What's DAT Smell? Untangling and Weaving the Disjoint Assertion Tangle Test Smell

Monil Narang
University of California, Irvine
Irvine, California, USA
moniln@uci.edu

Hang Du
University of California, Irvine
Irvine, California, USA
hdu5@uci.edu

James A. Jones
University of California, Irvine
Irvine, California, USA
jajones@uci.edu

I. ABSTRACT

Abstract*—In this work, we characterize a novel test-code smell—Disjoint Assertion Tangle (DAT)—which occurs when a test method verifies multiple, logically unrelated behaviors that can be separated. We propose a program analysis-based approach that automatically detects DAT and refactors DAT tests into separate focused test methods. We implemented this approach as a tool called U2W. By separating unrelated testing logic, U2W enhances readability, maintainability, and fault localization, while exposing hidden test clones and duplicated code. It then seizes these opportunities by converting structurally similar tests into compact, parameterized unit tests (PUTs), reducing redundancy and enabling more scalable, extensible test designs. To evaluate our approach and tool, we conducted a number of evaluations: (1) a large-scale, quantitative study to study the prevalence of the test smell and the effects of their refactoring, (2) a user survey to assess developers' opinions and preferences of the unrefactored and refactored test code, and (3) pull requests that were issued to original project maintainers to assess the acceptability of our refactorings. Our quantitative study was conducted on 42,334 tests across 49 open-source projects. We found the DAT smell in 95.9% of the subject projects, affecting an average of 8.59% of analyzed tests. In total, we identified and refactored 3,638 smelly tests, untangled them into 31,837 test-execution logics, and then weaved 14,343 of them into 1,713 extensible PUT methods. These refactorings reduced the executable test-code lines in smelly tests by an average of 36.33%. Our user survey involving 49 industrial and academic participants demonstrated strong preference for our refactored test cases over their original, unrefactored versions. Additionally, we submitted 19 pull requests based on our automated refactorings; 16 of these were accepted by project maintainers. These results suggest that U2W effectively improves test-suite quality, and validate our novel test smell aligns closely with developers' intuitions and practices.

II. INTRODUCTION

Writing high-quality tests is essential for ensuring software correctness, maintainability, and long-term evolution. However, test suites frequently devolve to accumulate suboptimal design decisions, resulting in what are known as *test smells* [1]–[3]. These test smells are not technically incorrect but may foster deeper issues, such as reduced clarity and readability, compromised maintainability during long-term software evolution, or diminished effectiveness of downstream tasks, such as fault localization and program repair [4]–[7].

A common source of test smells stems from developers' urgency to keep pace with rapidly evolving codebases. In such scenarios, developers often reuse existing test methods as templates—renaming methods, tweaking inputs, or modifying assertions—without fully reconsidering the underlying test structure [8]. This "copy-and-paste" behavior has been documented in empirical studies of how developers write unit tests [8]. Consequently, test methods often become overly lengthy and tangled, combining multiple independent testing logics that should ideally be isolated into distinct unit tests.

Such lengthy test methods often simultaneously exhibit multiple previously documented test smells. Examples of such test smells include *eager tests* [1] that validate several behaviors in a single test method, *verbose tests* [9] spanning numerous lines, and *duplicate tests* [1] redundantly covering similar logic with slight variations. Although these classic fine-grained smells have been well-established since 2001 [1], their detection typically involves heuristic-based thresholds such as the number of production method calls [9]–[13], line count [9], [14], and textual similarity [15], and their refactoring is often subject to developer judgment and development context [1], [6], [16].

In this paper, we identified this previously undocumented test smell through a pilot study, which we describe in the motivation section. We then formalize this identified test smell, which we call *Disjoint Assertion Tangle (or DAT)*, with a precise definition, characterized by a test method that contains multiple semantically independent *Assertion Clusters*—a logically cohesive unit consisting of one or more assertions that share a common setup and collectively verify a specific behavior or concern. We surmise that this smell may typically arise from incremental edits or mechanical reuse of test code, a phenomenon highlighted by prior studies [8]. We hypothesize that the presence of DAT undermines test focus, readability, and maintainability by entangling independent test-execution logics in a single test method.

To address the DAT test smell, we introduce a fully automated technique that both *detects* and *refactors* affected test methods into cleaner, modular forms. Specifically, our approach decomposes lengthy, smelly test methods into multiple *conventional unit tests* (CUTs), synthesizes them into *parameterized unit tests* (PUTs), and generates descriptive names for each refactored test method and extracted variables. Moreover, for tests that have been parameterized, our approaches also can optionally suggest new value sets to provide greater test thoroughness. Consequently, while our technique is primarily designed to address DAT smells, it also has the potential

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00108

to mitigate other well-known classic test smells—such as Eager Test and Verbose Test—by promoting more focused, less redundant, and clearly named tests.

To evaluate our approach, we conducted multiple studies using various empirical methods. First, we conducted a quantitative study by implementing our approach in a tool called U2W to evaluate the prevalence of DAT and assess the quantitative effects of their refactorings on 49 open-source Java repositories. Our tool identified 3638 instances of DAT, isolating 31,837 independent behavioral logics subsequently refactored into focused unit tests, including 1713 parameterized test methods. This transformation resulted in a 36.33% reduction in test code duplication. Second, we conducted a developer survey with 49 industrial and academic developers focusing on non-functional attributes and overall preferences for the refactored tests. The results indicated strong developer preference for the refactored tests. Finally, we submitted 19 pull requests containing our transformations to open-source projects. Of these, 16 pull requests were accepted and merged by their project maintainers, which provides strong support for our hypothesis that DAT is undesirable and that their refactored versions are preferred.

The primary contributions of this paper are:

- Formalizing and defining the new test smell, *Disjoint Assertion Tangle*, with actionable refactoring guidelines.
- Designing and implementing a fully automated technique that detects and refactors DAT instances into conventional and parameterized unit tests.
- Empirically evaluating our technique across 49 widely-used Java repositories, detailing the prevalence, characteristics, and impact of refactoring.
- Conducting a developer survey and validating the practical acceptance of our refactoring approach through real-world pull request submissions.

Data Availability. All code, experimental setup, survey, results, and links to pull requests are available [17].

III. Motivation

This section presents an illustrative example that highlights a hypothetical scenario leading to a smelly test, grounded in test-engineering behaviors observed in prior research [8]. We then describe a pilot study on an open-source project to empirically validate our hypothesis regarding the presence of this test smell in real-world test-development practices. Additionally, we discuss our manual refactoring of the identified smelly tests to gain preliminary insights into effective refactoring strategies.

Motivating Example. Figure 1(a) presents an example of a smelly unit test. A developer tasked with testing a banking application begins by creating an initial test case — instantiating an `Account` object and verifying its `name` and `balance` via two assertions. To expedite coverage of additional scenarios, the developer resorts to a common shortcut: copying and pasting existing test code. Specifically, they repeatedly duplicate the account creation and make minimal edits, *i.e.,* updating arguments in `account` instantiation, to validate multiple accounts. They also replicate the original assertions, adjusting only the expected values and the verified variable names.

Although a developer's intentions may be for such changes to be temporary, such incremental edits may often expand and persist due to project pressures and delayed refactoring efforts. Consequently, the test method grows increasingly complex, exhibiting a clear test smell: it intertwines semantically independent test code by instantiating three separate `Account` objects and verifying each with distinct groups of assertions.

Although this scenario presents a hypothetical and simplified test-engineering workflow, it closely reflects common testing practices familiar to many developers. Such "lazy" test development is not merely theoretical; Aniche *et al.* [8] observed 13 developers performing think-aloud testing tasks on real-world open-source projects and found that developers frequently resort to "copy-and-paste" or "template-based" test creation with minimal modifications as a productivity shortcut.

Observing DAT Test Smell in Real-World Code. This work was first motivated by the authors' experience observing test code that accrued multiple, independent test assertions. Particularly for students enrolled in software-testing courses, submitted homework assignments often contain a single test method that contain several test assertions (*e.g.,*`assertEquals(2, sum(1,1); assertEquals(3, sum(1,2)); assertEquals(4, sum(2,2);`) The first author also independently conducted a pilot study of open-source test code, with the purpose of discovering opportunities to parameterize test cases. The first author also observed a frequent phenomenon of test methods that contained multiple sets of lines of test code that were independent, in terms of data and control dependence. These observations, once shared, became the impetus for us to define this test smell and to pursue further investigation of it.

