Jurnal Nasional Teknik Elektro dan Teknologi Informasi
Volume 15 Number 1 February 2026

DOI: 10.22146/jnteti.v15i1.24073

Test-First Protocol for Deriving Unit Tests from Use Case Specifications

Muhammad Ridho Kurniawan Pratama[1], Deni Utama[1], Rauhil Fahmi[1]

[1] Information Systems and Technology Study Program, Faculty of Engineering, Universitas Negeri Jakarta, Jakarta, DKI Jakarta 13220, Indonesia

[Received: 24 September 2025, Revised: 12 December 2025, Accepted: 2 February 2026]
Corresponding Author: Muhammad Ridho Kurniawan Pratama (email: muhammadridho@unj.ac.id)

ABSTRACT — Early and systematic derivation of unit test scenarios remains challenging in software engineering, particularly in aligning functional requirements with executable tests. Graduate-level observations reveal that most students operate without granular traceability, standardized structures, or alternate flow testing. This study explored a structured test-first protocol that transformed use case specifications into coverage-aware test scenarios by applying object-oriented analysis and design principles. The protocol integrated sequence diagrams via behavioral modeling. Internal logic was extracted from sequence diagrams and visualized using control flow graphs. Basis path testing identified independent paths, serving as foundations for deriving unit test cases using the arrange-act-assert pattern. The "Pay the Order" use case in a hypothetical e-commerce system demonstrated the feasibility of the protocol. Cyclomatic complexity analysis yielded a complexity of 2, indicating that two independent test paths were required for complete coverage. The protocol successfully derived two-unit test cases with 100% basis path coverage, demonstrating complete traceability from functional requirements to unit test scenarios with one-to-one mapping between control flow paths and test cases. Results highlight the protocol's ability to support early verification and validation processes. Unlike prior works focused on automated system-level test generation, this protocol offers a lightweight, human-centric approach promoting testability, traceability, and strong semantic alignment between requirements and implementation. The protocol is well-suited for educational settings and environments that prioritize traceability. Future research should pursue empirical validation, scalability investigations, semi-automated tool development, domain generalization across paradigms, and longitudinal impact assessment.

KEYWORDS — Unit Test, Basis Path Testing, Use Case-Based Testing, Object-Oriented Analysis and Design.

I. INTRODUCTION

Ensuring software quality from the earliest stages of development is a central concern in software engineering. Among various approaches to maintain software quality, test-driven development (TDD) has gained attention for its test-first nature to support early validation and to drive analysis, design and programming decision [1], increase the developer productivity and speed [2], [3], improving engagement level [4] leads to the better maintainability [3], [4], create the higher deliverable quality and achieve satisfaction [3], [5]. However, TDD practices are often performed informally, and there is no systematic traceability between early requirements modelling and concrete unit-level test design.

In parallel, use case modelling has become a well-established method in requirements engineering for capturing system functionality from a user's perspective. Use cases are often considered high-level artifacts suitable for deriving system-level or integration tests [6], [7]. However, their potential to guide unit test case derivation, especially in the context of object-oriented analysis and design (OOAD), remains underexplored. These challenges transcend theoretical concerns. Analysis of graduate student work across multiple iterations of an Information Systems Requirements Analysis and Design course revealed systematic difficulties in deriving unit tests from use case specifications. Three recurring patterns emerged consistently.

First, traceability between artifacts was weak. Students typically referenced high-level functional requirements without establishing granular links to specific use case steps or behavioral model interactions. Test documentation often stated general purposes without indicating which use-case steps or sequence-diagram messages were being validated, making systematic coverage verification nearly impossible.

Second, test case structures lacked standardization. Submissions mixed setup activities, method execution, and outcome verification into sequential narratives without clear boundaries. For instance, test procedures might combine "call validation function with empty parameters, observe returned messages" without distinguishing input arrangement, execution, and assertion logic. Some formats compressed all test phases into a single cell, obscuring the logical separation between test arrangement, action, and assertion.

Third, path coverage remained incomplete. Students predominantly tested successful execution paths while omitting documented alternate flows explicitly. Critically, no students applied systematic techniques, such as basis path analysis, to determine test completeness. When asked to justify the test suite size, responses relied on subjective judgment rather than analytical reasoning grounded in control-flow analysis. These observations demonstrate a systematic gap between requirements modelling and unit test derivation that our protocol directly addresses.

To address these pedagogical and practical challenges systematically, model-driven development (MDD) offers a promising pathway. By leveraging behavioral and structural Unified Modeling Language (UML) models such as sequence diagrams, class diagrams, and behavioral state machine diagrams, MDD supports systematic reasoning about software behavior and architecture [8]–[10]. When aligned with white-box strategies such as basis path testing and combined with cyclomatic complexity calculation [11], these models can

provide a structured basis for extracting test logic from early design artifacts.

Despite the established benefits of TDD and MDD approaches, a critical gap persists in both literature and practice. Prior research has predominantly focused on automated test generation at the system or integration level [6], [7], [12], [13], leaving unit-level test derivation from use cases systematically underexplored. Existing model-driven testing approaches tend to emphasize automation at the expense of semantic alignment [9], [14], while others target hardware and cyber-physical systems [15], [16] rather than object-oriented software units. Although basis path analysis [11] provides rigorous coverage criteria and use case modeling effectively captures functional requirements, no prior work has systematically integrated these techniques with OOAD principles to derive unit tests with explicit requirement-to-test traceability.

The pedagogical challenges observed in educational settings, where students consistently fail to establish granular traceability and systematic coverage, further underscore the practical need for such integration. This research addresses this void through a human-centric, traceability-focused protocol that bridges requirements modeling and unit test derivation.

The protocol's novelty emerges from its systematic integration of use case modeling, behavioral modeling via sequence diagrams, control flow analysis, and basis path testing into a unified, replicable workflow. This contribution offers significant value across three dimensions: providing a pedagogical framework for teaching systematic test derivation in educational contexts, establishing auditable traceability suitable for regulated industries, and preserving semantic alignment between requirements and tests that automated approaches often sacrifice. By maintaining human interpretability throughout the transformation process, the protocol proves particularly valuable for contexts that require a deep understanding of the relationships between functional specifications and validation artifacts, distinguishing it from automation-focused prior work [9], [14], [17], [18].

