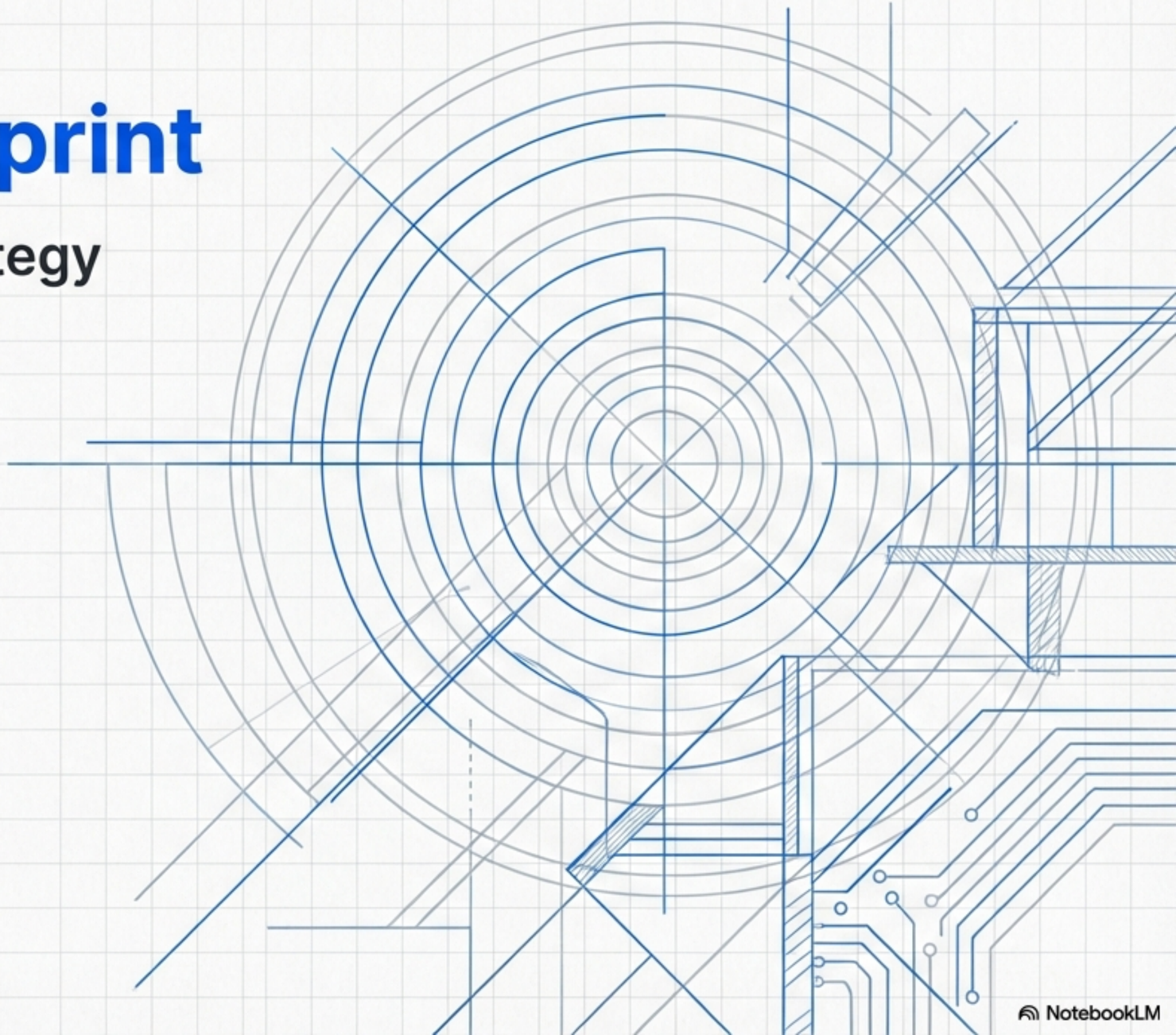
The Quality Engineering Blueprint

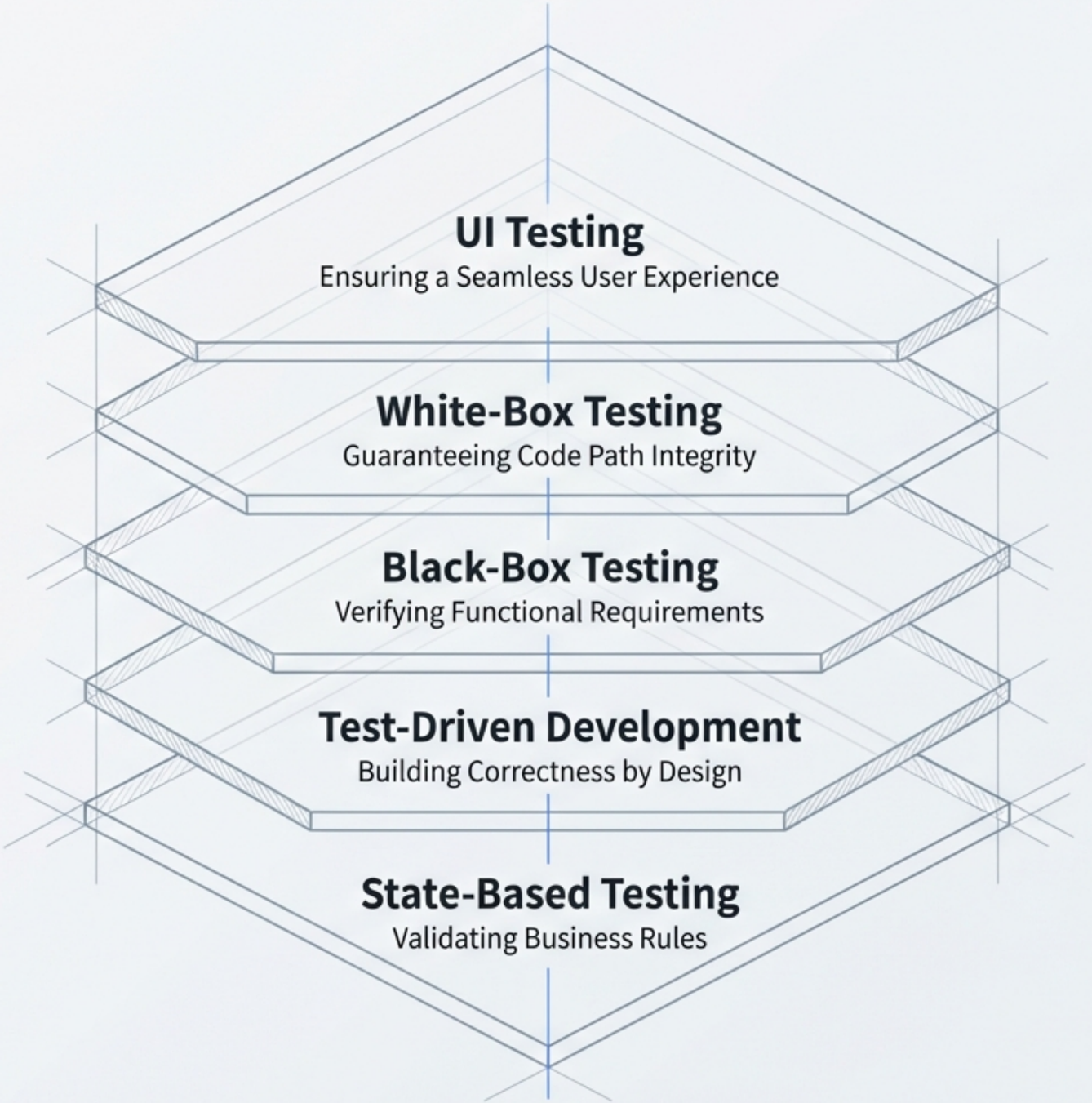
A Multi-Faceted Testing Strategy for a Core Banking System

Amir Tamer (23p0248)
Moaz Ahmed Fathy (23p0049)
Mohamed Wael Badra (23p0059)
Mohamed Wael Hadary (23p0043)
Mostafa Amr Nabil (23p0206)
Omar Foad (23p0146)



A Layered Approach to Ensuring System Integrity

This project documents a comprehensive testing strategy applied to a banking application. We treat each testing methodology as a distinct blueprint, systematically building layers of quality assurance—from abstract business logic and code implementation to the final user interface.



BLUEPRINT 1: STATE-BASED TESTING

How do we ensure the system enforces complex business rules?