From our manual analysis in the pilot study, we found that smelly tests in real-world projects can be considerably more difficult to identify — and sometimes exhibit more severe structural issues — than our contrived example. Figure 2 presents a real-life test case with two distinct sets of test-code lines that share no control or data dependence. We call each such set of test code as an "assertion cluster," and a test method that contains multiple such assertion clusters as exhibiting the DAT test smell. In this figure, each assertion cluster is highlighted in a different color to illustrate their potential to be isolated into separate tests. These semantically independent execution logics are interwoven within the same test method, which significantly obscures the test's behavior and the developer's intent. Refactoring these clusters into clearly named, standalone test methods — as illustrated in the legend of Figure 2 — may improve both comprehension and extensibility.

Additionally, we manually refactored the identified smelly tests into multiple distinct test methods and found that the isolated methods often share identical method invocation sequences, differing only in input values. For example, in our illustrative scenario depicted in Figure 1(a), a single test method was effectively split into three methods (depicted in Figure 1(c). We found further opportunities to minimize code duplication by transforming conventional unit tests (CUTs) into

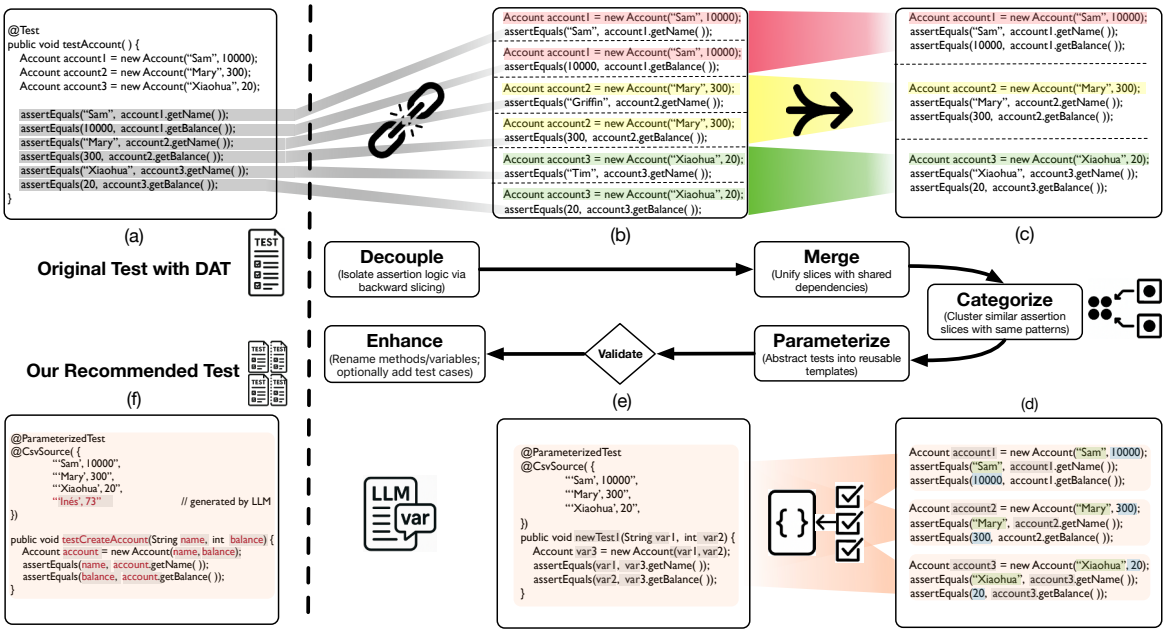

Fig. 1: Technique workflow: detection and refactoring phase

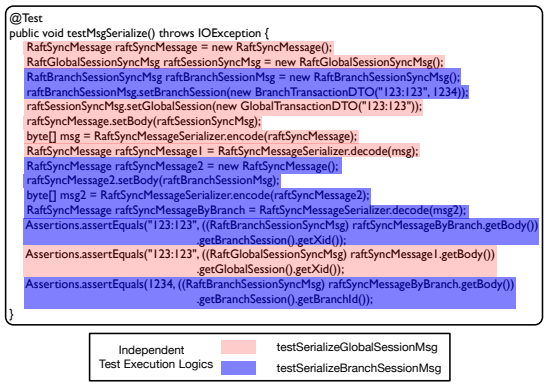

Fig. 2: Example APACHE SEATA test with independent logic

parameterized unit tests (PUTs) [18]. Such parameterization provided a reusable template structure, conveniently supporting the more test scenarios with additional value sets.

Our pilot study confirmed that smelly tests with interwoven independent logic clusters exist within open-source projects. Moreover, we found practical evidence of developers applying the "copy-and-paste" and "template-based" test-engineering style beyond previous human studies [8]. Our refactoring demonstrated the potential to isolate smelly tests into clearly focused, properly named methods and also the significant additional benefits from adopting parameterized tests, which further reduces code duplication and induces new value sets.

IV. FORMALIZING DAT TEST SMELL, BACKGROUND, AND REFACTORING PRINCIPLES

In this section, we formally define this newly identified test smell, *Disjoint Assertion Tangle (DAT)*. Then, we provide a detailed comparison between DAT and classic test smells. Finally, we formalize the principles of refactoring DAT instances.

A. Test Smell: Disjoint Assertion Tangle

Let T be a test case containing a set of assertion statements $\{a_1, a_2, \ldots, a_n\}$, and let S_i denote all setup statements from T required to execute assertion a_i, i.e., the set of program statements that may influence the outcome of a_i.

We construct an undirected graph $G = (V, E)$ where:

- $V = \{a_1, a_2, \ldots, a_n\}$
- An edge $(a_i, a_j) \in E$ exists iff $S_i \cap S_j \neq \emptyset$

The test T exhibits DAT if G contains more than one connected component, *i.e.,* T can be partitioned into two or more assertion clusters with non-overlapping logic dependencies. This indicates that T is composed of multiple semantically disjoint sub-tests and would benefit from decomposition for improved maintainability, clarity, and fault isolation.

B. Comparison of DAT with Classic Test Smells

Our formalized definition of Disjoint Assertion Tangle (DAT) advances beyond classic test smells by grounding "testing too much" in the precise, actionable notion of *behavioral isolation* and, as a byproduct, enabling detection of *intra-method code duplication*—a dimension overlooked by existing smells.

Behavioral Isolation. Table I contrasts DAT with classic smells that allude to the concept of "testing too much" (*e.g.,* Eager Test, Verbose/Long Test), sometimes framed as violations of the single-responsibility principle [22]. Those smells rely on ambiguous definitions and heuristic, threshold-based proxies that lack actionable precision and resist full automation. Specifically, Tran *et al.* [23] systematically analyzed test-smell detection tools, identifying at least six distinct rules

Smell	Core Concepts		Detection		Refactoring
Disjoint Assertion Tangle (DAT)	Precise	Tangled assertions verifying disjoint logic and dependencies	Precise	Structural slicing + disjoint-set algorithms	Fully Automatable
Eager Test	Ambiguous	"Eagerly" verifies multiple behaviors in one test	Heuristics	Production method count (most common) [9]–[13]; calls to multiple production classes [19]; information retrieval-based metrics [20]; machine learning-based approaches [21]	Developer expertise required (*e.g.,* limited UI support [20])
Verbose Test		Excessive verbosity / long setup		Lines of Code (10–30 [9], [14], [19])	

Note: The core concepts reflected in test smell naming and the associated detection metrics are often conflated or used interchangeably in prior literature.

