

How Do Developers Structure Unit Test Cases? An Empirical Analysis of the AAA Pattern in Open Source Projects

Chenhao Wei , Member, IEEE, Lu Xiao , Member, IEEE, Tingting Yu , Member, IEEE,
Sunny Wong , Member, IEEE, and Abigail Clune 

Abstract—The AAA (Arrange, Act, Assert) pattern provides a unified structure for unit test cases, potentially benefiting comprehension and maintenance. However, its adoption and implementation in practice remain insufficiently understood. This study investigates the prevalence of AAA pattern usage, identifies recurring deviations and design issues within AAA structures, and assesses developers’ receptiveness to AAA-based improvements. We conducted an empirical study on 735 real-life unit test cases randomly selected from seven open-source projects. We manually analyzed these test cases, identified AAA-related issues, and proposed fixes to developers. Our analysis found that 77% of test cases follow the AAA structure. We identified three recurring patterns deviating from AAA and four design issues within A blocks. Comparison with classic test smells revealed unique insights provided by AAA analysis. Of 27 improvement proposals sent to developers, 78% received positive feedback. These findings show that the AAA pattern is widely adopted in practice, but deviations from and design issues within AAA patterns are common. Our analysis provides a novel perspective on test case quality, complementing traditional test smell analysis. The high acceptance rate of our improvement proposals suggests that developers value AAA-based enhancements. These findings can guide the development of tools for improving AAA practice in unit tests.

Index Terms—Software testing, unit testing, design quality, refactoring, test smell, AAA pattern.

I. INTRODUCTION

THE AAA pattern refers to the three-section layout of writing a unit test case: *arrange*(AR), *act*(AC), and *assert*(AS) [1], offering a natural and intuitive flow for creating a unit

Received 17 July 2024; revised 13 December 2024; accepted 22 January 2025. Date of publication 31 January 2025; date of current version 18 April 2025. This work was supported in part by U.S. National Science Foundation (NSF) under Grant CCF-1909085 and Grant CCF-1909763. Recommended for acceptance by Y. Wang. (Corresponding author: Lu Xiao.)

Chenhao Wei and Lu Xiao are with the Department of Systems and Enterprises, Stevens Institute of Technology, Hoboken, NJ 07030 USA (e-mail: cwei7@stevens.edu; lxiao6@stevens.edu).

Tingting Yu is with the Department of Computer Science and Engineering, University of Cincinnati, Cincinnati, OH 45221 USA (e-mail: tingting.yu@uconn.edu).

Sunny Wong is with Envestnet, Inc., Berwyn, PA 19312 USA (e-mail: sunny@computer.org).

Abigail Clune is with AGI, Ansys Company, Exton, PA 19341 USA (e-mail: abigail.clune@ansys.com).

Digital Object Identifier 10.1109/TSE.2025.3537337

```
1 public void testGetByPrefix() {
2     Config con = new Config(); //AR
3     tc.set(PROP_PREFIX); //AR
4     var p = tc.getProperties(); //AC
5     assertEquals("prop", p); //AS
6 }
```

(a) AAA Example

```
7 public void testGetByPrefix() {
8     Config con = new Config(); //AR
9     tc.set(PROP_PREFIX); //AR
10    var p = tc.getProperties(); //AC
11    assertEquals("prop", p); //AS
12
13    tc.set(SCANT_PREFIX); //AR
14    p = tc.getProperties(); //AC
15    assertEquals("scan", p); //AS
16 }
```

(b) AAA Violation Example

Fig. 1. Examples illustrating correct and incorrect applications of the AAA (Arrange-Act-Assert) pattern in unit tests.

test case. In *arrange*, the required environment, such as object creation and mock setup, is prepared. In *act*, the target function being tested is executed. In *assert*, the actual output from the *act* is checked against expectation. Fig. 1(a) illustrates such a test case.

Although scientific evidence of its benefits is absent, the AAA pattern benefits the comprehension and maintenance of test cases, as is well advocated in technical blogs and tutorials. For example, “Following this pattern does make the code quite well structured and easy to understand” [2]; “The AAA pattern is simple and provides a uniform structure for all tests in the suite. This uniform structure is one of its biggest advantages: once you get used to this pattern, you can read and understand the tests more easily. That, in turn, reduces the maintenance cost for your entire test suite”; [3] and “It is a structure or a way of thinking about and arranging your tests so that they can be clearly understood” [4].

Despite the benefits, AAA is only a structure, a way of thinking, or a guideline for writing test cases. There is no tool

to enforce the AAA pattern. Instead, the practice of the AAA pattern defers to the developers who actually write the test cases. In the scenario of test-driven development [5], the test cases are written before the production functions are completed. The developer may start a test case with *assert*, which describes the expected behavior. The test case is intended to fail with only the *assert* when not enough understanding of the actual function is available. Developers may also imprudently violate the AAA pattern due to insufficient design. For example, a test case may contain multiple AAA sequences combining different test scenarios in one test case. Fig. 1(b) is an example based on a real-life test case we observed. It combines different testing scenarios of *getByPrefix* in one test case. This case is a blunt violation of the “single responsibility” principle [6], [7]. The test case could fail due to either of the two scenarios, adding extra difficulty to comprehension, maintenance, and debugging. This test case should be broken down into separate test cases with each focusing on one test scenario.

There is little understanding of whether and to what extent real-life test cases actually follow the AAA pattern in the current literature. Of a particular note, Ma’ayan’s study [8] also provided insights into the usage of the AAA pattern in real-world JUnit tests. However, their study mainly focused on the linear structure of Fixture and Assert in test cases. This means they 1) exclude test cases with conditional logic from their scope—thus linear and 2) treat arrangement and action together as “Fixture” without differentiating them, which is essential for understanding the core intention of test cases and conformance check of their AAA structure. Their insights also mainly focus on the best practices of assertions, instead of the full AAA pattern. In comparison, our study provided an in-depth and full analysis of the AAA pattern, by carefully recognizing each of the A’s (arrangement, action, or assertion). Also, we consider the usage of conditional logic an orthogonal aspect of the AAA pattern—meaning the AAA pattern and logical conditions are not mutually exclusive in test cases. As such, we were able to contribute insights on the various AAA problems, whose analysis reply on the separation of the A’s and the inspection of the conditional logic.

While our recent work developed a machine learning approach to automatically tag AAA statements in test cases [9], there remains little understanding of how developers actually structure their test cases in practice. This leaves open questions about the broader adoption of the AAA pattern across different testing frameworks and projects. Furthermore, in test cases that do not follow the AAA pattern, are there recurring deviation patterns that merit restructuring? And, if test cases follow the AAA pattern, could they still contain design issues inside the A blocks? How can we improve the test cases based on the AAA context to enhance its behavior and design quality? And, will real-world developers receive such improvements? If not, what are their considerations? These are the questions that we are interested in addressing in this study to fill the current gap of understanding in the practice of the AAA pattern.

This study investigates the adoption and implementation of the AAA pattern in unit test cases through a mixed-methods

approach combining empirical analysis and developer feedback. It analyzes 735 real-life unit test cases randomly sampled from seven open-source projects across community-driven and corporate-driven contexts. The study involves manually tagging test cases to identify adherence to the AAA structure and deviations, categorizing recurring design issues within the AAA context, and comparing these issues to traditional test smells. Additionally, 27 representative test cases with identified AAA problems are restructured, and improvement proposals are submitted to developers for validation. By integrating qualitative analysis, quantitative evaluation, and practical validation, the study provides insights into the prevalence, deviations, and implications of AAA-based testing practices, paving the way for improved test case quality and maintenance.

This work makes the following contributions: 1) first of its kind empirical study to investigate whether and how often the AAA pattern is practiced; 2) novel analysis of AAA problems in test cases, which shed light on root causes to surface problem indications, compared to test smells; and 3) real-life developers’ perspectives and considerations regarding whether they favor fixing the AAA problems. This study paves the way to related future research.

II. BACKGROUND

In this section, we introduce the basics of the AAA pattern in unit testing, as well as the open-source community culture related to unit testing and the AAA adoption.

A. Unit Testing

Unit testing is a foundational phase within software testing, concentrating on examining individual units or components of a software system [10]. A unit typically refers to the smallest testable part of an application, such as a function, method, or class. The principal aim of unit testing is to ensure that each unit functions as intended and aligns with its design specifications. This process holds significant importance in guaranteeing software quality and ease of maintenance. By testing units in isolation, developers can identify and fix defects early in the development process, reducing the cost and effort required for debugging and maintenance [11], [12]. Moreover, unit testing serves as a protective measure for code alterations, empowering developers to modify and refactor the codebase confidently.

B. AAA in Unit Testing

The Arrange-Act-Assert (AAA) pattern is first introduced by Bill Wake in 2018 [13]. It is a widely adopted and unified structure for organizing unit test cases [1]. While commonly associated with Java, AAA is also used in other languages such as Python [14] and JavaScript [15]. Moreover, the AAA concept extends beyond unit testing, appearing in acceptance testing as the “Given-When-Then” structure [16]. AAA pattern promotes a consistent and readable format that enhances the maintainability and comprehension of test code. As implied by its name, the AAA pattern comprises three distinct sections: Arrange,

which sets up the requisite preconditions and inputs for the unit under test—this may involve creating objects, initializing data, or configuring mocks and stubs; Act, encompassing the invocation of the unit(s) being examined—typically a function call that triggers the behavior under scrutiny; Assert, the concluding segment—which validates the anticipated outcomes of the unit under test, such as verifying return values, state alterations, or expected exceptions. Fig. 1(a) illustrates a test case that follows the AAA pattern, with clear separation of arrangement (object creation), action (method invocation), and assertion (output verification). In contrast, Fig. 1(b) demonstrates an anti-pattern where multiple AAA sequences are combined in a single test case, making it more complex to understand and debug.

By adhering to the AAA pattern, developers can create unit tests that are more comprehensible and maintainable. The explicit division of responsibilities among the arrangement, action, and assertion steps enhances the readability and maintainability of test code. This aspect holds particular significance in the context of test-driven development and shared code ownership, where multiple developers collaborate on the same codebase [17], [18]. The AAA structure enables developers to swiftly grasp the intent and progression of test cases, even when dealing with unfamiliar code segments [12]. Furthermore, the AAA pattern can aid in diagnosing test failures by furnishing a higher-level abstraction for understanding the failure's underlying cause. Developers can prioritize their investigation based on the “A” section where the failure occurred, potentially streamlining the debugging process. Moreover, adherence to the standardized AAA structure in test cases can aid in identifying shortcomings in test code design, such as missing assertions or overly complex test cases, which can then be addressed through refactoring.

C. Test Smells in Unit Testing

Test smells are design flaws in test code that indicate potential problems with test maintainability and effectiveness [19]. Similar to code smells in production code, test smells suggest violations of fundamental design principles but may not always indicate actual problems. Common test smells include Eager Test (a test method that checks several methods of the tested object), Assertion Roulette (a test method that contains several assertions with no explanation), and Mystery Guest (a test that relies on external resources that are not visible in the test) [20].

Several tools are available for detecting test smells [21], [22], [23], [24]. JNose [25] has emerged as the most comprehensive tool based on a recent systematic mapping study [26]. It detects 21 distinct types of test smells, making it the current state-of-the-art in test smell detection.

The relationship between test smells and test code quality has been established through empirical studies [27]. Research shows that test smells often correlate with increased change- and defect-proneness in both test code and the associated production code [28]. For example, tests with assertion roulette tend to be more difficult to maintain, as failures provide unclear indications of what went wrong. Similarly, eager tests that

verify multiple behaviors in a single test case often become more complex and harder to understand over time [29].

III. SURVEY ON UNIT TESTING CULTURE

Before we start our core investigation and empirical analysis of the AAA practice reflected in real-world test cases, we first set out to understand the testing culture across Open Source Communities (OSCs) through a survey. Note that the survey does not analyze specific test cases or identify AAA problems; instead, it aims to offer a broader perspective on the cultural and procedural factors that may shape testing practices in OSCs. The survey result data can be accessed here <https://github.com/Codegass/Unit-Test-Empirical-Study-from-AAA-Perspective>.

A. Survey Design

a) Survey Questions: The survey consisted of four parts that focus on 1) participants' general background, 2) their experience and expertise in unit testing, 3) testing practices employed in the projects they contribute to, and 4) their knowledge and experience with the AAA pattern. For participants' general background, we ask about the types of projects they primarily work on and the years of related experience. In part two, we ask about their level of engagement in unit testing and their familiarity with testing frameworks. Part three focused on the testing practices within the open-source software projects that the participants have been involved with. We inquired about the strictness of unit test case standards, the clarity of unit testing guidelines, and the feedback quality on pull requests containing unit tests. In the last part, we asked participants to rate on a scale of 0 to 10 regarding their expertise in the AAA pattern, as well as the frequency with which they adopt the AAA pattern in their test cases. We also asked for their open comments regarding the benefits and/or drawbacks of using the AAA pattern.

b) Community Selection: We focused on the Apache Software Foundation, Microsoft, Google, and Netflix due to their significant influence in the software industry and their different organizational models. The Apache Software Foundation represents one of the largest and most respected community-driven open-source ecosystems, with over 200 active Java projects and a well-established governance model for software development. In comparison, Microsoft, Google, and Netflix, on the other hand, represent corporate-driven open-source development, where the 223 active projects often reflect industry best practices and professional software engineering standards. These organizations contribute significantly to the open source landscape while maintaining high standards of code quality, making them ideal examples for studying software engineering practices, including the AAA pattern.

c) Participant Recruitment: We distributed a survey to a total of 381 developers actively contributing to Java projects within these OSCs. We use stars of more than 10 to filter out less impactful repositories. Participants were selected based on their ranking in terms of lines of code contributed to unit tests for the Java projects within each OSC. To account for the difference in community sizes, we targeted the top 50% of contributors from the larger Apache community and the top 95% of contributors

TABLE I
SURVEY DELIVERY AND RESPONSE RATES

Community	Selected	Delivered	#Participants
Community-driven	158 (Top 50%)	151	6
Corporate-driven	310 (Top 95%)	230	5

from the relatively smaller Corporate-driven OSCs. As shown in Table I, we received a total of 11 responses. The response rate was 4% in the Community-driven OSC and 2% in the Corporate-driven OSCs.

d) Statistical Analysis: One of our key hypotheses is that community-driven and corporate-driven projects may show different testing cultures. Given our limited sample size (11 responses total), we use the Mann-Whitney U test to examine the statistical significance between the survey results of the two types of projects, with a threshold of $p \leq 0.05$. The Mann-Whitney U test is appropriate here as it does not assume normal distribution and is suitable for small sample sizes. The null hypothesis is that there is no significant difference between the two groups being compared.

B. Survey Results

a) Participants Background and Expertise: Although we only received 11 responses, the participants are well qualified, work in diverse domains, and thus are representative of the broader software development community. Ten of the eleven survey participants have more than five years of experience as a software developer. The projects the participants work on focus on domains including development and testing (73%), web technology (9%), infrastructure, networking, IoT (9%), and others (9%). Hence, the survey results should provide general knowledge of the unit testing culture in open-source communities.

As for the participants' experience and expertise in unit testing, nine participants reported that unit testing is a fundamental part of their daily practice. The other two developers—both from community-driven projects—reported that they only practice unit testing when necessary. This likely indicates a stronger emphasis on unit testing in corporate-driven projects, if not a coincidence. Ten participants reported using JUnit as the testing framework, demonstrating familiarity with various JUnit features, such as the Assumptions API and annotations. Notably, there is a strong consensus among all 11 participants that unit testing has a strong impact on the maintainability of the production code.

b) Testing Practice: Nine participants indicated that their community provided guidelines for unit testing. The other two participants—indicating the lack of related guidelines—were both from community-driven projects. Additionally, we observed a statistically significant difference ($p\text{-value} = 0.024$) regarding the rating of the strictness of unit testing between the corporate-driven and community-driven projects. More specifically, all six participants from the corporate-driven projects rated the unit testing strictness 7 or above (on a scale of 0 to

10). In contrast, five participants from the community-driven projects rated the strictness between 4 and 6; and one rated the strictness level as 9. Despite the above differences, all participants agreed that their projects had high-quality unit tests, and they received relevant feedback for pull requests that involved changes to unit tests. These findings together underscore the importance of unit testing and related practices in open-source communities.

c) Knowledge of AAA: Regarding the familiarity and practice of the AAA pattern, all participants from corporate-driven projects rated themselves 7 or above (on a scale of 0-10). In contrast, the participants from the community-driven projects exhibited diverse ratings. Two participants rated 0—entirely unfamiliar with AAA; two rated 10—very familiar with AAA; the remaining two rated 5-7—moderate to good familiarity. This divergence between the two participant groups suggests potentially better adherence to AAA within corporate-driven projects. As for the benefits of AAA, participants highlighted improved test clarity, documentation, and readability. However, the drawbacks mentioned included increased verbosity in simple testing scenarios.

C. Overall Insights From Survey

In summary, the survey's findings first underscore the significance of unit testing in ensuring code maintainability, with all participants acknowledging its importance. This cultural emphasis on testing motivates the empirical investigation of the AAA practices reflected in test cases. Additionally, this cultural emphasis on testing also supports the conclusions (presented in our study results) regarding the prevalence of AAA practices and the potential benefits of addressing deviations and design issues within the AAA context.

Furthermore, the diversity of expertise and project domains among survey participants enriches the study's findings, suggesting that the observed patterns in AAA adoption and deviations may reflect broader industry practices. By connecting developer perspectives with empirical analysis, the survey serves to contextualize the study's results and reinforces the practical relevance of its recommendations for improving test case quality.

Most importantly, the survey also reveals that corporate-driven projects demonstrate a more rigorous testing culture compared to community-driven projects, characterized by stricter adherence to unit testing guidelines and higher familiarity with the AAA pattern. This motivates us to investigate how the AAA practices differ between these two types of projects. As we will discuss in the results section, these findings in the survey align with the study's observation that corporate-driven projects exhibit a greater percentage of test cases adhering to the AAA structure (85%) compared to community-driven projects (71%).

Overall, the survey provides contextual insights that help explain variations in AAA adoption observed in the empirical analysis and thus plays a supporting role, providing a cultural lens through which to interpret the study's results.

TABLE II
STUDY SUBJECTS

Project Name	Domain	Version	#Commits	#Contributors	#Files			#Test Cases	
					All	Test	Test (%)	All	Selected
Accumulo	Data Management	2.0.0	10,087	146	2,454	607	24.7%	2,333	77
Druid	Data Management	0.19.0	10,471	507	7,324	1,297	17.7%	6,532	239
Cloudstack	Networking & IoT	4.13.1.0	32,250	366	7,816	514	6.6%	3,002	96
Dubbo	Web Technologies	2.7.7	4,307	424	6,524	1,376	21.1%	2,765	88
Gson	Development & Infrastructure	2.10.1	1,772	147	227	122	53.7%	1,300	100
Mssql-jdbc	Data Management	12.6.1	2,645	93	374	118	31.6%	1,074	100
Archaius	Networking & IoT	2.7.5	702	29	171	39	22.8%	243	100
Total			62,234	1,712	24,890	4,073	16.4%	17,249	800

IV. RESEARCH QUESTIONS AND STUDY SUBJECTS

This study investigates four key research questions (RQs) concerning the AAA practice based on selected projects and test cases from diverse open-source communities.