This paper presents a structured test-first protocol that transforms use case specifications into unit test scenarios by systematically integrating use case modeling, sequence diagrams, control flow analysis, and basis path testing. The protocol ensures complete traceability from functional requirements to test scenarios while maintaining semantic consistency between design models and validation artifacts. Unlike automated approaches, this human-centric methodology preserves interpretability for educational, regulated, or documentation-intensive environments. A hypothetical e-commerce case study validates the protocol's feasibility, demonstrating how use cases decompose into sequence diagrams, translate into control flow graphs, and generate complete, traceable unit test scenarios.

II. RELATED WORKS

A. USE CASE BASED TESTING

Use case modelling has long served as a bridge between requirements and testing, especially at the system or integration level. A structured approach was proposed to generate test cases from use case contracts, demonstrating how requirement artifacts could be directly mapped to system-level test scenarios [6]. Similarly, the transformation of the use case diagram to system test scenario from activity diagram, sequence diagram, and data variation equivalence classes was introduced [7]. Advancing the formalization, the domain-specific language called test case specification language (TCSL) and use case specification language (USL) was developed and used to derive the test cases using the use case specification language-based test generation (USLTG) method [12]. In parallel, machine learning applications further advanced use-case-based testing. A natural language processing (NLP)-based approach was used to derive acceptance test cases from the use case specifications [13]. While these works focus on higher-level testing, they reinforce the argument for use-case-centric testing processes, which this study applies at the unit level.

B. MODEL DRIVEN TESTING AND CONTROL FLOW ANALYSIS

The MDD advocates using formal system models, such as UML diagrams, to drive design and generate testing artifacts, including unit and application programming interface (API) test cases [19]. Test paths can be derived from UML models using traversal algorithms, such as depth-first search (DFS), highlighting the role of sequence and activity diagrams in informing test logic [9]. The protocol presented in this paper adopts a similar philosophy while emphasizing manual derivation to promote understanding and maintain semantic alignment.

Recent efforts have incorporated artificial intelligence techniques, specifically ant colony optimization, to automate test case generation from activity diagrams [14]. State chart diagrams, combined with formal methods, derive integration tests for distributed systems [10]. Beyond traditional UML diagrams, user interface testing tools enhance static model construction and leverage approaches to retrieve runtime information [20]. Model-driven testing extends to hardware verification, automating hardware-in-loop test generation [15] and validating industrial cyber-physical systems [16] using tools such as GraphWalker.

Traditional testing approaches, such as differential testing [21], program analysis [22], and mutation analysis [23], [24], [25], can be integrated with large language models (LLMs). Novel approaches use LLMs to retrieve relevant inputs from bug reports, improving test generation [17]. ChatGPT-based iterative enhancement tools increase the number compliant tests generated by over 20% [18]. While this signals a trend toward automation, this work emphasizes traceable, analyst-driven derivation to preserve fidelity between functional design and unit validation.

To support unit-level validation, this protocol integrates basis path testing, a white-box approach rooted in control flow analysis. Cyclomatic complexity [11] quantifies the number of independent paths in a program, which forms the core of our structural testing process. Control flow graphs constructed from behavioral models yield the minimum set of test cases required for full path coverage. Mapping object interactions from sequence diagrams to control logic and applying basis path analysis ensures that each independent logic path is exercised by at least one test case, enabling early detection of logic defects.

C. REQUIREMENT TRACEABILITY IN TESTING

Requirements traceability has emerged as critical in software quality assurance, particularly in regulated industries requiring audit trails. Traditional approaches struggle with maintaining bidirectional traceability between requirements and test artifacts [26]–[31], while matrix-based methods suffer from scalability issues despite being comprehensive [32]–[34].

Recent advances in automated traceability link generation using NLP have shown promise; however, they lack the semantic depth required for unit-level testing [35]–[39]. This gap highlights the need for human-interpretable, design-time traceability approaches that maintain semantic consistency throughout the development lifecycle.

D. TEST-FIRST DEVELOPMENT

Test-first development, particularly TDD, yields significant improvements in software design and quality compared to test-last approaches. TDD encourages frequent incremental refactoring, leading to cleaner, more modular code and reduced technical debt [40]–[42]. Enhanced TDD methodologies that integrate testability measurement and refactoring techniques yield 77.8% improvements in class testability [41]. Writing tests-first enables early defect discovery, reducing costs and debugging time, ultimately resulting in productivity enhancements [41], [43]–[45]. AI-supported test automation further increases coverage and accuracy [43]. However, test-first development faces challenges. Small teams find that the approach has a steep learning curve [46], necessitating iterative cycles [47], systematic learning, standardized curricula [48], and proper mentoring [46]. Implementation results are affected by insufficient familiarity and low adherence with the methodology [49], [50]. To mitigate the identified challenges, a systematic protocol is needed to provide structured guidance for deriving test cases from use case specifications, thereby offering the scaffolding for developers to effectively implement the approach.

III. METHODOLOGY

A. OVERVIEW

This protocol systematically transformed use case specifications into unit test cases through four sequential phases, ensuring complete traceability from functional requirements to executable test scenarios. Phase 1 (use case modelling) established functional requirements through structured use case specifications. Phase 2 (behavioral modeling) transformed narratives into sequence diagrams that depicted object interactions using the entity-control-boundary pattern. Phase 3 (logic extraction and control flow analysis) derived algorithmic logic from sequence diagrams, constructed control flow graphs, calculated cyclomatic complexity ($V(G)$) and identified independent execution paths. Phase 4 (unit test case design) transformed these paths into executable test cases using the arrange-act-assert pattern with explicit traceability links to use case steps and sequence diagram messages.

B. PHASE 1: USE CASE MODELLING

Use case modelling bridged communication between system analysts and stakeholders during requirements analysis. The modeling involved creating a detailed use case specification that described actor-system interactions through structured narratives. The specification documented normal flows (primary success scenarios) and alternate flows (exception scenarios) using numbered steps, enabling precise traceability to subsequent artifacts.

C. PHASE 2: BEHAVIORAL MODELLING

High-level functional requirements were transformed into lower-level system interactions using sequence diagrams. Sequence diagrams illustrated the communication and information exchange between internal system objects, capturing both the behavioral flow and the participating classes along with their method signatures.

D. PHASE 3: LOGIC EXTRACTION, AND CONTROL FLOW ANALYSIS

The algorithmic logic of each object was derived from sequence diagram messages, which were then translated into implementation steps. Control flow graphs were constructed to model decision points and execution flow, representing statements as nodes and flow between statements as edges [51]. Using the control flow graph, basis path analysis was then applied to identify the independent paths of the logic.

