Software Testing Project

Amir Tamer 23p0248
Moaz Ahmed Fathy 23p0049
Mohamed Wael Badra 23p0059
Mohamed Wael Hadary 23p0043
Mostafa Amr Nabil 23p0206
Omar Foad 23p0146
Github Link:
https://github.com/AmirTamer-27/Software-Testing-Project.git

State-Based testing

State-based testing is used in this project to verify that the banking system correctly handles account behavior based on its current status. The testing focuses on the defined account states— Unverified, Verified, Suspended, and Closed—and ensures that only valid operations and state transitions are allowed. Test cases are designed to confirm that restricted actions are properly rejected and that state changes occur only under the correct conditions.

States

State Name	Description
Unverified	Account is created but not yet verified
Verified	Account is active and fully operational
Suspended	Account has restricted operations
Closed	Account is permanently inactive

State Transition Table

Current State	Event	Next State
Unverified	Verification completed	Verified
Unverified	Violation detected	Suspended
Verified	Violation detected	Suspended
Suspended	Appeal accepted	Verified
Verified	Admin action to close	Closed
Suspended	Admin action to close	Closed
Unverified	Admin action to close	Closed

Valid Transition Test

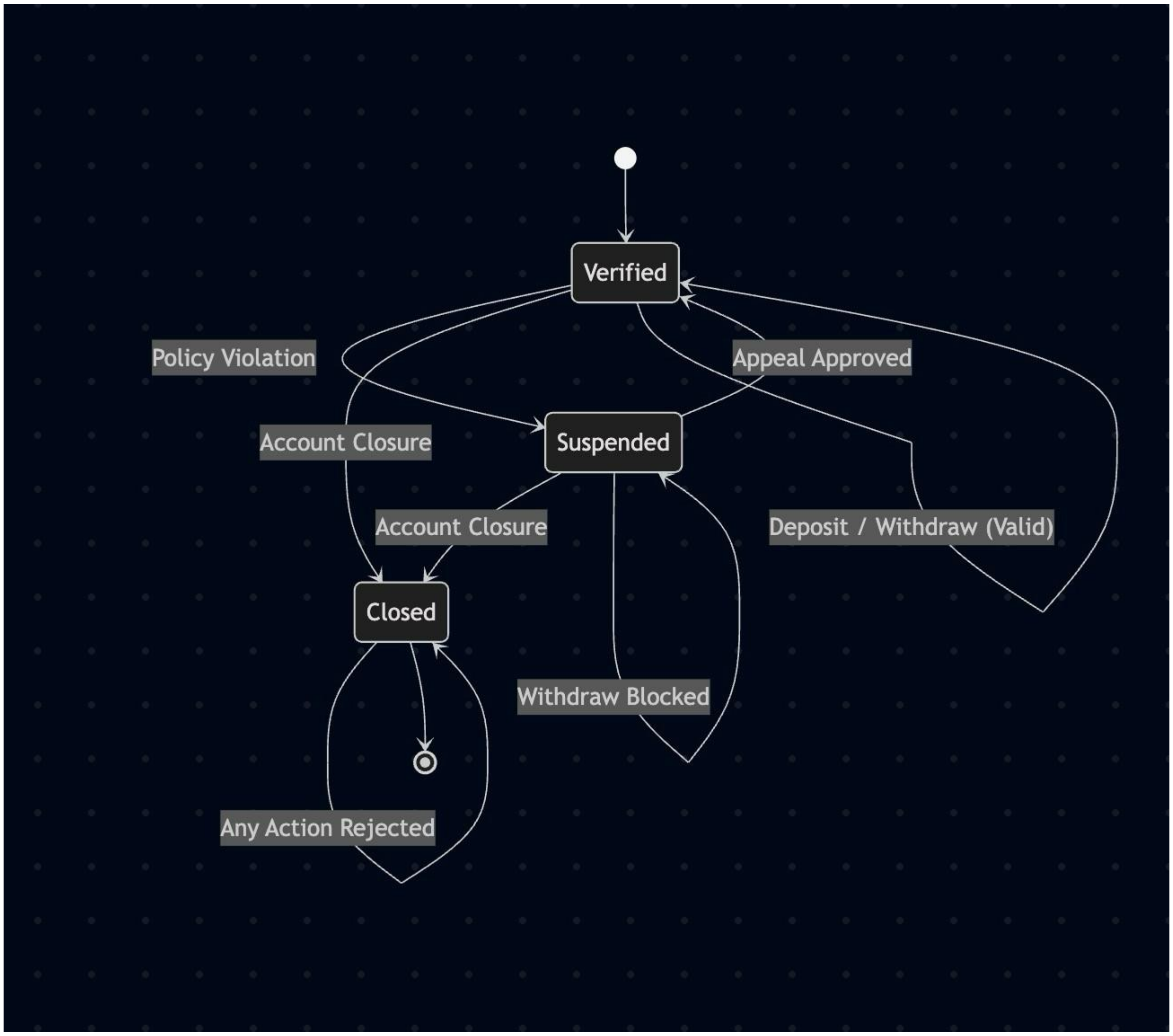
Test ID	Initial State	Event	Expected Result
ST-01	Unverified	Verification completed	State changes to Verified
ST-02	Unverified	Violation detected	State changes to Suspended
ST-03	Verified	Violation detected	State changes to Suspended
ST-04	Suspended	Appeal accepted	State changes to Verified
ST-05	Verified	Admin action to close	State changes to Closed
ST-06	Suspended	Admin action to close	State changes to Closed
ST-07	Unverified	Admin action to close	State changes to Closed

