions, it can still be an eager test when containing multiple assert statements verifying the outcomes of multiple method calls. For example, the test case below has only one cycle of non-verification instructions followed by verification instructions. However, it verifies multiple method calls; hence, it should not be discarded.

```
Example Test Case #6
@Test
public void testEquals() throws Exception {
    Category a = new Category();
    a.setName("giraffes");

    Category b = new Category();
    b.setName("camels");

    Category c = new Category("giraffes");

    assertEquals(a, c);
    assertEquals(c, a);
    assertTrue(!a.equals(b));
    assertTrue(!b.equals(c));
    assertTrue(!b.equals(null));
    assertTrue(!b.equals(new Object()));
}
```

The second problem of this detection rule is that a test case with at least two required cycles might not be an eager test. For example, a test case can contain an assert statement verifying the outcome(s) of one method call from either CUT or production code in the first cycle. In the second cycle, one or more **get** methods are called to acquire extra information regarding the outcome(s) of the method call in the first cycle. This test case is not an eager test as all the assert statements verify the outcome(s) of one single method call. For example, one test case falls into this scenario is shown below.

```
Example Test Case #7
@Test
public void testBasic() throws Exception {
    BeanBinDAO dao = getDAO();

    TestEntity entity = new TestEntity();
    entity.setAnInt(5);
    entity.setString("a sample string");

    Transaction tx = new Transaction(dao,
        TestEntity.class);
    tx.addCommand(new AddEntity(entity));
    tx.commit();

    Query q = new Query(new Criteria("string", "a
        sample string", SearchType.EQUALS));
    List<Object> results =
        dao.search(TestEntity.class, q);

    assertEquals(1, results.size());
    TestEntity saved = (TestEntity) results.get(0);
    assertEquals("a sample string",
        saved.getString());
}
```

The last detection rule in our manual assessment was DR4, which emphasizes that an eager test needs to have at least two assert statements, and at least one of the assert statements is not based on a **get** method's result. The first thing to note is that this detection rule does not come with a clear description of **get** methods. A get method can be a simple getter such as *getName()*, or it can be something less traditional such as *isEmpty()*. In our assessment, we used our definition of **get** methods for this detection rule, i.e., a get method returns an object's attribute without modifying the attribute. Like the other detection rules discussed above, DR4 can also miss detecting eager tests. It occurs when a test case contains multiple assert statements, and each of the assert statements invokes a get method to verify the outcome(s) of a different method like a mutator or a constructor, etc. The test case clearly satisfies the original definitions of the Eager Test smell while being missed by DR4. One example is as follows:

```
Example Test Case #8
@Test
public void testGetSetter() throws Exception {
    Method getter =
        EntityUtils.getMethod(IndexedEntity.class,
        "keywords");
    Method setter = EntityUtils.getSetter(getter);
    assertEquals("setKeywords", setter.getName());

    getter =
        EntityUtils.getMethod(IndexedEntity.class,
        "generatedKeywords");
    setter = EntityUtils.getSetter(getter);
    assertEquals(null, setter);
    }
```

This detection rule DR4 can also falsely flag a non-eager test. The common scenario we learned from our manual assessment is that a test case has at least two assert statements, and at least one invokes an *external producer* instead of a **get** method. Such external producers can be *toString()* or *contains()*. These external producers act like helper methods, i.e., to extract information regarding the outcome(s) of a single method call. Hence, in this scenario, the test case is actually not an eager test, as shown in the example below.

```
Example Test Case #9
@Test
public void testFindImplementations() throws
    Exception {
  Resolver resolve = new Resolver();
  List<Class> list =
      resolve.findImplementations(Base.class);
  assertTrue(list.contains(ImplOne.class));
  assertTrue(list.contains(ImplTwo.class));
  assertEquals(2, list.size());
  }
```

7.1. Implications

Through our manual analysis, we have identified multiple patterns of test cases that often appear in practice but are misclassified by the detection rules. Basically, there are three common patterns where non-eager test cases are misclassified as eager tests by the widely-used detection rules. These include test cases that (1) require multiple method calls for setup but verify only one method, (2) use multiple **get** methods to validate the outcome(s) of a single method call, or (3) invoke helper methods that are not part of the production code to acquire information regarding the outcome(s) of a single method call such as *size()*, *contains()*. Additionally, our analysis highlights two patterns of eager tests that are missed by these rules but detected by the proposed heuristic: (1) tests verifying multiple constructors and (2) tests with multiple get methods validating the outcomes of different methods.