Cyclomatic complexity, denoted as $V(G)$, quantifies the number of independent paths and represents the minimum number of test cases needed to ensure all paths are exercised. The cyclomatic complexity was calculated using (1).

$$V(G) = E - N + 2P \quad (1)$$

where E represents the number of edges in the control flow graph, N represents the number of nodes, and P represents the number of connected components. For a single function or method, P is always equal to 1.

E. PHASE 4: UNIT TEST CASE AND SCENARIO DESIGN

After identifying the independent paths, the next step was to design a test case that ensured each path was executed. Each test case should be explicitly associated with a distinct control path and include clearly defined input conditions and expected outcomes. The test scenario could be systematically structured using the arrange-act-assert pattern [52], providing a natural and intuitive flow for defining unit tests and enhancing test maintainability [53]. In the arrange phase, testers set up the required preconditions and inputs necessary for the test scenario to proceed. The act phase then executed the unit under test with these preconditions in place. Finally, the assert phase compared the actual result against the expected result to determine correctness. A test was considered to have failed when the observed outcome deviated from the expected result. This structure promoted clarity, repeatability, and alignment with unit testing best practices across different languages and toolchains.

IV. CASE STUDY: DERIVING UNIT TESTS FROM THE "PAY THE ORDER" USE CASE

A. CASE STUDY OVERVIEW

This case study demonstrated the protocol's application through the "Pay the Order" use case from a hypothetical e-commerce system (XYZ Mart). The use case was selected for its balance of demonstrative value and analytical clarity, representing a critical business function with sufficient complexity (V(G)=2) to exercise all protocol phases while remaining comprehensible. The case study validated protocol feasibility by transforming a realistic use case into complete unit test scenarios, examined the quality of traceability through explicit linkages from use case steps to test assertions, and verified coverage completeness by confirming that derived test cases achieve 100% path coverage as predicted by cyclomatic complexity analysis.

B. USE CASE SPECIFICATION

The "Pay the Order" use case encompassed normal and alternate execution flows. The normal flow (NF) began when the buyer verified cart contents (NF-1) and proceeded to payment (NF-2). The system displayed available payment

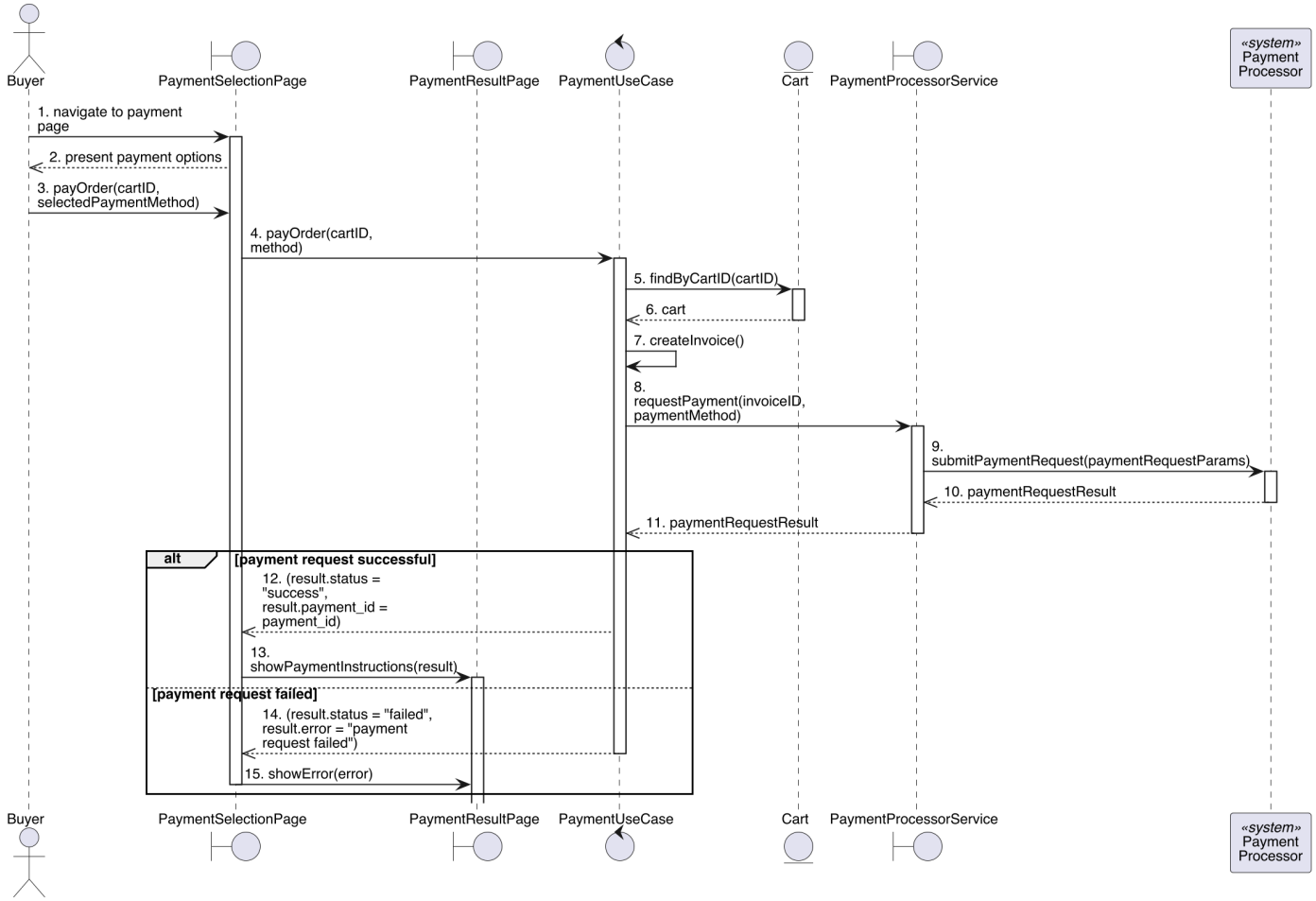

Figure 1. Pay the Order sequence diagram.

methods (NF-3), the buyer selected a method (NF-4), the system submitted the payment request to the payment processor (NF-5), and the system then presented payment instructions (NF-6). After the buyer completed payment (NF-7), the payment processor returned a successful notification, triggering "Change Order Status" (NF-8). The system notified the buyer that the order was being processed (NF-9) and notified the seller about the paid order (NF-10).

Two alternate flows (AF) handled exceptions. AF-1 addressed payment request submission failures (branching from NF-5): the system displayed an error (AF-1.1) with the option to retry (AF-1.2). AF-2 addressed payment processing failures (branching from NF-8): the system triggers "Change Order Status" (AF-2.1), notified the buyer of failure and refund (AF-2.2), then terminated (AF-2.3).