Transaction Tests

Test ID	State	Operation	Expected Result
ST-01	Verified	Deposit	Allowed
ST-02	Verified	Withdraw	Allowed
ST-03	Verified	Transfer	Allowed
ST-04	Suspended	Deposit	Allowed
ST-05	Suspended	Withdraw	Rejected
ST-06	Suspended	Transfer	Rejected
ST-07	Closed	Deposit	Rejected
ST-08	Closed	Withdraw	Rejected
ST-09	Closed	Transfer	Rejected

Transfer is included as a state-based operation per the specification; however, it is not implemented in the provided sample Java code and is therefore validated at the design/specification level in this section.

State Diagram



Test-Driven Development (TDD)

1. Feature Description: Client Credit Score Check

Objective: Implement a new feature to determine if a client is eligible for a loan based on their credit score using the **Red-Green-Refactor** TDD cycle.

Business Rules:

- **Valid Range:** Credit scores must be between **300 and 850**.
- **Approval:** Scores of **600 or higher** are "Approved".
- **Rejection:** Scores **below 600** are "Rejected".
- **Invalid Input:** Scores below 300 or above 850 are returned as "Invalid".

2. Test Plan and Expected Behavior

The following test cases were designed *before* any functional code was written.

Test ID	Input (Score)	Expected Output	Logic Description
TDD-01	500	"Rejected"	Score is within valid range but below the approval threshold (600).
TDD-02	600	"Approved"	Score is exactly at the boundary. Boundary condition check.
TDD-03	750	"Approved"	Score is well above the threshold. Valid positive case.
TDD-04	200	"Invalid"	Score is below the minimum valid limit (300).
TDD-05	900	"Invalid"	Score is above the maximum valid limit (850).

3. TDD Implementation Process

Phase 1: Red State (Writing the Test First)

We created the CreditScoreServiceTest class and defined the 5 test cases. At this stage, the service only existed as a "stub" (empty method), causing all tests to **FAIL** initially.

Unit Test Code (CreditScoreServiceTest.java):

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class CreditScoreServiceTest {

  CreditScoreService service = new CreditScoreService();

    @Test
    public void testScoreBelowThreshold_ShouldReturnRejected() {
        String result = service.checkLoanEligibility(500);
        assertEquals("Rejected", result);
    }

    @Test
    public void testScoreAtThreshold_ShouldReturnApproved() {
        String result = service.checkLoanEligibility(600);
        assertEquals("Approved", result);
    }

    @Test
    public void testScoreAboveThreshold_ShouldReturnApproved() {
        String result = service.checkLoanEligibility(750);
        assertEquals("Approved", result);
    }

    @Test
    public void testScoreTooLow_ShouldReturnInvalid() {
        String result = service.checkLoanEligibility(200);
        assertEquals("Invalid", result);
    }