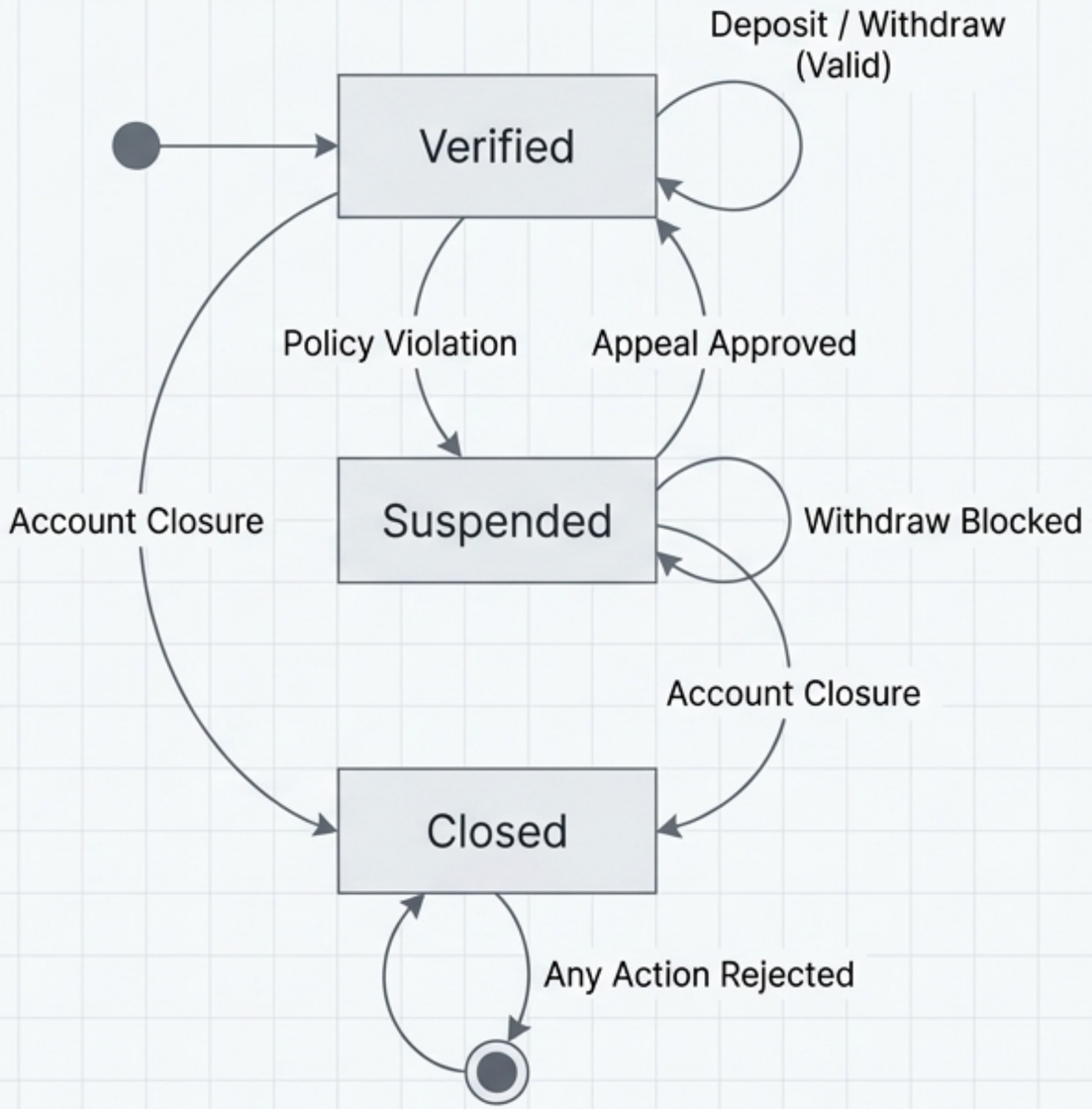
State-based testing was used to model and verify account behavior. By defining a finite set of states and the events that trigger transitions between them, we can systematically test that the application adheres to its core operational logic.

States Defined

- **Unverified**: Newly created account.
- **Verified**: Fully operational account.
- **Suspended**: Account with restricted operations.
- **Closed**: Permanently inactive account.

Goal

To confirm that only valid operations and state transitions are allowed, and invalid ones are rejected.