C. BEHAVIOURAL MODELING

From the use case description, internal system interactions were derived using a sequence diagram, as shown in Figure 1. The entity control boundary pattern [54] segregated class responsibilities, ensuring separation of concerns and semantic consistency between user interactions and system responsibilities. The sequence diagram captured distinct messages spanning from initial cart validation through final payment confirmation.

Entity objects (cart and invoice) represented persistent business data and encapsulated domain-specific business rules. Control objects (*PaymentUseCase*) coordinated the workflow and implemented use-case-specific logic. Boundary objects (*PaymentSelectionPage* and *PaymentProcessorService*) managed external interfaces and system interactions, isolating core business logic from external dependencies.

D. LOGIC EXTRACTION AND CONTROL FLOW ANALYSIS

This study focused on the *PaymentUseCase* class, selected for its role in handling the core business logic of order payments and for its sufficient complexity to demonstrate the test case derivation methodology. From the sequence diagram messages 4-14, the algorithm was derived (shown in Listing I).

Listing I
Pay Order Pseudocode

```
PaymentUseCase.payOrder(cartID, method):
1. cart = Cart.findByCartID(cartID)
2. invoice = Invoice.createFromCart(cart)
3. result = processorService.requestPayment(
        invoice.id,
        invoice.amount,
        method)
4. if (result.failed) then
5.    return failure(result.error)
6. else
7.    return success(result.paymentID)
```

A control flow graph was constructed to visualize the algorithm's decision points and execution flow, as shown in Figure 2. Using the cyclomatic complexity formula, the graph obtained a cyclomatic complexity of 2. With one conditional branch (decision node 4), two independent paths existed, requiring two test cases to ensure complete basis path coverage.

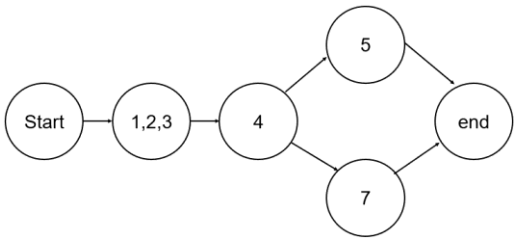

Figure 2. Control flow graph.

E. UNIT TEST CASE AND SCENARIO DESIGN

Based on the two independent paths, test cases in Listing II were designed using the arrange-act-assert pattern. The test scenario targeted the *PaymentUseCase* class' *payOrder()* method as the unit under test to maintain test isolation and adhere to the dependency inversion principle [55]. The PaymentProcessorService was mocked to simulate a payment processor's behavior, enabling independent verification of the PaymentUseCase's logic. The test format was implementation-agnostic, executable with any unit testing framework that supports object-oriented principles, aligning with test-first development and model-based engineering principles for early validation.

Listing II
Pay Order Unit Test Scenario

```
Test Suite: PaymentUseCase.payOrder()
Setup: Mock PaymentProcessorService with configurable
requestPayment() behavior

Run UT-01: Payment Request Fails
  Arrange: Mock returns status="FAILED"
  Act: payOrder("cart-001", "BANK_XYZ")
  Assert:
    - status="FAILED"
    - error="Payment Request Failed"

Run UT-02: Payment Request Succeeds
  Arrange: Mock returns status="SUCCESS",
payment_id=generate_id()
  Act: payOrder("cart-001", "BANK_XYZ")
  Assert:
    - status="SUCCESS"
    - error=null
    - payment_id=defined
```

V. RESULTS

The application of the proposed protocol to the "Pay the Order" use case yielded quantifiable metrics demonstrating systematic derivation of unit test scenarios from functional requirements. Control flow analysis of the *payOrder()* method yielded measurable outcomes in terms of structural complexity, test coverage, and traceability relationships, validating the protocol's ability to establish clear linkages between requirement specifications and executable test cases, while ensuring comprehensive path coverage through basis path testing methodology.

A. CONTROL FLOW METRICS

Table I summarizes the structural characteristics of the *payOrder()* method's control flow graph, which directly determined the minimum number of required test cases. The cyclomatic complexity of 2 indicated exactly 2 test cases were necessary and sufficient for complete basis path coverage, eliminating subjectivity in test suite sizing. This deterministic relationship between *V(G)* and required test count exemplified the protocol's strength in providing objective coverage criteria.

While this case study demonstrated the approach with *V(G)* = 2, the protocol scaled linearly with complexity: *V(G)* = N requires exactly N test cases, enabling predictable estimation of test effort. As a practical guideline, methods with *V(G)* > 10 should be considered candidates for refactoring into smaller, more testable components.

B. TEST CASE DERIVATION OUTCOMES

Table II documents the complete specification of derived test cases demonstrating a one-to-one correspondence between the independent execution paths identified through basis path analysis and the test scenarios structured using the arrange-act-assert pattern. Each test case corresponded to exactly one independent path through the control flow graph, confirming the mathematical relationship between cyclomatic complexity (*V(G)*=2) and required test count (2 test cases).

Traceability was maintained at multiple levels. Forward traceability linked each test case to specific use case steps (e.g., UT-01 validates NF4, NF5, and AF1.1), sequence diagram message exchanges (messages 4-14), and control flow paths (specific node sequences). Backward traceability enabled impact analysis: if the use case step AF1.1 changed, its mapping to UT-01 immediately identified the affected test cases that required updates. This explicit traceability addressed the systematic gap observed in student work, namely the absence of granular mappings between requirements and tests.

Table III quantifies the protocol's achievement in test completeness and requirement traceability, showing 100% coverage across path, statement, and branch dimensions. The one-to-one correspondence between cyclomatic complexity (*V(G)*=2) and test cases (2 tests) demonstrated that no redundant, yet complete coverage was achieved.

Complete bidirectional traceability, verified from use cases to tests and back, enabled precise impact analysis during maintenance. When the use case AF1.1 (payment request failure) changed, the mapping immediately identified UT-01 as requiring review. This explicit traceability addressed the pedagogical challenges observed, where students established no step-level requirement mappings. The protocol's deterministic nature ensured reproducibility: any analyst applying the protocol to the same use case would derive identical test cases with identical coverage, eliminating the subjectivity inherent in traditional test-first development, where developers rely on intuition rather than a mathematical foundation.