    @Test
    public void testScoreTooHigh_ShouldReturnInvalid() {
        String result = service.checkLoanEligibility(900);
        assertEquals("Invalid", result);
    }
}
```

Phase 2: Green State (Implementing the Logic)

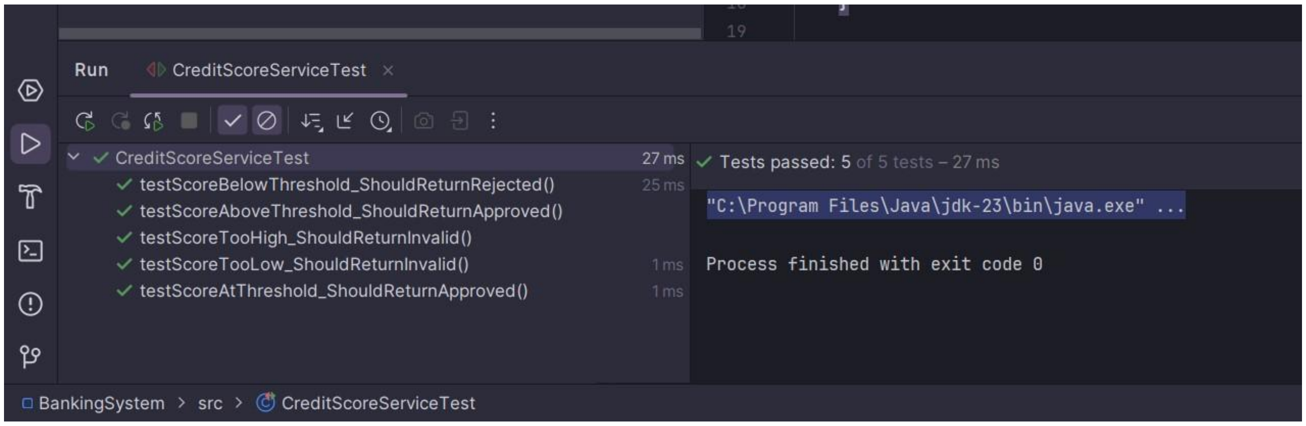
We then wrote the implementation code to handle the business rules. This moved the tests from "Red" (failing) to "Green" (passing).

Implementation Code (CreditScoreService.java):

```java
public class CreditScoreService {

    public String checkLoanEligibility(int creditScore) {
        // Check for invalid ranges first
        if (creditScore < 300 || creditScore > 850) {
            return "Invalid";
        }

        // Business logic for approval
        if (creditScore >= 600) {
            return "Approved";
        } else {
            return "Rejected";
        }
    }
}
```

4. Execution Results

The following screenshot confirms that all 5 test cases passed successfully, validating that the new feature works according to the specifications.



```
Run    CreditScoreServiceTest

CreditScoreServiceTest                                27 ms   Tests passed: 5 of 5 tests – 27 ms
  testScoreBelowThreshold_ShouldReturnRejected()      25 ms
  testScoreAboveThreshold_ShouldReturnApproved()              "C:\Program Files\Java\jdk-23\bin\java.exe" ...
  testScoreTooHigh_ShouldReturnInvalid()
  testScoreTooLow_ShouldReturnInvalid()                1 ms   Process finished with exit code 0
  testScoreAtThreshold_ShouldReturnApproved()          1 ms