BLUEPRINT 1: STATE-BASED TESTING

Validating State Transitions and Permitted Operations

Valid Transition Tests

Confirming Logic Flow

Test ID	Initial State	Event	Expected Next State
ST-01	Unverified	Verification completed	Verified
ST-03	Verified	Violation detected	Suspended
ST-04	Suspended	Appeal accepted	Verified
ST-05	Verified	Admin action to close	Closed

Transaction Tests

Enforcing State-Based Restrictions

Test ID	Account State	Operation	Expected Result
ST-01	Verified	Deposit	Allowed
ST-05	Suspended	Withdraw	Rejected
ST-06	Suspended	Transfer	Rejected
ST-07	Closed	Deposit	Rejected

**The 'Transfer' operation was validated at the design/specification level as it was not present in the sample code.*

How do we build new features that are correct by design?

We applied the ‘Red-Green-Refactor’ TDD cycle to implement a client credit score check. This approach ensures that every requirement is first translated into a failing test case *before* any production code is written, forcing a focus on edge cases and requirements from the outset.

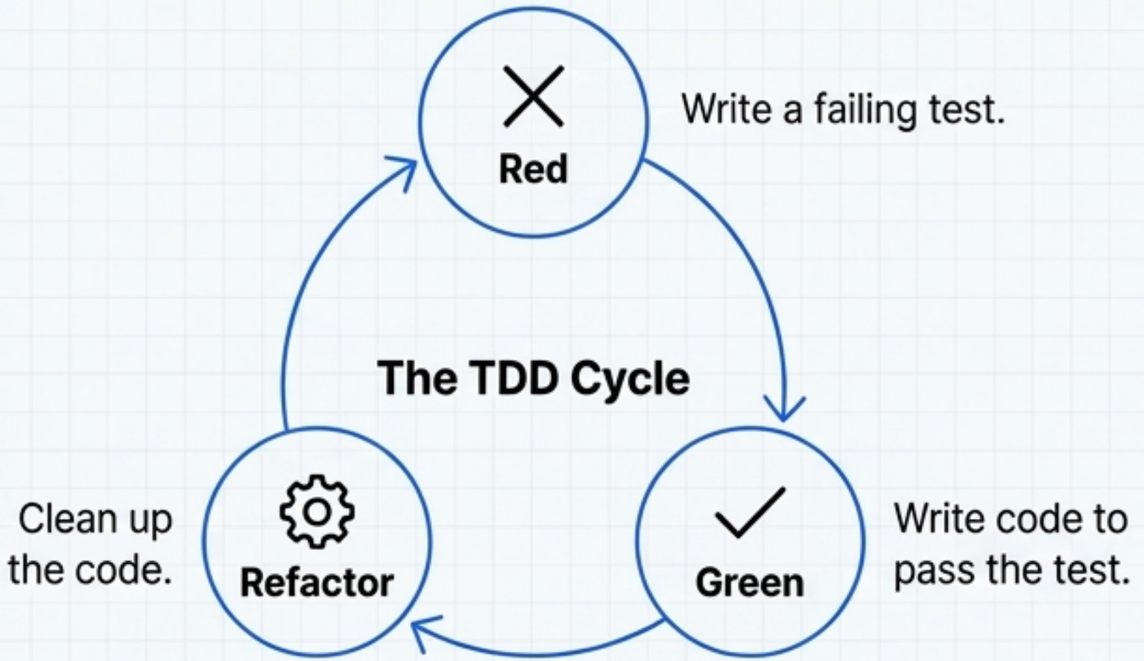
Feature Specification

Objective

Check client loan eligibility based on credit score.

Business Rules

- Score ≥ 600 → “Approved”
- Score < 600 → “Rejected”
- Score outside 300-850 range → “Invalid”



BLUEPRINT 2: TEST-DRIVEN DEVELOPMENT

From Failing Tests to a Verified Feature

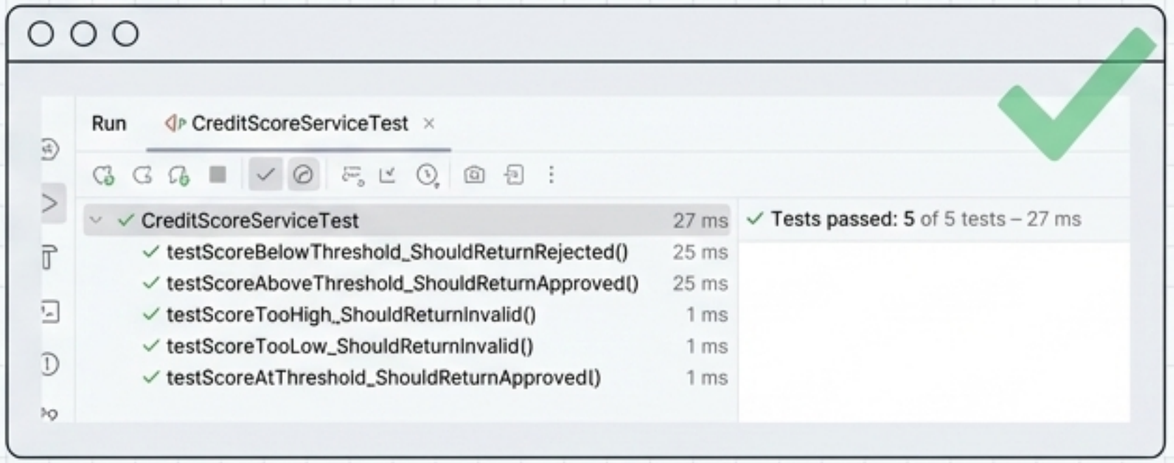
Step 1: The "Red" State - Write Tests First

