Assignment 3: Test-Driven Development

A report produced for the individual testing project in the course CS4004 Software testing and Inspection at University of Limerick

Christoffer Näs - 24271322

2024-12-08

1 TDD implementation

The chapter provides a description of the Test-Driven Development process followed in implementing the system.

1.1 Workflow

The following workflow was followed for all iterations.

1. Text for test case and code for test case was written.
2. Test was run and failed.
3. Code was written for implementation to pass test.
4. Test was tried and passed.
5. The potential for refactoring was reviewed and then performed.
6. Process restarted.

1.2 Iteration 1

1.2.1 Test case 1: Successful account creation operation

Test name: Successful account creation operation

Test Objective: Verify that an account can be successfully created with valid inputs.

Test Description: Create an account with account number 1 and an initial balance of 1000.0.

Test Steps:

1. Initialize the bank account management system.
2. Call the *createAccount* method with account number 1 and balance 1000.0.
3. Retrieve the list of accounts using the *getAccount* method.
4. Check if the new account is present.

Expected Outcome: The account is successfully added to the system, and the balance is correctly stored.

Test rationale: Ensures system can create accounts with valid inputs to fulfil requirement 1.1.

Test method name: *testCreateAccount*

1.2.2 Implementation

A class BankAccountsManagementSystem with attribute accounts and the method *createAccount* to fulfil the basic description of the system. The method *getAccount* is also created on the request of test case 1 to be able to test the *createAccount* method properly. To maintain encapsulation and protect the internal state of the HashMap accounts it is returned as a package-private unmodifiable copy.

```java
import java.util.HashMap;
import java.util.Map;

public class BankAccountManagementSystem {

    private Map<Integer, Double> accounts;

    public BankAccountManagementSystem() {
            this.accounts = new HashMap<>();
    }

    public void createAccount(int accountNumber, double initialBalance) {
            accounts.put(accountNumber, initialBalance);
    }

    Map<Integer, Double> getAccounts() {
            return Collections.unmodifiableMap(new HashMap<>(accounts));
    }
}
```

Figure 1. Code for implementation 1 to pass test case 1.

1.2.3 Refactoring

No refactoring was needed for this iteration.

1.3 Iteration 2

1.3.1 Test case 2: Validate rejection of creation on negative initial balance

Test name: Validate rejection of creation on negative initial balance

Test Objective: Verify that an account cannot be created using negative values.

Test Description: Create an account with account number 1 and an initial balance of -500.0.

Test Steps:

1 Initialize the bank account management system.
2 Call the *createAccount* method with account number 1 and balance -500.0.
3 Retrieve the list of accounts using the *getAccount* method.
4 Check if the new account is absent.

Expected Outcome: The account is not added to the system.

Test rational: Ensures system rejects the creation of accounts with negative initialBalance to prevent faults in the system and to cover requirement 1.2.

Test method: *testNegativeInitialBalance*

1.3.2 Implementation

An if statement to check that initial balance is positive before adding account to system is added to provide minimal code to pass test case 2.

```
public void createAccount(int accountNumber, double initialBalance) {
            if (initialBalance < 0) {
                        accounts.put(accountNumber, initialBalance);
            }
}
```

Figure 2. createAccount method updated to pass test case 2.

1.3.3 Refactoring

Added module comment for testing file and comments for test file were improved.

1.4 Iteration 3

1.4.1 Test case 3: Validate rejection of duplicate account creation

Test name: Validate rejection of duplicate account creation

Test Objective: Verify that system reject creation of duplicate accounts.

Test Description: Create an account with account number 1 when there is already an account with account number 1.

Test Steps:

1 Initialize the bank account management system.
2 Call the *createAccount* method with account number 1 and balance 1000.0.
3 Call the *createAccount* method again with account number 1 and balance 1000.
4 Retrieve the list of accounts using the *getAccount* method.
5 Check if the new account is absent.

Expected Outcome: The account is not added to the system.

Test rational: Ensures system rejects the creation of duplicate accounts to prevent faults and to cover requirement 1.3.

Test method name: *testDuplicateAccountCreation*

1.4.2 Implementation

To provide minimal code to pass test case 3, the if statement is updated to check if there is already an account with the same account number in accounts.

```
public void createAccount(int accountNumber, double initialBalance) {
            if (!accounts.containsKey(accountNumber) && initialBalance > 0) {
                        accounts.put(accountNumber, initialBalance);
            }
}
```

Figure 3. createAccount method updated to pass test case 3.

1.4.3 Refactoring

Added correct spacing between functions in test file.