Given the high occurrence of these patterns in practice, their misclassification significantly affects detection results. This raises concerns about the reliability of current tools, potentially undermining both practitioner trust and academic claims regarding the prevalence and impact of eager tests.

Ultimately, the main reason for the misclassification of these common patterns of test cases is that the existing rules do not take into account what a method actually does, which is the determining factor for eager test recognition. The question here is whether the authors of these rules and tools took this limitation into account while proposing the rules and implementing the detection tools. In fact, we have not seen any studies where any limitations of these rules and tools were well documented. Only Bavota et al. (2015), whose proposed detection rule is "JUnit classes having at least one method that uses more than one method of the tested class", explained briefly that their rule is "very simple that overestimates the presence of test smells in the code". Their reason was to not miss any test smell instances. Our heuristic, in contrast, analyzes the purpose of each method call by systematically studying the production code and the test code based on the method type before drawing conclusions. Therefore, even though it might require more effort to implement our heuristic due to the very step of analyzing the code, we believe that the trade-off is worth it as it is important to reduce such a high misclassification rate in terms of eager tests.

8. Conclusion and future work

Eager Test has been one of the most common test smells discussed by many researchers. There are over ten test smell detection tools, including the Eager Test smell in their test smell lists. Nevertheless, the existing detection rules and approaches used by those detection tools are all based on inadequate interpretations of the Eager Test smell. Due to this reason, conclusions on the prevalence and the negative impacts of the Eager Test smell in practice might become invalid. To address this problem, we conducted a literature review to obtain a better understanding of the definitions of the Eager Test smell as well as the existing detection rules and approaches for this test smell. Based on the information obtained from the literature review, we proposed a novel, unambiguous definition of the Eager Test smell and, accordingly, a heuristic (with pseudocode) to detect eager tests in unit test cases written in Java. We also performed a manual assessment to analyze how the heuristic could overcome the shortcomings of the existing detection rules.

Overall, the contribution of this study is two-fold: (1) the original definitions of the Eager Test smell together with the current detection rules found in the literature, and (2) the novel, unambiguous definition of the Eager Test smell and, accordingly, a heuristic (with pseudocode) to detect eager tests. Our findings provide researchers with an overview of how the Eager Test smell has been defined and detected in the literature. On top of that, the manual assessment of the proposed heuristic highlights the importance of having a more precise approach to detect eager tests in the context of unit testing and the concrete shortcomings of the existing detection rules. In our recent work, we are also developing a tool, namely EagerID,[1] that operationalizes the proposed heuristic. For future work, we plan to use our tool to conduct automatic assessment of eager tests on a larger scale, including test cases from industrial projects, to further validate our heuristic in real-world scenarios. Additionally, we aim to compare our heuristic with other prominent test smell detection tools for Java. We also plan to explore defining the Eager Test smell for non-object-oriented programming languages, such as C, to broaden the applicability of our approach across diverse programming paradigms.

CRediT authorship contribution statement

Huynh Khanh Vi Tran: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Nauman bin Ali:** Writing – review & editing, Validation, Supervision, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Michael Unterkalmsteiner:** Writing – review & editing, Validation, Supervision, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Jürgen Börstler:** Writing – review & editing, Validation, Supervision, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been supported by ELLIIT, the Strategic Research Area within IT and Mobile Communications, funded by the Swedish Government. The work has also been supported by research grants for GIST (reference number 20220235) and SERT projects, funded by the Knowledge Foundation in Sweden.