We defined 5 test cases covering valid, boundary, and invalid inputs for the `checkLoanEligibility` method. Initially, all tests failed as the implementation was just a stub.

```
@Test
public void testScoreAtThreshold_ShouldReturnApproved() {
    assertEquals("Approved", service.checkLoanEligibility(600));
}
@Test
public void testScoreTooLow_ShouldReturnInvalid() {
    assertEquals("Invalid", service.checkLoanEligibility(200));
}
```

Step 2: The "Green" State - Write Code to Pass

We implemented the business logic in the `CreditScoreService` class, satisfying the test requirements.

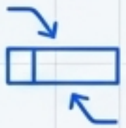


BLUEPRINT 3: BLACK-BOX TESTING

How do we verify system behavior without seeing the code?

Black-box testing focuses on validating the `Account` class's external behavior against its requirements. We treated the class as a 'black box,' designing tests based solely on its specification (inputs and expected outputs) using Equivalence Partitioning and Boundary Value Analysis.

Techniques Applied



Equivalence Partitioning (EP)

Goal: To divide input data into partitions of equivalent data from which test cases can be derived.

Example: For `Deposit Amount`, we identified two partitions: Valid (amount > 0) and Invalid (amount ≤ 0).

Boundary Value Analysis (BVA)

Goal: To focus on the "edges" or boundaries of input partitions, where errors often occur.

Example: For `Deposit Amount`, we tested the boundary value of `0` and a value just above it.

BLUEPRINT 3: BLACK-BOX TESTING

Systematic Test Case Design and Execution

Test ID	Method	Initial Balance	Status	Input	Expected Output	Notes
BB-01	deposit	100	Verified	-100	false	Invalid Partition (Negative)
BB-02	deposit	100	Verified	0	false	Boundary Value
BB-03	deposit	100	Verified	50	true	Valid Partition
BB-05	withdraw	100	Verified	200	false	Overdraft Prevention
BB-07	withdraw	100	Suspended	20	false	State Restriction