1.5 Iteration 4

1.5.1 Test case 4: Validate successful deposit operation

Test name: Validate successful deposit operation

Test Objective: Verify that system can successfully perform a deposit operation on an account.

Test Description: Try to perform a deposit operation on one of the systems accounts.

Test Steps:

1 Initialize the bank account management system.
2 Call the *createAccount* method with account number 1 and balance 1000.0.
3 Call the *deposit* method with accountNumber 1 and amount 100.
4 Retrieve the list of accounts using the *getAccount* method.
5 Check if system contains accounts with correct balances.

Expected Outcome: The accounts balance is not updated by the system and still contains 900.0.

Test rational: Ensures system are able to perform the deposit method the creation and to fulfil requirement 2.1.

Test method name: *testDuplicateAccountCreation*

1.5.2 Implementation

To provide minimal code to pass test case 4, a new method *deposit* is created to provide deposit operation to the system.

```
public void deposit(int accountNumber, double amount) {
              Double balance =  accounts.get(accountNumber);
              double newBalance = balance + amount;
              accounts.put(accountNumber, newBalance);
}
```

Figure 4. deposit method added to pass test case 4.

1.5.3 Refactoring

Added docstring to implementation file for all added functions in implementation file.

1.6 Iteration 5

1.6.1 Test case 5: Validate that system rejects negative amount deposits

Test name: Validate that system rejects of negative amount deposits

Test Objective: Verify that system can successfully reject a deposit operation provided a negative amount.

Test Description: Try to perform a deposit operation on one of the systems accounts using a negative deposit amount.

Test Steps:

1. Initialize the bank account management system.
2. Call the *createAccount* method with account number 1 and balance 1000.0.
3. Call the *deposit* method with accountNumber 1 and amount -100.
4. Retrieve the list of accounts using the *getAccount* method.
5. Check if system contains account with correct balances and account numbers.

Expected Outcome: The account is not updated and still contains a balance of 1000.0.

Test rational: Ensures system are able to reject deposits with a negative amount to fulfil requirement 2.2.

Test method name: *testDepositOperation*

1.6.2 Implementation

To provide minimal code to pass test case 5, the method *deposit* is updated to reject deposit operations on the system when the deposit amount is negative.

```
public void deposit(int accountNumber, double amount) {
            if (amount > 0) {
                        Double balance =  accounts.get(accountNumber);
                        double newBalance = balance + amount;
                        accounts.put(accountNumber, newBalance);
            }

}
```

Figure 5. deposit method updated to pass test case 5.

1.6.3 Refactoring

Added docstring to all test functions in the test file.

1.7 Iteration 6

1.7.1 Test case 6: Validate successful withdraw operation

Test name: Validate successful withdraw operation

Test Objective: Verify that system can successfully perform a withdrawal operation on an account.

Test Description: Try to perform a withdrawal operation on one of the systems accounts.

Test Steps:

1. Initialize the bank account management system.
2. Call the *createAccount* method with account number 1 and balance 1000.0.
3. Call the *withdraw* method with accountNumber 1 and amount 100.
4. Retrieve the list of accounts using the *getAccount* method.
5. Access right account with accountNumber 1.
6. Check if system contains account with correct balances and account numbers.

Expected Outcome: The account is updated and contains a balance of 900.0.

Test rational: Ensures system are able to perform withdraw operations with a valid withdraw amount on one of the systems accounts to fulfil requirement 3.1.

Test method name: *testWithdrawOperation*

1.7.2 Implementation

To provide minimal code to pass test case 6, the method *withdraw* is created to enable withdraw operations for the systems accounts when the withdraw amount is valid.

```
public void withdraw(int accountNumber, double amount) {
            Double balance = accounts.get(accountNumber);
            double newBalance = balance - amount;
            accounts.put(accountNumber, newBalance);

}
```

Figure 6. withdraw method created to pass test case 6.

1.7.3 Refactoring

No refactoring was needed during this iteration.

1.8 Iteration 7

1.8.1 Test case 7: Validate that system rejects negative withdraw amounts

Test name: Validate that system rejects negative withdrawal amounts

Test Objective: Verify successful rejection of withdrawal operations when trying to withdraw a negative amount.

Test Description: Try to perform a withdrawal operation on one of the systems accounts using a negative withdrawal amount.

Test Steps:

1. Initialize the bank account management system.
2. Call the *createAccount* method with account number 1 and balance 1000.0.
3. Call the *withdraw* method with accountNumber 1 and amount -100.
4. Retrieve the list of accounts using the *getAccount* method.
5. Check if system contains accounts with correct balances and account numbers.