VI. DISCUSSION

A. CASE STUDY VALIDATION AND INTERPRETATION

The case study demonstrated how the protocol fulfills its design goals by establishing a traceable, model-driven approach linking high-level functional requirements to concrete unit-level test scenarios. The workflow transitioned from use case narratives through behavioral modelling via sequence diagrams, logic extraction formalized into pseudocode, control flow graph visualization, and basis path analysis. Test cases designed from independent paths ensured coverage of both functional expectations and internal logic branches, reinforcing requirement-to-test traceability and supporting early-stage validation.

The "Pay the Order" use case (*V(G)*=2) was selected for its balance of demonstrative value and analytical clarity,

TABLE I
CONTROL FLOW GRAPH STRUCTURAL METRICS

Metric	Value	Measurement Method	Interpretation
CFG Node (*N*)	6	Manual count from Figure 2	6 distinct statement blocks including entry/exit nodes
CFG Edges (*E*)	6	Manual count from Figure 2	6 control flow transitions between nodes
Connected Component (*P*)	1	Single method	One continuous control flow (no disconnected subgraphs)
Cyclomatic Complexity	2	$V(G) = E - N + 2P = 6 - 6 + 2(1) = 2$	Indicates 2 independent execution paths requiring test coverage

TABLE II
DERIVED TEST CASES WITH COMPLETE TRACEABILITY

Element	Test Case UT-01 (Failure Path)	Test Case UT-02 (Success Path)
Test Name	Payment Request Fails	Payment Request Succeeds
Independent Path	Start → 1,2,3 → 4 → 5 → End	Start → 1,2,3 → 4 → 7 → End
Triggering Condition	*PaymentProcessor* returns status="FAILED"	*PaymentProcessor* returns status="SUCCESS"
Input Setup (Arrange)	• *cartID* = "cart-001" • *paymentMethod* = "BANK_XYZ" • Mock: *processorService.requestPayment()* returns *{status: "FAILED", error: "Payment Request Failed"}*	• *cartID* = "cart-001" • *paymentMethod* = "BANK_XYZ" • Mock: *processorService.requestPayment()* returns *{status: "SUCCESS", paymentID: generated_id}*
Action (Act)	Execute: *payOrder("cart-001", "BANK_XYZ")*	Execute: *payOrder("cart-001", "BANK_XYZ")*
Expected Outcome (Assert)	• Return *status* = "FAILED" • Return *error* = "Payment Request Failed" • *Payment ID* = null	• Return *status* = "SUCCESS" • Return *error* = null • *Payment ID* = defined (non-null)
Traceability: Sequence Diagram	Messages 4, 5, 6, 7, 8, 11, 14 (Figure 1)	Messages 4, 5, 6, 7, 8, 11, 12 (Figure 1)
Traceability: Use Case	NF4 (Select Payment Method) NF5 (Submit Payment Request) AF1.1 (Payment Request Fails)	NF4 (Select Payment Method) NF5 (Submit Payment Request) NF8-10 (Successful Payment)
Method Under Test	*PaymentUseCase.payOrder()* (Listing I)	*PaymentUseCase.payOrder()* (Listing I)
Test Implementation	Listing II (Run UT-01)	Listing II (Run UT-02)

exercising all protocol phases while remaining comprehensible. The transformation process reveals several insights. First, the entity-control-boundary pattern facilitates clear separation of concerns during sequence diagram construction, enabling focused analysis on the *PaymentUseCase* control class while abstracting infrastructure details. Second, the protocol's deterministic nature becomes evident through direct correspondence between the use-case alternate flows (AF1.1, AF2.1) and control flow graph branches, illustrating how functional requirements systematically translate into structural test coverage. Third, the dependency inversion principle, as demonstrated through the *PaymentProcessorService*, highlights the protocol's compatibility with modern software design practices, enabling isolated unit testing while preserving the semantic intent of external service interactions.

However, the case study's relatively simple control flow ($V(G)$=2) may not adequately demonstrate the protocol's behavior with complex conditional logic, nested exception handling, or concurrent processing scenarios. The hypothetical nature of XYZ Mart eliminates real-world constraints such as legacy system integration, incomplete specifications, or evolving requirements. The one-to-one correspondence between independent paths and test cases, although mathematically sound, raises questions about practical maintainability in scenarios where business logic changes frequently, as the protocol's emphasis on early test derivation assumes relatively stable requirements.

B. ADDRESSING OBSERVED EDUCATIONAL CHALLENGES

The protocol addressed three systematic issues observed in graduate-level requirements analysis courses. Weak traceability was resolved through explicit mapping requirements in phases 2 and 4, mandating numbered sequence diagram messages from use case narratives with test cases documenting specific validations (e.g., "validates messages 4-8, 11-15; use case steps NF4, NF5, AF1.1"), transforming vague references into precise traceability links.

Incomplete path coverage was systematically addressed through cyclomatic complexity analysis in phase 3. By calculating $V(G)$ from control flow graphs extracted from sequence diagrams, the protocol provided objective test sufficiency criteria: $V(G)$ = N required exactly N test cases for complete basis path coverage. Each decision point in the use case narratives corresponded to control flow branches requiring test coverage, ensuring both normal and alternate flows were validated.

Inconsistent test structures were standardized through the mandatory arrange-act-assert pattern in phase 4. Rather than conflating setup, execution, and verification, the protocol required explicit separation: arrange documented preconditions and mocks; act contained one method invocation; and assert specified concrete outcomes. This standardization ensured reproducibility and made coverage gaps immediately visible.

Beyond producing higher-quality artifacts, the protocol provides pedagogical scaffolding by making explicit the transformation steps from requirements to tests, which experienced practitioners often perform intuitively. This helps students develop systematic reasoning about the relationships between functional specifications, design, and validation strategies.

C. PRACTICAL BENEFITS AND CONTRIBUTION

The proposed protocol bridges the gap between requirement modeling and early-stage unit testing, an area often overlooked in use-case-based and model-driven testing

TABLE III
COVERAGE AND TRACEABILITY ANALYSIS

Metric Category	Metric	Value	Calculation/Verification Method
Path Coverage	Independent Paths Identified	2	Basis path analysis from CFG
	Independent Paths Tested	2	Count of derived test cases (Table II)
	Basis Path Coverage	100%	(Paths Tested / Paths Identified) × 100 = (2/2) × 100%
Statement Coverage	Total CFG Nodes	6	Manual count from Figure 2
	Nodes Covered by Tests	6	Union of all paths: {Start, 1, 2, 3, 4, 5, 7, End}
	Node Coverage	100%	(Nodes Covered / Total Nodes) × 100 = (6/6) × 100%
Branch Coverage	Decision Points	1	Node 4 in CFG (if-else condition)
	Decision Outcomes	2	True branch (→7) and False branch (→5)
	Branches Covered	2	Both outcomes tested (UT-01: false, UT-02: true)
	Branch Coverage	100%	(Branches Covered / Decision Outcomes) × 100 = (2/2) × 100%
Traceability	Use Case Steps Total	6	NF4, NF5, NF8, NF9, NF10, AF1.1
	Use Case Steps Covered	6	All steps mapped to at least one test case
	Traceability Links Established	2	UT-01 → AF1.1; UT-02 → NF8-10
	Bidirectional Traceability	Complete	Forward (UC → Test) and Backward (Test → UC) verified

research. Unlike prior works focused on system-level scenarios and activity flow transformation [6], [7], this study emphasizes a unit-level focus through white-box testing via basis path analysis, ensuring test derivation in the design phase aligns with OOAD principles.