Illustrative JUnit Test

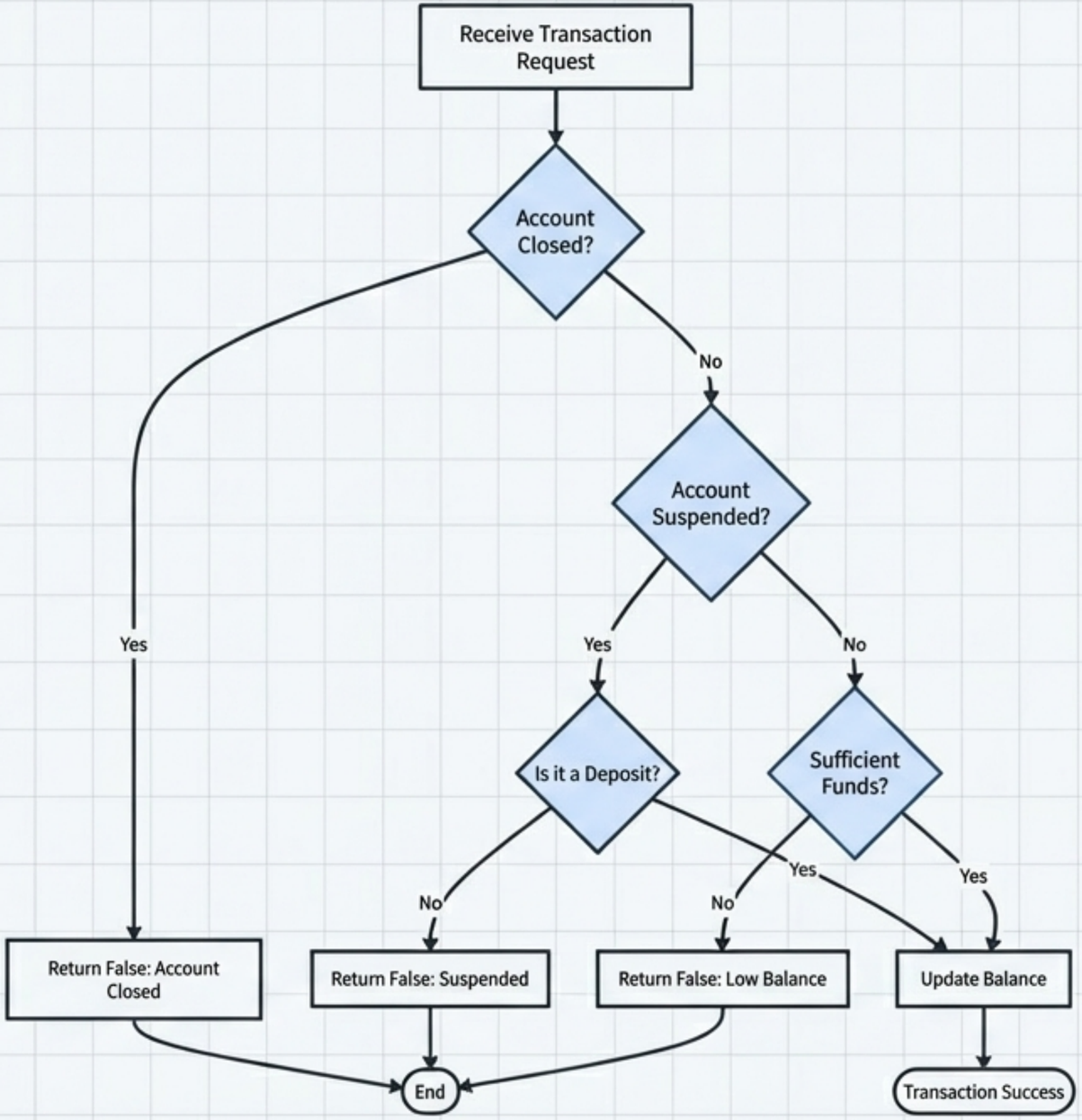
```
@Test
public void withdrawExceedBalance_shouldFail() {
    Account acc = new Account(100, "Verified");
    assertFalse(acc.withdraw(200));
}
```

BLUEPRINT 4: WHITE-BOX TESTING

How do we guarantee every line of code is tested?

While black-box testing validates *what* the system does, white-box testing validates *how* it does it. Our goal was to achieve.

100% branch coverage for the `Account` class, ensuring that every possible decision point (`if/else` logic) in the `deposit` and `withdraw` methods was executed by our test suite.



BLUEPRINT 4: WHITE-BOX TESTING

Achieving 100% Branch Coverage

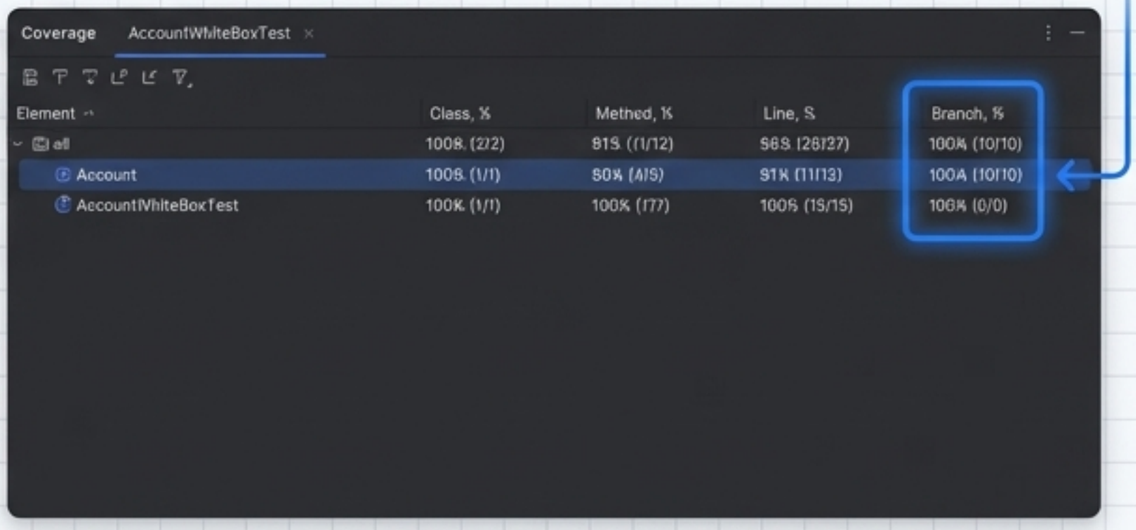
Test Case Strategy

A test case was designed specifically for each logical branch to ensure complete coverage.