A. Research Questions

RQ1: To what extent do real-world test cases adhere to the AAA pattern? How do the characteristics of AAA compliant test cases compare to those exhibiting AAA deviations? Our objective is to present the proportion of real-world test cases that conform to the AAA pattern. Furthermore, we seek to contrast test cases following the AAA paradigm with those deviating from it in terms of their complexity. The analysis will also explore whether project type—Community-driven versus Corporate-driven—affects adherence to the AAA pattern.

RQ2: What are the typical ways in which test cases diverge from the AAA pattern? Additionally, what underlying design issues might exist within the “A” blocks? We aim to uncover the various ways in which the AAA pattern is disregarded in practice. Through our observations, we aim to identify recurring deviations from the AAA structure. Our objective is to explore avenues for improving the behavior and quality of test cases within the framework of the AAA context. As with RQ1, we will also compare these issues between community-driven and corporate-driven projects.

RQ3: How do the AAA problems differ from the traditional test smells? Test smells have been extensively studied, focusing on the design quality of test cases. Our objective is to elucidate the distinctions between AAA analysis and test smells. We hypothesize that AAA analysis delves into the underlying root causes beneath the surface indicators of smells, potentially offering more essential, feasible, and appropriate resolutions to enhance the behavior and quality of test cases.

RQ4: Is there interest among real-world developers in addressing the AAA issues revealed in our study? We plan to apply refactoring to selected test cases exhibiting AAA issues and submit issue reports with our proposed solutions to assess the tangible impact of rectifying these issues. Non-responsiveness or rejection from developers would suggest limited practical significance.

B. Study Subjects

Answering the above research questions requires a thorough examination and comprehension of the structure of test cases

reflecting real-world practices. Aligning with the survey, we focus on the Apache Software Foundation, representing one of the largest community-driven open source ecosystems with well-established governance, and projects from Microsoft, Google, and Netflix, representing corporate-driven development with high industry standards. From their 435 Java projects (212 Apache and 223 corporate-driven), our strategy is to select projects of varied attributes and focus on randomly sampled test cases from these projects. All selected projects use Java as their primary programming language and *JUnit* as their testing framework, ensuring consistency in our analysis. While focusing on Java allows for a controlled study environment, we acknowledge this as a potential threat to validity and the value of future work for extending this analysis to other programming languages. Our project selection process takes into account several pivotal factors, including project domain, scale, activity level, and commitment to testing. This ensures that the chosen projects constitute a diverse and representative subset of the respective communities.

Table II shows the key information of the selected projects. The first column shows the names of selected projects. We adopted the categorization from a previous study [30], which classifies Apache projects into four broad categories: Networking & IoT, Web Technologies, Development & Infrastructure, and Data Management. As shown in column two, the selected projects cover all four categories. The projects are also in different scales, activity levels, and percentages of test files, forming a diverse dataset for our study. Druid represents the largest scale in our selection, with 7,324 total source files, including 1,297 test files and 6,532 test cases. In contrast, Archaius is the smallest, with only 171 source files and 39 test files, and 243 test cases. The other projects fall in between, illustrating a spectrum of scales. In terms of the activity level, CloudStack stands out as highly active, containing 32,250 commits from 366 contributors, demonstrating a vibrant and dynamic development environment. On the other end, Archaius is the least active project, with just 702 commits from 29 core contributors. Other projects lie in between, showing a blend of projects with different activity levels. Regarding the commitment to testing, we measure the percentage of test files relative to the total number of source files, Gson dedicating 53.7% of its files to testing. Conversely, CloudStack dedicates only 6.6% of its files to testing, highlighting potentially different development and testing strategies. Meanwhile, projects like

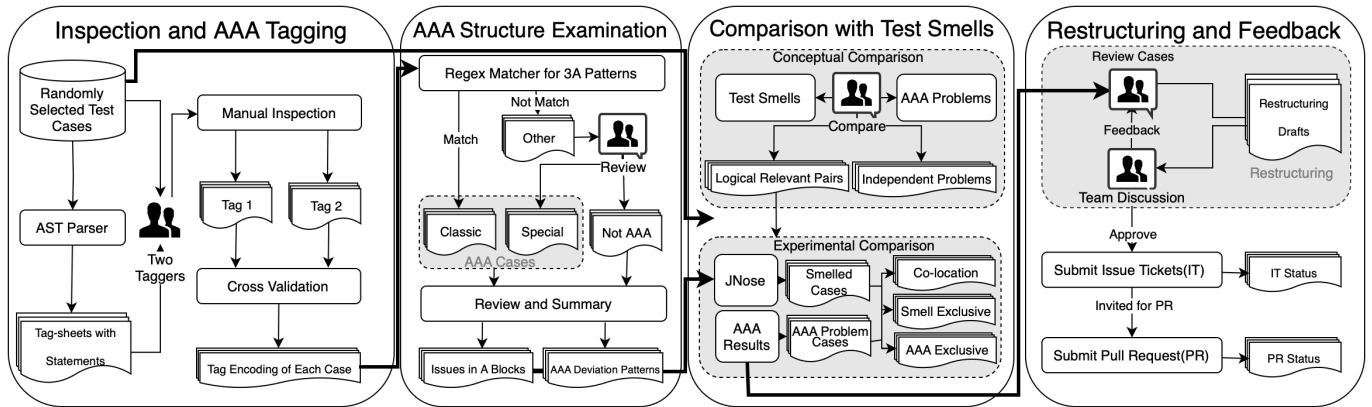


Fig. 2. Study overview.

Druid, Dubbo, Mssql-jdbc, and Accumulo fall in between, with the percentage of test files equal to 17.7%, 21.1%, 31.6%, and 24.7% respectively, reflecting the non-trivial commitment to testing in these projects.

While we employ careful selection of open-source projects for our study, the test cases in these projects are our actual primary units of analysis. We random sample 3% or at least 100 test cases from each project as a pragmatic balance between feasibility and representativeness. Given the large number of test cases in the projects studied, analyzing the entire set was infeasible within our research timeframe. Specifically, the test case selection was influenced by the total number of test cases in each project. For larger projects like Accumulo, Druid, Cloud-Stack, and Dubbo, we randomly selected approximately 3% of the total test cases to ensure a manageable yet representative sample. While, for smaller projects such as Gson, Mssql-jdbc, and Archaius, we set a minimum threshold of 100 test cases to maintain a sufficient sample size for meaningful analysis.

In total, 800 test cases were selected for our study, providing a comprehensive dataset for analyzing unit testing practices. According to Yamane’s formula for sample size determination with a 95% confidence level and a 5% margin of error [31], our sample of 800 test cases exceeds the minimum required sample size for a population of 17,249 test cases.

Random sampling was chosen to ensure an unbiased representation of the test case population without introducing systematic selection biases that could arise from manual or stratified sampling. Random sampling is a commonly used technique in software engineering research. For instance, Chen et al. used random sampling to select test cases for their study on test case prioritization [32].

The raw and immediate data for this study are hosted here. <https://github.com/Codegass/Unit-Test-Empricial-Study-from-AAA-Perspective>.

V. STUDY APPROACH

Fig. 2 describes the overview of our study steps, including 1) Inspecting and tagging the AAA structure of test cases; 2) Examining the AAA structure, its deviations, and design issues; 3) Comparing the AAA problems with *Test Smells*; and

4) Restructuring test cases with AAA problems and collecting developer feedback.

A. Step 1: Inspection and AAA Tagging

First of all, we manually tag the statements in each test case as *Arrange*, *Act*, or *Assert*, based on our understanding.

Taggers: To avoid personal bias, two taggers work on this task independently from each other. Both taggers are Ph.D. students focusing on Software Engineering, with one of them having approximately two years of prior industrial experience as a software engineer.

Tag-Sheets: To smooth the process, we create a Java parser based on the Eclipse JDT, version 4.17.¹, which parses unit test case to generate abstract syntax trees (ASTs), and then traverses these ASTs to output a full and expanded list of invocation statements. Each method invocation in the test case, including those in input parameters of compound statements, is expanded into its internal call trace until it cannot be further expanded. Note that the parser does not expand a production or external library method, since it should be tagged “atomically” as one of the “A”s—usually *arrange* or *act*. The taggers use the “tag-sheet”, and review the original code in a test case, to annotate the type of each statement in the sheet. Later, from each “tag-sheet” we can get an encoding of the test case layout as a sequence composed of “*arrange*”, “*act*”, and “*assert*” in any order. The output Tag-sheet is structured as a CSV file with four columns:

- testClassName: the name of the test class
- testMethodName: the name of the test method
- potentialTargetQualifiedname: the fully qualified name² of the invoked method
- AAA: the AAA tag type for taggers to tag (0 for Arrange, 1 for Action, 2 for Assert)

Fig. 3 demonstrates how our parser processes a test case into its tag-sheet representation. The upper part shows a test case with both compound statements (*assertEquals* containing *stmt.execute*) and expandable test utility methods

¹<https://archive.eclipse.org/eclipse/downloads/drops4/R-4.17-202009021800/>

²<https://docs.oracle.com/javase/specs/jls/se10/html/jls-6.html>

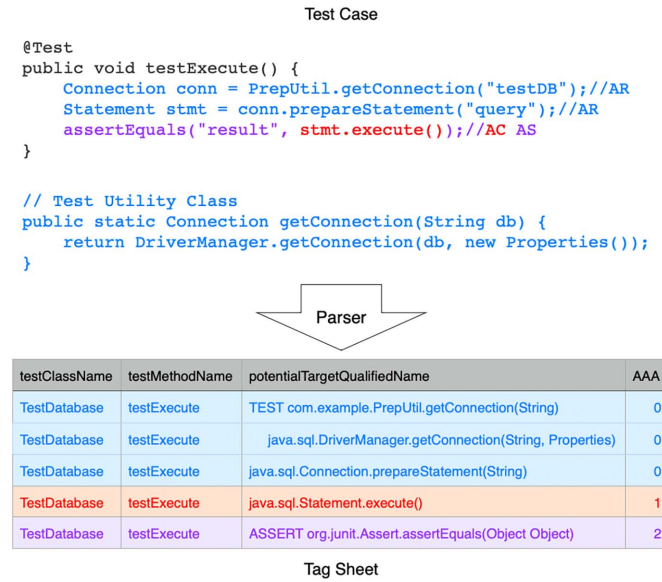


Fig. 3. Example of test case parsed to tag-sheet.

(*PrepUtil.getConnection*). The parser expands the compound statement by separating the assertion and the execution into distinct entries in the tag-sheet. Similarly, it expands the test utility method *getConnection* until reaching the production code *DriverManager.getConnection*. The lower part shows the resulting tag-sheet, where indentation indicates the expansion hierarchy. Note that while the figure shows AAA tags (0,1,2) for demonstration purposes, the actual tag-sheet leaves the AAA column blank for taggers to manually assess and tag each statement. For example, after manual review, *PrepUtil.getConnection* and its expansion to *DriverManager.getConnection* would be tagged as Arrange (0), while *stmt.execute* within the assertion would be tagged as Act (1). This expansion and tagging process ensures we capture the complete structure of test cases, even when they contain nested method calls or compound statements.

Manual Tagging: To ensure consistency and accuracy in the tagging process, we implemented a tagger training approach with three independent taggers. Initially, the first tagger reviewed all test cases and identified uncertain test statements, which were then discussed in group meetings with all authors but without the second and third taggers. These discussions led to the development of a tagging guide document, summarizing the experience and establishing clear tagging strategies. This approach was particularly valuable given the absence of a standardized AAA tagging framework. After the guidelines were finalized, the second and third taggers independently conducted their analyses, following the documented procedures.

The actual tagging process follows three main steps. First, taggers analyze the test case and class names, which often provide crucial context about test intentions. Good names help with a quick grasp of the test case intention. It is typical to see a test case name starting with the verb “test”, followed by the function and scenario under test; and the test class name is usually “AlphaTest”, where “Alpha” is a higher summary of the tested

functions. For example, test cases “*testInvoker_normal*” and “*testInvoker_fail*” are under test class “*ClusterInvokerTest*”. The test case name usually reveals the function under test—pointing to the *act*. For example, “*testInvoker_normal*” and “*testInvoker_fail*” should both *act* “*cluster.invoke()*”.

Second, the taggers dive deep into the internal logic of the test case. The taggers review the code within the test case, and also the expanded code from methods invoked by the test case. This is critical for gaining a flattened view of the full AAA structure of a test case. For example, test case *testReleaseDedicatedGuestVlanRange* [33] from CloudStack only contains three lines of code by itself. But it calls the method *runReleaseDedicatedGuestVlanRangePostiveTest*, which contains 9 expanded statements with all three “A”s. In addition, the taggers often also have to carefully review the production functions called in the test case, especially for test cases that do not follow good naming conventions. For example, we often see test cases named “testX”, where “X” is a number or an alphabet with no clue for the intention. For example, *test2* [34] is a real test case from Accumulo.

Finally, taggers apply special case rules established in our guide: (1) statements that assist in data retrieval for assertions are tagged as Assert, (2) expected exception fixtures like “@Test(expected = Exception.class)” count as Assert statements, and (3) test cases ending with “IT” in the name are excluded as integration tests.

Cross Validation: After working independently, all three taggers compare and cross-validate their results. The cases with disagreements are brought into group discussions with all authors—two of which are full-time developers. We use Fleiss’ kappa [35] to assess the agreement between the three taggers. This is assessed for the three “A”s separately—with 0.89 on *arrange*, 0.83 on *act*, and 0.89 on *assert*. The tagging of *assert* is most straightforward—mostly relying on the JUnit Assert APIs. We also recognize some common syntax features for *arrange*, such as invocations to *setter* and *constructor*. Tagging *act* is most critical, relying on the understanding of the test case intention. The most frequently recurring disagreement is the different *act* in test cases with vague names, such as “testX”. The other frequent disagreement is on *assert*, when it is used for checking pre-conditions, which will be discussed in RQ2.

B. Step 2: AAA Structure Examination

This step aims to examine whether test cases follow the AAA pattern and the related problems. Thus, we first differentiate AAA test cases from their deviations and then summarize the patterns of deviation and issues that reside inside of the A blocks.

Regex Matching and Manual Inspection: We take the encoding of each test case from the “tag-sheet” as input, and match it against a regular expression, namely “[arrange]+[act]+[assert]+”. This expression implies that the encoding of a test case **should** be composed of at least one *arrange*, followed by at least one *act*, and followed by at least one *assert*, assuming the adherence to the AAA structure. In other

words, if a test case matches the regular expression, it indicates that it follows the AAA pattern.

For cases that do not match the regular expression, we revisit the source code to understand whether the AAA pattern is truly violated. Here, we expand our inspection to the entire scope of the test class where the test case resides, as there could be special design considerations. For example, a test case may access global *arrange* or *assert* shared across different test cases, encapsulated in the *@Before* or *@After* methods. We consider such cases as *Special AAA* since the “violation” of AAA pattern is superficial. We summarize recurring patterns that lead to *Special AAA*. Thus, the AAA test cases contain both the classic and special AAA. Test cases that are confirmed to truly deviate from the AAA pattern due to improper design are referred to as *AAA-Deviation* cases.

Quantitative Analysis: We quantitatively compare the complexity of AAA cases and *AAA-Deviation* cases, using LOC and Cyclomatic complexity. We also compare their AAA layout in terms of the number of *arrange*, *act*, and *assert* in the two types of test cases. LOC provides a basic measure of code size, offering insights into the scale of the test cases. Cyclomatic complexity measures the number of unique execution paths, reflecting the logical complexity of the test cases. While these metrics are not perfect indicators of complexity—a longer test case might be more readable than a shorter, densely written one, and high cyclomatic complexity might result from necessary conditional logic—they are commonly used in empirical software engineering studies to assess code complexity and maintainability. For instance, Athanasiou et al. used both metrics to examine test code maintainability and its impact on issue handling [36]. In the context of our study, they provide objective, reproducible measures for comparing the complexity of test cases that adhere to AAA patterns vs. those that do not.

Additionally, motivated by the testing culture difference between community-driven and corporate-driven projects observed from the survey results, we also compare the percentages of test cases that adhere to AAA pattern in these two types of projects. The assumption is that the corporate-driven projects are likely to demonstrate stronger adherence to AAA patterns in their test cases, which should align with the survey observations of their testing culture and AAA familiarity.

To ensure statistical rigor, for both comparisons introduced above—i.e. 1) complexity of AAA test cases and non-AAA test cases, and 2) percentages of test cases adhering to the AAA pattern in the community- and corporate-driven projects—we employ a sequence of tests. We first assess whether parametric tests are appropriate by testing two key assumptions. We use the Shapiro-Wilk test for normality and Levene’s test for homogeneity of variances, with a significance level of 0.05. As both tests confirm normal distribution ($p > 0.05$) and equal variances ($p > 0.05$) in the two analyses, we finally confirmed using the t-test for both comparisons, with threshold $p \leq 0.05$ indicating a significant difference between the compared groups.

AAA Problem Summarization: After distinguishing test cases as adhering to AAA and deviating from AAA, we proceed with revealing recurring design problems in the context of AAA and the reason for their corresponding improvement resolution.

Our analysis involves a collaborative approach, where one individual team member, referred to as “tagger,” first independently analyzes and summarizes recurring patterns of how the *AAA-Deviation* diverges from the classic AAA structure. This initial analysis includes a detailed review of both the encoding and source code of the test cases. Specifically, we identify the drawbacks of each deviation—discussing whether and how a test case might be restructured to better adhere to the AAA structure and how such improvements could enhance the behavior and quality of the test case.

In parallel, for cases that already follow the AAA structure, we focus on examining each “A” block to identify potential issues within these blocks and discuss strategies for better design. This involves a meticulous evaluation of the arrangement, action, and assertion segments of each test case. The team member then brings his findings to group meetings where everyone discusses and reviews the work. These discussions help us challenge and refine our initial observations and combine our insights.

While we strive to minimize subjectivity in our analysis, it is important to acknowledge that a certain degree of subjectivity is inherently embedded in the process due to its reliance on the collective judgment of the research group. Our methodology aligns with the grounded theory approach [37], constructing findings iteratively and bottom-up through detailed notes and group discussions. This rigorous process enabled us to identify seven distinct AAA problems, which are discussed in detail in Section VI-B.

As elaborated in the definitions of each problem type, six of the identified issues are based on clear and objective structural patterns. For instance, the Multiple AAA problem can be objectively identified by matching patterns of *<arrange, act, assert>*, ..., *<arrange, act, assert>* in the test code. The only category with a higher degree of subjectivity is *Obscure Assert*, relying on individual interpretations of complex logic within the assertion block.

Overall, the identification process is grounded in objective criteria derived from observable patterns in the test code. These criteria include structural deviations from the AAA sequence, missing components in the test structure, and specific code constructs that affect test behavior, such as multiple sequences of AAA blocks, missing verification steps, or control flow structures like try-catch blocks. For issues involving complex logic, particularly within assertion blocks, we use measures such as cyclomatic complexity as an initial filter and rely on group discussions for case-by-case evaluation.