[1] https://github.com/vi-tran1987/ET_heuristic

Appendix. Comparison of the outcomes for Eager test classifications of our heuristic (EagerID) and existing detection rules

Data set (a): 100 manually written test cases from Panichella et al.'s study

		DR1	
		YES	NO
EagerID	YES	51	6
	NO	23	20

		DR2.1	
		YES	NO
EagerID	YES	36	21
	NO	11	32

		DR2.2	
		YES	NO
EagerID	YES	17	40
	NO	6	37

		DR2.3	
		YES	NO
EagerID	YES	36	21
	NO	20	23

		DR3	
		YES	NO
EagerID	YES	49	8
	NO	8	35

		DR4	
		YES	NO
EagerID	YES	25	32
	NO	15	28

		DR2.1	
		YES	NO
DR1	YES	47	27
	NO	0	26

		DR2.2	
		YES	NO
DR1	YES	23	51
	NO	0	26

		DR2.3	
		YES	NO
DR1	YES	47	27
	NO	9	17

		DR3	
		YES	NO
DR1	YES	50	24
	NO	7	19

		DR4	
		YES	NO
DR1	YES	25	49
	NO	15	11

		DR2.2	
		YES	NO
DR2.1	YES	23	24
	NO	0	53

		DR2.3	
		YES	NO
DR2.1	YES	36	11
	NO	20	33

		DR3	
		YES	NO
DR2.1	YES	35	12
	NO	22	31

		DR4	
		YES	NO
DR2.1	YES	15	32
	NO	25	28

		DR2.3	
		YES	NO
DR2.2	YES	23	0
	NO	33	44

		DR3	
		YES	NO
DR2.2	YES	17	6
	NO	40	37

		DR4	
		YES	NO
DR2.2	YES	8	15
	NO	32	45

		DR3	
		YES	NO
DR2.3	YES	39	17
	NO	18	26

		DR4	
		YES	NO
DR2.3	YES	15	41
	NO	25	19

		DR4	
		YES	NO
DR3	YES	21	36
	NO	19	24

Data set (b): 100 manually written test cases from Sharma et al.'s study

		DR1	
		YES	NO
EagerID	YES	49	7
	NO	29	15

		DR2.1	
		YES	NO
EagerID	YES	36	20
	NO	8	36

		DR2.2	
		YES	NO
EagerID	YES	25	31
	NO	7	37

		DR2.3	
		YES	NO
EagerID	YES	33	23
	NO	12	32

		DR3	
		YES	NO
EagerID	YES	42	14
	NO	0	44

		DR4	
		YES	NO
EagerID	YES	49	7
	NO	10	34

		DR2.1	
		YES	NO
DR1	YES	43	35
	NO	1	21

		DR2.2	
		YES	NO
DR1	YES	32	46
	NO	0	22

		DR2.3	
		YES	NO
DR1	YES	44	34
	NO	1	21

		DR3	
		YES	NO
DR1	YES	36	42
	NO	6	16

		DR4	
		YES	NO
DR1	YES	46	32
	NO	13	9

		DR2.2	
		YES	NO
DR2.1	YES	27	17
	NO	5	51

		DR2.3	
		YES	NO
DR2.1	YES	33	11
	NO	12	44

		DR3	
		YES	NO
DR2.1	YES	26	18
	NO	16	40

		DR4	
		YES	NO
DR2.1	YES	30	14
	NO	29	27

		DR2.3	
		YES	NO
DR2.2	YES	32	0
	NO	13	55

		DR3	
		YES	NO
DR2.2	YES	20	12
	NO	22	46

		DR4	
		YES	NO
DR2.2	YES	21	11
	NO	38	30

		DR3	
		YES	NO
DR2.3	YES	26	19
	NO	16	39

		DR4	
		YES	NO
DR2.3	YES	30	15
	NO	29	26

		DR4	
		YES	NO
DR3	YES	40	2
	NO	19	39

Data set (c): 100 automatically generated test cases from Panichella et al.'s study