BankingSystem > src > CreditScoreServiceTest
```

5. TDD Reflection & Summary

The Test-Driven Development approach ensured that we considered edge cases (like 200 or 900) before writing the code.

1. **Requirement Analysis:** We first broke down the "Loan Eligibility" rule into strict boundaries (300, 600, 850).
2. **Test First:** Writing testScoreTooLow_ShouldReturnInvalid forced us to remember to handle negative or small numbers, which might have been forgotten in a standard coding approach.
3. **Validation:** The final green bar provides high confidence that the credit score feature is robust and ready for integration.

Black Box Testing

1. Functional Requirement Understanding (External Behavior Only)

Black-box testing is applied to verify that the Account class behaves correctly without knowing its internal code structure. Testing focuses on inputs vs expected outputs according to system requirements:

- Deposits must succeed only when:
  • Account is not “Closed”
  • Amount > 0
- Withdraw must succeed only when:
  • Account status is Verified
  • Amount ≤ Balance
- Withdraw must fail if:
  • Account is Closed or Suspended
  • Amount > Balance

2. Equivalence Partitioning (EP)

Feature	Valid Partitions	Invalid Partitions
Deposit Amount	amount > 0	amount ≤ 0
Deposit Account Status	Verified	Closed
Withdraw Status	Verified	Suspended / Closed
Withdraw Amount	amount ≤ balance	amount > balance

3. Boundary Value Analysis (BVA)

Scenario	Boundary	Expected
Deposit	0	Fail
Deposit	Positive (>0)	Success
Withdraw vs Balance	Equal to balance	Success
Withdraw vs Balance	Greater than balance	Fail

4. Black Box Test Cases

Test ID	Method	Initial Balance	Status	Input	Expected Output	Notes
BB-01	deposit	100	Verified	-100	false	Invalid negative deposit
BB-02	deposit	100	Verified	0	false	Boundary value