To enhance transparency and reproducibility, we have shared our problem identification criteria and related materials in a publicly accessible data package³.

C. Step 3: Comparison With Test Smells

To understand the strength of our AAA analysis over classic test smells, we compare them conceptually and empirically. Here, we select a state-of-the-art tool, called *JNose*. The reason

³<https://github.com/Codegass/Unit-Test-Empirical-Study-from-AAA-Perspective>

is that it is the most up-to-date tool that covers the most comprehensive detection of the test smells discussed in the literature [26]. *JNose* [25]—detects a total of 21 test smells discussed in detail later.

Conceptual Comparison: First, we make a one-on-one comparison of the definition and detection rules of each test smell with each AAA problem. The objective is to identify logically relevant problems from the two perspectives. **Formally, if a test smell type X and an AAA problem Y check on similar or logically related code features in a test case, we call X and Y a *Logically Relevant Pair*, denoted as $\langle X, Y \rangle$.** For instance, *Conditional Test Logic* (a test smell) checks the usage of *if*, *else*, *while*, *for* in test cases, which is usually a symptom of *Obscure Assert* (an AAA problem), and thus they are likely to identify relevant and co-located problems in the same test case. In comparison, *Magic Number Test* (another test smell) could happen as long as there is a numerical literal in the assertion. It is logically independent of the AAA structure in the test case. Though the same test case could contain a magic number and an AAA problem at the same time, this co-location is incidental.

In our analysis, we noticed that several *Logically Relevant Pairs* involve the same AAA problem, so we treat these test smells together. That is, test smell types X_i to X_j , each forms a *Logically Relevant Pair* with AAA problem Y_p , we examine them as $\langle \{X_i, \dots, X_j\}, Y_p \rangle$. The reason is that, as we will show later, test smells usually focus on specific, detailed code features; while AAA problems are based on the overall AAA structure. Thus, putting the smells together provides a more cohesive comparison between the two perspectives.

Empirical Comparison: For each *Logically Relevant Pair*, we compare whether and how the actually detected problems from one perspective overlap with and/or differ from the other, especially in providing different insights for following-up improvements.

More specifically, let $\langle \{X_{smell}\}, Y_{AAA} \rangle$ be a *Logically Relevant Pair*. Then, $S_x = \{tc_i\}$ is the set of test cases identified with test smell type X by *Jnose*; while $A_y = \{tc_j\}$ is the set of test cases with AAA problem Y . We investigate this *Logically Relevant Pair* in three parts: 1) **Co-locations:** $S_x \cap A_y$, which contains test cases with co-located smell X and AAA problem Y . The objective is to reveal whether and how reviewing the same test case from the two perspectives provides different insights regarding how to improve it. 2) **AAA (Y) Exclusives:** $A_y \setminus S_x \cap A_y$, which contains test cases only detected with the AAA problem Y , but not smell X . The objective is to investigate what new problems Y captures compared to X despite their logical relevance. And, 3) **Smell (X) Exclusives:** $S_x \setminus S_x \cap A_y$, which contains test cases only detected with smell X , but not any AAA problem Y . We assume that test smells could be quite aggressive in raising unnecessary alarms due to their focus on superficial problem indications.

Of particular note, smells that are conceptually independent of AAA problems, discussed in Section IX, are not examined in the experiment, since they may offer orthogonal checks.

D. Step 4: Restructuring and Feedback

This step evaluates the practical value of the AAA problems to developers. Towards this, we fix the AAA problems on sample cases and then send improvement proposals as Issue Tickets (ITs) and/or Pull Requests (PRs) for developer feedback. In the context of this study focusing on open source communities, using ITs and PRs to gather feedback is not only practical but also likely to yield similar insights to those obtained from interviews (as confirmed by our other study focusing on commercial projects through interviews [38]). The IT/PR approach aligns with open-source developers' regular workflows, enabling them to evaluate and apply improvements directly to their projects. This practical incentive makes IT/PR engagement inherently more motivating for open-source developers compared to participating in interviews or surveys, which are often seen as detached from their immediate goals. By focusing on how our proposals could improve their projects, developers are naturally more inclined to provide thoughtful feedback and critiques. In contrast, direct interviews with open-source developers present significant challenges. For example, in the unit testing culture survey in Section III, we reached out to 381 developers and followed up multiple times, but we received only 11 responses. This difficulty stems from the limited availability and geographic dispersion of open-source contributors.

Sample Case Selection: From our dataset of 735 manually analyzed test cases, we carefully select one sample case of each issue type from each project. Our selection strategy carefully balances two key aspects: 1) ensuring comprehensive coverage of the seven identified AAA problem types across all projects in our study, and 2) avoiding overwhelming any single project community with too many pull requests at once. To achieve this balance, we selected one representative case of each issue type within each project. This deliberate approach ensured that all problem types were represented while accounting for the diversity of project contexts. In cases where a particular problem type was not applicable within a project, no issue or pull request was submitted for that type. This strategy resulted in a total of 27 cases, which we believe are sufficient for our exploratory analysis. By carefully distributing these cases across problem types and projects, we were able to gather diverse qualitative insights into developer responses to AAA-related improvements. This allowed us to observe common themes and practical considerations, such as preferences for code readability, maintainability, and adherence to project-specific guidelines, that influence developers' decisions to accept or reject proposed changes.

Case Restructuring: For each selected case, we propose tentative improvement resolutions. The objective is to enhance its behavior and/or improve understandability and maintainability. For a *AAA-Deviation* case, we aim to restructure it to follow the AAA pattern. We also fix the design issues inside of the A blocks. Each case is extensively discussed among all authors (two are real-life developers) in weekly group meetings. The team discusses the resolution, suggests improvements, and reasons about the benefits of refactoring. This may trigger iterative discussions and improvements on a test case. Once

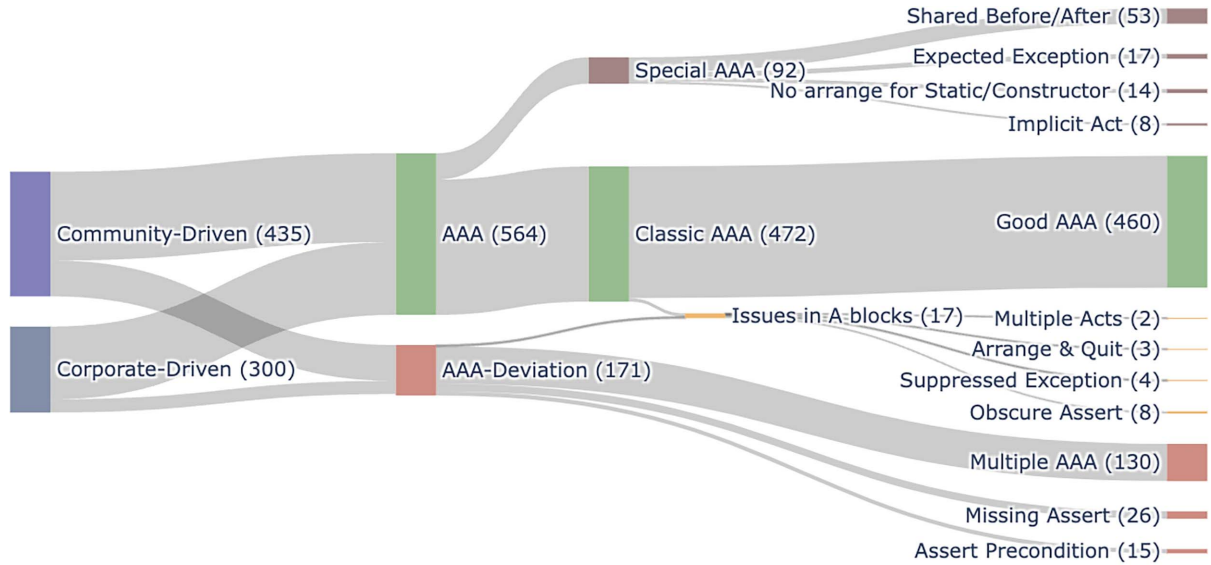


Fig. 5. Test cases categorization.

approved by the entire team, we proceed the case to the next phase—preparing and submitting an issue ticket to the project.

Issue Tickets: We prepare and submit an *Issue Ticket (IT)* describing the problem we found and the resolution we offer. If a *IT* confirms that the problem is valuable or a *PR* is welcome, we move on to a *PR*. When creating an *IT*, we describe 1) what problem is being identified; 2) how to fix the problem; and 3) what is the benefit of fixing the problem. Following is an *IT* sent to Dubbo:

Bring in the Junit Assume to testSubscription: I noticed that *testSubscription* is asserting the precondition (*pList*) before the actual test target (*multipleRegistry.subscribe()*) is executed. So when the test fails, it may be due to 2 reasons: 1) The *pList* has a bug; or 2) *multipleRegistry.subscribe()*—which is the actual target—has a bug. I suggest replacing lines 147-148 with *Assume: Assume functions is a set of methods useful for stating assumptions about the conditions in which a test is meaningful. A failed assumption does not mean the code is broken, but that the test provides no useful information.*

```

145 @Test
146 public void testGetByPrefix_Drop(){
147     Config con = new Config();//arrange
148     tc.set(PROP_PREFIX);//arrange
149     var p = tc.getAllProperties();//act
150     assertEquals("prop", p);//assert
151     tc.set(SCAN_PREFIX);//arrange
152     p = tc.getAllProperties();//act
153     assertEquals("scan", p);//assert

```

Original Code Proposed Code

Fig. 4. *IT* Example.

This can help us focus on the test target, and clearly identify the source of failure.

Pull Requests: We proceed with a *PR* when developers respond positively in an *IT*, or when there is no response after one week. For example, if an *IT* receives feedback such as “Good idea. Would you please submit a *PR*?” we proceed. In contrast, if a developer asks for clarification in an *IT* (e.g.,

about how a function works) and there’s no follow-up after our response, we do not proceed. As we know the *IT* was reviewed but did not draw attention. However, for *ITs* without any response for more than one week, then we will proceed with the *PR*, assuming developers may review *PRs* more diligently than *ITs*.

VI. STUDY RESULTS

The AAA pattern provides a uniform structure to unit test cases, but may not be appropriate to handle the complexity of other tests, such as integration tests. In manual tagging, we exclude 65 test cases from the initial random selection since they are not unit test cases, and thus are out of the scope of this study. The excluded test cases are often integration tests that focus on testing a flow of functions, with names ending with an “IT”. For the remaining 735 test cases, which—to our best understanding—are unit test cases, their categorization of whether they follow the AAA pattern is illustrated in Fig. 5 and discussed in detail below:

A. *RQ1: Adherence to AAA*

Overall, 564 (77%) test cases in our dataset follow the AAA pattern. As shown in Fig. 5, 472 (64%) test cases follows the *classic* AAA pattern, which shows the *arrange*, *act*, and *assert* layout. In addition, 92 (13%) test cases do not directly manifest the *classic* AAA structure, but they essentially follow the AAA design due to special design.

a) **Special AAA:** There are four *Special AAA* patterns: 1) **No Arrange for Static/Constructor:** As illustrated Fig. 6(a), the target of the test case is a static function or a constructor (line 3), thus the test case has no *arrange*. 2) **Shared Before/After:** As exemplified in Fig. 6(b), the *arrange* or *assert* sections are all encapsulated in methods with special annotation, such as @Before (line 1) or @After (line 3). They execute before or

```

1 public void testStatic() {
2     int a = SomeClass.StaticFunc();
3     assertEquals(1,a);}

```

(a) No Arr. for Static/Constructor

```

1 public class DataTest {
2     private final DataConfig config1 = new
        DataConfig();
3
4     @Before
5     public void setup(){data = new DT();}
6
7     @After
8     public void verify(){assertNotNull(data.
        getValue());}
9
10    @Test
11    public void testConfigBig(){data.config(
        config1);}
12 }

```

(b) Shared Before/After

```

1 @Test(expected = ClientException.class)
2 public void testEmptyClientExce() throws
    Exception {
3     try (Client client =new Client()){
4         client.createProfile();}

```

(c) Expected Exception

```

1 @Test
2 public void testEquals() throws Exception
    {
3     Client a = new Client("Bob");
4     Client b = new Client("Bob");
5     assertEquals(a, b);}

```

(d) Implicit Act

Fig. 6. Special AAA example cases.

after the execution of each test case by default. Alternatively, the test class may define an attribute initialized at the start of the class, serving as a common setup for all test cases within that class. 3) **Expected Exception**: As shown in Fig. 6(c), the test case uses the *expected* attribute of the `@Test` annotation to declare that it expects an exception to be thrown (line 1), which is equivalent to *assert*. And 4) **Implicit Act**: The test case does not have an explicit *act*; while the JUnit *assert* function executes the *act* function through dynamic binding. These cases are all associated with testing the *equals* overridden by user-defined functions. An example is shown in Fig. 6(d), where line 5 implicitly contains *act*.

b) Comparison of AAA vs. AAA-Deviation Cases: We compared the two types of test cases across several metrics using t-tests. The results showed no statistically significant difference in LOC, cyclomatic complexity, number of *arrangement* statements, or number of *assertion* statements. However, we found a significant difference in the number of *act* statements ($p < 0.001$), with AAA-Deviation cases containing 3.3 times more *act* statements than AAA cases. This indicates that AAA-Deviation cases often violate the single responsibility

```

1 @Test
2 public void testGetByPrefix(){
3     Config con = new Config(); //AR
4     tc.set(PROP_PREFIX); //AR
5     var p = tc.getProperties(); //AC
6     assertEquals("prop", p); //AS
7
8     tc.set(SCAN_PREFIX); //AR
9     p = tc.getProperties(); //AC
10    assertEquals("scan", p); //AS

```

(a) Before Restructuring

```

1 @Test
2 public void testGetByPrefix_PROP(){
3     Config con = new Config(); //AR
4     tc.set(PROP_PREFIX); //AR
5     var p = tc.getProperties(); //AC
6     assertEquals("prop", p); //AS
7
8     @Test
9     public void testGetByPrefix_SCAN(){
10        Config con = new Config(); //AR
11        tc.set(SCAN_PREFIX); //AR
12        var p = tc.getProperties(); //AC
13        assertEquals("scan", p); //AS

```

(b) After Restructuring

Fig. 7. Multiple AAA example case.

principle [6], which states that each test case should focus on testing one specific behavior.

c) Adherence to AAA in Community-Driven vs. Corporate-Driven Projects: 85% of test cases in a corporate-driven project adhere to the AAA structure; while 71% of test cases in a community-driven project follow the AAA structure, indicating stronger adherence to the AAA pattern in corporate-driven projects. The percentages of test cases adhering to AAA structure differ significantly between community-driven and corporate-driven projects ($p = 0.005$) based on the t-test. This observation aligns with our survey findings, where corporate-driven project participants reported higher familiarity and more consistent application of AAA structure in their practice, compared to the more varied responses from participants of community-driven projects.

RQ1 Summary: Overall, 77% test cases follow the AAA structure, indicating AAA's general adoption in practice. In particular, following the AAA structure could be a way to facilitate the single responsibility principle in test case design—focusing only on one target function.

B. RQ2: AAA Problems

1) AAA-Deviation Patterns: We summarized three recurring patterns that deviate from the AAA, explained in detail below:

a) Multiple AAA: A test case is composed of more than one AAA sequences, namely `<arrange, act, assert>`,...`<arrange, act, assert>`. As shown in Fig. 7(a), the test case contains two sequences of AAA. Line 8 to line 11, the first sequence, intends to test *getAllProperties* with

```

1 public void testDataGenerator() {
2     Data d = new Data(); //AR
3     d.generate(); //AC
4     printData(d.getData());
5 private void printData(var input) {
6     for (String d: input) {
7         System.out.println(d);
8     }
9 }

```

(a) Before Restructuring

```

1 public void testDataGenerator() {
2     Data d = new Data(); //AR
3     d.generate(); //AC
4     Vector<String> exp = load("gen.dat");
5     for (String d: input) {
6         assertEquals(exp, d);
7     }
8 }

```

(b) After Restructuring

Fig. 8. Missing assert example case.

PROP_PREFIX; while line 13 to line 15, the second sequence, tests scenario *SCAN_PREFIX*.

Problem: In practice, the test case could get very large with multiple AAA sequences combined, compromising the comprehension and maintenance of the test case. This anti-pattern also violates the single responsibility principle in software design [6], as each test case should focus on verifying one specific behavior. This is also a violation of the equivalence class partitioning (ECP) [39] in testing theory - each test should examine one distinct input partition (such as *PROP_PREFIX* or *SCAN_PREFIX*) with its expected behavior. When the scenarios representing different input partitions are combined, a failure in testing one scenario prevents the verification of others, potentially masking additional issues and increasing debugging complexity.

Restructuring: Break this test case into separate test cases, each with a single AAA sequence. As such, each test case follows the classic AAA pattern, focuses on one test scenario, and fails due to only one reason. Fig. 7(b) shows the resolution of the example case split into two test cases, starting on line 1 and on line 6 respectively.

If restructuring leads to code clones in multiple test cases that only vary in the input parameters [40], the developer could use the JUnit5 parameterized test feature to eliminate code clones. That is only one test case with annotated parameters. However, based on our observation, *Multiple AAA* usually only share the same action, and will not lead to code clone.

b) Missing Assert: A test case does not specify any expected behavior, which manifests this structure: <arrange, act>. An example is shown in Fig. 8(a), which uses *printData* (lines 6-8) to delegate the inspection of results to manual effort.

Problem: Regardless of the correctness of the target function, the test case provides no explicit verification of expected outcomes. While runtime exceptions (implicit oracles) might detect some failures, explicit assertions significantly improve fault detection in test cases [41].

Restructuring: Specify the expected behavior of the test case so that it will fail when something went wrong. Note that

```

1 public void testPoll() {
2     Snapshot s = sqlMng.createSnapshot();
3     assertNotNull(s);
4     String v = s.poll();
5     assertEquals("8/22/2022", v);
6 }

```

(a) Before Restructuring

```

1 public void testPoll() {
2     Snapshot s = sqlMng.createSnapshot();
3     assumeNotNull(s);
4     String v = s.poll();
5     assertEquals("8/22/2022", v);
6 }

```

(b) After Restructuring

Fig. 9. Assert pre-condition example case.

this could be done in different ways, such as using JUnit's *assert*, mocking frameworks' *verify*, or the *@expected* annotation. Fig. 8(b) shows the resolution of the example. The expected value (from the *printData*) could be stored in external resources (line 5), and loaded into the test case for assertion (line 7).

c) Assert Pre-Condition: The test case asserts a pre-condition of the arranged objects before acting the function under test, which shows a structure of <arrange, assert, act, assert>. As shown in Fig. 9(a), *assertNull* (line 4) makes sure that the *snapshot* is acquired (i.e. not null) from the database, checking a pre-condition before executing the *action* in line 5.

Problem: It is not clear why the test case fails: 1) the pre-condition is not met, or 2) the function under test contains errors. Also, the *act* will not execute if the pre-condition does not hold.

Restructuring: Replace the *assert* by the Junit *assume* for checking pre-condition. The *assume* is a set of methods introduced since JUnit 4 for stating assumptions about the conditions in which a test is meaningful. A failed *assume* does not mean the code is broken, but that the test provides less useful information. The restructuring of the example is shown in Fig. 9(b), with the change in line 4.

