

# Assignment 3: Test-Driven Development & Data Flow Testing

Davit Chuntishvili

Student ID: 24278688

Module: CS4004 - Software Testing & Inspection

## Introduction

In this assignment, Test-Driven Development (TDD) principles were applied, emphasizing writing test cases first and then implementing or modifying code based on the test results. This approach ensured thorough testing of all methods in `BankAccountManagementSystem_24278688` class and led to improve its functionality.

## Task 1

As TDD development principles suggest to unit test the code first and then implement the right calls, I went with this approach in the assignment as well. I started with testing each method's functionalities without looking explicitly using method implementations to judge what unit tests should've been done.

First method tested was "createAccount()" method, where I thought of 4 logical ways the method should be tested. Program should be tested if it can be successfully created or unsuccessful due to duplication, or negative balance entered in the argument, or both. These tests showed that original implementation method for "createAccount()" method was valid. Thus, no changes needed to be done inside the code.

The createAccount() method was tested for the following scenarios:

1. Successful account creation.
2. Unsuccessful creation due to duplicate accounts.
3. Unsuccessful creation with a negative balance.
4. Unsuccessful creation with both duplication and negative balance.

Second method tested was "deposit()" method. On this one, deposit method either has to be successful, or unsuccessful for these reasons: No precreated active account, negative deposit, negative deposit whilst having no precreated active account, or deposit is zero. These test results made me change method implementation's structure.

The deposit() method was tested for the following logical cases:

1. Successful deposit into an existing account.
2. Unsuccessful deposit due to:
  - No pre-created account.

- Negative deposit amount.
  - Negative deposit amount with no pre-created account.
  - Deposit of zero.
- Validation for negative amounts:
    - Original method didn't check for negative deposit amounts.
    - Updated method checks with `if (amount < 0)` and returns -1.0
  - Validation for zero amounts:
    - Original method didn't handle deposits of zero amount.
    - Updated method checks with `if (amount == 0)` and returns -3.0

Third method tested was “`withdraw()`” method. This method in its sense is the most complex one out of all the methods from the class, therefore I tested all logically possible cases for deposit: Successful withdrawal; when the withdrawal amount is more than the balance; when withdrawal is negative amount; when withdrawal happens from a non-existent account; and withdraw no amount or there is no money on the balance.

The `withdraw()` method, being the most complex, was tested for the following cases:

1. Successful withdrawal.
  2. Withdrawal amount exceeding the account balance.
  3. Negative withdrawal amount.
  4. Withdrawal from a non-existent account.
  5. Withdrawal of zero or when the balance is zero.
- Validation for zero amounts:
    - Original method didn't stand a chance to check zero amounts, since other checks such as `if (amount <= 0)` and `if (amount == balance)` were checked first.
    - In updated method zero check condition is moved on the top of other conditions, thus this case is checked first.
  - Validation for negative amounts:
    - Original method checked if the amount was less than or equal to zero.
    - Updated method is changed to check if the amount is less than zero, as zero is handled separately.
  - Validation for amount that's greater than balance:
    - Original method checked if the amount was greater than or equal to balance.
    - Updated method is changed to check if the amount is greater than the balance.

Fourth method tested was “`getAccountBalance()`”. This method was halfway implemented as it was tested for handling of non-existing account's balance which it failed. Therefore in the implementation the code to check if the account is created was added that returns 0.0.

## Task 2

Data flow testing focuses on examining the lifecycle of variables, from their definition (where they are assigned values) to their uses in conditions or computations. The goal is to verify all paths where a variable is defined and used to complete coverage of critical code paths.

### DU-Pair table for “deposit()”

Variable	Definition	Use	Type of Use
accountNumber	Parameter of method	accounts.containsKey(accountNumber)	p-use (Predicate)
amount	Parameter of method	amount < 0	p-use (Predicate)
amount	Parameter of method	balance += amount	c-use (Computation)
balance	balance = accounts.get(accountNumber)	balance += amount	c-use (Computation)
balance	balance = accounts.get(accountNumber)	return balance	c-use (Computation)

### Test Plan table for “deposit()”

Test case	Inputs	Expected output	DU-Pairs cover
1. Valid deposit	createAccount(1001, 200.0) deposit(1001, 400)	Success	All variables
2. Non-existent account	createAccount(N/A), deposit(1001, 400)	-1.0 (account missing)	accountNumber: p-use
3. Negative deposit	createAccount(1001, 200.0) deposit(1001, -400)	-1.0 (invalid amount)	amount: p-use
4. Non-existent account & Negative deposit	createAccount(N/A), amount = -400	-1.0 (account missing, invalid amount)	accountNumber: p-use
5. Zero deposit	createAccount(1001, 200.0) deposit(1001, 0)	-3.0 (can't deposit 0)	amount: p-use

---

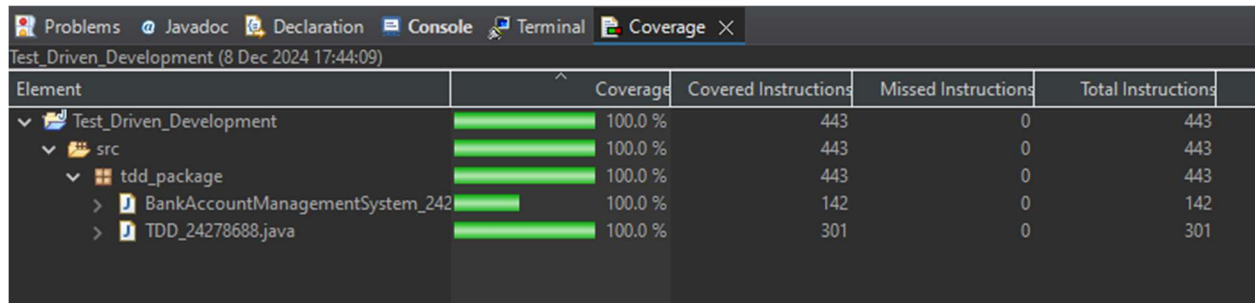
### Coverage level calculation

---

**COVERAGE LEVEL = (DU-Pairs covered)/(Total DU-Pairs identified) \* 100 = %**

**In our case, 5/5 \* 10 = 100%**

Coverage level shown by eclipse:



The screenshot shows the Eclipse IDE's Coverage view. The top bar includes tabs for Problems, Javadoc, Declaration, Console, Terminal, and Coverage. The Coverage view displays a tree of project elements with their respective coverage percentages and instruction counts. All elements show 100.0% coverage, indicated by green bars and text.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
Test_Driven_Development	100.0 %	443	0	443
src	100.0 %	443	0	443
tdd_package	100.0 %	443	0	443
BankAccountManagementSystem_242	100.0 %	142	0	142
TDD_24278688.java	100.0 %	301	0	301