Software testing and Inspection
Assignment 3: Test Driven Development

Kirtika Thakur
23241233

Table of Contents

Introduction

This report outlines the development process of a Bank Account Management System using Test-Driven Development (TDD) principles. TDD is a software development approach where the tests are written before the actual code gets developed. This report details the steps taken, iterations made, and the rationale behind each decision throughout the development process.

Development Process Overview

The development process was divided into several requirements, each aligned with a specific functionality of the Bank Account Management System. For each requirement, I followed the TDD cycle: writing a failing test, implementing the functionality to make the test pass, and then refactoring the code as necessary.

Requirement 1: Account Creation

Step 1: Write Failing Tests

Writing tests to ensure that the system allows the creation of a new bank account with a specified initial balance. The tests included:

- Test Case: `testAccountCreationWithPositiveInitialBalance`
    - Purpose: check if a new account can be created with a positive initial balance.
- Test Case: `testAccountCreationWithNegativeInitialBalance`
    - Purpose: Ensure that an exception is thrown when an attempt is made to create an account with a negative initial balance.
- Test Case: `testPreventDuplicateAccountCreation`
    - Purpose: Check that the system prevents the creation of duplicate accounts.

Step 2: Implement Functionality

Implemented the BankAccount class to handle account creation, ensuring that it checks for positive initial balances and duplicate account IDs.

Step 3: Refactor

The initial implementation was straightforward and did not require significant refactoring.

Requirement 2: Deposit

Step 1: Write Failing Tests

Next requirements were regarding the deposit functionality. The tests included:

- Test Case: `testDepositPositiveAmount`
  - Purpose: check that a positive amount can be deposited into the account successfully.
- Test Case: `testDepositNegativeAmount`
  - Purpose: Ensure that an exception is thrown when attempting to deposit a negative amount.

Step 2: Implement Functionality

Added a deposit method to the BankAccount class, which checks for positive deposit amounts before updating the balance.

Step 3: Refactor

The deposit method was simple and did not require any refactoring.

Requirement 3: Withdrawal

Step 1: Write Failing Tests

Implemented the withdrawal functionality with the following tests:

- Test Case: `testWithdrawValidAmount`
  - Purpose: Verify that a valid amount can be withdrawn from the account.
- Test Case: `testWithdrawNegativeAmount`
  - Purpose: Ensure that an exception is thrown when attempting to withdraw a negative amount.

Step 2: Implement Functionality

Added a withdraw method to the BankAccount class, which checks for positive withdrawal amounts and ensures that the withdrawal does not exceed the available balance.

Step 3: Refactor

The withdrawal method was straightforward and did not require significant changes.

Requirement 4: Overdraft Protection

Step 1: Write Failing Tests

Wrote the following test to implement overdraft protection:

- Test Case: `testWithdrawMoreThanBalance`
    - Purpose: Ensure that an exception is thrown when attempting to withdraw more than the available balance.

Step 2: Implement Functionality

Updated the withdraw method to include a check for sufficient funds before allowing a withdrawal.

Step 3: Refactor

The implementation was clean and did not require further refactoring.

Requirement 5: Balance Inquiry

Step 1: Write Failing Tests

Implemented the balance inquiry functionality with the following test:

- Test Case: `testBalanceInquiry`
    - Purpose: Verify that the current balance can be retrieved correctly.

Step 2: Implement Functionality

Added a getBalance method to the BankAccount class, which simply returns the current balance.

Step 3: Refactor

The balance inquiry method was straightforward and did not require any changes.

Conclusion

The Bank Account Management System was successfully developed using TDD principles. Each requirement was handled through a structured process of writing tests first, implementing the corresponding functionality, and performing necessary refactoring. This approach ensured the code was reliable, maintainable, and aligned with the defined requirements.

Task 2: Data Flow Testing

In this task, we will assess the deposit and withdraw methods of the Bank Account Management System created in Task 1 by applying data flow testing techniques. We will focus on two coverage criteria: DU-pair coverage for the deposit method and definition coverage for the withdraw method.

Part 1: Testing the deposit Method with DU-Pair Coverage

Step 1: Identify Relevant Data Variables

In the deposit method, the relevant data variables are:

- amount: The amount to be deposited.
- balance: The current balance of the account.

Step 2: Establish Definition-Use (DU) Pairs

A DU-pair consists of a variable's definition (where it is assigned a value) and its subsequent use (where it is read or utilized).

For the deposit method, the DU-pairs are:

1. DU Pair 1:
    - Definition: amount is defined when passed as a parameter to the method.
    - Use: balance is updated with balance += amount.
2. DU Pair 2:
    - Definition: balance is defined when the method is called.
    - Use: balance is used in the expression balance += amount.

Step 3: Create a Test Plan

Test Case	Description	Input	Expected Outcome	DU-Pairs Covered
TC1	Deposit a positive amount	`amount = 50.0`	Balance should increase by 50.0	DU Pair 1, DU Pair 2
TC2	Deposit a negative amount	`amount = -20.0`	Exception should be thrown: "Deposit amount must be positive"	DU Pair 1
TC3	Deposit zero amount	`amount = 0.0`	Exception should be thrown: "Deposit amount must be positive"	DU Pair 1

Step 4: Determine Coverage Level

- DU Pair 1: Covered by TC1, TC2, and TC3.
- DU Pair 2: Covered by TC1.

Coverage Level:

- DU Pair 1: Covered (3 test cases)
- DU Pair 2: Covered (1 test case)

Part 2: Testing the withdraw Method with Definition Coverage

Step 1: Identify Definitions of Data Variables

In the withdraw method, the relevant data variables are:

- amount: The amount to be withdrawn.

- balance: The current balance of the account.

Step 2: Establish Definitions

The definitions in the withdraw method are:

1. Definition 1: amount is defined when passed as a parameter to the method.
2. Definition 2: balance is defined when the method is called.

Step 3: Create a Test Plan

Created three test cases based on these definitions.

Test Case	Description	Input	Expected Outcome	Definitions Covered
TC1	**Withdraw a valid amount**	`amount = 50.0`	**Balance should decrease by 50.0**	**Definition 1, Definition 2**
TC2	**Withdraw a negative amount**	`amount = -30.0`	**Exception should be thrown: "Withdrawal amount must be positive"**	**Definition 1**
TC3	**Withdraw an amount greater than balance**	`amount = 200.0`	**Exception should be thrown: "Insufficient funds"**	**Definition 1, Definition 2**

Step 4: Determine Coverage Level

- Definition 1: Covered by TC1, TC2, and TC3.
- Definition 2: Covered by TC1 and TC3.

Coverage Level:

- Definition 1: Covered (3 test cases)
- Definition 2: Covered (2 test cases)

Summary of Coverage Levels

- Deposit Method (DU-Pair Coverage):
    - DU Pair 1: Covered
    - DU Pair 2: Covered
    - Overall Coverage: 100% for DU pairs.
- Withdraw Method (Definition Coverage):
    - Definition 1: Covered
    - Definition 2: Covered
    - Overall Coverage: 100% for definitions.

Conclusion

Using data flow testing techniques, I identified key data variables, defined DU-pairs for the deposit method, and outlined definitions for the withdraw method. The test plan thoroughly addresses all identified pairs and definitions, ensuring a comprehensive evaluation of the methods' functionality. This approach strengthens the reliability and robustness of the Bank Account Management System by thoroughly testing critical paths and data interactions. The achieved coverage reflects a solid commitment to testing principles, significantly enhancing the software's overall quality.

References:
https://stackoverflow.com/questions/19884711/data-flow-coverage