2) Issues in "A" Blocks: Following the AAA pattern does not imply that the test case is perfect. Here we present four issues we observed that reside in the A blocks of the AAA context:

a) Obscure Assert: The assert block contains obscured logic, which hinders the understanding of what is asserted. Admittedly, the definition of this issue by its nature could be subjective to an analyst's interpretation. In our analysis, we examined all the test cases whose *Assertion* block has a cyclomatic complexity [42] greater than 2, which requires the presence of at least two layers of nested conditional logic that incorporate assertions. However, this condition does not always lead to *Obscure Assert*. Fig. 10(a) shows an example, where *for* loop plus the *if-else* block (lines 5 to 8) asserts that the elements in a collection satisfy certain conditions.

Problem: As the name suggests, this design flaw adds unnecessary complexity to the assert logic, obscures the intention of assert, and adds difficulty to comprehension and maintenance.

Restructuring: Simply the logic of *assert* to make its intention explicit. Fig. 10(b) shows how the case in Fig. 10(b)

```

1  @Test
2  public void testCluster() {
3      ...
4      Boolean foundValid = false;
5      for(int cluster:clusterList){
6          if(cluster != 1){fail("Err");}
7          else{ foundValid = true;}}
8      assertTrue(foundValid);}

```

(a) Before Restructuring

```

1  import org.hamcrest.Matchers.*;
2  @Test
3  public void testCluster() {
4      ...
5      assertThat(clusterList, everyItem(
        equalTo(1)));}

```

(b) After Restructuring

Fig. 10. Obscure assert example case.

```

1  @Test
2  public void testSetException() {
3      Throwable thr = buildXExp();
4      if (thr == null) {return;}
5      App app = new App().setExp(thr);
6      assertEquals(0, app.getMsg());} //
    Corrected Java assert method

```

(a) Before Restructuring

```

1  @Test
2  public void testSetException() {
3      Throwable thr = buildXExp();
4      assumeNotNull(thr); // Corrected
        usage of JUnit assume method
5      App app = new App().setExp(thr);
6      assertEquals(0, app.getMsg());} //
    Corrected Java assert method

```

(b) After Restructuring

Fig. 11. Arrange & quit example case.

could be improved with one single *assert* statement by using the *hamcrest* API (line 5). Note that the resolution could be case-by-case depending on the concrete logic.

b) Arrange & Quit: The test case *returns silently if the arranged object does not meet a certain condition, as such it has this structure* *<arrange, if-return, act, assert>*. This is comparable to the *<arrange, assert, act, assert>* structure of *Assert Precondition*. See Fig. 11(a) for an example, where the test case *returns* when *thr* is *null* (line 4).

Problem: A bug in the target function hides unnoticed when the *return* happens.

Restructuring: Replace the *if-return* block by an *assume* for the precondition, as shown in Fig. 11(b), line 4. The resolution is similar to that of *Assert Precondition*.

c) Multiple Acts: The test case contains more than one *action*, thus with a structure of *<arrange, act₁, ...act_n*,

```

1  @Test
2  public void testCreateAndInfo() {
3      ...
4      qemu.create(file);
5      Map info = qemu.info(file);
6      assertTrue(file.exist());
7      assertEquals(SIZE, info.size());
8      ...}

```

(a) Before Restructuring

```

1  @Test
2  public void testCreate() {
3      qemu.create(file);
4      assertTrue(file.exist());}
5  @Test
6  public void testInfo() {
7      qemu.create(file);
8      assumeTrue(file.exist());
9      Map info = qemu.info(file);
10     assertEquals(SIZE, info.size());}

```

(b) After Restructuring

Fig. 12. Multiple acts example case.

assert>. The example in Fig. 12(a) contains two *actions*: *create* (line 4) and *info* (line 5), suggested by test case name, *testCreateAndInfo*.

Problem: If the test case fails, it is difficult to tell which *action* leads to the failure. In addition, *assert* usually focuses on the final output, leaving the first *action* not sufficiently checked.

Restructuring: Break into multiple test cases, each focusing on one single *action*. As shown in Fig. 12(b), there should be two test cases, focusing on *create* (line 3) and *info* (line 9), separately.

However, the refactoring should be taken with caution. Our observation also revealed that not all multiple-act test cases can or should be split into separate tests. Some unit tests inherently rely on executing a sequence of intrinsically related operations to verify a single unit of functionality. These are not integration tests, as they still focus on a single unit of functionality, but that unit may involve multiple logically connected actions. For example, when testing a serialization-deserialization cycle, the meaningful outcome only emerges after both operations have occurred in sequence. Splitting them into separate tests would not faithfully represent the intended validation, since the correctness depends on the combined effect of both steps.

d) Suppressed Exception: The test case *try to execute an action, and uses a catch to suppress a possible exception in the action. The structure may look like* *<arrange, try{action}, catch{}, assert>*. Fig. 13(a) shows such an example, with the *try-catch* block (line 4 to line 6), and there is a *printStackTrace* inside of *catch*.

Problem: If there is an *exception* with the execution of the *action*, the *catch* suppresses it and will not raise a failure. Thus, it hides the problem of *action* from developers.

Restructuring: Replace the *catch* by *throws* so that the exception will raise a failure to catch developers' attention when something went wrong with the *action*. Fig. 13(b) shows the

```

1  @Test //Suppressed Exception
2  public void testHttpClient() {
3      ...
4      try { client.execute(); } //AC
5      catch (final ClientException e) {
6          e.printStackTrace();
7      }

```

(a) Before Restructuring

```

1  @Test
2  public void testHttpClient() throws
   ClientProtocolException {
3      ...
4      client.execute(); //AC
5      ...}

```

(b) After Restructuring

Fig. 13. Suppressed exception example case.

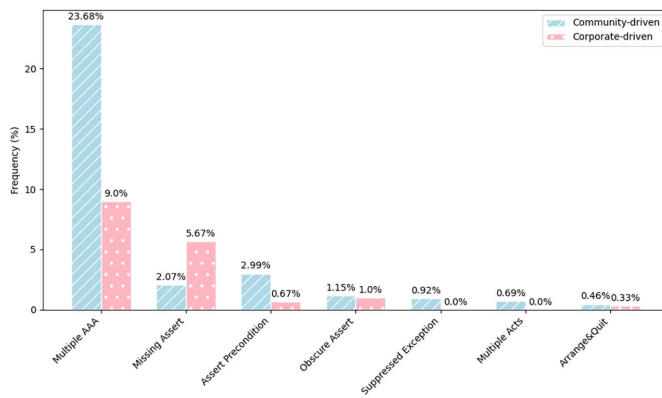


Fig. 14. Comparison of AAA deviation and issue frequencies in different project types.

corresponding resolution for the example case, where *throw* is added in line 2.

Fig. 14 shows the distribution of the issue types in community-driven projects and corporate-driven projects. Overall, there is no statistically significant difference between the two groups of projects. Notably, *Multiple AAA* is most common in both groups. While *Suppressed Exception* and *Multiple Acts* were exclusively noted in community-driven projects, we cannot reason about the generalizability of such absence from corporate-driven projects with the limited number of projects studied.

RQ2 Summary: We summarized three recurring AAA-Deviation Patterns and four types of design issues reside in the “A” blocks that merit attention. In particular, except *Obscure Assert*, the proposed restructuring of all the other six problems enhances test case behavior by exposing failures with better clarity. Meanwhile, fixing seven AAA problems all should benefit the overall design quality of the test cases.

C. RQ3: Comparison With Test Smells

We identified six *Logical Relevant Pairs* between six (out of twenty-one) types of test smells and six (out of seven) types of AAA problems. Note that the *Arrange & Quit* from our AAA analysis is independent of any existing smells and is not the focus of this RQ.

During our manual analysis of test smells detected by *JNose*, we find three technical shortcomings in *JNose* that contribute to both false negatives and false positives, affecting the comparison with AAA issues. Firstly, *JNose* does not expand method calls within test cases during its smell detection process. Consequently, it overlooks symptoms of smells that are embedded within these methods, leading to false negatives across various smell types. For instance, this oversight may cause *JNose* to miss assertion statements, leading to false positives in the detection of the *Unknown Test* smells. Secondly, *JNose* struggles to accurately capture conditional logic involving nested structures such as *if*, *else*, *switch*, *for*, and *while*, resulting in false negatives in conditional test logic. Lastly, in its search for *Unknown Test*, *JNose* only recognizes *assert* statements from JUnit, neglecting other syntactic forms, such as *verify* from mocking frameworks, which results in additional false positives. The impact of these issues will be further detailed in the pairwise comparison later.

Comparison Overview: Fig. 15 shows an overview of how the six *Logical Relevant Pairs* actually co-locate and distinguish from each other. *Pair 1*, involving three smells and *Multiple AAA*, is in Fig. 15(a); and the other five one-on-one pairs are in Fig. 15(b). In each matrix, the rows list the smells and the columns list the AAA problem(s). The yellow cells on the diagonal show the number of *co-locations*. For instance, there are 2 *co-location* between *Exceptional Handling* and *Suppressed Exception*. The blue cells on the first two columns show the number of *Smell Exclusives*; while the green cells on the first two rows show the number of *AAA Exclusives*. For example, there are 60 test cases identified with *Exception Handling* only, and 2 test cases identified with *Suppressed Exception* only.

Before we dive into the detailed comparison of AAA issues and test smells, we can make two key observations: 1) As shown in Fig. 15(a), the *Multiple AAA* and the three logically relevant test smells (i.e. *Eager Test*, *Duplicate Assert*, and *Verbose Test*) only have a relatively small amount of co-locations. In contrast, more test cases exclusively contain either *Multiple AAA* or test smells, but not both. 2) As shown in Fig. 15(b), the five AAA issues and the respective logically relevant test smells only have minimal co-locations against a large number of alarms of test smell exclusives. **Both observations indicate that the AAA issues provide a different perspective on the test case design from the test smells.** Following, we elaborate on each *Logical Relevant Pair*—how they differ conceptually and empirically:

1) *Pair 1: <{Eager Test, Duplicate Assert, Verbose Test}, Multiple AAA>*:

- *<{Eager Test, Duplicate Assert, Verbose Test}>*: These three smells detect a test case with the following problems respectively: 1) more than one production function calls; 2) with identical *assertions*; and 3) the LOC ≥ 30 .

Co-locations	Smell Exclusive	Multiple AAA		
AAA Exclusive		94	103	90
Eager Test	136	36		
Duplicate Assert	14		27	
Verbose Test	86			40

(a) Pair 1

Co-locations	Smell Exclusive	Suppress-exception	Missing Assert	Obscure Assert	Multiple Acts	Assert Precon.
AAA Exclusive		2	17	8	1	11
Exception Handling	60	2				
Unknown Test	44		9			
Conditional Test Logic	65			0		
Eager Test	171				1	
Duplicate Assert	37					4

(b) Pair 2 to Pair 6

Fig. 15. JNose and AAA problem comparison matrix.

- *Multiple AAA* detects a test case if it contains multiple blocks of *Arrange-Act-Assert* sequences.

Conceptual Comparison: On the one hand, the three smells reveal the fragments and surface symptoms of the *Multiple AAA* which is the actual root problem of the test case. More specifically, a test case with *Multiple AAA* could satisfy all three smell definitions due to the multiple blocks of AAA sequences. However, chasing the symptoms of smells could be misleading or confusing—such as removing additional production functions (i.e. *Eager Test*) or eliminating duplicated assertions (i.e. *Duplicate Assert*), or struggling without any clue on how to reduce its lines of code (i.e. *Verbose Test*). In comparison, treating it as *Multiple AAA* provides the right direction of guidance to restructure the test case—breaking it down into multiple test cases, each containing a separate AAA sequence, which naturally eliminates the symptoms of the three smells.

On the other hand, the *Multiple AAA* cases do not necessarily contain *Duplicate Assert* or *Verbose Test* since 1) the *assertions* from different AAA blocks could look different, while *Duplicate Assert* requires identical assertions; and 2) the LOC of a *Multiple AAA* test case could be below the threshold of 30. Thus, the two test smells cannot serve as a proxy for *Multiple AAA*. Note that *Multiple AAA* will always have the symptoms of *Eager Test* due to the multiple *actions* from the AAA sequences. However, examining the case as *Eager Test* misses the related context of the other two A's for guiding proper improvement.

Empirical Comparison: As shown in Fig. 15(a), the *Multiple AAA* issues are co-located with 36 *Eager Test*, 27 *Duplicate Assert*, and 40 *Verbose Test*, respectively; while there are much more test cases containing *Multiple AAA* exclusively without the smells. **This indicates that *Multiple AAA* focus on design issues in test cases that are quite distinctive from test smells, despite the logical relevance.**

Fig. 16 shows an example of how co-location happens based on a simplified real test case. The test case *testParseUrl* contains all the smells co-located with the *Multiple AAA*. Using the smell detection tool, one would receive a report that reads like “*Duplicate Assert* detected on line# 6 and 9; *Verbose Test* with more than 30 lines of code; *Eager Test* detected on line# 5,6, and 9”. This does not provide much useful information regarding how to improve. In comparison, examining it based on the rationale of *Multiple AAA* would suggest that one should break the test case down into two test cases, based on the two AAA sequences—lines 3 to 6, and lines 7 to 9—each focuses on one scenario of *parseUrl*. Note that real-life test cases are

```

1  @Test
2  public void testParseUrl{
3      String url = "192.168.1.1"; //AR
4      Params p = parseUrl(url); //AC
5      assertEquals(p.getPort(), 3000); //AS
6      assertEquals(p.getType(), Type.a); //AS
7
8      url += "iscsi"; //AR
9      p = parseUrl(url); //AC
10     assertEquals(p.getType(), Type.a); //AS
11     ...}

```

Fig. 16. Multiple-AAA co-location.

more complicated, and reviewing them with the right guidance is even more critical.

Besides the relatively small amount of co-locations, there are more test cases exclusive to *Multiple AAA* or the three test smells. On the one hand, 136, 14, and 86 test cases contain the *Eager Test*, *Duplicate Assert*, and *Verbose Test* respectively without being *Multiple AAA*. This aligns with our conceptual analysis that the three smells could be the symptoms of *Multiple AAA*, but having the smell symptoms on its own does not warrant *Multiple AAA*—for instance, a test case longer than 30 LOC (*Verbose Test*) does not necessarily have multiple blocks of AAA in it. On the other hand, 94, 103, and 90 test cases are *Muptile AAA* without being detected with *Eager Test*, *Duplicate Assert*, or *Verbose Test*, respectively. This confirms our conceptual analysis that *Multiple AAA* does not always show the relevant smell symptoms. For instance, a *Multiple AAA* test case may be less than 30 LOC.

Of a particular note though, we do expect *Multiple AAA* to **always** show the symptom of *Eager Test*, as the multiple AAA blocks imply more than one *action*. Therefore, there should not be any *Multiple AAA* exclusive with regard to *Eager Test*. Upon further investigation, the 103 *Multiple AAA* were missed by JNose as *Eager Test* because the latter does not expand method calls inside of test cases. In other words, if a AAA block hides inside a method call, JNose would miss it. We argue that expanding method calls, which may contain reused testing logic, is important to capture the full context of a test case when analyzing the AAA structure or test smells. However, it is largely missed by the test smell analysis, leading to the 103 false negatives by JNose.

In summary, these three smells are fragmentary and surface symptoms that lay above the *Multiple AAA*. Thus checking for *Multiple AAA* provides the right direction

of guidance to restructure the test case, which would naturally eliminate the smell symptoms altogether. On the contrary, the three smells cannot serve as a proxy to *Multiple AAA* due to their aggressive and superficial detection strategy.

2) **Pair 2: <Exception Handling, Suppress-Exception>:**

- *Exception Handling* detects a test case as long as it contains any “try” clause. The suggestion is to use JUnit exception handling, such as *Throw* or *expectThrow*.
- *Suppress-exception* detects a test case when it tries to catch an exception in the *action*. The suggestion is to *throw* (but not to *catch*) the *exception* to raise a failure so as to draw the developer’s attention.

Conceptual Comparison: They are logically relevant since they both check exceptions in test cases. However, the difference is that *Exception Handling* specifically searches “try” as a symptom without considering the context of the exception; while *Suppress-exception* searches a suppressed *exception* that won’t raise a failure. We reason that *Exception Handling* would be a super-set of *Suppress-exception*. The super-set does not necessitate restructuring the test case, but the subset must be fixed to make sure that an exception in the *action* raises a failure, instead of being hidden.

Empirical Comparison: There are 2 co-location test cases, where the *action* of a test case is wrapped inside of a *try-catch* block. In these cases, the restructuring suggestion is consistent from both perspectives—replacing the *try-catch* with *throw* so that JUnit raises a failure when the exception happens.

Note that there are 2 test cases exclusive to *Suppress Exception*, where *JNose* failed to detect. This is counter-intuitive to our conceptual analysis. Further investigation reveals that *JNose* did not scan the test class files for these two cases because it only scans test folders within the specific *Maven* project structure. In contrast, our method scans all folders named ‘test’, which is why *JNose* missed these files initially.

There are a total of 60 test cases detected as *Exception Handling* exclusively by *JNose*. Manual inspection confirmed our conceptual analysis that none of the *Exception Handling* cases necessitate or are feasible for restructuring. They can be categorized into three main types (with overlap): 1) *Try-Resource* (21 cases): The test cases employ the *try-with-resource* feature from Java [43] or use the *try-final* structure to close a resource. This is necessary and irrelevant to the *exception handling*, and thus cannot be changed. An example is shown in Fig. 17(a). 2) *Fail-with-Catch* (10 cases): Such a test case wraps the *action* in a *try* block, which follows a *catch* block with *fail* inside so that the test case will fail with the caught exception. An example is shown in Fig. 17(b). Given that the exception is not suppressed with the explicit *fail*, there is no need nor feasible to improve such cases. And 3) *Expected-Exception* (35 cases): The test case wraps the *action* and *Assert.fail* in a *try* block, which follows a *catch* block. In some cases, inside the *catch* blocks check more details about the *exception*. The intention of the test case is to pass with the *exception* and fail without the *exception*. Therefore, there is no need to improve such cases. An example is shown in Fig. 17(c). Note that with JUnit 5 or above, such cases can be improved

```
1 try (FileReader fr = new FileReader(path))
2 {
3     String str = fr.readLine();
4     assertNull(str);}
```

(a) Try Resource

```
1 try {
2     func();
3 } catch (Exception e) {
4     fail("something wrong");}
```

(b) Fail with Catch

```
1 try {
2     f.write(str);
3     Assert.fail();
4 } catch (IOException e) {
5     assertEquals(1, f.size());}
```

(c) Expected Exc.

Fig. 17. Exception handling exclusive.

with *assertThrow* for a better coding style. But this change is subjective to developers’ expertise and project convention.

In summary, *Exception Handling* tends to raise unnecessary alarms. While *Suppressed Exception* focuses on a subset necessitates restructuring for the sake of revealing suppressed test failure with exceptions.