EagerID \ DR1	YES	NO
YES	44	19
NO	24	13

EagerID \ DR2.1	YES	NO
YES	26	37
NO	13	24

EagerID \ DR2.2	YES	NO
YES	2	61
NO	1	36

EagerID \ DR2.3	YES	NO
YES	13	50
NO	5	32

EagerID \ DR3	YES	NO
YES	29	34
NO	4	33

EagerID \ DR4	YES	NO
YES	53	10
NO	16	21

DR1 \ DR2.1	YES	NO
YES	39	29
NO	0	32

DR1 \ DR2.2	YES	NO
YES	3	65
NO	0	32

DR1 \ DR2.3	YES	NO
YES	18	50
NO	0	32

DR1 \ DR3	YES	NO
YES	26	42
NO	7	25

DR1 \ DR4	YES	NO
YES	43	25
NO	26	6

DR2.1 \ DR2.2	YES	NO
YES	3	36
NO	0	61

DR2.1 \ DR2.3	YES	NO
YES	16	23
NO	2	59

DR2.1 \ DR3	YES	NO
YES	16	23
NO	17	44

DR2.1 \ DR4	YES	NO
YES	22	17
NO	47	14

DR2.2 \ DR2.3	YES	NO
YES	3	0
NO	15	82

DR2.2 \ DR3	YES	NO
YES	1	2
NO	32	65

DR2.2 \ DR4	YES	NO
YES	1	2
NO	68	29

DR2.3 \ DR3	YES	NO
YES	13	5
NO	20	62

DR2.3 \ DR4	YES	NO
YES	12	6
NO	57	25

DR3 \ DR4	YES	NO
YES	29	4
NO	40	27

Data availability

I have shared link under Information about Available Data/Artifacts.

References

Aljedaani, W., Peruma, A., Aljohani, A., Alotaibi, M., Mkaouer, M.W., Ouni, A., Newman, C.D., Ghallab, A., Ludi, S., 2021. Test smell detection tools: A systematic mapping study. In: ACM International Conference Proceeding Series. pp. 170–180.

Athanasiou, D., Nugroho, A., Visser, J., Zaidman, A., 2014. Test code quality and its relation to issue handling performance. IEEE Trans. Softw. Eng. 40 (11), 1100–1125.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., Binkley, D., 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: IEEE International Conference on Software Maintenance. ICSM, pp. 56–65.

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

Beck, K., 2002. Test-driven Development: by Example. Addison-Wesley, Boston, MA.

Binder, R., 2000. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional.

Bowes, D., Hall, T., Petrić, J., Shippey, T., Turhan, B., 2017. How good are my tests? In: Proceedings of the International Workshop on Emerging Trends in Software Metrics. WETSoM, pp. 9–14.

Breugelmans, M., Van Rompaey, B., 2008. Testq: Exploring structural and maintenance characteristics of unit test suites. In: WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques. Citeseer, p. 11.

Camara, B., Silva, M., Endo, A., Vergilio, S., 2021. On the use of test smells for prediction of flaky tests. In: ACM International Conference Proceeding Series. pp. 46–54.

Cohen, J., 1960. A coefficient of agreement for nominal scales. Educ. Psychol. Meas. 20 (1), 37–46.

Damasceno, H., Bezerra, C., Coutinho, E., Machado, I., 2023. Analyzing test smells refactoring from a developers perspective. In: Proceedings of the XXI Brazilian Symposium on Software Quality. SBQS '22, Association for Computing Machinery, New York, NY, USA.

De Bleser, J., Di Nucci, D., De Roover, C., 2019a. Assessing diffusion and perception of test smells in scala projects. In: IEEE International Working Conference on Mining Software Repositories, vol. 2019-May, pp. 457–467.

De Bleser, J., Di Nucci, D., De Roover, C., 2019b. SoCRATES,: Scala radar for test smells. In: Scala 2019 - Proceedings of the 10th ACM SIGPLAN International Symposium on Scala, Part of ECOOP 2019. pp. 22–26.

Dragan, N., Collard, M.L., Maletic, J.I., 2006. Reverse engineering method stereotypes. In: 2006 22nd IEEE International Conference on Software Maintenance. IEEE, pp. 24–34.