TABLE I: Comparison of Disjoint Assertion Tangle (DAT) with classic test smells regarding behavioral isolation

for detecting Eager Tests, and proposed finer-grained heuristics to improve detection accuracy. DAT, instead, is *structurally* defined: a tangle is present when assertions in a single test verify *disjoint* logic and dependencies. This yields a crisp, algorithmically checkable criterion aligned with behavioral isolation and directly supports automated refactoring, rather than merely flagging symptoms. Overlaps exist—DAT instances may also appear verbose (too long) or eager (invoking too many production methods), and conversely, tests identified as Eager or Verbose can themselves inadvertently manifest as DAT instances (*e.g.,* Figure 2). Yet length or call count alone does not imply a tangle: a multi-line, multi-call test can still exercise a single coherent behavior and thus be non-DAT, which typical heuristics would misclassify and still require developer judgment for detection and refactoring. Conversely, a test that invokes only a few production methods below the detection threshold may still constitute a DAT if isolatable execution logics are present. Examples of such overlaps can be identified from the file *examples.md* in our artifact [17].

Code Duplication. As a byproduct of isolating tangled behaviors, identifying a DAT instance also exposes intra-method test code duplication. The classic Duplicated Code smell [9] highlights redundancy in test code, but detection is typically limited to inter-method clone metrics [9], [12], which often miss duplication embedded within a single test. DAT's structural segmentation of execution logic naturally reveals such duplication, thereby complementing traditional inter-method clone detection with intra-method analysis. Moreover, the presence of intra-test duplication aligns with developers' observations of copy-paste patterns inside tests [8]. This perspective broadens the scope of duplication detection to capture the full spectrum of redundancy in test code.

C. Refactoring Principles

Refactoring a DAT smell instance requires careful consideration not only to eliminate the DAT itself, but also to prevent the introduction of additional test smells during the process. Drawing from both our analytical assessment and practical refactoring experience, we derive a set of structured and targeted guidelines specifically designed to address the DAT smell:

1) **Tangled Behaviors**: Partition DAT-smelly tests into distinct, self-contained behaviors, while avoiding extremes such as overly *eager* or excessively *verbose* testing.
2) **Duplication Reduction**: Identify duplicated code across isolated test methods and, where appropriate, refactor into shared helpers or parameterized unit tests, potentially enabling systematic exploration of new value sets.
3) **Understandability Refinement**: Improve clarity by applying meaningful test method names (see also Anonymous Test [14]) and variable naming conventions (see also Magic Number Test [24]). This ensures that isolated test logic remains both interpretable and maintainable.

V. Technique

This section describes our proposed approach and our tool, U2W (Untangle to Weave), designed to automatically detect and refactor tests affected by Disjoint Assertion Tangle. Our technique follows two phases for each test method under analysis: (1) smell *detection* and (2) test *refactoring*.

Figure 1 illustrates these phases, where Steps (a), (b), and (c) refer to the smell detection phase, and Steps (b)–(f) correspond to the test-refactoring phase. The provided figure utilizes the illustrative example from Figure 1(a). In the remainder of this section, we depict our technique comprehensively and discuss the rationales behind our refactoring process, emphasizing the three fundamental quality guidelines (Behavioral Isolation, Duplication Reduction, and Understandability Refinement) introduced earlier in Section IV-C.

Detection Phase. We leverage the Disjoint Set algorithm [25] to determine whether a test method aggregates multiple independent test behaviors. Concurrently, this step identifies and isolates distinct behavioral segments, *i.e.,* assertion clusters, within the identified smelly tests.

The first two steps in Figure 1, Decoupling (b) and Merging (c), are shared between the detection and refactoring phases. In the detection phase, these steps are performed virtually (*i.e.,* in memory) and in the refactoring phase, the cumulative result of both phases is then output to new suggested test files.

Decoupling: Backward Slicing from Assertions. Initially, U2W transforms each test method under analysis into its Abstract Syntax Tree (AST) representation. It subsequently identifies all assertion statements in the test code and applies static dependency analysis (backward slicing) on each of them. By performing backward slicing from each assertion, the algorithm determines minimal sets of statements (*i.e., slices*) required to independently reproduce the setup of each assertion.

The result from this step may contain instructions that form multiple "purified" tests [26], each containing exactly one assertion statement and minimal setup code. Such purified tests, or "atomized" tests in the fault-localization literature [26], allow isolation and debugging of individual behaviors.

However, producing these highly granular, single-assertion test cases may result in a *Lazy Test* anti-pattern [9], [22], [27], where each test verifies excessively narrow behavior, causing unnecessary redundancy across test methods. Therefore, we refrain from generating atomized tests directly. For instance, as depicted by Figure 1(b), instantiating the same `account` object repeatedly results in unnecessary duplication of setup code.

Merging. To optimize granularity and minimize code redundancy, U2W next merges atomized tests into assertion clusters when they share identical statements according to their AST representation. Each resulting assertion cluster reflects one coherent, independent execution logic, including the appropriate setup statements (illustrated as Figure 1(c)). Ultimately, if a test contains multiple independently verified assertion clusters, the algorithm reports a DAT smell in that test method.

Refactoring Phase. After detecting a DAT instance, our approach refactors the affected test methods and further reduces code duplication by identifying parameterization opportunities. The resulting in-memory data structures that represent the independent assertion-cluster from the Decoupling/Merging steps are first written to new test files for tests with the DAT smell. Then, optionally, the technique can further refine these refactored tests potentially parameterizing them, renaming method and parameter identifiers, and generating new value sets. Each such step is described here.

Assertion-Cluster-Based Refactoring. Based on the in-memory assertion clusters that were detected for smelly tests in the previous phase, the technique then extracts statements from the original tests and consolidated them into new test methods. This approach preserves the original statement order and retains associated comments.

Categorization of Similar Tests. The technique next identifies suitable candidates for parameterization by analyzing and grouping similar assertion clusters based on structural equivalence of their ASTs. Specifically, two assertion clusters are considered similar if they fulfill the following criteria:

1) Their AST structures are identical.
2) They differ exclusively in literal values or variable identifiers within their fixture code. Variables are replaced with abstract versions and compared for equivalence. Literals are made into parameterized value sets.
3) Their assertion statements are equivalent in terms of assertion type and purpose.

Such grouping criteria correspond conceptually to Type-2 code clones [28], characterized by structural identity with variation limited to identifiers and literals. Please note that our technique supports grouping similar assertion clusters across multiple DAT-impacted test methods within the same class, ensuring that such cases are not overlooked in parameterization.

Parameterization. Upon identifying similar tests as suitable parameters, the tool generates parameterized test cases from these assertion clusters. During this step, our tool automatically formulates parameterized unit test (PUT) templates by annotating the test methods with appropriate JUnit annotations (e.g., `@ParameterizedTest`, `@MethodSource`, or `@CSVSource`), extracting variable inputs for parameterization, and preparing corresponding source methods.

Initially, the PUT templates employ temporary placeholders for variables and method names (e.g., placeholders such as *var1*, *var2*, etc., demonstrated in subfigure (e) of Figure 1), awaiting further name refinements.

Validation. After leveraging parameterization opportunities, U2W executes the refactored tests against the codebase to ensure semantic correctness. This execution-outcome validation discards any refactored test that fails to reproduce the same outcome as its original counterpart (*e.g.,* fails instead of passes), leaving the original tests unaffected. Moreover, the tool also enforces structural consistency checks to guarantee that the sequence of statements and the total number of statements preserved within each assertion cluster remain unchanged.

LLM-Based Enhancement. Finally, our method utilizes Large-Language Model (LLM) capabilities to provide meaningful and descriptive identifiers for method and variable names. To achieve this, we feed the LLM prompt with sufficient context information, including: (1) the original (unrefactored) smelly test method, (2) relevant test input values guiding parameterization, (3) signatures of the methods under test, and (4) descriptive comments (if available within the test context).

We prompt the LLM to generate clean, formatted method names and descriptive variable identifiers. The placeholders initially defined in the PUT method are replaced with these clear and meaningful identifiers, significantly enhancing the readability and understandability of the refactored tests. Additionally, U2W provides an optional feature that leverages LLM to retrofit new value sets into the parameterized test methods.

VI. Evaluation and Experimental Methodology

In this section, we detail the experimental design used in evaluating our proposed technique. We first define the research questions guiding our evaluation, along with their rationales. Subsequently, we describe our experimental procedures and metrics used to investigate these research questions. Specifically, our experiments examine (1) the prevalence of the DAT smell, (2) quantifiable structural improvements achieved through refactoring (*i.e.,* behavioral isolation and code de-duplication), and (3) developers' perspectives gathered via practical feedback and pull requests to open-source projects.