3) **Pair 3: <Unknown Test, Missing Assert>:**

- *Unknown Test* detects a test case if the number of asserts is 0 or no *exception* is expected. Such a test case would never fail and should be fixed by adding *assertion*.
- *Missing Assert* detects a test case when there is missing *assertion* for its *action*. The suggestion is to add the missing *assertion* so every *action* is checked.

Conceptual Comparison: Both concepts check for any missing *assertion* in a test case. However, *Missing Assert* considering the full context of the AAA structure of a test case. As such, a test case may contain *asserts* yet still lack critical verifications. For example, a test case may have multiple actions but not all of them are asserted; or existing assertions check the pre-conditions, leaving the actual actions not checked. By considering the AAA context, *Missing Assert* captures such cases where assertions are inadequate; while *Unknown Test* only targets cases where assertions are completely absent.

Empirical Comparison: There are 9 test cases with *Co-locations*, with the same follow-up suggestion from both checks—adding missing *assertion*.

There are 17 test cases exclusively identified with *Missing Assert*, which *JNose* failed to detect. A manual examination of these cases revealed that *JNose* mistakenly considers 16 test cases as having legitimate assertions when a special annotation like *@Test(expected=xxxException.class)* appeared with other test cases of their test classes. The remaining case [44], shown in a simplified form in Fig. 18, supports our conceptual comparison that *Missing Assert* is more comprehensive considering the AAA context. This case contains an assertion that verifies a

```

1  @Test
2  public void deleteTest() {
3      Strategy s = Strategy(); //AR
4      Data data = createData(); //AR
5      AssertNotNull(data); //AS-Pre.
6      s.delete(data) //AC
7  }

```

Fig. 18. Missing assert exclusive.

precondition with the *arranged* object—*data*, but lacks the most critical *assert* for the actual *action*'s outcome. *JNose* missed it as an *Unknown Test* since its unawareness of the AAA context.

However, counter-intuitive to our conceptual analysis, we observed 44 test cases uniquely identified as *Unknown Test*. As discussed earlier, *Missing Assert* should be more comprehensive than *Unknown Test* so exclusives to *Unknown Test* are not expected. Our manual review revealed that all these cases were false positives, due to the following two reasons: 1) *JNose* recognizes assertion syntax focusing on *assert* and *fail* keywords, but missing other syntactic elements like *verify* used in mocking frameworks, *ExpectedException* employed by JUnit Rules [45], and native assert statements from Java. 2) *JNose* does not extend its analysis to method calls in test cases which may hide assertions. Consequently, some test cases were incorrectly categorized as having no *assertions* if the *assertions* appeared inside of method calls.

In Summary, while *Unknown Test* and *Missing Assert* conceptually target similar issues, the *Missing Assert* from our analysis tends to be conceptually more comprehensive for ensuring that all actions in test cases are properly asserted, even when assertions are already present.

4) Pair 4: <Conditional Logic, Obscure Assert>:

- *Conditional Logic* detects a test case as long as there is *if*, *else*, *for*, *while*, etc; The suggestion is to eliminate the usage of the conditional logic (which is not always feasible).
- *Obscure Assert* detects a test case whose assertion logic is implicit for understanding. The suggestion is to make the assertion logic more explicit.

Conceptual Comparison: *Conditional Test Logic* simply focuses on the symptoms—the usage of *conditional logic* in test cases, regardless of its context and effects. In contrast, *Obscure Assert* targets test cases whose *assertion* logic is difficult to understand, due to the result of complicated conditional logic that could be simplified and straightened out. Admittedly, the discernment of *Obscure Assert* could be subjective based on individual's comprehension. However, the key point is that *Conditional Test Logic* is not necessarily always problematic simply because of the usage of *conditional logic* in test cases. As we will show later through the empirical comparison, it is often necessary and cannot be changed. Therefore, we reason that *Conditional Logic* becomes *Obscure Assert* when the conditional logic appears in the *assertion* block and obscures the logic, but the former tends to raise a large number of alarms that not necessarily problematic.

```

1  @Test
2  public void testSelection() {
3      ...
4      for (int i = 0; i < values.length; i++)
5      {
6          agg.aggregate(); //AR
7          selector.increment(); //AC
8      }
9      assertTrue(selector.isExpected()); //AS

```

(a) Iterative Arrange and Act

```

1  @Test
2  public void testServer() {
3      String envSet = readEnvSetting();
4      if (envSet == null) {
5          setProperty(Key, envSet); //AR
6      } else {
7          replaceProperty(Key, envSet); //AR
8      }
9      server.start(); //AC
10     assertTrue(server.isRunning()); //AS
11 }

```

(b) Conditional Arrange

Fig. 19. Conditional logic exclusive.

Empirical Comparison: Contrary to our conceptual analysis, there is *no* test cases with *Co-locations*. Our manual review revealed that *JNose* failed to detect the 8 *Obscure Assert* cases due to two implementation issues: 1) *JNose* fails to capture nested conditional logic (e.g. two *if* statements nested within each other) which seems to be a bug, and 2) *JNose* misses the conditional logic if it appeared within method calls.

As expected, *JNose* raises 65 test cases with *Conditional Logic* smell, where improvements are either not feasible or unnecessary. These cases are in three types with overlaps: 1) *Simple Conditional Assertions* (20 cases): These cases have assert blocks characterized by a single layer of conditional logic. They generally do not warrant restructuring due to their simplicity. 2) *Iterative Arrange and Act* (43 cases): In these test cases, arrange and/or act blocks are encapsulated within loops (e.g. *for*, *while*, *foreach*). This structure is necessary for configuring arranged objects in collections and performing iterative actions, making the removal of the conditional logic unfeasible. Fig. 19(a) illustrates such an example. 3) *Conditional Arrange* (22 cases): These test cases use conditional logic blocks like *if-else* to establish consistent test environments while addressing potential variations in external conditions for the *action*. Fig. 19(b) demonstrates this pattern, where line 4 to line 8 ensures the appropriate property of *envSet*.

In summary, *Conditional Logic* aggressively tags test cases that are often not necessary or feasible to improve. *Obscure Assert* aims to make *assertion* logic more explicit.

5) Pair 5: <Eager Test, Multiple Acts>:

- *Eager Test* tags a test case that involves multiple calls to production functions. The rationale is that each test case should focus on testing one function. This approach, however, doesn't distinguish whether a call serves as an *arrange* or an *act* within the test case.

- *Multiple Acts* detects a test case that contains multiple *actions*. Likewise, each test case should contain one *action*.

Conceptual Comparison: Both aim to detect test cases that try to test more than one target function. However, *Eager Test* does not differentiate the roles of the production functions in the context of AAA. It is possible that a production function call serves as *arrange* but not as *action*. In comparison, *Multiple Acts* is based on a comprehensive analysis of the AAA context to identify actual *actions*. Therefore, *Eager Test* should be the super-set of the *Multiple Acts*, blaming its unawareness of the AAA context.

Empirical Comparison: There is one test case with *Co-locations*. Furthermore, 171 test cases are identified exclusively as *Eager Test* smell. Upon manual inspection, these were false alarms due to the erroneous recognition of production method calls, intended for *arrange* purposes, as *action*. Meanwhile, one test case is identified as *Multiple Acts* exclusively. *JNose* fails to recognize this case because the *actions* are hidden in method calls, and as described in earlier comparison, *JNose* does not expand and analyze method calls in test cases.

In summary, *Eager Test* tends to raise a large number of false alarms, blaming its unawareness of the AAA context.

6) **Pair 6: <Duplicate Assert, Assert Precondition>:**

- *Duplicate Assert* identifies a test case with two identical *assertions*. The elimination of duplication is suggested.
- *Assert Precondition* detects a test if it *asserts* a precondition from the *arrangements*. We suggest replacing this *assert* with *assume*.

Conceptual Comparison: Both checks the *assertion*. However, *Duplicate Assert* simply checks for duplication. While *Assert Precondition* checks the purpose of the *assertion* in the AAA context. We reason that they could overlap when the *Assert Precondition* happens to exactly duplicate an actual *assertion* later. Otherwise, these two problems actually focus on completely different problems.

Empirical Comparison: There are 4 occurrences of *co-locations*. However, the suggestions from the two perspectives are completely different. For instance, Fig. 20 illustrates test case *testRepeatConnect*, where *Duplicate Assert* flags lines 6 and 8 as identical *asserts*. A direct interpretation would be removing one of the *asserts*. However, examining the AAA context, the first call to *connect* (line 5) and the subsequent *assert* (line 6) set up and validate the context for the second *connect* (line 7), which is the real *action*. We should replace the first *assert* with *assume*. As such, the test fails only due to the *action*—second *connect*.

There are 37 cases of *Duplicate Assert Exclusive*. They all seem to be a truly naive duplication or on purpose duplication mentioned in pair one. On the other hand, **there are 11 cases in *Assert Precondition Exclusives*.** In these 11 cases, the *assertion* for the precondition is not identical to any other assertion in the test case. Therefore, *Duplicate Assert* cannot serve as a detection proxy for *Assert Precondition*.

In summary, although *Duplicate Assert* and *Assert Precondition* share focus on the *assertion*, they essentially check completely different problems with different resolutions.

```

1 @Test
2 public void testRepeatConnect() {
3     URL url1 = "host:Port/url1"; //AR
4     Client C1 = connect(url1); //AR
5     assertEquals("url1", C1); //AS-Pre.
6     C1 = connect(url1); //AC
7     assertEquals("url1", C1); //AS
8 }

```

Fig. 20. Duplicate assert, assert precondition co-location.

RQ3 Summary: We would like to highlight two significant strengths of focusing on AAA problems over test smells: 1) AAA problems, particularly *Multiple AAA*, point to the actual root causes of symptomatic smells and thus provides the right direction of guidance for improvement, which would naturally eliminate the test smell symptoms altogether. And 2) AAA problems are feasible and necessary to be fixed for enhanced test behavior and design quality; while *Test smells* generally aggressively raise a large number of unnecessary alarms and suggest superficial fixes, due to their focus on naive symptoms.

D. RQ4: Developers' Feedback

Table III lists the IT/PR status for the 27 refactoring proposals we submitted, covering all the seven AAA problems across the seven projects. The table shows the details of each refactoring proposal, regarding the project it is for, the test case it is applied on, the AAA problem, the turn-around time, and the status of the IT and/or the PR of this proposal. The final status column indicates whether the proposal was merged, rejected, or is still pending a response. **Overall, our response rate stands at 100%, with 78% of responses arriving within five days. Among the 27 proposals, 19 were merged, 5 were rejected, and 3 are still pending for response. This prompt and highly positive engagement with the proposals demonstrates a high level of interest in AAA improvements.**

a) **Accepted Proposals:** Among our submissions, 19 proposals (70%) were successfully merged into their respective projects. These accepted proposals can be categorized into three distinct response patterns:

Internal PR (Cases 1-3): For our 3 proposals to Dubbo, developers demonstrated exceptional receptiveness by creating pull requests themselves immediately after reviewing our issue tickets. These cases addressed Multiple AAA, Missing Assert combined with Assert Precondition, and Assert Precondition issues. This proactive response suggests that our proposed improvements align well with the project's quality requirements and standards.

Merged PR without IT Response (Cases 14-17): Four proposals to Archaius were merged without receiving any response to our issue tickets. After sending our very first IT to Archaius, we waited for a week; without any response to the IT, we submitted a corresponding PR (as described in Section V-D), which was promptly addressed and merged.

TABLE III
SUMMARY OF IT/PR STATUS

ID	Project	Test Case	Problem	T/A	IT	PR	Final Status
1	Dubbo	testAll [46]	Multiple AAA	2 D	Internal PR	M	Merged
2	Dubbo	testClear [47]	Missing Assert & Assert Precond.	10 D	Internal PR	M	Merged
3	Dubbo	testSubscription [48]	Assert Precond.	2 D	Internal PR	M	Merged
4	Dubbo	testSetException* [49]	Arrange & Quit	13 D	PR-Invited	M	Merged
5	Accumulo	verifyException* [50]	Obscure Assert	1 D	PR-Invited	M	Merged
6	Accumulo	testGetByPrefix [51]	Multiple AAA	1 D	-	R	Rejected
7	Accumulo	testSasl [52]	Assert Precond.	1 D	-	R	Rejected
8	CloudStack	testCreateSuccess [53]	Missing Assert	1 D	Rejected	-	Rejected
9	CloudStack	testAutoRenewal* [54]	Assert Precond.	4 M	PR-Invited	M	Merged
10	CloudStack	testCRUDacl [55]	Multiple AAA	1 D	PR-Invited	M	Merged
11	CloudStack	testHttpClient [56]	Suppress-Excep.	1 D	PR-Invited	M	Merged
12	CloudStack	searchForNonExist* [57]	Obscure Assert	1 D	PR-Invited	M	Merged
13	CloudStack	testCreateAndInfo [58]	Multiple Acts	1 D	PR-Invited	P	Pending
14	CloudStack	checkStrictMode* [59]	Obscure Assert	3 D	No Response	M	Merged
15	Archaius	validateApiWhenRemovingChild [60]	Assert Precond.	2 w	No Response	M	Merged
16	Archaius	testBasicReplacement [61]	Missing Assert	2 w	No Response	M	Merged
17	Archaius	PropertyTest.test [62]	Multiple AAA	2 w	No Response	M	Merged
18	Mssql-jdbc	testGetLocalDateTimeTypes [63]	Multiple AAA	1 D	-	R	Rejected
19	Mssql-jdbc	WarningTest.testWarnings [64]	Obscure Assert	2 D	Rejected	-	Rejected
20	Mssql-jdbc	RequestBoundaryMethodsTest.testWarnings [65]	Arrange & Quit	4 D	PR-Invited	M	Merged
21	Mssql-jdbc	SharedTimerTest.getTimer [66]	Missing Assert	4 D	PR-Invited	M	Merged
22	Gson	testWriterClosesIdempotent [67]	Missing Assert	1 D	PR-Invited	M	Merged
23	Gson	testWriteHtmlSafe [68]	Multiple AAA	1 D	PR-Invited	M	Merged
24	Druid	testIsTaskCurrent [69]	Multiple AAA	3 D	PR-Invited	P	Pending
25	Druid	testNormal [70]	Missing Assert	2 D	PR-Invited	M	Merged
26	Druid	testReadParquet* [71]	Arrange & Quit	1 D	PR-Invited	M	Merged
27	Druid	testPollOnDemand [72]	Assert Precond.	4 D	Pending	-	Pending

‘-’ mark means no IT or PR is submitted.

‘*’ mark means the test case’s name is abbreviated due to space limit.

We observed this general pattern occurring in many other ITs and PRs in Archaius—the community is active on PRs without relying on ITs. Thus, we adjusted our strategy for this project: ITs were still submitted first but were followed by PRs on the same day. These cases, addressing Assert Precondition, Missing Assert, and Multiple AAA, demonstrate that Archaius values and accepts our refactoring proposals to fix these issues.

IT-Invited-PR (Cases 4-5, 9-12, 20-23, 25-26): The majority of our accepted proposals followed a standard process where developers explicitly invited pull requests after reviewing our issue tickets, and then merged our PRs into their projects. Notably, some projects like CloudStack (Cases 9-12) showed consistent positive engagement across different types of AAA issues, responding with comments like “*could you propose a PR*” and “*CLGTM*” (*commit looks good to me*). In Accumulo (Case 5), the developers not only accepted our Obscure Assert improvements but also suggested extending similar changes to other test cases: “*It looks like a few other test cases in OpTimerTest could get the same treatment too...*” In this same comment, the developer also provided valuable suggestion regarding adding comments to further improve understandability of test cases “*...Also, if you are interested in further improving these tests, you could optionally add descriptive messages to the assert statements.*” These cases indicate that open source

developers do care about test case quality and is willing to spend efforts on fixing relevant issues.

b) Pending Proposals: Three proposals (11%) remain “Pending” as shown in Table III. For Case 13 (testCreateAndInfo from CloudStack), which addresses Multiple Acts issues by separating distinct testing scenarios, developers responded positively to our proposal. However, CI/CD infrastructure issues prevented us from submitting the pull request successfully. Despite our follow-up through the project’s mailing list, we received no further response. Similarly, for Case 24 (testIsTaskCurrent from Druid), which suggests restructuring a Multiple AAA case, developers accepted our proposal but requested modifications to reduce code duplication in the common setup code. While preparing these changes, we encountered CI/CD issues that prevented us from updating the pull request. Our attempts to reach developers through the project’s Slack channel remained unanswered. In the IT to Case 27 (testPollOnDemand from Druid), a developer commented “*I don’t know this JUnit feature well...if getDataSourcesSnapshot returns null, the statement Assume.assumeThat(dataSourcesSnapshot, is(null)); will force the current test to exit and mark it as something instead of failure?*”. Although we provided detailed explanations, “*JUnit Assume will let test skip if the condition is not satisfied...Only when getDataSourcesSnapshot not returns null, assume will just skip the test and leave an assume*

exception(which is weak than assert exception) in the test log...”, we received no further response.

Overall, while the final statuses of the cases remain pending, it is worth highlighting that Case 13 and Case 24 both received favorable responses to the proposal during the IT phase. The suspension of the PRs was due to factors beyond our control, such as issues with the CI/CD pipeline. In the case of Case 27, the developer showed interest in the Assume API, though there were no follow-ups. These cases collectively suggest that developers generally demonstrate interest, if not preference, in addressing AAA problems. However, we acknowledge that the pending statuses likely reflect the lower priority assigned to refactoring tasks in the development workflow.

c) Rejected Proposals: Five proposals (19%) were rejected, providing valuable insights into the practical constraints around AAA adoption. We analyze each rejection case and its implications in detail:

In Case 8 (testCreateSuccess from CloudStack), we proposed adding explicit assertions to verify successful creation operations. The developer rejected this proposal, stating *“I think you can leave this. If the creation is not successful it would have thrown an exception. On the other hand, this is one that is tested implicitly a lot of times as well.”* This response shows their reluctance to modify test cases that have worked consistently over time. It also shows a common practice where developers rely on exceptions for error detection rather than adding explicit assertions to positive tests.

In Cases 6 (testGetByPrefix) and 7 (testSasl) from Accumulo, we proposed refactoring to address Multiple AAA and Assert Precondition issues respectively. The developer acknowledged that our changes would increase simplicity and test granularity, but raised concerns about code maintainability. For Case 6, they noted: *“The increased simplicity in individual test cases would be offset by the additional complexity in managing more test methods. Given that this test has never failed, the effort required for review and approval seems unnecessary.”* In Case 7, they emphasized project-level priority, stating that *“this type of general improvement does not align with any coherent plan for Accumulo at this time.”* These rejections demonstrate how refactoring decisions in open-source projects balance theoretical improvements against practical factors like maintenance overhead, test reliability history, and alignment with project roadmaps. In projects where test cases have performed reliably, developers may prioritize stability and minimal maintenance burden over structural improvements.

In Case 18 (testGetLocalDateTimeTypes from Mssql-jdbc) we suggested separating different datetime conversion test scenarios into distinct test cases. The developers explained that the scenarios share required setup logic and test interrelated *date-time* conversion behaviors, with the second scenario depending on database state modifications from the first scenario. Their response noted: *“Separating tests can lead to an increased test time, and this is not something we want.”* Their response emphasizes that test execution efficiency is a critical consideration in testing, where setup operations like establishing connections and preparing test data can be time-intensive. Splitting a test case that shares setup context would require repeating these

costly operations multiple times. This rejection demonstrates that addressing Multiple AAA issues must be balanced against practical constraints, particularly in domains where test execution performance significantly impacts development workflows.