Expected Outcome: The account is not updated and still contains a balance of 1000.0.

Test rational: Ensures system are able to reject withdraw operations with a negative withdraw amount on one of the systems accounts to fulfil requirement 3.2.

Test method name: *testWithdrawNegativeAmount*

1.8.2 Implementation

To provide minimal code to pass test case 7, the method *withdraw* is updated to prevent withdraw operations with a negative withdraw amount.

```
public void withdraw(int accountNumber, double amount) {

            if (amount > 0) {
                        Double balance = accounts.get(accountNumber);
                        double newBalance = balance - amount;
                        accounts.put(accountNumber, newBalance);
            }
}
```

Figure 7. Update of withdraw method to pass test case 7.

1.8.3 Refactoring

No refactoring was performed during this iteration.

1.9 Iteration 8

1.9.1 Test case 8: Validate that system prevents overdraft on withdraw

Test name: Validate that system prevents overdraft on withdraw

Test Objective: Verify that system prevents account withdrawals if the account balance would become negative after withdrawal.

Test Description: Try to perform a withdrawal operation on one of the systems accounts which have a withdraw amount higher than the account balance.

Test Steps:

1. Initialize the bank account management system.
2. Call the *createAccount* method with account number 1 and balance 1000.0.
3. Call the *withdraw* method with accountNumber 1 and amount 2000.
4. Retrieve the list of accounts using the *getAccount* method.
5. Check if system contains accounts with correct balances and account numbers.

Expected Outcome: The account is not updated and still contains a balance of 1000.0.

Test rational: Ensures system are able to provide overdraft protection to fulfil requirement 4.1 and 4.2.

Test method name: *testOverdraftProtection*

1.9.2 Implementation

To provide minimal code to pass test case 8, the method *withdraw* is updated to prevent withdraw operations were the balance end up negative.

```
public void withdraw(int accountNumber, double amount) {
            Double balance = accounts.get(accountNumber);
            if (amount > 0 && balance >= amount) {
                        double newBalance = balance - amount;
                        accounts.put(accountNumber, newBalance);
            }
}
```

Figure 8. Update of withdraw method to pass test case 8.

1.10 Iteration 9

1.10.1 Test case 9: Validate that system successfully implements inquiry operation

Test name: Validate that system successfully implements inquiry operation

Test Objective: Verify that system successfully implements inquiry operation that lets user check their balance.

Test Description: Try to perform an inquiry operation on one of the systems accounts and check that it provides the user with the correct balance.

Test Steps:

1. Initialize the bank account management system.
2. Call the *createAccount* method with account number 1 and balance 1000.0.
3. Call the *checkAccountBalance* method with accountNumber 1.
4. Check if system contains the right balance for the account with account number 1.

Expected Outcome: The retrieved balance is 1000.0.

Test rational: Ensures system are able to provide inquiry operation to fulfil requirement 5.1.

Test method name: *testInqueryOperation*

1.10.2 Implementation

To provide minimal code to pass test case 9, the method *checkAccountBalance* is created to enable inquiry operations on the systems accounts.

```
public Double checkAccountBalance(int accountNumber) {
            Double balance = accounts.get(accountNumber);
            return balance;
}
```

Figure 9. Creation of the checkAccountBalance method to pass test case 9.

1.10.3 Refactoring

Incorrect comments were corrected and code where refactored to follow coding standards. The method *getAccounts* used for testing is moved to the bottom to make the other functions more visible for the user.

1.11 Reported known risks and problems

To write as little code as possible during the TDD process, the following known risks and problems were overlooked since they were not explicitly needed to fulfil system requirement.

1. If methods are called with non-existing accounts, there will be an error.
2. The method *getAccounts* only purpose is to be able to test the system properly and doesn't suit a purpose to fulfil requirements.

2 Data flow testing

A description of the data flow testing performed with different strategies to test the methods *deposit* and *withdraw*.

2.1 DU-pair coverage of deposit method

A description of the Definition and use-pair testing performed to test the method *deposit*.

2.1.1 Assumptions and comments for the DU-pair testing

- p-use is included as use cases
- get method calls for an internal map is counted as a use case
- put method calls for an internal map is counted as a use case
- constructor initializations are counted as definitions for the internal variables

2.1.2 Code and line definitions for testing

```
    private Map<Integer, Double> accounts;

    public BankAccountManagementSystem() {
(1)             this.accounts = new HashMap<>();
    }

(2) public void deposit(int accountNumber, double amount) {
(3)             if (amount > 0) {
(4)                     Double balance =  accounts.get(accountNumber);
(5)                     double newBalance = balance + amount;
(6)                     accounts.put(accountNumber, newBalance);
                }
  }
```

Figure 10. Line definitions for testing of deposit function. Line number I marked as (x).