Research Questions:

- RQ1: How prevalent is the Disjoint Assertion Tangle smell in real-world open-source test suites? **[Prevalence]**
- RQ2: How effectively does our proposed approach isolate distinct execution logics from smelly test code into individual refactored tests? **[Behavioral Isolation]**
- RQ3: To what extent can U2W's parameterization module reduce code duplication in unit tests exhibiting test smells? **[Code De-duplication]**
- RQ4: How do developers perceive unit tests refactored by U2W? **[Developer Preference]**

RQ1 ***Prevalence***. For this research question, we systematically investigate its prevalence in real-world codebases. Particularly, we present the percentage of tests flagged with DAT test smell relative to the unit tests analyzed by our technique.

We also quantify overlaps between DAT and existing smells to highlight their points of convergence and distinction. For this comparison, we adopt the most representative detection metrics for Eager Test and Verbose Test used in prior studies [29]. While these smells have been defined using different heuristics and thresholds across tools—leading to variation in interpretation—our goal is not to revalidate those definitions. Instead, we aim to illustrate where DAT instances coincide with or diverge from them, and, more importantly, to demonstrate that U2W provides *fully automated refactoring support* not provided by prior approaches.

Please note that the common Eager Test heuristic threshold is only set to be 1, making any tests with 2 or more production method calls detected as Eager Test. As such, we use the number of production method calls ($>= 4$) and the lines of code ($>= 13$) as thresholds for Eager Test and Verbose Test, respectively, following prior empirical work that identified these values as "medium severity" [30].

RQ2 ***Behavioral Isolation***. Then, we analyze the degree to which tangled assertion clusters exist within identified smelly tests. Specifically, we quantify the amount of "behavioral entanglement" by counting how many distinct test-execution logics (referred to as assertion clusters) are extractable into cohesive, focused tests as a result of our refactoring approach. This demonstrates the effects of the "Decoupling" and "Merge" components ((b) and (c) in Figure 1). The numeric results enable us to measure how effectively our technique mitigates "eagerness" in the smelly tests.

RQ3 ***Code De-duplication***. Behavioral isolation within a single test method enables intra-method clone detection and parameterized refactoring—capabilities that prior techniques do not offer. We then assess the effectiveness of our parameterization module in reducing redundant code from test methods. This corresponds to the "Categorize" and "Parameterize" components ((b) and (c) in Figure 1). Particularly, we measure executable lines of code (eLOC) in each test before and after refactoring, and compute the duplication reduction rate as:

$$\text{De-duplication Rate} = \frac{\text{eLOC}_{\text{Before}} - \text{eLOC}_{\text{After}}}{\text{eLOC}_{\text{Before}}} \times 100\%$$

Additionally, we report the number of value sets extracted per PUT method. This gives us quantitative insights into our approach's capacity to generalize repetitive test logic.

RQ4 ***Developer Preference***. *RQ2* and *RQ3* offer quantitative evaluations regarding measurable improvements in behavioral isolation and code duplication in refactored tests. To complement this, *RQ4* investigates the human-centric viewpoint, focusing on real-world developers' perspectives after reviewing refactored tests generated by U2W. To obtain qualitative feedback, we surveyed participants with different backgrounds to evaluate their preferences and perceptions of original (smelly) versus refactored tests. We also submitted 19 pull requests containing refactored tests to open-source projects and evaluated responses from maintainers. This evaluation not only assesses the understandability of the refactored tests—focusing on the test method and variable naming enhancement module of our tool—but also validates that our novel test smell is consistent with developers' intuitions and practices.

Quantitative Evaluation (RQ1–RQ3).

Dataset. The dataset used in this study incorporates all the subjects from [31], [32] and [33], originally derived from previous works [34]–[37]. Additionally, we augmented our dataset with six additional projects identified during preliminary analysis as likely candidates for containing DAT instances. These extra projects were opportunistically selected based on their popularity and ongoing, active development status.

Overall, our final dataset comprises 49 diverse open-source Java projects with varying levels of popularity. The subject identifiers and names are shown in the first two columns in Table II. Among the subjects, 13 projects have more than 10,000 stars, 14 projects fall within the range of 1,000 to 10,000 stars, and 22 have fewer than 1,000 stars. Collectively, these projects represent a broad spectrum of Java projects with JUNIT tests, including frameworks designed for distributed systems and microservices (e.g., APACHE DUBB, APACHE SEATA), big data and stream processing solutions (e.g., APACHE FLINK, APACHE HADOOP), web-oriented libraries and APIs (e.g., JSOUP, SPRING-WS), tools designed for testing and observability (e.g., APACHE SKYWALKING, GRAYLOG), as well as various utility libraries and domain-specific tools extensively leveraged throughout enterprise software development.

Specifically, we excluded test methods containing conditional logic constructs (`if-else`, `switch`), as we consider these indicative of deeper-level design issues that should be addressed prior to addressing DAT. Previous research characterizes these constructs as *Conditional Test Logic* test smell [29]. Moreover, our tool automatically filters out test case methods that are not compatible with our rudimentary slicing implementation, which includes code involving lambda expressions, mocking frameworks, and other complex constructs (e.g., `@Override`, parameterized tests, `Thread.sleep`). These exclusions, while notable, do not impact the conceptual validity or effectiveness of our approach, as our technique remains applicable and generalizable to these scenarios given enhanced tool support. The resulting refined dataset constitutes the basis for our analysis and evaluation. Detailed filtering statistics per repository are available in our provided artifacts. As a result, our dataset comprises 42,334 test case methods from 49 subjects.

Experimental Environment. We conducted our experiments using a 2022 MacBook equipped with an Apple M2 CPU (3.49 GHz, 8-core CPU), 8 GB of RAM, operating on macOS 15.3.2. We used GPT-4o Mini [38] to implement our LLM-based enhancements. We selected this model due to its publicly accessible API via the LangChain library and its reduced environmental footprint compared to larger alternatives. Additionally, for evaluation purposes, we disabled

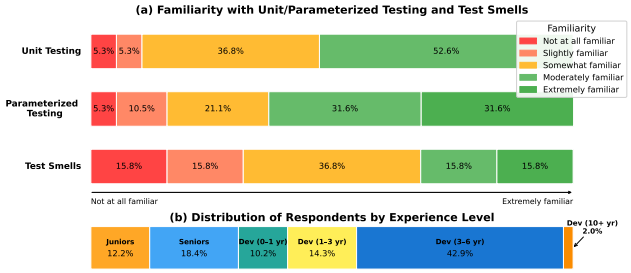

Fig. 3: Demographics of participants

the optional feature in U2W that uses LLM to augment the generated parameterized unit tests with extra value sets. Overall, excluding the time spent on LLM-assisted enhancements and execution validation, the complete refactoring effort across all subject projects required approximately 10 minutes. During this process, 8 (out of 3646) refactored tests were rejected due to failures identified during the execution-validation process.

Survey (RQ4). Our survey was conducted in two stages. The first stage captured participants' preferences on unit test styles through written evaluation tasks, while the second stage was a post-survey profile that gathered additional background information. This separation was intended to minimize priming effects [39]: if participants had been introduced to terms such as "test smell" or labels like "Eager Test" beforehand, they might have evaluated the code examples by rigidly applying predefined rules rather than relying on their own judgment of readability, maintainability, and overall quality.

Survey Design. The first stage of the survey presented six representative code examples from our technique. For each, participants were asked to compare randomized side-by-side pre- and post-refactoring tests. They indicated their preference across five dimensions: readability, maintainability, extensibility, debugging capability, and overall preference. Each question also included a text box for optional open-ended comments.

The six example pairs were selected to balance realism, comprehensibility, and diversity. Among them, four examples were drawn directly from open-source projects, and two were lightly simplified (*e.g.,* renaming identifiers or removing unrelated code) to improve clarity while preserving the essential characteristics of the code and the DAT smell. The set covers a range of DAT patterns our tool addresses, including: (1) isolated tests with non-trivial logic, (2) parameterized tests with concise or multi-line logic, (3) long smelly tests split into one or more parameterized templates, and (4) parameterized tests with varied input sizes (2–5). All six pairs are available in our public artifact for transparency.