Dragan, N., Collard, M.L., Maletic, J.I., 2009. Using method stereotype distribution as a signature descriptor for software systems. In: 2009 IEEE International Conference on Software Maintenance. IEEE, pp. 567–570.

Fowler, M., 2018. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

Fraser, G., Arcuri, A., 2014. A large-scale evaluation of automated unit test generation using evosuite. ACM Trans. Softw. Eng. Methodol. (TOSEM) 24 (2), 1–42.

Freeman, S., Pryce, N., 2009. Growing Object-Oriented Software, Guided by Tests. Pearson Education.

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

Goodenough, J.B., Gerhart, S.L., 1975. Toward a theory of test data selection. IEEE Trans. Softw. Eng. SE-1 (2), 156–173.

Grano, G., De Iaco, C., Palomba, F., Gall, H.C., 2020. Pizza versus pinsa: On the perception and measurability of unit test code quality. In: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 336–347.

Hamill, P., 2004. Unit Test Frameworks: Tools for High-Quality Software Development. O'Reilly Media, Inc.

Jabbari, R., Ali, N.B., Petersen, K., Tanveer, B., 2018. Towards a benefits dependency network for DevOps based on a systematic literature review. J. Softw. Evol. Process. 30 (11).

Junior, N.S., Rocha, L., Martins, L.A., Machado, I., 2020. A survey on test practitioners' awareness of test smells. arXiv preprint arXiv:2003.05613.

Kapfhammer, G.M., Soffa, M.L., 2003. A family of test adequacy criteria for database-driven applications. ACM SIGSOFT Softw. Eng. Notes 28 (5), 98–107.

Khorikov, V., 2020. Unit Testing Principles, Practices, and Patterns. Simon and Schuster.

Kochhar, P.S., Xia, X., Lo, D., 2019. Practitioners' views on good software testing practices. In: Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice. ICSE-SEIP 2019, pp. 61–70.

Lambiase, S., Cupito, A., Pecorelli, F., De Lucia, A., Palomba, F., 2020. Just-in-time test smell detection and refactoring: The DARTS project. In: IEEE International Conference on Program Comprehension. pp. 441–445.

Landis, J.R., Koch, G.G., 1977. The measurement of observer agreement for categorical data. Biometrics 33 (1), 159–174.

Lemos, O.A.L., Vincenzi, A.M.R., Maldonado, J.C., Masiero, P.C., 2007. Control and data flow structural testing criteria for aspect-oriented programs. J. Syst. Softw. 80 (6), 862–882.

Lilis, Y., Savidis, A., 2019. A survey of metaprogramming languages. ACM Comput. Surv. 52 (6), 1–39.

Martin, F., 2014. Unit test. URL: https://martinfowler.com/bliki/UnitTest.html.

Martins, L., Costa, H., Machado, I., 2023. On the diffusion of test smells and their relationship with test code quality of Java projects. J. Softw.: Evol. Process.

Meszaros, G., 2007. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley Professional.

Moreno, L., Marcus, A., 2012. Jstereocode: automatically identifying method and class stereotypes in java code. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 358–361.

Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M., 2017. Curating github for engineered software projects. Empir. Softw. Eng. 22, 3219–3253.

Neukirchen, H., Zeiss, B., Grabowski, J., 2008. An approach to quality engineering of TTCN-3 test specifications. Int. J. Softw. Tools Technol. Transf. 10 (4), 309.

Palomba, F., Zaidman, A., De Lucia, A., 2018. Automatic test smell detection using information retrieval techniques. In: Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution. ICSME 2018, pp. 311–322.

Panichella, A., Panichella, S., Fraser, G., Sawant, A.A., Hellendoorn, V.J., 2020. Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities. In: 2020 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 523–533.

Panichella, A., Panichella, S., Fraser, G., Sawant, A.A., Hellendoorn, V.J., 2022. Test smells 20 years later: detectability, validity, and reliability. Empir. Softw. Eng. 27 (7).

Pei, K., Cao, Y., Yang, J., Jana, S., 2017. DeepXplore: Automated whitebox testing of deep learning systems. In: Proceedings of the 26th ACM Symposium on Operating Systems Principles - SOSP. pp. 1–18.