In Case 19 (testWarnings from Mssql-jdbc), we proposed using Hamcrest matchers to simplify complex assertion logic. The developer rejected this proposal citing dependency management concerns: *“I think we would rather keep this simple and not add any additional dependencies on Third-party frameworks.”* This feedback sheds light on the complex landscape of open-source contributions in corporate-driven OSCs, where considerations that prefer to minimize dependencies on external frameworks can be critical.

In summary, analysis of these five rejections (Cases 8, 6, 7, 18, and 19) reveals three key factors in developers’ evaluation of test code restructuring proposals. First, the *“Why fix it if it ain’t broken?”* mindset emerged unsurprisingly. For example, in CloudStack (Case 8) and Accumulo (Cases 6 and 7), developers were reluctant to modify test cases that were functioning without any obvious issues. This resistance to change stable code reflects a common perspective in software refactoring where the risks of modifying working systems are carefully weighed against potential benefits. While in Accumulo (Case 7), this was explicitly tied to project-level priorities, with developers noting that test improvements needed clear justification within the project roadmap. Second, Mssql-jdbc (Case 19) demonstrated adherence to dependency hygiene principles, cautious are taken on changes that would introduce additional external framework dependencies. Third, Mssql-jdbc (Case 18) highlighted that test execution efficiency can be a critical factor, particularly for testing where setup operations are time-intensive. The developers explicitly rejected breaking up test cases due to concerns about increased test execution time. These findings indicate that successful AAA-based refactoring requires not only technical merit but also careful consideration of project-specific constraints around code stability, dependency management, and performance requirements.

Of particular note, we did not observe significant differences in the responses to refactoring proposals between community-driven and corporate-driven projects. This suggests that the acceptance of refactoring proposals is primarily driven by the perceived value of the improvements and practical considerations, rather than the nature of the project community.

RQ4 Summary: We received 100% response rate and 78% responses are positive—we are invited to submit a PR or a PR is merged. This indicates that developers value the AAA problems and accept our resolution. Rejections also point to valuable lessons—return-on-investment is a key concern, which could consider the failure rate of test cases, performance of test codebase, and granularity of code change.

VII. REFACTORING PROPOSAL CASE STUDIES

In this section, we provide in-depth case studies of real-world refactoring proposals that we submitted to the projects,

```

1  @Test
2  public void testWriteHtmlSafe() {
3      // Arrange common test data
4      List<String> contents = Arrays.
5          asList("<", ">", "&", "=", "'"); //AR
6      // First AAA sequence: test with
7      // default HTML escaping
8      StringWriter writer = new
9          StringWriter(); //AR
10     new Gson().toJson(contents, new
11         JsonWriter(writer)); //AC
12     assertEquals(writer.toString(),
13         "['<','>','&','=','\"']"); //AS
14     // Second AAA sequence: test with
15     // HTML escaping disabled
16     writer = new StringWriter(); //AR
17     new GsonBuilder().
18         disableHtmlEscaping().create().
19         toJson(contents, type, new
20             JsonWriter(writer)); //AC
21     assertEquals(writer.toString(),
22         "['<','>','&','=','\"']"); //AS
23 }

```

Fig. 21. Multiple AAA accepted cases - original.

as outlined in Table III. Due to space constraints, we are unable to discuss all 27 cases in depth. Instead, for each AAA problem type, we have selected two cases wherever possible—one accepted case and one rejected case—for comprehensive illustration. We present code snippets illustrating the test cases before and after refactoring each case. We add code comments in the snippets for illustrating purposes. Based on the code snippets, we offer readers a clearer and more intuitive understanding of the benefits of refactoring, the reasons behind their acceptance, and the potential factors leading to rejection, all grounded in the technical details of the test cases. We believe these case studies could inform future research directions and guide the development of practical refactoring strategies to address AAA problems.

A. Multiple AAA

We present both an accepted case and a rejected case of Multiple AAA, offering a clear contrast between the concrete characteristics that may explain their differing outcomes.

a) Accepted Case: Consider Case 23 (*testWriteHtmlSafe* [68]) from Gson, as listed in Table III. The case is illustrated in Fig. 21. The test case violates the single responsibility principle by combining two distinct test scenarios in one test case: testing default HTML escaping (from line 5) and disabled HTML escaping (from line 9) behaviors. Each AAA sequence contains a complete arrange-act-assert flow, involving setting up the writer configuration, writing content, and verifying the expected output format. This setup creates a situation where the test case can fail due to either the default escaping configuration or the disabled escaping scenario.

Our proposed refactoring separated these scenarios into two distinct test cases, whose code snippet is shown in Fig. 22:

```

1  @Test
2  public void
3      testWriteHtmlSafeWithEscaping() {
4      //AR
5      List<String> contents = Arrays.
6          asList("<", ">", "&", "=", "'");
7      StringWriter writer = new
8          StringWriter();
9      //AC
10     new Gson().toJson(contents, new
11         JsonWriter(writer));
12     //AS
13     assertEquals(writer.toString(),
14         "['<','>','&','=','\"']");
15 }
16
17 @Test
18 public void
19     testWriteHtmlSafeWithoutEscaping() {
20     //AR
21     List<String> contents = Arrays.
22         asList("<", ">", "&", "=", "'");
23     StringWriter writer = new
24         StringWriter();
25     //AC
26     new GsonBuilder().
27         disableHtmlEscaping().create().
28         toJson(contents, new
29             JsonWriter(writer));
30     //AS
31     assertEquals(writer.toString(),
32         "['<','>','&','=','\"']");
33 }

```

Fig. 22. Multiple AAA accepted case - refactored.

testWriteHtmlSafeWithEscaping focusing on default HTML escaping behavior, and *testWriteHtmlSafeWithoutEscaping* testing the disabled HTML escaping scenario. The two separate test cases follow the “equivalence class partitioning (ECP) [39]” discussed in Section VI-B. This separation ensures each test case has a single responsibility and enhances test case quality in three ways. First, it improves failure diagnosis - if one fails, developers immediately know which HTML escaping configuration caused the issue by looking at the name of the test case. In comparison, if the original test case fails, it could be either of the configuration scenarios—requiring additional effort to analyze. Second, the separate test names *testWriteHtmlSafeWithEscaping* and *testWriteHtmlSafeWithoutEscaping* provide a clear “documentation” of the distinct test scenarios, making the test intentions more apparent for future maintenance. In contrast, the original test case name *testWriteHtmlSafe* provides less information on the tested scenarios. Third, the refactored structure allows independent verification of each escaping behavior. In the original version, if the first assertion fails, the second escaping scenario would not be executed at all, potentially hiding issues with the disabled escaping behavior.

b) Rejected Case: In contrast, the *Mssql-jdbc* project rejected our proposal to refactor a Multiple AAA case, *testGetLocalDateTimeTypes* [63], Case #18 listed in Table III. The test

```

1  @Test
2  public void testGetLocalDateTimeTypes()
3  {
4      // Complex shared setup for both
       scenarios
5      DateTime testValue =
        setupComplexDateTime();//AR
6      Connection conn = setupDBConnection
       ();//AR
7      // First AAA: test with default
       conversion
8      PreparedStatement stmt1 = conn.
        prepareStatement(QUERY);//AR
9      ResultSet rs1 = executeQuery(stmt1,
        testValue);//AC
10     verifyTimeZoneConversion(rs1,
        testValue);//AS
11     // Second AAA: test with modified
       conversion
12     conn.setTimeZoneConversion(
        IGNORE_OFFSET);//AR
13     PreparedStatement stmt2 = conn.
        prepareStatement(QUERY);//AR
14     ResultSet rs2 = executeQuery(stmt2,
        testValue);//AC
15     verifyTimeZoneConversion(rs2,
        testValue);//AS
16 }

```

Fig. 23. Multiple AAA rejected case - original.

case examines *datetime* conversion behavior in two scenarios: default timezone conversion (starting on line 6) and modified timezone conversion (starting on line 10), whose simplified code snippet is shown in Fig. 23. We proposed to split this test case into *testGetLocalDateTimeTypesWithDefault* for testing default timezone behavior and *testGetLocalDateTimeTypesWithIgnore* for testing modified timezone behavior. Each test case requires its own database connection setup and test data preparation:

A key difference between this rejected case and the previously accepted case is in the coupling between the two test scenarios. In the accepted case, the two scenarios were mostly independent, testing distinct behaviors that could be verified separately. Here, the scenarios are interdependent as they share the same setup in lines 4-5 of the original test case. Splitting these scenarios would require duplicating the database connection setup and test data preparation in both test cases, as seen in lines 4-6 and 15-18 of the refactored code snippet. This duplication increases test execution time, as now the same setup needs to be repeatedly executed in two test cases. As the developers rejected the case and noted “*separating tests can lead to an increased test time, and this is not something we want.*”

This comparison of the accepted and rejected cases points to future research focusing on the unique characteristics of Multiple AAA test cases, such as the size, complexity, and coupling among AAA blocks, that may impact whether it is beneficial and worthwhile to conduct the refactoring to split them.

```

1  @Test
2  public void
3  testGetLocalDateTimeTypesWithDefault
4  () {
5      //AR
6      DateTime testValue =
        setupComplexDateTime();
7      Connection conn = setupDBConnection
       ();
8      PreparedStatement stmt = conn.
        prepareStatement(QUERY);
9      //AC
10     ResultSet rs = executeQuery(stmt,
        testValue);
11     //AS
12     verifyTimeZoneConversion(rs,
        testValue);
13 }
14 @Test
15 public void
16 testGetLocalDateTimeTypesWithIgnore()
17 {
18     //AR
19     DateTime testValue =
        setupComplexDateTime();
20     Connection conn = setupDBConnection
       ();
21     conn.setTimeZoneConversion(
        IGNORE_OFFSET);
22     PreparedStatement stmt = conn.
        prepareStatement(QUERY);
23     //AC
24     ResultSet rs = executeQuery(stmt,
        testValue);
25     //AS
26     verifyTimeZoneConversion(rs,
        testValue);
27 }

```

Fig. 24. Multiple AAA rejected case - refactored.

B. Missing Assert

Our analysis of the accepted and rejected Missing Assert cases highlights different perspectives on verification strategies.

a) *Accepted Case:* Consider the *getTimer* [66] (case #21 in Table III) test case from project *Mssql-jdbc*. The original implementation is shown in Fig. 25, which lacked verification of the timer’s state as annotated in line 9. While this test case might still fail if the timer operations throw exceptions (implicit oracles), it cannot verify specific behavioral requirements such as proper timer cleanup in a multi-threaded environment [41].

As shown in Fig. 26, our proposed refactoring added verification of the *Timer* status (line 10 in the following code snippet), with two key benefits. First, it explicitly verifies the timer’s cleanup behavior after task execution - without this verification, resource leaks in the multi-threaded context could go undetected in the original implementation. Second, it documents the expected state transition through the assertion message - the original implementation provided no indication of the expected timer state, making it unclear to maintainers what constitutes

```

1  @Test
2  void getTimer() throws Exception {
3      //AR
4      ExecutorService executor =
5          createExecutor();
6      List<Future<?>> tasks = createTasks(
7          executor);
8      //AC
9      executeAllTasks(tasks);
10     executor.shutdown();
11     // No Assert - missing verification
12     // of timer state
13 }

```

Fig. 25. Missing assert accepted case - original.

```

1  @Test
2  public void testCreateSuccess() {
3      //AR
4      UserVmService service = mockService
5          ();
6      UserVm userVm = mockVm();
7      setupServiceMock(service, userVm);
8      //AC
9      service.upgradeVirtualMachine();
10     //AS
11     verify(service, times(1)).getValue();
12     verify(userVm, times(1)).getId();
13 }

```

Fig. 28. Missing assert rejected case - refactored.

```

1  @Test
2  void getTimer() throws Exception {
3      //AR
4      ExecutorService executor =
5          createExecutor();
6      List<Future<?>> tasks = createTasks(
7          executor);
8      //AC
9      executeAllTasks(tasks);
10     executor.shutdown();
11     //AS
12     assertEquals("Timer should stop after all
13         tasks complete",
14         Timer.isRunning(),
15         false);
16 }

```

Fig. 26. Missing assert accepted case - refactored.

```

1  @Test
2  public void testCreateFailure() {
3      //AR
4      UserVmService service = mockService
5          ();
6      setupServiceMockForFailure(service);
7      //AC
8      try {
9          service.upgradeVirtualMachine();
10     } catch (ServerApiException e) {
11         //AS
12         assertEquals("Failed to scale vm
13             ", e.getMessage());
14     }
15 }

```

Fig. 29. testCreateFailure case.

```

1  @Test
2  public void testCreateSuccess() {
3      //AR
4      UserVmService service = mockService
5          ();
6      UserVm userVm = mockVm();
7      setupServiceMock(service, userVm);
8      //AC
9      service.upgradeVirtualMachine();
10     // No explicit Assert, relies on
11     // exception for verification
12 }

```

Fig. 27. Missing assert rejected case - original.

correct behavior. The Mssql-jdbc developers accepted this improvement, recognizing the value of explicitly verifying the timer's cleanup behavior.

b) Rejected Case: In comparison, adding explicit assertions is not favored in all cases, which is exemplified by the rejected proposal for *testCreateSuccess* [53] (case #8 in Table III) in CloudStack. The simplified code snippet is shown in Fig. 27. As we manually annotated in line 9, no explicit assertion in the test case.

We proposed to verify the execution status using Mockito's APIs, ensuring that mocked functions are called as expected, as annotated in line 10 and line 11 in Fig. 28. As discussed in Section VI-D, the developers rejected our proposal and explained that “if the creation is not successful it would have thrown an exception...”, suggesting that they consider the absence of exceptions as sufficient verification for passing the test case. The developer also noted “this is one that is tested implicitly a lot of times”—indicating a reluctance to change a test case considered working properly and consistently.

However, our broader examination of the entire test class revealed that another test case, named *testCreateFailure* (shown in Fig. 29), complements this test case, *testCreateSuccess*, by explicitly focusing on testing the scenario of an exception and verifying the status of the exception with “CreateFailure”, as outlined in line 11. This explicit assertion underscores the advocacy of explicit assertion strategy, which is missing from the positive scenario of “CreateSuccess”.

This rejection reveals two practical factors that impact the verification strategy and related refactoring. First, developers are hesitant to modify test cases that they believe are working consistently and properly. Second, they sometimes rely on implicit verifications through exception handling—the absence of an exception indicates a pass. However, this approach has

```

1  @Test
2  public void testAutoRenewalDisabled() {
3      // AR
4      setupRenewalConfig(false);
5      setupCertificateMap();
6      //AS for precondition
7      Assert.assertTrue(certMap.size() ==
8                          1);
9      //AC
10     task.runInContext();
11     //AS
12     verify(manager).checkCertificate(
13         host);
14 }

```

Fig. 30. Assert precondition accepted case - original.

limitations - while it detects failures that trigger exceptions, it cannot identify incorrect results which may not necessarily raise an exception. Particularly, as the codebase evolves, the test case might need to verify specific conditions beyond just no exceptions. The missing assertion potentially makes the test case less reliable over time.

C. Assert Pre-Condition

The accepted and rejected cases for fixing Assert Preconditions reveal differing perspectives on handling test preconditions, impacted by project-specific factors.

a) *Accepted Case:* In test case *testAutoRenewalDisabled* [54] (case #9 in Table III) from CloudStack (shown in Fig. 30), line 7 used an assertion to verify if *certMap.size* equals 1 before executing the actual test target *task.runInContext()* in line 9. Test failures could be caused by an invalid test environment (*certMap.size != 1*) or by a defect in the actual functionality being tested (*task.runInContext*). In other words, this test case could fail due to reasons outside of its target—coupling the debugging of test dependency and invalid the pre-condition with that of the test target.

In the proposed refactoring, we replaced the assertion with JUnit's *Assume.assumeTrue()* method in line 6, illustrated in Fig. 31. While both *Assert.assertTrue()* and *Assume.assumeTrue()* verifies that *certMap.size* equals 1, they differ in how they handle the checking results. When *Assert.assertTrue()* fails, it marks the test as failed—treating it with the same level of “alarm” as a failure in the target function. In contrast, a failure in *Assume.assumeTrue()* marks the test as skipped and signals that the test environment was not properly configured. This approach allows developers to focus on critical failures in the target function while still providing a low-priority alert for misconfigured test environments.

In summary, the proposed refactoring improves test reliability and debugging efficiency. Before refactoring, the use of *Assert.assertTrue()* treated precondition failures with the same severity as functional defects. Thus incorrect environmental setup issues could lead to a false negative (i.e. failing test case with correct target function). This coupling could complicate debugging and misdirect developer attention. By replacing

```

1  @Test
2  public void testAutoRenewalDisabled() {
3      //AR
4      setupRenewalConfig(false);
5      setupCertificateMap();
6      Assume.assumeTrue(certMap.size() ==
7                        1); //Assume
8      //AC
9      task.runInContext();
10     //AS
11     verify(manager).checkCertificate(
12         host);
13 }

```

Fig. 31. Assert precondition accepted case - refactored.

```

1  @Test
2  public void testSasl() throws Exception
3  {
4      // AR
5      Properties clientProps = new
6          Properties();
7      clientProps.setProperty(
8          ClientProperty.SASL_ENABLED.
9          getKey(), "true");
10     // ... more property setup as AR...
11     EasyMock.replay(factory, context,
12         siteConfig); //AR
13
14     // AS used for precondition
15     assertEquals(ThriftServerType.SASL,
16         context.getThriftServerType());
17     // AC
18     SaslServerConnectionParams
19         saslParams = context.
20         getSaslParams();
21     // AS
22     assertEquals(new
23         SaslServerConnectionParams(conf,
24             token), saslParams);
25     assertEquals(username, saslParams.
26         getPrincipal());
27 }

```

Fig. 32. Assert precondition rejected case - original.

Assert.assertTrue() with *Assume.assumeTrue()*, the refactoring ensures that tests are skipped when preconditions are unmet, clearly signaling setup issues without conflating them with target function failures. This distinction helps developers prioritize critical issues while still addressing test environment reliability. The CloudStack developers accepted this improvement, recognizing that using JUnit's Assume API better communicates test intentions.

b) *Rejected Case:* In case #7 of Table III, the *Accumulo* developer rejected a similar refactoring of an assertion precondition in *testSasl* [52]. As shown in Fig. 32, the original test case used assertion to verify the server type before testing the SASL parameters (in line 10). We proposed replacing the precondition assertion with *assumeThat* (in line 10 of Fig. 33) to

```

1  @Test
2  public void testSasl() throws Exception
3  {
4      // AR
5      Properties clientProps = new
6      Properties();
7      clientProps.setProperty(
8      ClientProperty.SASL_ENABLED.
9      getKey(), "true");
10     // ... more property setup as AR...
11     EasyMock.replay(factory, context,
12     siteConfig); //AR
13
14     // Assume used for checking
15     precondition
16     assumeThat(ThriftServerType.SASL, is
17     (context.getThriftServerType()));
18     // AC
19     SaslServerConnectionParams
20     saslParams = context.
21     getSaslParams();
22     // AS
23     assertEquals(new
24     SaslServerConnectionParams(conf,
25     token), saslParams);
26     assertEquals(username, saslParams.
27     getPrincipal());
28 }

```

Fig. 33. Assert precondition rejected case - refactored.

distinguish between setup verification and actual test assertions. Despite addressing the same technical issue as the accepted case, the refactoring was rejected, as developers thought it misaligned with the project's priorities and not worth the review effort. They noted that such changes “do not seem to be part of any coherent plan to make substantive improvements to Accumulo.”