2.1.3 Relevant data variables

1. accountNumber: Int
2. amount: Double
3. balance: Double
4. newBalance: Double
5. accounts: Map<Integer, Double>

2.1.4 Definition and use cases

1. Def accounts (constructor)
2. Def accountNumber, amount
3. Use amount
4. Use accounts. Def balance
5. Use balance, amount. Def newBalance
6. Use accounts, accountNumber, newBalance

2.1.5 Definition-use pairs

- accounts: (1, 4), (1, 6)
- accountNumber: (2, 6)

- amount: (2, 3), (2,5)
- balance: (4, 5)
- newBalance: (5, 6)

2.1.6 Test plan

2.1.6.1 Test case 1

Test name: Valid deposit into an existing account

Test case id: 1

Test Objective: Verify that the deposit operation works for a normal case.

Test Description: Perform a valid deposit operation on an existing account and ensure the account balance is updated correctly. The test validates the system's ability to handle normal deposit scenarios.

Test Steps:

1. Initialize the bank account management system.
2. Call the *createAccount* method with account number 1 and balance 1000.0.
3. Call the *deposit* method with account number 1 and balance number 100.0.
4. Retrieve the account balance to verify that it has been updated correctly.

Expected Outcome: The retrieved balance is 1100.0.

Test rational: This test ensures the system performs correctly under normal conditions by properly updating account balances for valid deposits. The test also ensures that all DU pairs are covered.

Tested DU-pairs:

- (1, 4) accounts defined in class constructor on line 1, used when account is retrieved on line 4.
- (1, 6) accounts defined in class constructor on line 1, used when account is retrieved on line 6.
- (2, 6) accountNumber is defined on line 2 as an input parameter, used on line 4 in get method.
- (2, 3) amount defined as an input parameter on line 2, used on line 3 to make sure amount is more than 0.
- (4, 5) balance defined as a variable when account balance is retrieved on line 4, used on line 5 to calculate and define the new balance of the account.
- (2, 5) amount is defined on line 2 as an input parameter and used on line 5 to calculate the new balance.
- (5, 6) newBalance is defined when calculating the new account balance on line 4 and used when put into accounts to update account balance.

Test case successfully covers all DU-pairs.

2.1.6.2 Test case 2

Test name: Verify deposit rejection when amount is 0

Test case id: 2

Test Objective: Verify that the deposit operation is rejected for deposits of 0.

Test Description: Try to perform an invalid deposit operation on one of the systems accounts and check that it provides the user with the correct balance.

Test Steps:

1. Initialize the bank account management system.
2. Call the *createAccount* method with account number 1 and balance 1000.0.
3. Call the *deposit* method with account number 1 and balance number 0.0.
4. Retrieve account balance to check if system contains the right balance for the account with account number 1.

Expected Outcome: The retrieved balance is 1000.0.

Test rational: Ensures system are able to reject deposit operation on invalid deposit amounts of 0. The test also ensures that certain DU-pairs are covered correctly for different conditions.

Tested DU-pairs:

- (2, 3) amount defined as an input parameter on line 2, used on line 3 to make sure amount is more than 0.

2.1.6.3 Test case 3

Test name: Verify that the deposit operation is rejected for deposits of negative amounts.

Test case id: 3

Test Objective: Ensure the system prevents deposits of negative values, leaving the balance unchanged.

Test Description: Verify that the deposit operation is rejected for deposits of negative values.

Test Steps:

1. Initialize the bank account management system.
2. Call the *createAccount* method with account number 1 and balance 1000.0.
3. Call the *deposit* method with account number 1 and balance number -100.0.
4. Retrieve account balance to check if system contains the right balance for the account with account number 1.

Expected Outcome: The retrieved balance is 1000.0.

Test rational: Ensures system are able to provide deposit operation to test normal case for deposit operation. The test also ensures that certain DU-pairs are covered correctly for different conditions.

Tested DU-pairs:

- (2, 3) amount defined as an input parameter on line 2, used on line 3 to make sure amount is more than 0.

2.1.7 Coverage level

Coverage criterion: All DU-pair

Coverage level of test cases: 100%

Rational: Test case 1 covers all 7 DU pairs for the deposit method.