BB-03	deposit	100	Verified	50	true (Balance = 150)	Valid partition
BB-04	withdraw	200	Verified	50	true (Balance = 150)	Valid withdraw
BB-05	withdraw	100	Verified	200	false	Overdraft prevention
BB-06	deposit	100	Closed	50	false	State restriction
BB-07	withdraw	100	Suspended	20	false	State restriction

5. Black Box JUnit Test Implementation

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class AccountBlackBoxTest {

    @Test
    public void depositNegativeAmount_shouldFail() {
        Account acc = new Account(100, "Verified");
        assertFalse(acc.deposit(-100));
    }

    @Test
    public void depositZero_shouldFail() {
        Account acc = new Account(100, "Verified");
        assertFalse(acc.deposit(0));
    }

    @Test
    public void depositValidAmount_shouldSucceed() {
        Account acc = new Account(100, "Verified");
        assertTrue(acc.deposit(50));
        assertEquals(150, acc.getBalance());
    }

    @Test
    public void withdrawWithinBalance_shouldSucceed() {
        Account acc = new Account(200, "Verified");
        assertTrue(acc.withdraw(50));
        assertEquals(150, acc.getBalance());
    }

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

1. Decision Path Analysis(White box testing)

The Account class contains several critical decision points (branches) that determine the flow of a transaction. To achieve 100% branch coverage, we analyzed the following logic:

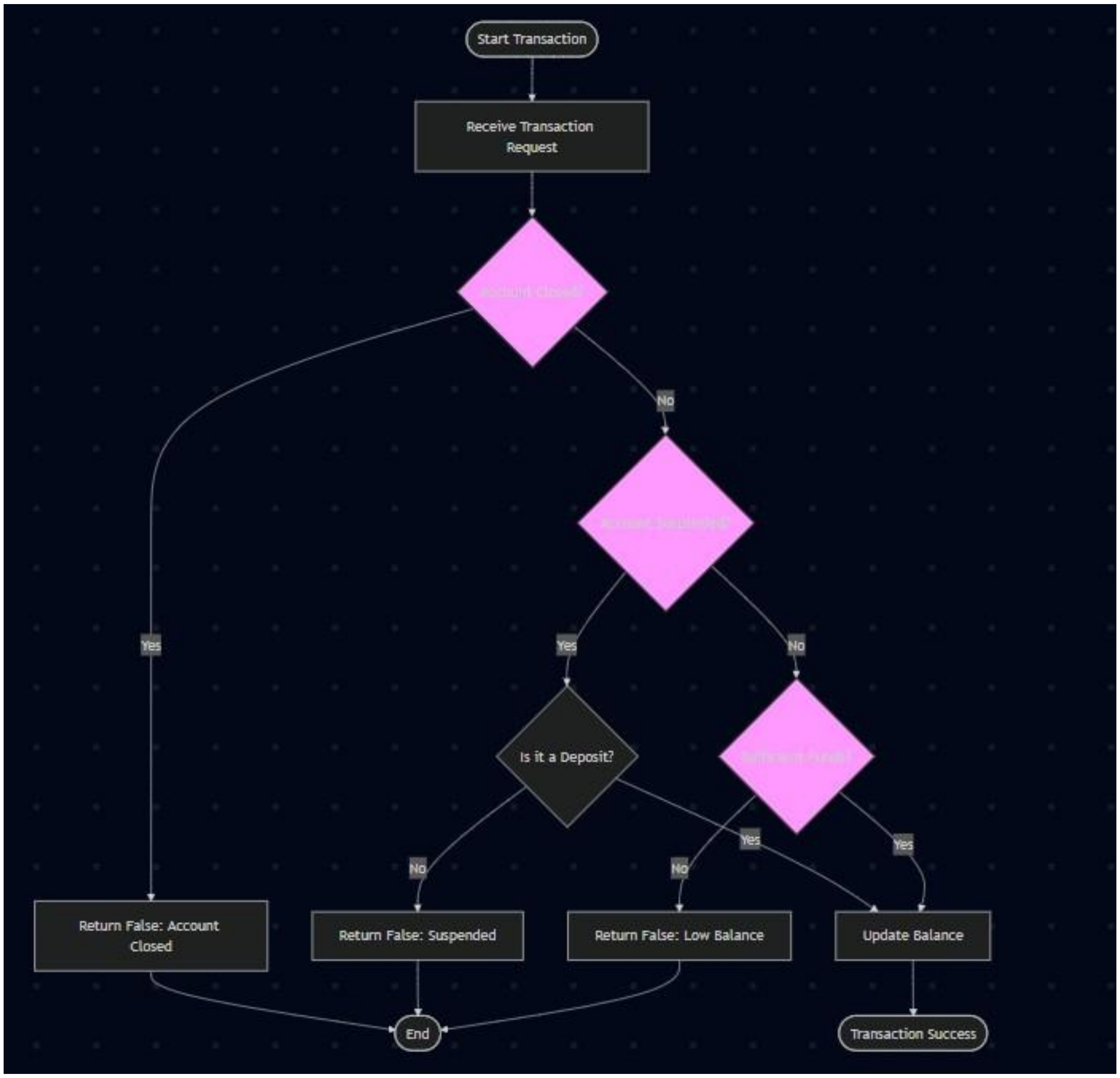
- **deposit(double amount) Decision Logic:**
  - **Path 1 (Invalid):** status.equals("Closed") is true OR amount <= 0 is true. Both lead to an immediate false return.
  - **Path 2 (Valid):** Both conditions are false, leading to the state change (balance += amount).
- **withdraw(double amount) Decision Logic:**