The second stage was a post-survey profile designed to capture background information. Participants reported their years of unit testing experience and rated their familiarity with unit testing, parameterized unit testing, and test smells on a 5-point Likert scale [40], ranging from "Not at all familiar" to "Extremely familiar."

Participants. We invited a total of 49 participants. Recruitment was conducted through the authors' professional networks and by contacting senior undergraduates from the university's ICS department with demonstrated experience in programming and testing, as well as software developers working in industry. The participants' demographics are summarized in Figure 3.

About 70% of the participants were industry practitioners, while the remaining were upper-division undergraduates (third- or fourth-year students). On average, respondents from industries reported 4.3 years of unit testing experience. Industry respondents averaged 4.3 years of unit testing experience; over half reported being "extremely familiar" with unit testing and "moderately familiar" with parameterized testing. In contrast, roughly 70% had limited familiarity with test smell terminology despite extensive exposure to unit tests."

Analysis Method. We analyzed the results of binary preference questions across participants grouped by participants' experience levels: (1) software engineers with ≥ 3 years of industry experience, (2) software engineers with 0-3 years of industry experience, and (3) junior and senior undergraduate students. We first aggregated all binary preference responses across participants and code examples to calculate the overall favoring percentages. Then, we investigated participants' preference patterns for specific code pairs to understand which types of refactoring changes influenced their choices. Then, two authors qualitatively analyzed the open-ended comments, discussing and extracting key insights from the participants' reasoning.

Pull Requests (RQ4). We also contributed back to the open-source community by submitting pull requests (PRs) containing the refactored tests from our technique. At the point of writing, our efforts yielded a total of 19 pull requests containing refactored tests from our technique. These PRs contain 1 to 4 refactored tests and were distributed across 6 open-source projects, including APACHE SEATA (6 PRs), APACHE BAREMAP (9 PRs), APACHE MORO (1 PR), APACHE COMMONS NET (1 PR), JSOUP (1 PR), and APACHE XTABLE (1 PR). Among them, 2 PRs include additional value sets in the refactored PUT method to further motivate the refactoring. With the outcome of PRs, we drew insights from real-life project maintainers' feedback.

VII. RESULTS

RQ1: Prevalence of Disjoint Assertion Tangle. Columns 3 and 4 in Table II present the total number of analyzed tests and the occurrence of DAT across each subject in our dataset. For instance, in COMMONS-CLI, we identified 21 tests with DAT among the total 171 analyzed tests. To further illustrate and complement the information from Table II, we visualize the prevalence of DAT per subject in Figure 5, sorting them in descending order to better depict the distribution. The IDs of the subjects on the horizontal axis of Figure 5 correspond to the subjects listed in Table II. We observed DAT occurrences in almost all subjects (47 out of 49) in our dataset; however, its prevalence varies. DAT instances can reach as high as approximately 30.0% of the analyzed tests in SPOTIFY-WEB-API and COMMONS-LANG. In total, we identified 3,638

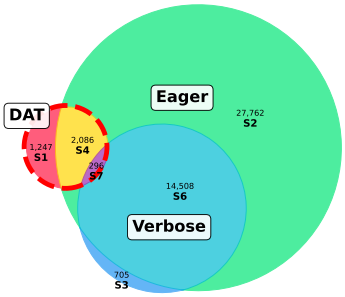

Fig. 4: Overlap of DAT, Eager, and Verbose Test Smells

smelly tests containing DAT among the 42,334 tests analyzed, representing 8.6% of the tests in our entire dataset.

Moreover, Figure 4 illustrates the overlaps between DAT, Eager Test, and Verbose Test. Eager Tests dominate the distribution, accounting for 54.5% of all analyzed tests, with a large exclusive region (S2) and a substantial overlap with Verbose Tests (S6). This prevalence and smell coexistence observation is consistent with prior empirical findings [5].

DAT, in contrast, constitutes a smaller portion overall but exhibits distinct characteristics. Notably, 1,247 tests (S1) are exclusively DAT, capturing behaviors overlooked by heuristic thresholds for Eager (≥ 4 method calls) and Verbose (≥ 13 LOC)—thresholds identified as medium severity in earlier studies [30]. Among DAT overlaps, the largest is with Eager Test (S4, 2,086 tests), while the overlap with Verbose Tests (S5) is minimal–just nine tests–too small to be shown in the proportional Venn diagram due to geometric layout constraints.

The most important insight lies in the DAT instances highlighted by the red-dotted circle: these are "perfectly separable" where completely independent execution logics exist within a single smelly test. Their structural independence provides an unambiguous signal that the test can be and maybe should be refactored, and our tool performs this automatically—something prior heuristics cannot achieve. As a result, our tool not only refactors DAT instances but also simultaneously restructures 2,391 tests flagged as medium-severity Eager or Verbose, thereby demonstrating practical value in addressing multiple smells within a unified framework.

RQ1: DAT smell is prevalent, appearing in 47/49 projects and affecting 3,638 test methods (8.6%). U2W automatically detected and refactored these instances, including 2,391 also marked as medium-severity *Eager* or *Verbose* tests.

RQ2: Isolation of Independent Execution Logics. For the 3,638 smelly test method, U2W successfully isolated a total of 31,837 distinct assertion clusters, which were then abstracted into 17,494 test case methods through U2W's parameterization module. Figure 6 illustrates the distribution of the number of assertion clusters our technique detected, isolated, and refactored from each non-refactored test. For example, 1,895 smelly tests containing DAT can be split into exactly two

TABLE II: Dataset Overview: Subject Projects & Test Analysis

No.	Project	Tests	DAT (%)	New PUTs	Avg Value Sets	Git Stars
1	commons-cli	171	21 (12.28)	15	3.67	368
2	commons-validator	267	38 (14.23)	22	47.91	213
3	spotify-web-api-java	317	95 (29.97)	0	0.00	1,048
4	cdk/data	141	1 (0.71)	1	5.00	528
5	commons-text	533	50 (9.38)	66	4.97	362
6	dyn4j	1202	30 (2.50)	2	3.00	509
7	commons-codec	403	81 (20.10)	35	10.37	470
8	joda-money	624	24 (3.85)	4	3.25	668
9	jline3	198	10 (5.05)	10	3.20	1,574
10	jfreechart	1579	19 (1.20)	2	7.00	1,293
11	incubator-seata	852	95 (11.15)	28	4.11	25,644
12	jsoup	844	36 (4.27)	24	6.17	11,175
13	commons-net	119	7 (5.88)	3	3.67	273
14	amoro	131	19 (14.50)	14	7.07	974
15	incubator-xtable	61	3 (4.92)	2	2.00	1,054
16	incubator-baremaps	136	26 (19.12)	7	4.00	539
17	Activiti	465	43 (9.25)	12	4.83	10,321
18	jitwatch	101	6 (5.94)	5	5.00	3,158
19	graylog2-server	1163	110 (9.46)	62	4.98	7,684
20	commons-configuration	713	44 (6.17)	14	3.29	208
21	commons-dbcp	180	25 (13.89)	3	2.67	350
22	commons-io	581	88 (15.15)	25	7.32	1,032
23	commons-lang	1554	465 (29.92)	357	5.74	2,802
24	commons-math	963	60 (6.23)	34	5.79	612
25	commons-pool	34	5 (14.71)	0	0.00	532
26	dubbo	1524	161 (10.56)	43	4.88	41,015
27	flink	4863	185 (3.80)	95	5.89	24,907
28	hadoop	4985	155 (3.11)	87	6.21	15,104
29	skywalking	128	17 (13.28)	11	3.82	24,336
30	rocketmq	470	24 (5.11)	8	3.38	21,809
31	storm	214	15 (7.01)	5	4.80	6,629
32	HikariCP	25	0 (0.00)	0	0.00	20,466
33	dropwizard	718	30 (4.18)	4	5.00	8,544
34	druid	5712	334 (5.85)	160	12.88	13,719
35	eclipse-collections	3701	664 (17.94)	203	3.99	2,513
36	elastic-job-lite	200	12 (6.00)	5	3.00	2
37	graphhopper	794	71 (8.94)	75	3.69	5,815
38	Openfire	324	15 (4.63)	5	2.80	2,932
39	java-design-patterns	438	16 (3.65)	4	3.25	91,910
40	marine-api	844	121 (14.34)	12	6.58	253
41	languagetool	934	224 (23.98)	194	21.50	13,138
42	spring-ws	366	16 (4.37)	5	3.60	330
43	asterisk-java	241	12 (4.98)	11	5.55	439
44	undertow	153	11 (7.19)	9	9.33	3,645
45	admiral	520	24 (4.62)	12	6.33	257
46	Wikidata-Toolkit	782	110 (14.07)	6	3.00	382
47	wildfly	781	18 (2.30)	11	6.18	3,110
48	carbon-apimgt	119	0 (0.00)	0	0.00	170
49	riptide	166	2 (1.20)	1	6.00	318
	Total	**42334**	**3638 (8.59)**	**1713**	**5.85**	375,134