This comparison highlights that while Assert Precondition refactoring can improve test clarity, its adoption depends not only on technical merit but also on project-specific priorities, including resource constraints and alignment with broader goals.

D. Obscure Assert

We will present an accepted case of Obscure Assert, where the developer values this improvement, and a rejected case due to caution in taking on new dependencies. These again reflect that the acceptance or rejection of a refactoring proposal is beyond simply technical merits.

a) Accepted Case: In test case *searchForNonExistingLoadBalancer* (case #12 in Table III) from CloudStack, the developers use a boolean flag *notFound* combined with *try-catch* and *if-else* blocks to verify that retrieving a non-existent load balancer throws an exception, as shown through line 4 to line 13 in Fig. 34.

As shown in Fig. 35, we proposed to refactor this test case by removing the try-catch block and boolean flag, instead using JUnit's expected attribute of the *@Test* annotation to directly

```

1  @Test
2  public void
3  searchForNonExistingLoadBalancer() {
4      boolean notFound = false; //AR
5      try {
6          LoadBalancer rule = _appLbSvc.
7          getApplicationLoadBalancer(
8          nonExistingLbId); //AC
9          if (rule != null) {
10             notFound = false; //AS
11         }
12     } catch (
13     InvalidParameterValueException ex
14     ) {
15         notFound = true; //AS
16     }
17     assertTrue("Found non-existing load
18     balancer; no invalid parameter
19     value exception was thrown",
20     notFound); //AS
21 }

```

Fig. 34. Obscure assert accepted case - original.

```

1  @Test(expected=
2  InvalidParameterValueException.class)
3  //AS
4  public void
5  searchForNonExistingLoadBalancer() {
6      _appLbSvc.getApplicationLoadBalancer
7      (nonExistingLbId); //AC
8  }

```

Fig. 35. Obscure assert accepted case - refactored.

specify the expected exception, as shown in line 1 of the refactored code. This refactored case structure aligned with our Special AAA type Expected Exception introduced in section VI-A, which essentially follows the AAA pattern design.

This refactoring offers two main benefits. First, it makes the test case reads easier for developers—rather than tracing through the multiple steps and state changes in the original test, they can now understand the test's purpose at a glance through a single annotation. Second, it makes the test more consistent by using JUnit's standard way of handling exceptions, which is simple and familiar to most developers. The CloudStack developers accepted this improvement proposal.

b) Rejected Case: In contrast, case #19 in Table III, featuring the *testWarnings* method from the *Mssql-jdbc* project. Illustrated in Fig. 36, there is complex assertion logic involving nested loops and boolean flags. The test code iterates through a list of expected warning messages, checking if each warning contains any of the expected strings. The nested loops and boolean flag make the verification logic hard to follow (line 7 to line 19).

We proposed to refactor this case using Hamcrest matchers, which provide built-in methods for string matching. As illustrated in Fig. 37, the refactored version eliminates the nested

```

1  @Test
2  public void testWarnings() throws
    SQLException {
3      // ... AR Statements ...
4      //AC
5      warn = conn.getWarning();
6      //AS
7      for (int i = 0; i < 5; i++) {
8          boolean found = false;
9          List<String> list = Arrays.
              asList(infoArray);
10         for (String word : list) {
11             if (warn.toString().contains
                (word)) {
12                 found = true;
13                 break;
14             }
15         }
16         assertTrue(found, "Warning not
            found: " + warn.toString());
17         warn = warn.getNextWarning();
18         found = false;
19     }
20     //... rest of the Assertion
        Statements...
21 }

```

Fig. 36. Obscure assert rejected case - original.

```

1  @Test
2  public void testWarnings() throws
    SQLException {
3      // ... AR Statements ...
4      //AC
5      warn = conn.getWarning();
6      //AS
7      List<String> infoList = Arrays.
          asList(infoArray);
8      infoList.forEach(item ->{assertThat(
        warn.toString(),anyOf(
            containsString(item));warn =
            warn.getNextWarning();});
9      //... rest of the Assertion
        Statements...
10 }

```

Fig. 37. Obscure assert accepted case - refactored.

loops and boolean flag, replacing them with *forEach* and *assertThat* constructs, asserting the list without loop and if-esle conditions (lines 7 and 8).

Despite the potential benefits of eliminating the nested conditional logic with this case, the developers rejected this change, stating “we would rather keep this simple and not add any additional dependencies on third-party frameworks.” This rejection reflects a project-level preference for minimizing external dependencies, even when they might improve code readability. The team chose to maintain test code using only core JUnit assertions rather than introducing additional testing libraries. This practice, often referred to as dependency management hygiene,

```

1  @Test
2  public void testHttpClient(String
    templateUrl) {
3      try {
4          // AR
5          HttpGet getMethod = new HttpGet(
              templateUrl);
6          //AC
7          entity = client.execute(
              getMethod).getEntity();
8          //AS
9          entity.writeTo(output); //It is
              writing to output for
              verification later
10     } catch (ClientProtocolException e)
        {
11         e.printStackTrace();
12     } catch (IOException e) {
13         e.printStackTrace();
14     } finally {
15         try {
16             if (output != null) {
17                 output.close();
18             }
19         } catch (IOException e) {
20             e.printStackTrace();
21         }
22     }
23 }

```

Fig. 38. Suppressed exception accepted case - original.

aims to reduce potential vulnerabilities, simplify maintenance, and ensure long-term stability by limiting external dependencies [73]. This case illustrates how dependency management policies can influence testing practices, sometimes favoring simplicity and minimal dependencies over more expressive testing approaches that require additional libraries.

E. Suppressed Exception

We identified four instances of Suppressed Exceptions all from the CloudStack. Since they exhibit similar patterns and are all from the same project, we submitted one such proposal to avoid overwhelming the community. This proposal was accepted by the developers. This left us with no rejected case to discuss here.

a) *Accepted Case:* Consider CloudStack’s *testHttpClient* [56] (case #11 in Table III). As shown in Fig. 38, the original test case contains multiple catch blocks through lines 10 to 13 and lines 19 to 21 that only print stack traces without failing the test. Most notably, when the action *client.execute()* in line 7 throws a *ClientProtocolException*, the catch block in lines 10-11 suppresses this failure by only printing the stack trace, allowing the test to pass despite the action’s failure. The catch blocks for *IOException* in lines 12-13 and lines 19-20 also represent exception suppression, but these are relevant to writing output for assertion *entity.writeTo(output)* and resource cleanup *output.close()*, which are less critical as they are not relevant to the problems in the target function.

```

1  @Test
2  public void testHttpClient(String
    templateUrl)
3      throws ClientProtocolException,
4      IOException {
5      try {
6          //AR
7          HttpGet getMethod = new HttpGet(
            templateUrl);
8          //AC
9          entity = client.execute(
            getMethod).getEntity();
10         //AS
11         entity.writeTo(output);
12     } finally {
13         if (output != null) {
14             output.close();
15         }
16     }

```

Fig. 39. Suppressed exception accepted case - refactored.

We refactored this test case by removing all catch blocks, and threw the exceptions in the method signature, as shown in Fig. 39 line 3 where we specify throws *ClientProtocolException* and *IOException*. The key motivation is to allow exceptions from the action *client.execute()* (line 8) to propagate up to fail the test.

The refactored version still maintains proper resource cleanup in the final block (lines 11-15):

This refactoring provides two key benefits. Most importantly, it ensures the visibility of action target function failure. The original test case hides exceptions in console logs, which is easily overlooked. This is particularly problematic for the action's potential *ClientProtocolException*. Developers are likely to miss this exception without raising a test failure. The refactored version ensures that exceptions trigger actual test failures, making issues immediately visible. Secondly, it simplifies the code and potential debugging process. The original version required navigating multiple catch blocks with *printStackTrace* calls; while the refactored version eliminates these for cleaner test logic delegate exception handling to *Throw*. The CloudStack developers accepted this improvement, recognizing that proper exception propagation leads to more reliable test results and clearer failure diagnosis.

F. Arrange & Quit

All proposed refactorings for the Arrange & Quit problem were accepted, so we only discuss one accepted case here.

a) *Accepted Case*: In the test case *testSetException* [49] (case #4 in Table III) from Dubbo, shown in Fig. 40, developers check if a *Throwable* object is created null; if so, the test case *return* without executing the rest—shown in line 5 to 7. In other words, if the setup fails to create the *Throwable* (i.e. not null), the test silently returns without executing the core test logic—masking potential errors in the target function on line 8.

```

1  @Test
2  public void testSetException() {
3      //AR
4      Throwable throwable =
5          buildEmptyStackTraceException();
6      if (throwable == null) {
7          return;
8      }
9      AppResponse response = new
10         AppResponse();
11      //AC
12      response.setException(throwable);
13      //AS
14      assertThat(response.getException(),
15         is(throwable));
16  }

```

Fig. 40. Arrange & quit accepted case - original.

```

1  @Test
2  public void testSetException() {
3      //AR
4      Throwable throwable =
5          buildEmptyStackTraceException();
6      assumeNotNull(throwable);
7      AppResponse response = new
8         AppResponse();
9      //AC
10     response.setException(throwable);
11     //AS
12     assertThat(response.getException(),
13         is(throwable));
14  }

```

Fig. 41. Arrange & quit accepted case - refactoring.

We refactored this test case by replacing the if-return statement with JUnit's *assumeNotNull* method in line 5, as shown in Fig. 41 for proper test pre-condition checking.

This refactoring offers two key benefits. First, it prevents the test from silently quitting when the precondition is not met. Instead of returning without executing the core logic, it logs the issue with pre-condition and makes it skipped. Second, replacing the if-return by Assume API makes the intention of checking the precondition formal and explicit. This improves the clarity and correctness of the test's structure. The Dubbo developers accepted this improvement.

G. Multiple Acts

Here we present the only proposal to fix Multiple Acts for CloudStack. Although this case is still in pending status (the merge of PR halted due to factors outside of our control), we still consider it as a case standing for “accepted” due to the favoring comments we received and the invitation to submit PR.

a) *Pending Case*: As shown in Fig. 42, the original test case, *testCreateAndInfo* [58] (case #13 in Table III), contains

```

1  @Test
2  public void testCreateAndInfo() {
3      //AR
4      QemuImgFile file = new QemuImgFile(
5          filename, size);
6      QemuImg qemu = new QemuImg();
7      //AC
8      qemu.create(file); //First AC
9      Map info = qemu.info(file); //Second
10     //AS
11     assertTrue(file.exists());
12     assertEquals(size, info.get("size"));
13 }

```

Fig. 42. Multiple acts pending case - original.

two Actions - creating a file in line 7 and retrieving its information in line 8. This design violates the single responsibility principle and makes failure diagnosis ambiguous, as a test failure could stem from either action.

As shown in Fig. 43, We refactored this test case by separating the two operations into individual test cases *testCreate* and *testInfo*. While *qemu.create(file)* serves as an action in *testCreate* as shown in line 7, it becomes part of the arrangement in *testInfo* since *qemu.info(file)* requires a pre-existing file, as shown in line 17. We use JUnit's *assumeTrue* in line 18 to validate that file creation succeeded before testing the info operation. This refactoring improves failure diagnosis. That is, when each test has a specific focus, the cause of a failure is more localized. If the *testCreate* fails, it's clear that something is wrong with the file creation. If the *testInfo* fails when *testCreate* passes, the cause is related to how info is parsed or the command's output, rather than file creation. Instead, the original test case combines the status of both *Create* and *Info* into only one pass or fail. A failure only tells you that something in that chain of operations (Create and/or Info) is broken.

Overall, this refactoring offers better "separation of concern" in two progressive levels. On the first level, this is reflected in the naming of the test cases. The original test case name *testCreateAndInfo* imply both *Create* and *Info* were tested in this one single test case. After the refactoring, the test case names were split into *testCreate* and *testInfo*, allowing developers to view these as two separate targets while they navigate through test cases. On the next level, this provides better separation in the failure reporting and respective debugging. Before refactoring, an error in either *Create* or *Info* could lead to a failure in the test case—requiring the developers to manually inspect and debug which target function caused the failure. After the refactoring, errors in *Create* and *Info* are isolated to their respective test cases, allowing developers to quickly focus on the core problem raised by the specific test case. Tests that try to do too much (like verifying both creation and retrieval in one go) are inherently more complex and can fail for multiple, unrelated reasons. Simpler tests are easier to read, maintain, and debug.

```

1  @Test
2  public void testCreate() {
3      //AR
4      QemuImgFile file = new QemuImgFile(
5          filename, size);
6      QemuImg qemu = new QemuImg();
7      //AC
8      qemu.create(file);
9      //AS
10     assertTrue(file.exists());
11 }
12
13 @Test
14 public void testInfo() {
15     //AR
16     QemuImgFile file = setupTestFile();
17     QemuImg qemu = new QemuImg();
18     qemu.create(file);
19     //AC
20     assumeTrue("File must exist for info
21     test", file.exists());
22     //AS
23     Map info = qemu.info(file);
24     assertEquals(size, info.get("size"));
25 }

```

Fig. 43. Multiple acts pending case - refactored.

VIII. DISCUSSION

A. Threats to Validity and Limitation

First, we acknowledge that focusing on only seven projects of a specific language (Java) and testing framework (JUnit) may potentially compromise the generalizability of our study. Different programming languages and testing frameworks may influence testing practices in unique ways. For instance, alternative frameworks like TestNG or Spock within Java, or testing tools in other languages such as Python's *unittest* or JavaScript's Jest, might encourage different test case structures or syntax. However, we believe our focus is justifiable because Java and JUnit are widely adopted, making our findings pertinent to a significant portion of the developer community. Moreover, the AAA pattern is fundamentally language- and framework-agnostic, concentrating on the logical structure of test cases rather than specific language or framework features. Therefore, the experience gained from this study could also provide valuable insights to practitioners working on projects of other languages and frameworks.

We also admit that our findings based on only seven projects may not directly apply to other projects. As mitigation, we carefully selected the seven projects that encompass a diverse range of domains, scales, activity levels, and commitments to testing, representing both Apache and corporate-driven open-source ecosystems. Specifically, the projects cover Data Management (Accumulo, Druid, Mssql-jdbc), Web Technologies (Dubbo), Networking & IoT (Cloudstack, Archaius), and Development & Infrastructure (Gson). These projects also vary widely in scale

(171 to 7,816 source files), age (702 to 32,250 commits), and community engagement (29 to 507 contributors). This diversity aims to provide insights that may extend beyond the specific projects studied. This study serves as a foundational step toward broader investigations, and future work will aim to expand the sample size, include projects of other programming languages, and analyze additional testing frameworks to enhance the generalizability of our findings.

Another limitation involves our test culture survey. Despite additional recruitment efforts, including sending reminders and expanding the participant pool from the top 50% to 60% for Apache projects and from 95% to 96% for corporate projects (adding 106 potential participants), we received no new responses beyond the initial 11. This reflects the inherent challenges of conducting surveys with open-source developers. While the low response rate potentially limits the generalizability of our findings, a study by Leslie suggests that response rates do not always directly affect the validity of results, particularly when respondents are knowledgeable and closely aligned with the topic of inquiry [74]. In our case, the participants' expertise—10 out of 11 with over five years of professional experience—and their representation of diverse domains, including development and testing (73%), web technology (9%), and infrastructure, networking, and IoT (9%), provide significant depth and value to the findings, mitigating concerns related to the limited sample size. Thus, while the response rate remains low, we believe the quality of the responses is noteworthy.

Our identification and summarization of AAA problems also have inherent limitations. First, the seven types of AAA problems we identified may not be exhaustive. Other AAA-related issues might exist that were either absent from our dataset or not recognized during our analysis. Second, the classification of certain AAA problems, particularly "Obscure Assert," involves an element of subjectivity. Although we used criteria such as cyclomatic complexity greater than 2 to identify Obscure Assert, determining whether assertion logic is truly "obscure" can depend on the experience and background of individual developers. This subjectivity is a natural aspect of any empirical research that seeks to interpret complex phenomena through qualitative analysis. Despite using iterative group discussions to refine and standardize our identification criteria, the potential for bias or differing interpretations cannot be eliminated.

We also acknowledge the limitation of our study in the relatively small sample size of 27 issues/pull requests. While our selection strategy was designed to balance comprehensive coverage of the seven AAA problem types across projects and avoid overwhelming any single project community with multiple similar proposals, this approach inherently limits the number of cases we could submit for analysis. However, this trade-off was necessary to ensure thoughtful engagement from contributors while enabling us to observe a range of developer responses across diverse contexts. The small sample size introduces potential threats to the validity of our findings. For instance, the feedback we received may not fully capture the breadth of developer perspectives across different projects or contexts. Moreover, the acceptance or rejection of proposals may have been influenced by factors unique to the specific projects or

individuals involved, such as familiarity with AAA principles, time constraints, or project-specific guidelines. These factors could limit the generalizability of our findings beyond the selected projects and cases. To address these limitations, future research will extend the study to a larger set of projects, increasing the number and diversity of analyzed issues and pull requests. Moreover, future investigations will focus on a more in-depth analysis of how the characteristics of individual issues—such as complexity, criticality, and alignment with project goals—impact the acceptance of refactoring proposals.

Finally, the study results may reflect bias from the authors' experience. We worked to mitigate this through team collaboration and weekly discussions. The significant effort in collecting and analyzing developer feedback helps ensure our findings reflect real-world problems in test case development.

B. Research Road-Map and Automation Plan

To maximize the practical impact of our work, we aim to develop a comprehensive suite of automation tools for improving software testing practices based on the AAA pattern. As part of this effort, we have already created an AAA Tagging Technique powered by machine learning, which automatically identifies Arrange, Act, and Assert statements in test cases [9]. Complementing this empirical study, we have also conducted case studies of the AAA tagging technique and examination of AAA practices in commercial projects [38]. Our prior efforts lay the groundwork for automated detection and resolution of AAA-related issues. Building on these foundations, we are currently developing an AAA problem Detection Tool to automatically identify the AAA problems. Looking ahead, we plan to create an AAA Refactoring Tool that will suggest and implement improvements to test cases with identified AAA problems. Together, this toolchain—encompassing AAA tagging, issue detection, and refactoring—has the potential to significantly enhance software testing practices by providing actionable feedback to developers, improving test case quality, and reducing the manual effort required for test case analysis and refactoring. To ensure the generalizability and practical utility of these tools, we plan to validate them across diverse projects. This holistic approach aims to not only address AAA problems but also advance the state of automated software testing.