Test ID	Method & Input	Path Covered
WB-01	`deposit(50, "Closed")`	`status.equals("Closed")` is true
WB-02	`deposit(-10, "Verified")`	`amount <= 0` is true
WB-06	`withdraw(1000, "Verified")`	`amount > balance` is true
WB-07	`withdraw(50, "Verified")`	All conditions false (Success Path)

The Proof of Coverage

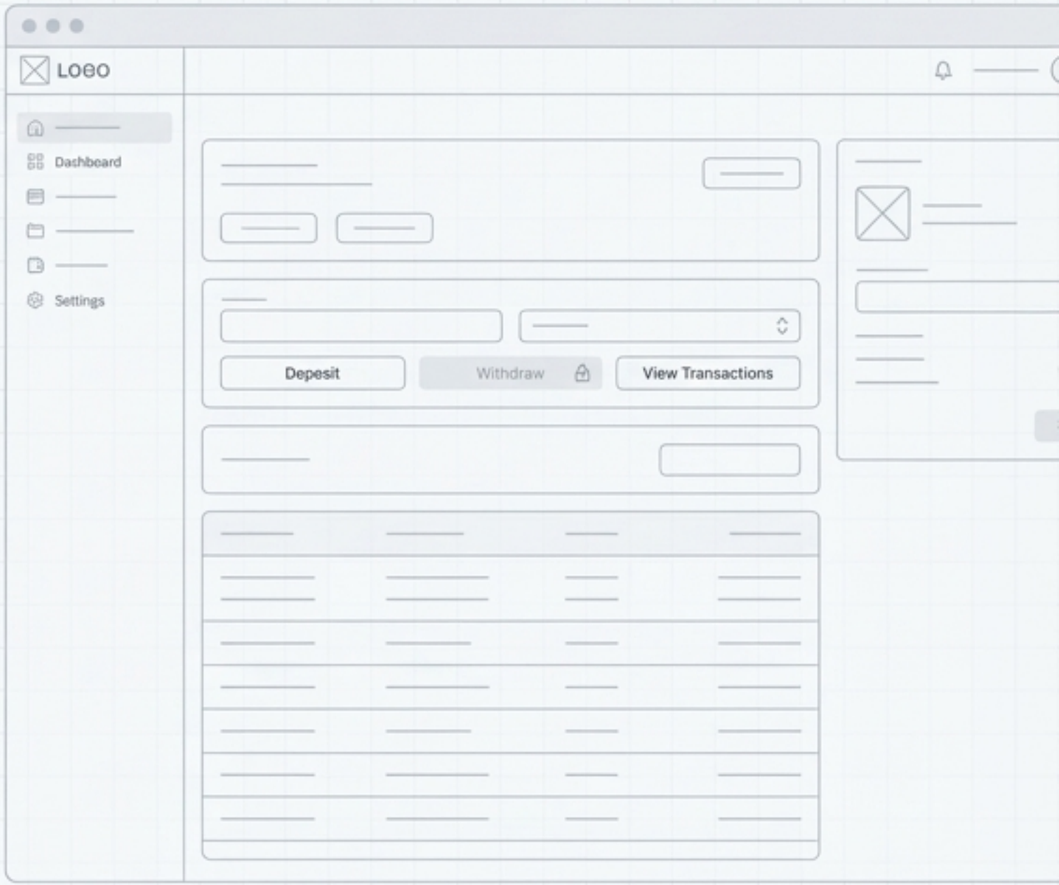
The execution of our 7 white-box test cases resulted in complete branch coverage for the `Account` class, as confirmed by the code coverage tool.



BLUEPRINT 5: UI TESTING

How do we ensure the user experience reflects the system's logic?