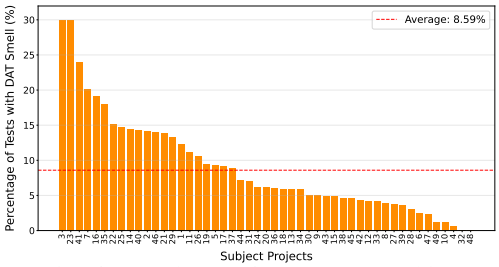

Fig. 5: Prevalence of DAT across subjects

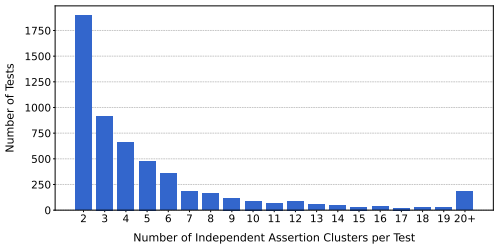

Fig. 6: Distribution of the number of extracted execution logics

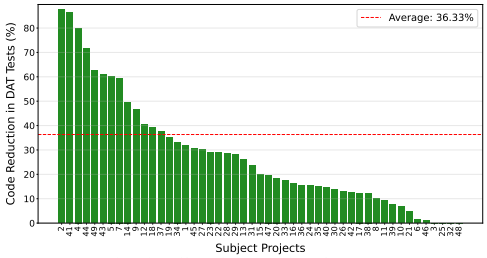

Fig. 7: RQ3: code duplication reduction among smelly tests

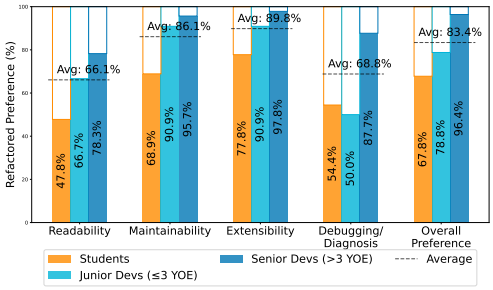

Fig. 8: Participant preference for our refactored tests

separate tests, each with its own independent execution logic.

We observe that most DAT-smelly tests contain a relatively small number (typically between 2 and 6) of assertion clusters. However, we also encountered some extreme cases: specifically, our approach identified 37 test methods containing more than 50 distinct execution logics each. In a particularly extreme instance, we found one test case method, *testDetNounRule* in LANGUAGE TOOL, containing 551 distinct execution logics. Through our manual investigation, we found that test case methods with unusually high numbers of execution logics often exhibit repetitive test logics with different value sets. Overall each smelly test contains a median of 3 assertion clusters.

RQ2: U2W successfully isolated 31,837 distinct execution logics from 3,638 DAT-smelly tests. Most DAT-smelly test could be refactored into 2–6 test execution logics.

RQ3: Code Duplication Reduction. Figure 7 illustrates the reduction in executable lines of code—achieved by U2W's parameterization module—among DAT-smelly tests across different subjects, sorted in descending order. For instance, parameterization achieved a reduction of up to 87.6% in duplicated executable lines of code among smelly tests in COMMONS-VALIDATOR (subject ID 2). We found that the de-duplication rate varies notably across different projects. Overall, U2W reduces test-code duplication for 45 out of the 47 subjects that exhibited the DAT smell, achieving an average duplication reduction rate of 36.3% among smelly tests.

RQ3 The parameterization module in U2W successfully reduced the duplicated executable lines of code in DAT-smelly tests for 45 out of the 47 DAT-smelly subjects, achieving an average duplication reduction of 36.3%.

RQ4: Developer Preferences for Refactored Tests. *Developer Preference Survey.* Figure 8 uses grouped bar charts to illustrate the percentage of favorable ratings for U2W-refactored tests across four quality dimensions and overall preferences. Within each dimension, results are categorized by participant demographics: students (orange), junior developers (light blue), and senior developers (dark blue). Additionally, the average favoring preference percentages (aggregated across all participant groups) are indicated with dashed horizontal lines and accompanying labels across each quality dimension.

Among all participants, we found that nearly all agreed refactored tests have enhanced maintainability (86.1%) and extensibility (89.8%). However, the perceived improvements in readability and debugging/diagnosis received more moderate support, with favorability rates averaging 66.1% and 68.8%, respectively. Regarding overall favorability, 83.4% of participants preferred U2W-refactored tests over the original ones.

Breaking down the results by participants' experience levels, we found senior developers with more than three years of professional experience expressed stronger preferences for our refactored tests, with 83.4% favoring U2W-refactored tests compared to lower preference rates among junior developers (78.8%) and undergraduate students (67.8%).

We also manually investigated developers' comments to gain insights into the refactored tests. Below, we highlight some of the common themes reflected in their feedback:

Behavioral Isolation:

> *"Each test case for different scenarios must be wrapped in different testing blocks, which is where [refactored code] shines. Secondly, I like the readability of function name which specifies what the actually the test case is doing." (P18)*

Code Duplication:

> *"Imagine having many assertEquals and there is a change you need to make across each line, that will be lot of code refactoring." (P17)*

Readability: Participants have diverged opinions on the readability of refactored tests under specific scenarios, especially for parameterized tests. One developer noted that variable names in parameterization can sometimes hinder readability.

> *"[non-refactored code] is more readable, just because you see the variables inline in the function calls in [refactored tests], but for all other purposes, [refactored code] is better." (P10)*

We investigated this case and speculated that developers may prefer inline values over readable variable names when the inline values themselves are intuitive and self-readable, which is against the definition of the classic test smell, Magic Test.

Moreover, when the extracted PUT method is simple, e.g., containing only one assertion with understandable inline values, it may appear less readable than the non-refactored one:

> *"[refactored code] is not as readable, could be improved by improving the formatting in the test cases, currently it's a long list of numbers. [unrefactored code] makes it obvious about what exactly is being calculated." (P10)*

Additionally, developers may prefer the adoption of parameterized tests when there are many new value sets to be used.

> *"It will depend on how much I want to test that method, generally you end up only testing for a few value in which case the first one makes sense. I will use the [non-refactored code] until I really need to use the [refactored code]." (P12)*

Pull Requests. Among 19 submitted pull requests at the time of submission, 16 were accepted by project maintainers, 2 are still open, and 1 is closed. One PR is closed because the project maintainer is happy with the current implementation. Among the two open PRs, one received positive feedback, but the maintainers noted that modifying a single test would be inconsistent with the project's overall test suite. In the other open PR, which includes a PUT method with three value sets, the maintainers discuss that PUT is most beneficial when dealing with numerous test cases and complex method body:

> *"I usually wait to use parameterized tests until there are a lot of test cases and the repeated code starts to get messy. When there are only a few checks, I think regular test lines are easier to read and understand. Parameterized tests can make things harder to follow if they don't really help with keeping the code cleaner or easier to update."*

Summary RQ4: Overall, 83.4% of survey participants preferred U2W-refactored tests, with a much higher preference observed among senior professionals. Additionally, 16 out of 19 pull requests were accepted. However, the adoption of parameterized tests may depend on both the number of retrofitted value sets and the complexity of the PUT method.