IX. RELATED WORK

Test Smells are a special group of code smells in test code. Numerous prior studies have examined different types of test smells [19], [20], [24], [27], [28], [29], [36], [75], [76], [77], [78], [79], [80], [81]. Garousi and Küçük [80] presented the most comprehensive catalog of 166 test smells, many of which came from grey literature and lacks verification of the value. These smells were grouped into six groups, focusing on 1) execution/behavior, 2) semantic/logic, 3) design, 4) test steps, 5) code, and 6) dependencies. However, none of these aspects concerns the AAA structure. Our study compares the AAA problems with 21 well-studied test smells, detectable by Jnose [25]. We notice that 15 smells are orthogonal to our AAA analysis,

grouped into three types based on their key focus. First, three smells, *Dependent Test*, *Lazy Test*, and *General Fixture*, focus on the relationship or interactions across different test cases. While the AAA analysis focuses on the internal structure of a test case. Secondly, there are 10 smells focusing on very specific code features, such as using `@Ignore` annotation, invoking `Thread.sleep()` or `toString`. The occurrences of these smells are independent of our AAA analysis. Finally, the *Assert Roulette* and *Redundant Assertions* sound relevant to our AAA analysis with a focus on *assert*. However, they check specific features inside an *assert*, orthogonal to our AAA analysis.

Refactoring: This study is also relevant to software refactoring, which aims at improving the design and quality of code without altering its external behavior. In particular, van Deursen et. al. suggest that *test refactoring* “do not add or remove test cases” and improve its quality [82]. We would like to clarify that fixing the AAA problems will enhance the test case behavior. More specifically, *Missing Assert* should be fixed by adding assertions to verify expected outcomes - empirical evidence from Shrestha’s study demonstrates that while implicit oracles catch only 11% of faults, explicit assertions detect 53% additional faults [41]; *Assert precondition* is fixed that the precondition should not fail the test case; *Arrange & Quit* is fixed to use assume such that the action will always get executed; *Suppressed Exception* is fixed to expose the exception; *Multiple Acts* and *Multiple AAA* both should be split into multiple test cases to raise failures separately. Meanwhile, all the restructuring may also benefit the overall design of the test cases, similar to the objective of refactoring.

Linear Fixture and Assertion Structure: Ma’ayan’s study [8] also provided insights into the usage of the AAA pattern in real-world JUnit tests. Essentially, the study uses algebraic representations to classify test case structures as F and A. F represents “fixtures” (setup commands), which are non-assertion commands like arranging test preconditions or acting on objects. A represents “assertions” that verify outcomes. It also excludes test cases that contain any conditional logic, thus they refer to their study as focusing on linear Fixture and Assertion Structure. This method leaves out the complexity of real-world test structures, thus providing a limited analysis of the full AAA pattern.

This limitation affects the analysis in three fundamental ways: **1)** First, as stated in their methodology, their analysis excludes all non-linear test cases - specifically those containing conditional logic or nested method calls. This exclusion removes a substantial portion of real-world test cases from consideration, limiting their findings to only simple, linear test structures. **2)** Second, by combining arrangement and action statements into a unified fixture component, their analysis cannot identify cases where a test contains multiple complete AAA sequences with grouped assertions. For example, a test case might perform multiple setup and execution steps before verifying all outcomes together. Such patterns often indicate structural issues that require refactoring. **3)** Third, their linear analysis cannot accurately process test cases that encapsulate assertions within helper methods defined in the test class. Without expanding these method calls, the analysis incorrectly classifies these

cases as missing assertions, producing false positives in their detection of test case issues. Our analysis of the complete AAA pattern reveals findings different from those of Ma’ayan’s study. While their work focused on analyzing linear test structures, our research identified seven distinct types of AAA-related issues by considering complex branching structures and nested method calls.

X. CONCLUSION

We conducted an empirical study analyzing the AAA structure of 735 real-world unit test cases from seven JAVA open source projects. We found that 77% of test cases indeed follow the AAA structure—confirming that it is well practiced. We discovered three recurring patterns in test cases that deviate from the AAA structure and four design issues that reside inside of the “A” blocks. We compared these seven AAA problems with classic test smells, underscoring the strength of AAA problems in focusing on root causes of smell symptoms, and guiding necessary and feasible improvements in test cases. The 27 representative proposals for fixing these problems are well-received by developers—with a 100% response rate and 78% responses favoring the improvements. From the rejections, we learned that developers are concerned about the return on investment, impacted by the change-proneness, failure-proneness, and granularity of change. This work paves the road for our future research in 1) auto-detection of the AAA problems, and 2) auto-/semi-auto fixing of the AAA problems.

REFERENCES

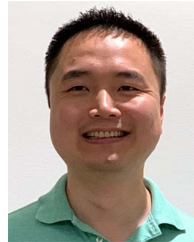
- [1] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. New York, NY, USA: Simon and Schuster, 2020.
- [2] P. Gomes, “Unit Testing and the Arrange, Act and Assert (AAA) Pattern,” 2017. Accessed: Jul. 1, 2024. [Online]. Available: <https://medium.com/@pjbgrf/title-testing-code-ocd-and-the-aaa-pattern-df453975ab80>
- [3] M. Publications, “Making better unit tests: Part 1, the AAA pattern,” 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://freecontent.manning.com/making-better-unit-tests-part-1-the-aaa-pattern/>
- [4] T. Eason, “The arrange, act, and assert (AAA) pattern: A functional approach,” 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://developers.mews.com/aaa-pattern-a-functional-approach/>
- [5] S. Gulati and R. Sharma, “Java unit testing with junit 5,” *Java Unit Testing with JUnit 5*, New York, NY, USA: Springer, 2017.
- [6] R. C. Martin, J. Newkirk, and R. S. Koss, *Agile Software Development: Principles, Patterns, and Practices*, vol. 2. Upper Saddle River, NJ, USA: Prentice-Hall, 2003.
- [7] R. I. Masel, *Principles of Adsorption and Reaction on Solid Surfaces*, vol. 3. Hoboken, NJ, USA: Wiley, 1996.
- [8] D. D. Ma’ayan, “The quality of junit tests: An empirical study report,” in *Proc. IEEE/ACM 1st Int. Workshop Softw. Qualities Dependencies (SQUADE)*, 2018, pp. 33–36.
- [9] C. Wei et al., “Automatically tagging the “AAA” pattern in unit test cases using machine learning models,” *IEEE Trans. Softw. Eng.*, vol. 49, no. 5, pp. 3305–3324, May 2023.
- [10] H. Zhu, P. A. Hall, and J. H. May, “Software unit test coverage and adequacy,” *ACM Comput. Surveys*, vol. 29, no. 4, pp. 366–427, 1997.
- [11] P. Gomes, “Unit testing and the arrange, act and assert (AAA) pattern,” *Retrieved Specific*, vol. 9, 2017, Art. no. 2017.
- [12] T. Eason, “The arrange, act, and assert (AAA) pattern: A functional approach,” *Mews*, Nov. 2020. Accessed: Nov. 23, 2020. [Online]. Available: <https://developers.mews.com/aaa-pattern-a-functional-approach/>
- [13] B. Wake, “3a – arrange, act, assert.” 2011. Accessed: Jul. 1, 2024. [Online]. Available: <https://xp123.com/3a-arrange-act-assert/>

- [14] Pytest, "Anatomy of a test function." 2024. Accessed: Jul. 1, 2024. [Online]. Available: <https://docs.pytest.org/en/stable/explanation/anatomy.html>
- [15] React, "Act." 2024. Accessed: Jul. 1, 2024. [Online]. Available: <https://react.dev/reference/react/act>
- [16] M. Fowler, "Given when then." 2024. Accessed: July 1st, 2024. [Online]. Available: <https://martinfowler.com/bliki/GivenWhenThen.html>
- [17] R. Modi, "Terraform unit testing," in *Deep-Dive Terraform on Azure*. New York, NY, USA: Springer, 2021, pp. 191–220.
- [18] S. Gulati, R. Sharma, S. Gulati, and R. Sharma, "Building software the correct way," *Java Unit Testing with JUnit 5: Test Driven Develop.* JUnit 5, pp. 1–23, Nov. 2017.
- [19] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? An empirical study," *Empirical Softw. Eng.*, vol. 20, no. 4, pp. 1052–1094, 2015.
- [20] M. Tufano et al., "An empirical investigation into the nature of test smells," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 4–15.
- [21] S. Reichhart, T. Gırba, and S. Ducasse, "Rule-based assessment of test quality," *J. Object Technol.*, vol. 6, no. 9, pp. 231–251, 2007.
- [22] M. Breugelmans and B. Van Rompaey, "TestQ: Exploring structural and maintenance characteristics of unit test suites," in *Proc. WASDeTT-1: 1st Int. Workshop Adv. Softw. Develop. Tools Techn.*, CiteSeer, 2008, p. 11.
- [23] M. Greiler, A. Zaidman, A. Van Deursen, and M.-A. Storey, "Strategies for avoiding text fixture smells during software evolution," in *Proc. 10th Work. Conf. Mining Softw. Repositories (MSR)*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 387–396.
- [24] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "tsDetect: An open source test smells detection tool," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1650–1654.
- [25] T. Virginio et al., "JNose: Java test smell detector," in *Proc. XXXIV Brazilian Symp. Softw. Eng.*, 2020, pp. 564–569.
- [26] W. Aljedaani et al., "Test smell detection tools: A systematic mapping study," in *Proc. Eval. Assess. Softw. Eng.*, 2021, pp. 170–180.
- [27] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 1–12.
- [28] A. Qusef, M. O. Elish, and D. Binkley, "An exploratory study of the relationship between software test smells and fault-proneness," *IEEE Access*, vol. 7, pp. 139526–139536, 2019.
- [29] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Piscataway, NJ, USA: IEEE Press, 2012, pp. 56–65.
- [30] L. Xiao et al., "An empirical study on the usage of mocking frameworks in apache software foundation," *Empirical Softw. Eng.*, vol. 29, no. 2, p. 39, 2024.
- [31] T. Yamane, *Statistics: An Introductory Analysis*. New York, NY, USA: Harper & Row New York, 1973.
- [32] J. Chen et al., "Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering," *J. Syst. Softw.*, vol. 135, pp. 107–125, Jan. 2018.
- [33] CloudStack, "testReleaseDedicatedGuestVlanRange." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/cloudstack/blob/b2ffa3efa58a1569dbaa8a34f723756926e22820/server/src/test/java/com/cloud/network/DedicatedGuestVlanRangesTest.java#L169-L183>
- [34] Accumulo, "test2." Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/accumulo/blob/0b34e30c34b7c31ebb84ca6b3b686c1b3e7b19af/core/src/test/java/org/apache/accumulo/core/iterators/system/MultiIteratorTest.java#L153-L175>
- [35] H. C. Kraemer, "Extension of the kappa coefficient," *Biometrics*, vol. 36, no. 2, pp. 207–216, 1980.
- [36] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *IEEE Trans. Softw. Eng.*, vol. 40, no. 11, pp. 1100–1125, Nov. 2014.
- [37] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: A critical review and guidelines," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 120–131.
- [38] C. Wei, L. Xiao, and S. Wong, "Analyzing AAA pattern in unit testing at envestnet: Tool enhancement and developer practices," in *Proc. 47th IEEE/ACM Int. Conf. Softw. Eng.*, under Review.
- [39] A. Bhat and S. Quadri, "Equivalence class partitioning and boundary value analysis-a review," in *Proc. 2nd Int. Conf. Comput. Sustain. Global Develop. (INDIACom)*, Piscataway, NJ, USA: IEEE Press, 2015, pp. 1557–1562.
- [40] E. Soares, M. Ribeiro, R. Gheyi, G. Amaral, and A. M. Santos, "Refactoring test smells with junit 5: Why should developers keep up-to-date," *IEEE Trans. Softw. Eng.*, vol. 49, no. 3, pp. 1152–1170, Mar. 2023.
- [41] K. Shrestha and M. J. Rutherford, "An empirical evaluation of assertions as oracles," in *Proc. 4th IEEE Int. Conf. Softw. Testing, Verification Validation*, Piscataway, NJ, USA: IEEE Press, 2011, pp. 110–119.
- [42] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante, "Cyclomatic complexity," *IEEE Softw.*, vol. 33, no. 6, pp. 27–29, Nov./Dec. 2016.
- [43] Oracle, "The try-with-resources Statement." 2022. Accessed: Jul. 1, 2024. [Online]. Available: <https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>
- [44] CloudStack, "SnapshotTest.deleteSnapshot." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/cloudstack/blob/b2ffa3efa58a1569dbaa8a34f723756926e22820/engine/storage/integration-test/src/test/java/org/apache/cloudstack/storage/test/SnapshotTest.java#L417-L437>
- [45] JUnit, "org.junit.rules.ExpectedException." Accessed: Jul. 1, 2024. [Online]. Available: <https://junit.org/junit4/javadoc/4.12/org/junit/rules/ExpectedException.html>
- [46] Dubbo, "testAll." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/dubbo/issues/10214>
- [47] Dubbo, "testClear." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/dubbo/issues/10212>
- [48] Dubbo, "testsubscription." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/dubbo/issues/10213>
- [49] Dubbo, "testSetExceptionWithEmptyStackTraceException." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/dubbo/issues/10487>
- [50] Accumulo, "verifyExceptionCallingStartWhenRunning." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/accumulo/issues/2886#issue-1344425211>
- [51] Accumulo, "testGetByPrefix." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/accumulo/pull/2553>
- [52] Accumulo, "testSasl." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/Codegass/accumulo/pull/2>
- [53] CloudStack, "testCreateSuccess." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/cloudstack/issues/6666>
- [54] CloudStack, "testAutoRenewalDisabled." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/cloudstack/issues/6203>
- [55] CloudStack, "testCRUDacl." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/cloudstack/issues/6665>
- [56] CloudStack, "testHttpClient." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/cloudstack/issues/6673>
- [57] CloudStack, "searchForNonExistingLoadBalancer." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/cloudstack/issues/6660>
- [58] CloudStack, "testCreateAndInfo." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/cloudstack/issues/6674>
- [59] CloudStack, "checkStrictModeWithCurrentAccountVmsPresent." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/cloudstack/pull/6676>
- [60] Archaius, "DefaultLayeredConfigTest.validateApiWhenRemovingChild." 2024. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/Netflix/archaius/issues/712>
- [61] Archaius, "ConfigManagerTest.testBasicReplacement." 2024. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/Netflix/archaius/issues/710>
- [62] Archaius, "PropertyTest.test." 2024. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/Netflix/archaius/issues/714>
- [63] Mssql-jdbc, "DataTypesTest.testGetLocalDateTimeTypes." 2024. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/microsoft/mssql-jdbc/pull/2360>
- [64] Mssql-jdbc, "WarningTest.testWarnings." 2024. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/microsoft/mssql-jdbc/issues/2358>
- [65] Mssql-jdbc, "RequestBoundaryMethodsTest.testWarnings." 2024. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/microsoft/mssql-jdbc/issues/2359>
- [66] Mssql-jdbc, "SharedTimerTest.getTimer." 2024. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/microsoft/mssql-jdbc/issues/2357>
- [67] Gson, "JsonWriterTest.testWriterCloseIsIdempotent." 2024. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/google/gson/issues/2650>

- [68] Gson, "MixedStreamTest.testWriteHtmlSafe." 2024. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/google/gson/issues/2651>
- [69] Druid, "testIsTaskCurrent." Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/druid/issues/12398>
- [70] Druid, "testNormal." Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/druid/issues/12399>
- [71] Druid, "testReadParquetDecimal32." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/druid/issues/12926>
- [72] Druid, "testPollOnDemand." 2020. Accessed: Jul. 1, 2024. [Online]. Available: <https://github.com/apache/druid/issues/12436>
- [73] T. Gustavsson, "Managing the open source dependency," *Computer*, vol. 53, no. 2, pp. 83–87, 2020.
- [74] L. L. Leslie, "Are high response rates essential to valid surveys?" *Social Sci. Res.*, vol. 1, no. 3, pp. 323–334, 1972.
- [75] A. Peruma, K. S. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the distribution of test smells in open source android applications: An exploratory study," 2019.
- [76] T. Virgínio, R. Santana, L. A. Martins, L. R. Soares, H. Costa, and I. Machado, "On the influence of test smells on test coverage," in *Proc. XXXIII Brazilian Symp. Softw. Eng.*, 2019, pp. 467–471.
- [77] V. Garousi, B. Kucuk, and M. Felderer, "What we know about smells in software test code," *IEEE Softw.*, vol. 36, no. 3, pp. 61–73, May/Jun. 2018.
- [78] J. De Bleser, D. Di Nucci, and C. De Roover, "Assessing diffusion and perception of test smells in scala projects," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 457–467.
- [79] R. O. Spínola, N. Zazworka, A. Vetro, F. Shull, and C. Seaman, "Understanding automated and human-based technical debt identification approaches—a two-phase study," *J. Brazilian Comput. Soc.*, vol. 25, no. 1, pp. 1–21, 2019.
- [80] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *J. Syst. Softw.*, vol. 138, pp. 52–81, Apr. 2018.
- [81] N. S. Junior, L. Rocha, L. A. Martins, and I. Machado, "A survey on test practitioners' awareness of test smells," 2020, *arXiv:2003.05613*.
- [82] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," *Rep. - Softw. Eng.*, pp. 1–6, Jul. 2001.



Tingting Yu (Member, IEEE) received the Ph.D. degree in computer science from the University of Nebraska-Lincoln. She is an Associate Professor with the Computer Science and Engineering Department, University of Connecticut. Her research focuses on software engineering, with a focus on developing methods and tools for improving the reliability and security of complex software systems, testing for concurrent software, regression testing, and performance testing.



Sunny Wong (Member, IEEE) received the Ph.D. degree in computer science from Drexel University. He is a Senior Software Architect with Envestnet, with over a decade of experience in the aerospace/defense, healthcare, and finance industries. He was named Young Engineer of the Year in 2019 by the IEEE Philadelphia Section. His research interests include software architecture and design modeling, tools support for improving developer productivity, and applications of AI techniques to software development.



Abigail Clune received the B.S. and M.S. degrees in computer engineering from Drexel University, in 2021. She is a Software Engineer with Ansys Government Initiatives. Her research interests include software architecture, software usability and reusability, and Agile software development.



Chenhao Wei (Member, IEEE) is currently working toward the Ph.D. degree with the Department of Systems and Enterprises, Stevens Institute of Technology, advised by Dr. Lu Xiao. His research interests lie in software testing and artificial intelligence for software engineering (AI4SE). He received the Stevens Excellence Doctoral Fellowship (2024–2025) and the third-place prize at the ASE Student Research Competition in 2022.



Lu Xiao (Member, IEEE) received the bachelor's degree in computer science from Beijing University of Posts and Telecommunications, in 2009, and the Ph.D. degree in computer science from Drexel University, in 2016, advised by Dr. Yuanfang Cai. She is an Associate Professor with the Department of Systems and Enterprises, Stevens Institute of Technology. Her research interests lie in the broad area of software engineering, particularly in software architecture, software economics, cost estimation, and software ecosystems. She is an awardee of NSF

CAREER project in 2021. She has published her work in different conferences and journals, including IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, ICSE, FSE, and ICSA, etc. She received the first-place prize at the ACM Student Research Competition in 2015.