  - **Path 1 (Status Restriction):** If status is "Closed" OR "Suspended", the transaction is blocked.
  - **Path 2 (Constraint Check):** If amount > balance, the transaction is blocked even if the account is "Verified".
  - **Path 3 (Success):** Account is in a valid state AND has sufficient funds.

2. Branch & Loop Analysis

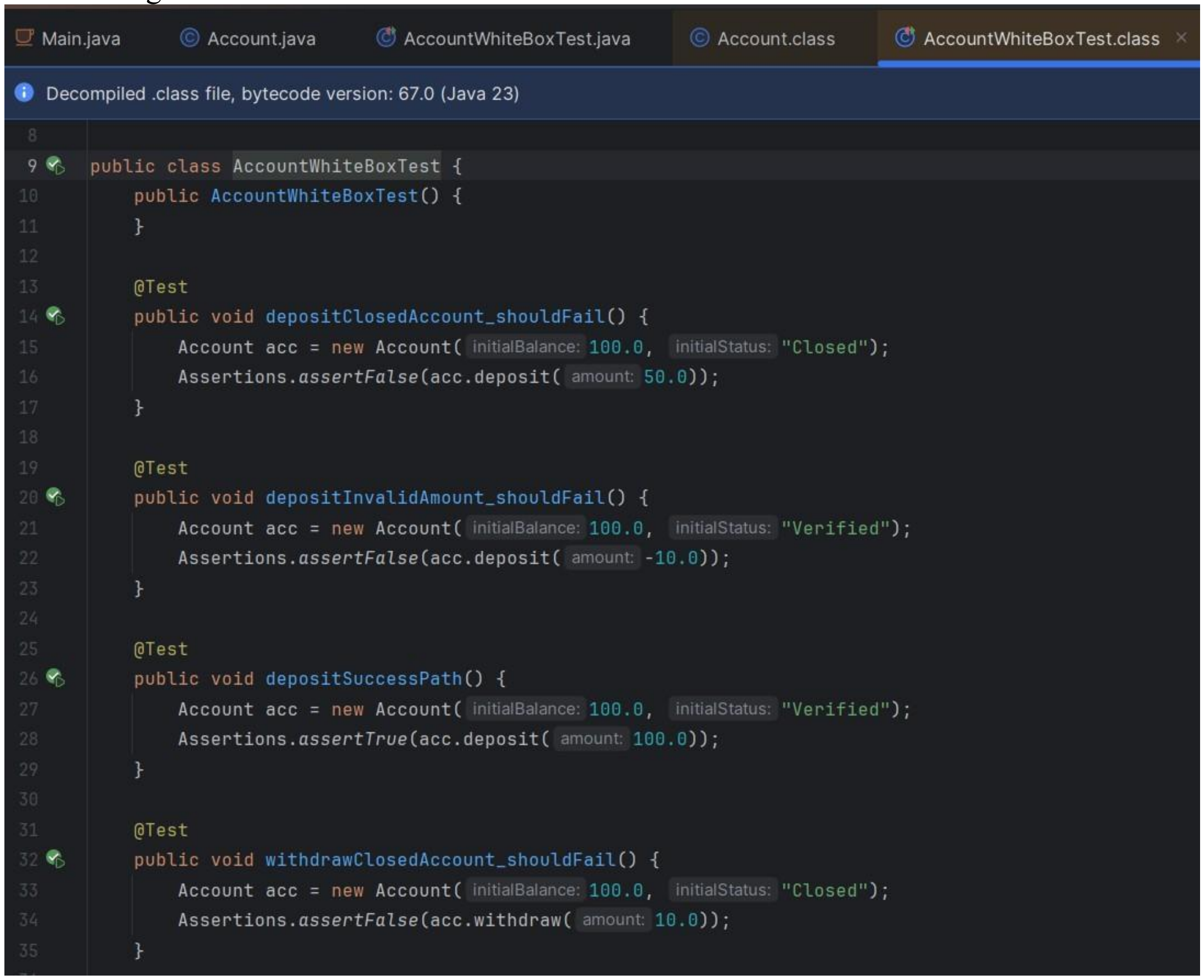
- **Branches:** There are **4 main decision branches** in this class (2 in deposit, 2 in withdraw).
    - *Note:* Even though the code uses || (OR) operators, white-box testing treats these as "Short-circuit" branches. For example, if status.equals("Closed") is true, the code never even checks if the amount is negative.
- **Loops:** There are currently **no loops** in this specific Account snippet. (If your TransactionProcessor class contains a for loop that iterates through a list of accounts, you would note that here as a "Loop Boundary" analysis).

Test ID	**Method**	**Input (Amount, Status)**	**Expected**	**Path/Annotation**
WB-01	deposit	(50, "Closed")	false	Decision: Status Check (True)
WB-02	deposit	(-10, "Verified")	false	Decision: Amount Check (True)
WB-03	deposit	(100, "Verified")	true	Full Execution Path
WB-04	withdraw	(10, "Closed")	false	Decision: Multi-Status Check
WB-05	withdraw	(10, "Suspended")	false	Decision: Multi-Status Check
WB-06	withdraw	(1000, "Verified")	false	Decision: Overdraft Logic
WB-07	withdraw	(50, "Verified")	true	Full Execution Path

CFG:

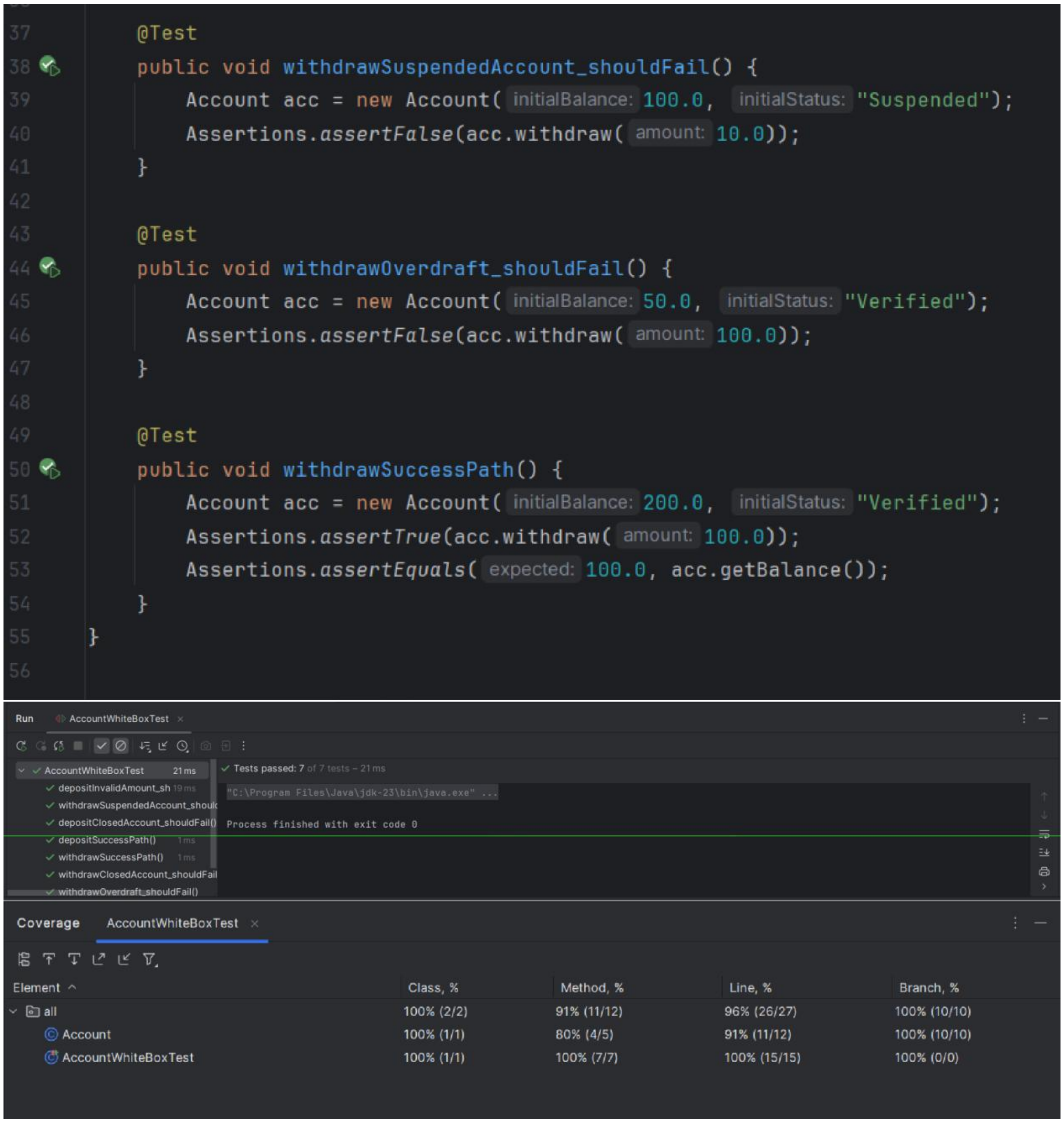

Start Transaction
Receive Transaction Request
Account Closed?
No
Account Suspended?
Yes
Yes
No
Is it a Deposit?
Sufficient Funds?
Yes
Yes
No
No
Return False: Account Closed
Return False: Suspended
Return False: Low Balance
Update Balance
End
Transaction Success

Unit Testing Code:



Decompiled .class file, bytecode version: 67.0 (Java 23)