The protocol's emphasis on early test derivation aligns with the "shift-left" testing paradigm, offering quantifiable benefits, including a 35.86% reduction in mean defect density and a 76.19% decrease in mean change density when using TDD compared to TLD over two releases [56], directly impacting maintenance costs. Additionally, TDD supports faster product simplification, reducing complexity and maintenance costs [57].

The protocol addresses the semantic gap problem prevalent in automated testing approaches. While automation excels at execution speed, it often lacks the contextual understanding necessary for meaningful test case design. Our human-centric approach preserves domain knowledge and business logic understanding throughout the derivation process, ensuring test cases validate business-critical scenarios beyond technical coverage. The systematic transformation process also supports compliance with safety-critical domain requirements by maintaining explicit traceability artifacts at each transformation step and generating auditable documentation suitable for certification processes in the aerospace, medical device, and automotive domains.

The protocol is lightweight, tool-agnostic, and easy to implement in both academic and industry settings, providing a hands-on pathway for students and practitioners to understand how abstract requirements evolve into detailed unit tests. Leveraging test-first development in software engineering courses has been demonstrated to lead to improvements in both student grades and satisfaction [48]. For software teams in traceability-sensitive contexts, the protocol provides a structured, auditable approach to test design.

D. LIMITATIONS

This study presents several important limitations. First, the research constitutes a proof of concept based on a single hypothetical use case with relatively low complexity ($V(G)$=2), which does not establish effectiveness with complex real-world scenarios involving higher cyclomatic complexity, nested conditions, or intricate exception handling patterns.

Second, all models and transformations were created manually by researchers, potentially introducing bias into both artifact quality and semantic interpretation. The absence of independent validation by domain experts or practitioners' limits confidence in the protocol's reproducibility and consistency across different users. The manual nature of transformations raises scalability concerns for larger systems with dozens of interacting use cases.

Third, the study lacks empirical validation of the impacts on students and professionals. Without such validation, claims about the protocol's superiority in semantic preservation and traceability cannot be substantiated beyond theoretical arguments. Fourth, the research focuses exclusively on object-oriented systems with well-defined use case specifications, leaving unexplored applicability to other development paradigms, legacy systems with incomplete documentation, or domains with different modeling practices. Finally, the effectiveness of derived test cases in actual defect detection has not been empirically measured. While the protocol ensures basic path coverage, the relationship between this coverage and real-world improvements in software quality requires empirical investigation through controlled experiments with practicing developers and actual software projects.

VII. CONCLUSION

This study explored a test-first protocol for deriving unit test scenarios from use case specifications by systematically integrating use case modeling, sequence diagrams, control flow analysis, and basis path testing, ensuring complete traceability from requirements to tests. The case study validation on the "Pay the Order" use case achieved 100% basis path coverage with two test cases ($V(G)$ = 2), demonstrating one-to-one mapping between execution paths and test assertions.

The protocol addresses gaps in use-case-based and model-driven testing research through a human-centric methodology that preserves domain knowledge and interpretability, unlike automation-focused approaches. Graduate-level observations confirmed students' difficulties with step-level traceability and alternate flow testing.

Future research should focus on empirical validation for students and professionals, scalability investigations, semi-automated tool development, domain generalization, and

longitudinal impact assessments on maintenance and productivity.

CONFLICTS OF INTEREST

The authors declare that they have no conflicts of interest.

AUTHORS' CONTRIBUTIONS

Conceptualization, Muhammad Ridho Kurniawan Pratama; methodology, Muhammad Ridho Kurniawan Pratama; software, Deni Utama; validation, Muhammad Ridho Kurniawan Pratama, Deni Utama, Rauhil Fahmi; formal analysis, Muhammad Ridho Kurniawan Pratama; writing—original draft preparation, Muhammad Ridho Kurniawan Pratama; writing—reviewing and editing, Muhammad Ridho Kurniawan Pratama; visualization, Rauhil Fahmi; project administration, Muhammad Ridho Kurniawan Pratama; Authorship must be limited to those who have contributed substantially to the work reported.

REFERENCES

[1] D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43–50, Sep. 2005, doi: 10.1109/MC.2005.314.

[2] D.S. Janzen and H. Saiedian, "On the influence of test-driven development on software design," in *19th Conf. Softw. Eng. Educ. Train. (CSEET'06)*, 2006, pp. 141–148, doi: 10.1109/CSEET.2006.25.

[3] M.S. Rahman *et al.,* "Evaluating the impact of test-driven development on software quality enhancement," *Int. J. Math. Sci. Comput. (IJMSC)*, vol. 10, no. 3, pp. 51–76, Sep. 2024, doi: 10.5815/ijmsc.2024.03.05.

[4] W. Ren and S. Barrett, "Test – Driven development, engagement in activity, and maintainability: An empirical study," *IET Softw.*, vol. 17, no. 4, pp. 509–525, Jul. 2023, doi: 10.1049/sfw2.12135.

[5] V. Bhadauria, R. Mahapatra, and S. Nerur, "Performance Outcomes of Test-Driven Development: An Experimental Investigation," *J. Assoc. Inf. Syst.*, vol. 21, pp. 1045–1071, Jul. 2020, doi: 10.17705/1jais.00628.

[6] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, "Automatic test generation: A use case driven approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 140–155, Mar. 2006, doi: 10.1109/TSE.2006.22.

[7] B. Hasling, H. Goetz, and K. Beetz, "Model based testing of system requirements using UML use case models," in *2008 1st Int. Conf. Softw. Test. Verif. Valid. Lillehammer*, 2008, pp. 367–376, doi: 10.1109/ICST.2008.9.

[8] B. Rumpe, "Model-Based Testing of Object-Oriented Systems," in *Form. Methods Compon. Objects*, F.S. de Boer, M.M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. 2003, pp. 380–402.