The final layer of quality assurance validates the user interface. Testing ensures that UI elements (buttons, labels) correctly display and respond to the underlying account state. For example, a 'Suspended' account should have its 'Withdraw' button disabled.



Methodology

Checklist Creation: A detailed checklist was developed to map UI element states and actions to different account statuses.

Automation with Selenium: Key scenarios were automated using a Selenium script to create repeatable, reliable UI tests.

BLUEPRINT 5: UI TESTING

Validating UI Behavior Through Checklists and Automation

UI Test Checklist (Sample)

Element	Action	Account Status	Expected Output
Withdraw Button	Click	Verified	Action successful, balance updates.
Withdraw Button	View	Suspended	Button is disabled/grayed out.
Deposit Button	Click	Verified	Input: -50, shows "Invalid amount".
Deposit Button	View	Closed	Button is disabled/grayed out.

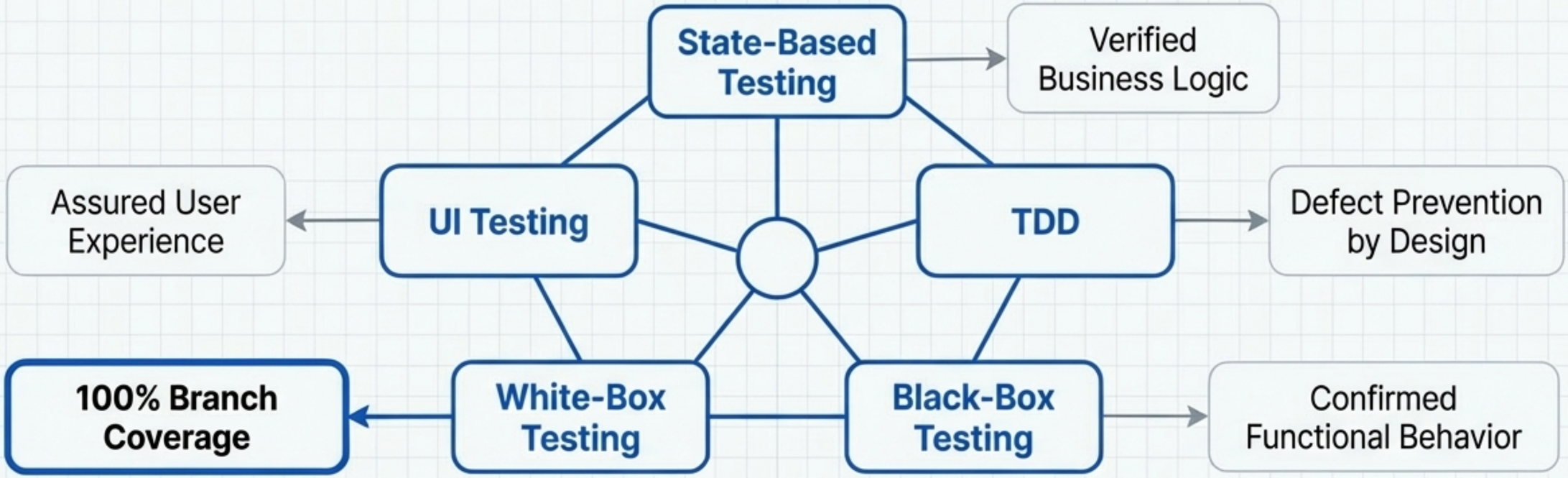
Automation with Selenium

A Python script automates browser interactions to verify critical UI restrictions.

```
# TC: Suspended accounts must block Transfers and Withdrawals.
# TC: Suspended accounts must block
def test_suspended_state_restrictions(self):
    status_label = self.driver.find_element(By.ID, "status-label")
    if "Suspended" in status_label.text:
        withdraw_btn = self.driver.find_element(By.ID, "withdraw-btn
        self.assertFalse(withdraw_btn.is_enabled(),
            "Withdraw should be disabled when Suspended.")
```

The Complete Quality Blueprint: A Unified Strategy

By combining five distinct testing methodologies, we constructed a robust quality assurance framework. Each 'blueprint' addressed a unique aspect of the system, creating overlapping layers of defense against defects.



Key Insight: This multi-faceted approach ensures that quality is not just tested at the end, but is engineered into the system at every stage of development.

From Blueprints to a Production-Ready System

This systematic application of diverse testing techniques provides high confidence in the banking system's reliability, security, and correctness. The result is not merely a 'tested' application, but one that has been deliberately engineered for quality, with verifiable proof at every layer of its architecture.