VIII. Discussion

Our methodology for investigating and learning about DAT test smells was inspired by previously observed human test-engineering patterns. These patterns offer a unique avenue for us to explore and formally define the test smell under study. The correlation between the DAT and classic test smells not only helps to explain the coexistence of multiple classic smells but also enables us to develop detailed refactoring guidelines. Our empirical analysis, developer surveys, and pull-request experiences have yielded several insights and implications:

Wide Presence and Impact of the DAT Smell. Our analysis reveals the widespread presence of the DAT test smell. Specifically, nearly all subject projects (47 out of 49) contained tests exhibiting the DAT smell, affecting an average of 8.59% of all analyzed tests. Developer surveys and pull request reviews further confirmed the negative impacts of the DAT smell on test readability, maintainability, debugging, and extensibility.

Mitigating Classic Test Smells. The precise definition of the DAT smell facilitates its detection and enables a fully automated refactoring approach—an uncommon achievement in the test smell community, where ambiguities in detection rules and refactoring guidelines have traditionally hindered automation. Unlike heuristic-based smells such as *Eager Test* or *Verbose Test*, which rely on arbitrary thresholds, DAT is grounded in structural isolatability, making automated transformation both feasible and reliable. The notion of isolatability itself provides a strong signal for refactoring: when independent behaviors coexist in a single test, they can be meaningfully separated.

Our results confirm that U2W automatically detected and refactored 3,638 of DAT instances, including 2,391 also labeled as medium-severity *Eager* or *Verbose* tests. In practice, DAT's refactoring solution inherently mitigates symptoms of these smells by splitting a smelly test into an average of 8.8 smaller, more focused tests (*RQ2*). This restructuring also decreases code duplication by 36.3% through parameterization (*RQ3*).

Despite these advances, many *Eager*/*Verbose* tests remain unresolved because their intertwined logic is not cleanly separable. Some are long merely to set up state—subjective cases that heuristics may over-report (over 50% labeled "Eager" in our study) and may not warrant refactoring. Addressing truly problematic cases likely requires developer-in-the-loop strategies. Still, DAT demonstrates feasible, fully automated refactoring and a path to progressively untangle complex tests."

On the Use of Parameterized Tests. Parameterized tests were proposed decades ago by researchers (e.g., [41], [42]) to abstract repetitive test logic and facilitate introducing new input values. However, our human studies found that parameterized tests can sometimes diminish readability. We speculate that the need for annotations (such as `csv` or `methodsource`) can pose minor visual readability challenges. Given this, the parameterized tests are generally more preferable when involving large sets of input values and with non-trivial execution logics. Consequently, future tools for refactoring DAT smells might prioritize parameterizing tests based on specific metrics to target the most beneficial improvements. Our current implementation scans the code and offers two refactoring options—split-and-merge or parameterized—leaving the choice to developers.

Test Smell Education. Our study also revealed varying opinions on DAT and refactored test cases among participants with

different levels of engineering experience. Notably, we found in *RQ4* that senior engineers showed a greater preference for refactored tests than junior engineers, who in turn preferred them more than undergrads. This suggests that academic programs could enhance software-testing curricula to promote better test-engineering practices and styles among students.

IX. Threats to Validity

Threats to Internal Validity stem from possible unintended behavioral changes, despite DAT being precisely defined and algorithmically separable. To mitigate this, U2W performs three validations: (1) automated structural checks to ensure consistency of statement order and line of code before parameterization; (2) parameterization restricted to Type-2 clones [28], which preserve structural identity with variations limited to identifiers and literals, excluding Type-3 clones that involve added, removed, or modified statements; and (3) execution validation as a pragmatic safeguard for behavioral preservation.

The last sanity check is necessary because static slicers, while theoretically sound, can occasionally be imprecise in practice due to unrecognized complex dependencies or external resources [43]. Our current implementation skips tests with complex constructs (*e.g.,* lambda expressions) and only considers first-party dependencies, excluding third-party libraries. In our experiments, 8 of 3,638 instances failed, all due to unrecognized external dependencies. For example, write–close–reopen of the same file path via a new variable creates a resource-level dependency that lies outside local def–use; static slicing therefore does not connect the two steps. As such, we reject such refactorings during execution validation.

To further validate correctness, the authors manually inspected randomly sampled 102 DAT instances (3%) and their refactored counterparts (447), finding no behavioral differences. Finally, all identified DAT instances and refactorings are released in our replication package, and refactorings are surfaced as recommendations, ensuring developers retain oversight.

There are *Threats to External Validity* relating to the ability to generalize of our findings for other projects, frameworks, and languages. That said, the observed "copy-paste" and "template-based" patterns are likely widespread, and our evaluation is performed on 49 diverse projects (42,334 tests), further supported by developer surveys and 19 submitted pull requests.

Threats to Construct Validity pertain to potential inaccuracies in characterizing the construct of "bad test design" or the "test smell" concept. Our definition of DAT might deviate from developers' intentions or perceptions. However, we grounded DAT clearly within established test-smell patterns. Furthermore, we demonstrated the practical harm of test-logic interweaving in our pilot study, reinforced our findings with developer surveys, and validated our construct through real-world pull requests.

X. Related Work

Classic Test Smells and Their Limitations. The concept of code smells was initially introduced by Deursen *et al.* [2] and later expanded to test code [1]. Since then, empirical studies extensively investigated its impacts [5], [6], [15], [16], [22], [29], [44]–[46], and various tools have emerged for test-smell detection [9], [13], [23], [24], [47], [48]. However, existing research seldom attempts full automation of refactoring detected test smells. Current detection is mainly semi-automated, offering suggested improvements yet still requiring substantial manual developer intervention during refactoring [9], [20].

Fully automating test-smell refactoring encounters several practical challenges: ambiguity in detection criteria, coexistence of multiple smells within a single test suite [9], and subjective developer evaluations of smell severity [6]. Our work listed the varied smell-detection metrics and thresholds in Section IV-B and adopted a "medium severity" threshold [30] to examine the empirical overlaps among DAT, *Verbose Test*, and *Eager Test*. Nevertheless, U2W uniquely supports the automated refactoring of DAT instances—capabilities not offered by prior detection metrics or existing tools. Another example is *Lazy Test* smell, which characterizes tests that insufficiently exercise intended behaviors, *i.e.,* "testing too little." However, detection is rare in existing tools and typically heuristic, identifying multiple tests that call the same methods with similar test fixtures [9], [12], [19], [24]. Moreover, *Duplicated Code* was usually addressed via inter-method clone detection with developer assessment in prior works [1], [9], [12] . By contrast, mitigating DAT enables intra-method duplication detection and parameterization of Type-2 Clones, complementing prior inter-method clone detection [12].

Parameterized Unit Tests. Parameterized Unit Tests (PUTs), originally introduced by Tillmann and Schulte [41], have gained attention due to their modularity and abstraction potential. Prior work has also explored manual refactoring of tests into PUTs [42] and automated approaches for retrofitting similar methods [49]. However, our technique instead targets long tests with multiple independent scenarios, emphasizing clear abstractions, meaningful names, and readability, offering fresh insights into developer acceptance and practical impact.

XI. Conclusion

In this work, we introduce a novel test smell, Disjoint Assertion Tangle (DAT), defined as having multiple independent assertion clusters within one test method. Our automated tool, U2W, identified and refactored DAT instances in 47 of 49 projects analyzed (8.59% of tests), restructuring 3,638 tests into 31,837 test cases, and grouping 14,343 cases into 1,713 parameterized unit tests, reducing test code lines by 36.33%. Developer feedback confirms that DAT refactoring aligns well with practitioner preferences. Our study highlights the tradeoffs involved in test parameterization and emphasizes the need for practitioner awareness of suboptimal test designs. In future work, we aim to investigate adaptive refactoring approaches that incorporate developer feedback or project-specific conventions to better align with human-centric test refactoring.

Acknowledgments

We thank all study participants, Professor Darko Marinov for his guidance, Shubhi Jain for her help, and the Spider Lab for its supportive research environment.