[9] Meiliana *et al.,* "Automated test case generation from UML activity diagram and sequence diagram using depth first search algorithm," *Procedia Comput. Sci.*, vol. 116, pp. 629–637, Oct. 2017, doi: 10.1016/j.procs.2017.10.029.

[10] B. Graics, M. Mondok, V. Molnár, and I. Majzik, "Model-based testing of asynchronously communicating distributed controllers using validated mappings to formal representations," *Sci. Comput. Program.*, vol. 242, pp. 1–40, May 2025, doi: 10.1016/j.scico.2025.103265.

[11] T.J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, doi: 10.1109/TSE.1976.233837.

[12] C.T.M. Hue, D.-H. Dang, N.N. Binh, and A.-H. Truong, "USLTG: Test case automatic generation by transforming use cases," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 29, no. 9, pp. 1313–1345, Sep. 2019, doi: 10.1142/S0218194019500414.

[13] C. Wang, F. Pastore, A. Goknil, and L.C. Briand, "Automatic generation of acceptance test cases from use case specifications: An NLP-based approach," *IEEE Trans. Softw. Eng.,* vol. 48, no. 2, pp. 585-616, Feb. 2022, doi: 10.1109/TSE.2020.2998503.

[14] P. Jha *et al.,* "Application of machine learning in software testing of healthcare domain," in *Proc. 7th Int. Conf. Adv. Comput. Intell. Eng.*, 2024, pp. 63–73, doi: 10.1007/978-981-99-5015-7_6.

[15] V.A. Karlsson *et al.,* "Automation of the creation and execution of system level hardware-in-loop tests through model-based testing," in *A-TEST 2022, Proc. 13th Int. Workshop Autom. Test Case Des. Sel. Eval.*, 2022, pp. 9–16, doi: 10.1145/3548659.3561313.

[16] M.N. Zafar *et al.,* "Model-based testing in practice: An industrial case study using graphwalker," in *ISEC '21, Proc. 14th Innov. Softw. Eng. Conf. (Former. Known India Softw. Eng. Conf.)*, 2021, pp. 1–11, doi: 10.1145/3452383.3452388.

[17] W.C. Ouédraogo *et al.,* "Enriching automatic test case generation by extracting relevant test inputs from bug reports," *Empir. Softw. Eng.*, vol. 30, no. 3, pp. 1–60, Mar. 2025, doi: 10.1007/s10664-025-10635-z.

[18] Z. Yuan *et al.,* "Evaluating and improving ChatGPT for unit test generation," in *Proc. ACM Softw. Eng.*, 2024, pp. 1703–1726, doi: 10.1145/3660783.

[19] A. Deljouyi and R. Ramsin, "MDD4REST: Model-driven methodology for developing RESTful web services," in *Proc. 10th Int. Conf. Model-Driven Eng. Softw. Dev.*, 2022, pp. 93–104, doi: 10.5220/0011006300003119.

[20] S. Wu *et al.,* "CydiOS: A model-based testing framework for iOS apps," in *ISSTA 2023, Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2023, pp. 1–13, doi: 10.1145/3597926.3598033.

[21] Y. Deng *et al.,* "TARGET: Automated scenario generation from traffic rules for testing autonomous vehicles via validated LLM-guided knowledge extraction," *IEEE Trans. Softw. Eng.,* vol. 51, no. 7, pp. 1950–1968, Jul. 2025, doi: 10.1109/TSE.2025.3569086.

[22] A.E.I. Brownlee *et al.,* "Enhancing genetic improvement mutations using large language models," in *Search-Based Software Engineering*, 2024, pp. 153–159, doi: 10.1007/978-3-031-48796-5_13.

[23] J. Ackerman and G. Cybenko, "Large language models for fuzzing parsers (registered report)," in *FUZZING 2023, Proc. 2nd Int. Fuzzing Workshop*, 2023, pp. 31–38, doi: 10.1145/3605157.3605173.

[24] C. Tsigkanos, P. Rani, S. Müller, and T. Kehrer, "Variable discovery with large language models for metamorphic testing of scientific software," in *Comput. Sci. – ICCS 2023*, 2023, pp. 321–335, doi: 10.1007/978-3-031-35995-8_23.

[25] M.L. Siddiq *et al.,* "Using large language models to generate JUnit tests: An empirical study," in *EASE '24, Proc. 28th Int. Conf. Eval. Assess. Softw. Eng.*, 2024, pp. 313–322, doi: 10.1145/3661167.3661216.

[26] R. Settimi *et al.,* "Supporting software evolution through dynamically retrieving traces to UML artifacts," in *Proc., 7th Int. Workshop Princ. Softw. Evol.*, 2004, pp. 49–54, doi: 10.1109/IWPSE.2004.1334768.

[27] N. Narayan, Y. Li, J. Helming, and M. Koegel, "Interaction centric requirements traceability," in *Proc. 6th Int. Conf. Eval. Nov. Approaches Softw. Eng.*, 2011, pp. 232–238, doi: 10.5220/0003463502320238.

[28] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *2017 IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, 2017, pp. 3–14, doi: 10.1109/ICSE.2017.9.

[29] M. Rahimi and J. Cleland-Huang, "Evolving software trace links between requirements and source code," *Empir. Softw. Eng.*, vol. 23, pp. 2198–2231, Aug. 2018, doi: 10.1007/s10664-017-9561-x.

[30] D. Fucci, E. Alégroth, and T. Axelsson, "When traceability goes awry: An industrial experience report," *J. Syst. Softw.*, vol. 192, pp. 1–10, Oct. 2022, doi: 10.1016/j.jss.2022.111389.

[31] C.C. Rațiu, C. Mayr-Dorn, W.K.G. Assunção, and A. Egyed, "Taming cross-tool traceability in the wild," in *2023 IEEE 31st Int. Requir. Eng. Conf. (RE)*, 2023, pp. 233–243, doi: 10.1109/RE57278.2023.00031.

[32] X. Chen, J. Hosking, J. Grundy, and R. Amor, "DCTracVis: A system retrieving and visualizing traceability links between source code and documentation," *Autom. Softw. Eng.*, vol. 25, pp. 703–741, Dec. 2018, doi: 10.1007/s10515-018-0243-8.

[33] C. Mills, J. Escobar-Avila, and S. Haiduc, "Automatic traceability maintenance via machine learning classification," in *2018 IEEE Int. Conf. Softw. Maint. Evol. (ICSME)*, 2018, pp. 369–380, doi: 10.1109/ICSME.2018.00045.