Equation 1. Coverage level for du pairs for deposit method calculated.

$$coverage = \frac{nr\ of\ covered\ DU\ pairs}{total\ nr\ of\ DU\ pairs} = \frac{7}{7} = 100\%$$

2.2 Definition coverage of the withdraw method

2.2.1 Code and line definitions for testing

```
public BankAccountManagementSystem() {
(1)     this.accounts = new HashMap<>();
  }
(2) public void withdraw(int accountNumber, double amount) {
(3)                 Double balance = accounts.get(accountNumber);
(4)                 if (amount > 0 && balance >= amount) {
(5)                             double newBalance = balance - amount;
(6)                             accounts.put(accountNumber, newBalance);
                    }
  }
```

Figure 11. Line number definitions for the withdraw method. Line numbers marked as (x).

2.2.2 Definition of relevant variables

1. accountNumber: Int
2. amount: Double
3. balance: Double
4. newBalance: Double
5. accounts: Map<Integer, Double>

2.2.3 Variables definition analysis

- accounts map is defined on line 1
- amount is defined on line 2
- accountNumber is defined on line 2
- balance is defined on line 3
- newBalance is defined on line 5

2.2.4 Test plan

2.2.4.1 Test case 1

Test name: Valid withdrawal from an existing account

Test case id: 1

Test Objective: Verify that the withdrawal operation works for a normal case.

Test Description: Perform a valid withdrawal operation on an existing account and ensure that the account balance is updated correctly. The test validates the system's ability to handle normal withdrawal scenarios.

Test Steps:

1. Initialize the bank account management system.
2. Call the *createAccount* method with account number 1 and balance 1000.0.
3. Call the *withdrawal* method with account number 1 and balance number 100.0.
4. Retrieve the account balance to verify that it has been updated correctly.

Expected Outcome: The retrieved balance is 900.0.

Test rational: This test ensures the system performs correctly under normal conditions by properly updating account balances for valid withdrawals. The test also ensures that all definitions are covered.

Tested definitions:

- accounts definition is covered in initialization of the bank account management system
- amount definition is covered when defined as an input parameter on line 2.
- accountNumber is covered when defined as an input parameter on line 2.
- balance definition is covered on line 3 when balance is received from the chosen account.
- newBalance is covered when the new balance is calculated on line 5.

The test case successfully covers all definitions.

2.2.4.2 Test case 2

Test name: Valid withdrawal rejection on invalid withdrawal amounts

Test case id: 2

Test Objective: Verify that the withdrawal operation is rejected for negative and zero values.

Test Description: Perform an invalid withdrawal operation on an existing account and ensure that the account balance is not updated. The test validates the system's ability to handle withdrawal rejection on invalid withdrawal amounts.

Test Steps:

1. Initialize the bank account management system.
2. Call the *createAccount* method with account number 1 and balance 1000.0.
3. Call the *withdrawal* method with account number 1 and balance number 0.0.
4. Retrieve account balance to check if system contains the right balance for the account with account number 1.

Expected Outcome: The retrieved balance is 1000.0.

Test rational: Ensures system are able to reject withdrawal operation on invalid withdraw amounts of 0. The test also ensures that certain definitions are covered for different conditions.

Tested definitions:

- accounts map is defined on line 1
- amount is defined on line 2
- accountNumber is defined on line 2
- balance is defined on line 3

2.2.4.3 Test case 3

Test name: Verify that the withdrawal operation is rejected for withdrawals of with higher amounts than account balance.

Test case id: 3

Test Objective: Ensure the system prevents withdrawal operation when withdraw amount exceed account balance.

Test Description: Perform an invalid withdrawal operation on an existing account and ensure that the account balance is not updated. The test validates the system's ability to handle withdrawal rejection when withdraw amount exceeds account balance.

Test Steps:

1. Initialize the bank account management system.
2. Call the *createAccount* method with account number 1 and balance 1000.0.
3. Call the *withdrawal* method with account number 1 and balance number 2000.0.
4. Retrieve account balance to check if system contains the right balance for the account with account number 1.

Expected Outcome: The retrieved balance is 1000.0.

Test rational: Ensures system are able to reject withdrawal operation when withdrawal amount exceeds account balance. The test also ensures that certain definitions are covered for different conditions.

Tested definitions:

- accounts map is defined on line 1
- amount is defined on line 2
- accountNumber is defined on line 2
- balance is defined on line 3

2.2.5 Coverage level

Coverage criterion: All definitions

Coverage level of test cases: 100%

Rational: Test case 1 covers all 5 definitions for the withdraw method.

Equation 2. Coverage level for definitions for withdrawal method calculated.

$$coverage = \frac{nr\ of\ covered\ definitions}{total\ nr\ of\ definitions} = \frac{5}{5} = 100\%$$