REFERENCES

[1] A. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," 08 2001.

[2] M. Fowler, *Refactoring: improving the design of existing code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 4–15.

[4] D. Campos, L. Soares Bastos, and I. Machado, "Developers perception on the severity of test smells: an empirical study," 07 2021.

[5] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Softw. Engg.*, vol. 20, no. 4, p. 1052–1094, Aug. 2015. [Online]. Available: https://doi.org/10.1007/s10664-014-9313-0

[6] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Test smells 20 years later: detectability, validity, and reliability," *Empirical Software Engineering*, vol. 27, no. 7, p. 170, 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10207-5

[7] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the relation of test smells to software code quality," in *2018 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2018, pp. 1–12.

[8] M. Aniche, C. Treude, and A. Zaidman, "How developers engineer test cases: An observational study," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4925–4946, 2022.

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

[10] B. Van Rompaey, B. Du Bois, and S. Demeyer, "Characterizing the relative significance of a test smell," in *2006 22nd IEEE International Conference on Software Maintenance*, 2006, pp. 391–400.

[11] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, 2007.

[12] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 56–65.

[13] J. De Bleser, D. Di Nucci, and C. De Roover, "Socrates: Scala radar for test smells," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, ser. Scala '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 22–26. [Online]. Available: https://doi.org/10.1145/3337932.3338815

[14] S. Reichhart, T. Gîrba, and S. Ducasse, "Rule-based assessment of test quality." *Journal of Object Technology*, vol. 6, pp. 231–251, 10 2007.

[15] F. Palomba, A. Zaidman, and A. De Lucia, "Automatic test smell detection using information retrieval techniques," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 311–322.

[16] G. Galindo-Gutierrez, M. N. Carvajal, A. Fernandez Blanco, N. Anquetil, and J. P. Sandoval Alcocer, " A manual categorization of new quality issues on automatically-generated tests ," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2023, pp. 271–281. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSME58846.2023.00035

[17] M. Narang, H. Du, and J. A. Jones, "U2W artifact," https://github.com/spideruci/U2W, 2025.

[18] S. Thummalapenta, M. R. Marri, T. Xie, N. Tillmann, and J. de Halleux, "Retrofitting unit tests for parameterized unit testing," in *Fundamental Approaches to Software Engineering*, D. Giannakopoulou and F. Orejas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 294–309.

[19] L. Martins, H. Costa, and I. Machado, "On the diffusion of test smells and their relationship with test code quality of java projects," *Journal of Software: Evolution and Process*, vol. 36, no. 4, p. e2532, 2024. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2532

[20] S. Lambiase, A. Cupito, F. Pecorelli, A. De Lucia, and F. Palomba, "Just-in-time test smell detection and refactoring: The darts project," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 441–445. [Online]. Available: https://doi.org/10.1145/3387904.3389296

[21] V. Pontillo, D. A. d'Aragona, F. Pecorelli, D. D. Nucci, F. Ferrucci, and F. Palomba, "Machine learning-based test smell detection," 2022. [Online]. Available: https://arxiv.org/abs/2208.07574

[22] G. Galindo-Gutierrez, "Automatically generating single-responsibility unit tests," 04 2025.

[23] H. K. V. Tran, N. Ali, M. Unterkalmsteiner, and J. Börstler, "A proposal and assessment of an improved heuristic for the eager test smell detection," 01 2024.

[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 *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1650–1654. [Online]. Available: https://doi.org/10.1145/3368089.3417921

[25] B. A. Galler and M. J. Fisher, "An improved equivalence algorithm," *Commun. ACM*, vol. 7, no. 5, p. 301–303, May 1964. [Online]. Available: https://doi.org/10.1145/364099.364331

[26] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 52–63. [Online]. Available: https://doi.org/10.1145/2635868.2635906

[27] Hacker News, "Discussion on assertion roulette," https://news.ycombinator.com/item?id=33479397, 2022, accessed: 2025-05-08.

[28] C. Roy and J. Cordy, "A survey on software clone detection research," *School of Computing TR 2007-541*, 01 2007.

[29] W. Aljedaani, A. Peruma, A. Aljohani, M. Alotaibi, M. W. Mkaouer, A. Ouni, C. D. Newman, A. Ghallab, and S. Ludi, "Test smell detection tools: A systematic mapping study," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 170–180. [Online]. Available: https://doi.org/10.1145/3463274.3463335

[30] D. Spadini, M. Schvarcbacher, A.-M. Oprescu, M. Bruntink, and A. Bacchelli, "Investigating severity thresholds for test smells," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 311–321.

[31] C. Li, A. Baz, and A. Shi, "Reducing test runtime by transforming test fixtures," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1757–1769. [Online]. Available: https://doi.org/10.1145/3691620.3695541

[32] H. Du, V. K. Palepu, and J. A. Jones, " Leveraging Propagated Infection to Crossfire Mutants ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 687–687. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00150

[33] ——, "Ripples of a mutation — an empirical study of propagation effects in mutation testing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639179

[34] O. Legunsen, A. Shi, and D. Marinov, "Starts: Static regression test selection," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17. IEEE Press, 2017, p. 949–954.

[35] C. Li, M. M. Khosravi, W. Lam, and A. Shi, "Systematically producing test orders to detect order-dependent flaky tests," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 627–638. [Online]. Available: https://doi.org/10.1145/3597926.3598083

[36] P. Nie, A. Celik, M. Coley, A. Milicevic, J. Bell, and M. Gligoric, "Debugging the performance of maven's test isolation: experience report," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 249–259. [Online]. Available: https://doi.org/10.1145/3395363.3397381

[37] A. Shi, P. Zhao, and D. Marinov, "Understanding and improving regression test selection in continuous integration," in *2019 IEEE 30th*

International Symposium on Software Reliability Engineering (ISSRE), 2019, pp. 228–238.

[38] OpenAI, "Gpt-4o mini," https://platform.openai.com, May 2025, large language model, accessed via OpenAI API using LangChain.

[39] J. A. Bargh, M. Chen, and L. Burrows, "Automaticity of social behavior: Direct effects of trait construct and stereotype activation on action." *Journal of personality and social psychology*, vol. 71, no. 2, p. 230, 1996.

[40] J. Robinson, *Likert Scale*. Dordrecht: Springer Netherlands, 2014, pp. 3620–3621. [Online]. Available: https://doi.org/10.1007/978-94-007-0753-5_1654

[41] N. Tillmann and W. Schulte, "Unit tests reloaded: parameterized unit testing with symbolic execution," *IEEE Software*, vol. 23, no. 4, pp. 38–47, 2006.

[42] N. Tillmann, J. de Halleux, and T. Xie, "Parameterized unit testing: theory and practice," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 483–484. [Online]. Available: https://doi.org/10.1145/1810295.1810441

[43] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "Orbs and the limits of static slicing," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015, pp. 1–10.

[44] A. Pizzini, "Behavior-based test smells refactoring: toward an automatic approach to refactoring eager test and lazy test smells," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 261–263. [Online]. Available: https://doi.org/10.1145/3510454.3517059

[45] Y. Yang, X. Hu, X. Xia, and X. Yang, "The lost world: Characterizing and detecting undiscovered test smells," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 3, Mar. 2024. [Online]. Available: https://doi.org/10.1145/3631973

[46] J. Delplanque, S. Ducasse, G. Polito, A. P. Black, and A. Etien, "Rotten green tests," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 500–511. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00062

[47] M. Greiler, A. Van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 322–331.

[48] R. Lima, K. Costa, J. Souza, L. Teixeira, B. Fonseca, M. d'Amorim, M. Ribeiro, and B. Miranda, "Do you see any problem? on the developers perceptions in test smells detection," in *Proceedings of the XXII Brazilian Symposium on Software Quality*, 2023, pp. 21–30.

[49] K. Tsukamoto, Y. Maezawa, and S. Honiden, "Autoput: an automated technique for retrofitting closed unit tests into parameterized unit tests," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1944–1951. [Online]. Available: https://doi.org/10.1145/3167132.3167340