[34] J. Lin, Y. Liu, and J. Cleland-Huang, "Supporting program comprehension through fast query response in large-scale systems," in *ICPC '20, Proc. 28th Int. Conf. Program Compr.*, 2020, pp. 285–295, doi: 10.1145/3387904.3389260.

[35] T. Hey, F. Chen, S. Weigelt, and W.F. Tichy, "Improving traceability link recovery using fine-grained requirements-to-code relations," in *2021 IEEE Int. Conf. Softw. Maint. Evol. (ICSME)*, 2021, pp. 12–22, doi: 10.1109/ICSME52107.2021.00008.

[36] A. Kicsi, V. Csuvik, and L. Vidacs, "Large scale evaluation of natural language processing based test-to-code traceability approaches," *IEEE Access,* vol. 9, pp. 79089–79104, May 2021, doi: 10.1109/ACCESS.2021.3083923.

[37] C.D. Laliberte, R.E. Giachetti, and M. Kolsch, "Evaluation of natural language processing for requirements traceability," in *2022 17th Annu. Syst. Syst. Eng. Conf. (SOSE)*, 2022, pp. 21–26, doi: 10.1109/SOSE55472.2022.9812649.

[38] W. Sun *et al.,* "Method-level test-to-code traceability link construction by semantic correlation learning," *IEEE Trans. Softw. Eng.*, vol. 50, no. 10, pp. 2656–2676, Oct. 2024, doi: 10.1109/TSE.2024.3449917.

[39] X. Jin *et al.,* "Tracets4J: A traceable unit test generation dataset," in *2025 IEEE Int. Conf. Softw. Anal. Evol. Reengineering (SANER)*, 2025, pp. 757–767, doi: 10.1109/SANER64311.2025.00077.

[40] E. Lautenschläger, "The perception of test driven development in computer science – Outline for a structured literature review," in *Bus. Inf. Syst. Workshops*, 2022, pp. 121–126, doi: 10.1007/978-3-031-04216-4_13.

[41] S. Parsa, M. Zakeri-Nasrabadi, and B. Turhan, "Testability-driven development: An improvement to the TDD efficiency," *Comput. Stand. Interfaces*, vol. 91, pp. 1–25, Jan. 2025, doi: 10.1016/j.csi.2024.103877.

[42] F. Uyaguari *et al.,* "Reliability of systematic literature reviews on test-Driven development," *Inf. Softw. Technol.*, vol. 184, pp. 1–23, Aug. 2025, doi: 10.1016/j.infsof.2025.107762.

[43] D.T. Penagos and N. Agudelo, "Agile testing using user language automation with artificial intelligence in Enjisst," in *2024 IEEE Lat. Am. Conf. Comput. Intell. (LA-CCI)*, 2024, pp. 1–5, doi: 10.1109/LA-CCI62337.2024.10814749.

[44] M.M. Moe and K.K. Oo, "Comparative results of dependent and independent variables focused on regression analysis using test-driven development," in *Proc. 2020 10th Int. Workshop Comput. Sci. Eng. (WCSE 2020)*, 2020, pp. 27–35, doi: 10.18178/wcse.2020.02.006.

[45] M. M. Moe and K. K. Oo, "Consequences of dependent and independent variables based on acceptance test suite metric using test driven development approach," in *2020 IEEE Conf. Comput. Appl. (ICCA)*, 2020, pp. 1–6, doi: 10.1109/ICCA49400.2020.9022828.

[46] H.A. Ramzan, S. Ramzan, and T. Kalsum, "Test-driven development (TDD) in small software development teams: Advantages and challenges," in *2024 5th Int. Conf. Adv. Comput. Sci. (ICACS)*, 2024, pp. 1–5, doi: 10.1109/ICACS60934.2024.10473291.

[47] P. Calais and L. Franzini, "Test-driven development benefits beyond design quality: Flow state and developer experience," in *2023 IEEE/ACM 45th Int. Conf. Softw. Eng.: New Ideas Emerg. Results (ICSE-NIER)*, 2023, pp. 106–111, doi: 10.1109/ICSE-NIER58687.2023.00025.

[48] F.G. Rocha, L.S. Souza, T.S. Silva, and G. Rodriguez, "Enhancing the student learning experience by adopting TDD and BDD in course projects," in *2021 IEEE Glob. Eng. Educ. Conf. (EDUCON)*, 2021, pp. 1116–1125, doi: 10.1109/EDUCON46332.2021.9453916.

[49] S. Romano *et al.,* "Results from a replicated experiment on the affective reactions of novice developers when applying test-driven development," in *Agile Process. Softw. Eng. Extreme Program.*, 2020, pp. 223–239, doi: 10.1007/978-3-030-49392-9_15.

[50] A. Santos *et al.,* "A family of experiments on test-driven development," *Empir. Softw. Eng.*, vol. 26, no. 3, pp. 1–53, Mar. 2021, doi: 10.1007/s10664-020-09895-8.

[51] D. Abuaiadah, M. Bosu, and S. Jayalal, "Revisiting the concepts of basis path testing," in *Inf. Syst. Intell. Syst. (ISBM 2024)*, 2025, pp. 503–515, doi: 10.1007/978-981-96-1747-0_41.

[52] V. Khorikov, *Unit Testing Principles, Practices and Patterns*. Shelter Island, NY, USA: Manning Publications Co., 2020.

[53] C. Wei *et al.,* "How do developers structure unit test cases? An empirical analysis of the AAA pattern in open source projects," *IEEE Trans. Softw. Eng.*, vol. 51, no. 4, pp. 1007–1038, Apr. 2025, doi: 10.1109/TSE.2025.3537337.

[54] I. Jacobson, *Object Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA, USA: Addison-Wesley, 1992.

[55] S. van Deursen and M. Seemann, *Dependency Injection Principles, Practices, and Patterns*. Shelter Island, NY, USA: Manning Publications Co., 2019.

[56] O.P.N. Slyngstad *et al.,* "The impact of test driven development on the evolution of a reusable framework of components – An industrial case study," in *2008 3rd Int. Conf. Softw. Eng. Adv.*, 2008, pp. 214–223, doi: 10.1109/ICSEA.2008.8.

[57] S. Bannerman and A. Martin, "A multiple comparative study of test-with development product changes and their effects on team speed and product quality," *Empir. Softw. Eng.*, vol. 16, no. 2, pp. 177–210, Apr. 2011, doi: 10.1007/s10664-010-9137-5.

This page is intentionally left blank