Peruma, A., Almalki, K., Newman, C.D., Mkaouer, M.W., Ouni, A., Palomba, F., 2020. 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. pp. 1650–1654.

Pizzini, A., Reinehr, S., Malucelli, A., 2023. Automatic refactoring method to remove eager test smell. In: Proceedings of the XXI Brazilian Symposium on Software Quality. SBQS '22, Association for Computing Machinery, New York, NY, USA.

Rwemalika, R., Habchi, S., Papadakis, M., Le Traon, Y., Brasseur, M., 2023. Smells in system user interactive tests. Empir. Softw. Eng. 28 (1).

Santana, R., Martins, L., Virgínio, T., Soares, L., Costa, H., Machado, I., 2022. Refactoring assertion roulette and duplicate assert test smells: a controlled experiment. In: CIbSE 2022 - XXV Ibero-American Conference on Software Engineering.

Schvarcbacher, M., Spadini, D., Bruntink, M., Oprescu, A., 2019. Investigating developer perception on test smells using better code hub - Work in progress. In: CEUR Workshop Proceedings, vol. 2510.

Sharma, T., Georgiou, S., Kechagia, M., Ghaleb, T.A., Sarro, F., 2023. Investigating developers' perception on software testability and its effects. Empir. Softw. Eng.: Int. J. 28 (5), 120.

Spadini, D., Schvarcbacher, M., Oprescu, A., Bruntink, M., Bacchelli, A., 2020. Investigating severity thresholds for test smells. In: Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories. MSR 2020, pp. 311–321.

Spadini2018On, D., Palomba, F., Zaidman, A., Bruntink, M., Bacchelli, A., 2018. On the relation of test smells to software code quality. In: Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution. ICSME 2018, pp. 1–12.

Tahir, A., Counsell, S., MacDonell, S.G., 2016. An empirical study into the relationship between class features and test smells. In: Proceedings - Asia-Pacific Software Engineering Conference. APSEC, pp. 137–144.

Tran, H.K.V., Ali, N.B., Börstler, J., Unterkalmsteiner, M., 2019. Test-case quality – understanding practitioners' perspectives. In: Proceedings of the 20th International Conference on Product-Focused Software Process Improvement. PROFES, Springer, pp. 37–52.

Tran, H.K.V., Unterkalmsteiner, M., Börstler, J., Ali, N.B., 2021. Assessing test artifact quality—A tertiary study. Inf. Softw. Technol. 139, 106620.

Tudose, C., 2021. JUnit in Action, third ed. Manning Publications.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2016a. An empirical investigation into the nature of test smells. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ASE 2016, pp. 4–15.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2016b. An empirical investigation into the nature of test smells. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ASE 2016, pp. 4–15.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D., 2015. When and why your code starts to smell bad. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, IEEE, pp. 403–414.

Van Deursen, A., Moonen, L., Van Den Bergh, A., Kok, G., 2001. Refactoring test code. In: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering. XP2001, Citeseer, pp. 92–95.

Van Rompaey, B., Du Bois, B., Demeyer, S., 2006. Characterizing the relative significance of a test smell. In: IEEE International Conference on Software Maintenance. ICSM, pp. 391–400.

Van 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 Trans. Softw. Eng. 33 (12), 800–816.

Virgínio, T., Martins, L., Rocha, L., Santana, R., Cruz, A., Costa, H., Machado, I., 2020. Jnose: Java test smell detector. In: Proceedings of the XXXIV Brazilian Symposium on Software Engineering. pp. 564–569.

Virgínio, T., Soares, L.R., Santana, R., Costa, H., Martins, L.A., Machado, I., 2019. On the influence of test smells on test coverage. In: ACM International Conference Proceeding Series. pp. 467–471.

White, R., Krinke, J., 2022. TCTracer: Establishing test-to-code traceability links using dynamic and static techniques. Empir. Softw. Eng. 27 (3).

Zhu, H., Hall, P.A., May, J.H., 1997. Software unit test coverage and adequacy. ACM Comput. Surv. 29 (4), 366–427.