```java

public class AccountWhiteBoxTest {
    public AccountWhiteBoxTest() {
    }

    @Test
    public void depositClosedAccount_shouldFail() {
        Account acc = new Account( initialBalance: 100.0,  initialStatus: "Closed");
        Assertions.assertFalse(acc.deposit( amount: 50.0));
    }

    @Test
    public void depositInvalidAmount_shouldFail() {
        Account acc = new Account( initialBalance: 100.0,  initialStatus: "Verified");
        Assertions.assertFalse(acc.deposit( amount: -10.0));
    }

    @Test
    public void depositSuccessPath() {
        Account acc = new Account( initialBalance: 100.0,  initialStatus: "Verified");
        Assertions.assertTrue(acc.deposit( amount: 100.0));
    }

    @Test
    public void withdrawClosedAccount_shouldFail() {
        Account acc = new Account( initialBalance: 100.0,  initialStatus: "Closed");
        Assertions.assertFalse(acc.withdraw( amount: 10.0));
    }
```



```java
    @Test
    public void withdrawSuspendedAccount_shouldFail() {
        Account acc = new Account( initialBalance: 100.0,  initialStatus: "Suspended");
        Assertions.assertFalse(acc.withdraw( amount: 10.0));
    }

    @Test
    public void withdrawOverdraft_shouldFail() {
        Account acc = new Account( initialBalance: 50.0,  initialStatus: "Verified");
        Assertions.assertFalse(acc.withdraw( amount: 100.0));
    }

    @Test
    public void withdrawSuccessPath() {
        Account acc = new Account( initialBalance: 200.0,  initialStatus: "Verified");
        Assertions.assertTrue(acc.withdraw( amount: 100.0));
        Assertions.assertEquals( expected: 100.0, acc.getBalance());
    }
}
```

```
AccountWhiteBoxTest                    21 ms
  ✓ depositInvalidAmount_sh            19 ms
  ✓ withdrawSuspendedAccount_shoul
  ✓ depositClosedAccount_shouldFail()
  ✓ depositSuccessPath()                1 ms
  ✓ withdrawSuccessPath()               1 ms
  ✓ withdrawClosedAccount_shouldFail
  ✓ withdrawOverdraft_shouldFail()

✓ Tests passed: 7 of 7 tests – 21 ms

"C:\Program Files\Java\jdk-23\bin\java.exe" ...

Process finished with exit code 0
```

Element ^	Class, %	Method, %	Line, %	Branch, %
all	100% (2/2)	91% (11/12)	96% (26/27)	100% (10/10)
Account	100% (1/1)	80% (4/5)	91% (11/12)	100% (10/10)
AccountWhiteBoxTest	100% (1/1)	100% (7/7)	100% (15/15)	100% (0/0)

Test Case Method	Lines / Branches Covered	Logic Description
depositClosedAccount_shouldFail()	if (status.equals("Closed")) → TRUE	Covers the **true branch** of the account status check, where deposits are rejected for closed accounts.
depositInvalidAmount_shouldFail()	if (amount <= 0) → TRUE	Covers the **true branch** of the amount validation condition, where invalid deposit amounts are rejected.
depositSuccessPath()	if (status.equals("Closed")) → FALSEif (amount <= 0) → FALSEbalance += amount; return true;	Covers the **false branches** of all conditional checks and executes the successful deposit path.

Test Case Method	Lines / Branches Covered	Logic Description
withdrawClosedAccount_shouldFail()	if (status.equals("Closed")) → TRUE	Covers the **true branch** of the first status check where withdrawals are rejected for closed accounts.
withdrawSuspendedAccount_shouldFail()	if (status.equals("Closed")) → FALSEif (status.equals("Suspended")) → TRUE	Covers the **false branch** of the closed check and the **true branch** of the suspended account check.
withdrawOverdraft_shouldFail()	if (status.equals("Closed")) → FALSEif (status.equals("Suspended")) → FALSEif (amount > balance) → TRUE	Covers the **true branch** of the overdraft condition where withdrawal exceeds available balance.
withdrawSuccessPath()	if (status.equals("Closed")) → FALSEif (status.equals("Suspended")) → FALSEif (amount > balance) → FALSEbalance -= amount; return true;	Covers the **false branches of all conditions** and executes the successful withdrawal path, achieving full branch coverage.

UI Testing Checklist:

Element	Action	Account Status	Input	Expected Output
Status Label	View	**Unverified**	N/A	Label displays "Unverified".
Verify Button	Click	**Unverified**	N/A	Status transitions to "Verified".
Deposit Button	Click	**Verified**	500.00	Notification: "Deposit successful"Balance updates.
Deposit Button	Click	**Verified**	-1.00	Notification: "Invalid amount" or "false".
Deposit Button	Click	**Verified**	0.00	Action blocked; Notification shows validation error.
Withdraw Button	Click	**Verified**	200.00	Notification: "Withdrawal successful"; Balance updates.
Withdraw Button	Click	**Verified**	2000.00	Notification: "Overdraft prevention" or "false".
Transfer Button	Click	**Verified**	100.00	Button is enabled, Transaction proceeds.
Status Label	View	**Suspended**	N/A	Label displays "Suspended".
Transfer Button	Click	**Suspended**	Any	Button is disabled/grayed out.

Element	Action	Account Status	Input	Expected Output
Withdraw Button	Click	**Suspended**	Any	Button is disabled/grayed out.
Appeal Button	Click	**Suspended**	N/A	Status transitions back to "Verified".
Deposit Button	Click	**Suspended**	10.00	(If View Only) Button disabled or shows error.
Status Label	View	**Closed**	N/A	Label displays "Closed".
Deposit Button	Click	**Closed**	Any	Button is disabled; "false" returned.
Withdraw Button	Click	**Closed**	Any	Button is disabled; Action blocked.
Transfer Button	Click	**Closed**	Any	Button is disabled/grayed out.
View Statement	Click	**Any State**	N/A	Button is always enabled, Statement renders.
Client Name	View	**Any State**	N/A	Displays "John Doe" correctly.
Acc. Number	View	**Any State**	N/A	Displays "123456789" correctly.

Selenium Script for UI testing:

```
import org.openqa.selenium.By;
import
   org.openqa.selenium.WebDrive
   r;
import
   org.openqa.selenium.WebElem
   ent;
import
   org.openqa.selenium.chrome.C
   hromeDriver;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class
   BankingSystemFullUITest {
   private WebDriver driver;

   @Before
   public void setUp() {
      // Setup ChromeDriver
      driver = new
   ChromeDriver();
      // Provide the absolute path
   to your dashboard HTML
   provided in the artifacts
      driver.get("file:///path/to/you
   r/banking_dashboard.html");
   }

   @Test
   public void
   testVerifiedStateFunctionality()
   {

      String status =
   driver.findElement(By.id("statu
   s-label")).getText();

      if (status.equals("Verified"))
   {
         assertTrue("Deposit
   button should be enabled",
   driver.findElement(By.id("dep
osit-btn")).isEnabled());
        assertTrue("Withdraw
button should be enabled",
driver.findElement(By.id("with
draw-btn")).isEnabled());
        assertTrue("Transfer
button should be enabled",
driver.findElement(By.id("tran
sfer-btn")).isEnabled());
    }
}

@Test
public void
testSuspendedStateRestrictions
() {

    WebElement statusLabel =
driver.findElement(By.id("statu
s-label"));

    if
(statusLabel.getText().contains(
"Suspended")) {
        WebElement transferBtn
=
driver.findElement(By.id("tran
sfer-btn"));
        WebElement withdrawBtn
=
driver.findElement(By.id("with
draw-btn"));


        assertFalse("Transfer
should be disabled when
Suspended",
transferBtn.isEnabled());
        assertFalse("Withdraw
should be disabled when
Suspended",
withdrawBtn.isEnabled());
    }
}

@Test
public void
```

```
testClosedStateLockdown() {

    WebElement statusLabel =
driver.findElement(By.id("statu
s-label"));

    if
(statusLabel.getText().contains(
"Closed")) {
        WebElement depositBtn =
driver.findElement(By.id("dep
osit-btn"));
        WebElement withdrawBtn
=
driver.findElement(By.id("with
draw-btn"));


        assertFalse("Deposit
blocked in Closed state",
depositBtn.isEnabled());
        assertFalse("Withdraw
blocked in Closed state",
withdrawBtn.isEnabled());
    }
}

@Test
public void
testInputValidationNegativeDe
posit() {

    WebElement depositInput =
driver.findElement(By.id("amo
unt-input"));
    WebElement depositBtn =
driver.findElement(By.id("dep
osit-btn"));
    WebElement notification =
driver.findElement(By.id("notif
ication-box"));

    depositInput.clear();
    depositInput.sendKeys("-
50");
    depositBtn.click();
```

```
        assertTrue("Notification box
    should be visible",
    notification.isDisplayed());
        assertTrue("Should display
    'Invalid' message",
    notification.getText().contains(
    "Invalid"));
    }

    @After
    public void tearDown() {
        if (driver != null) {
            driver.quit();
        }
    }
}
```