CS4004 – Assignment 2: Criteria-Based Testing

Members: Group 31

Megan Katete ID23064684
Thelma Ofoegbu ID23329009
Christoffer Nas ID24271322
Emma O'Reilly ID23304472

Table of Contents

SUMMARY

This assignment focuses on testing the “BankAccountManagementSystem” using criteria-based testing techniques. The main goal was to identify testing requirements based on statement, branch, and condition coverage, then design and run JUnit test cases to meet these criteria.

The report includes:

- Testing requirements for all methods in the system, based on the provided code and specifications.
- Detailed test cases that address these requirements and specify the coverage achieved for each.
- An overall evaluation of the coverage levels for statement, branch, and condition coverage across the system.

Each method was thoroughly tested to handle various scenarios, including valid inputs, edge cases, and potential errors. The tests aim to ensure that the system works as expected, meets the outlined requirements, and identifies any areas for improvement or further testing.

CREATE ACCOUNT METHOD:

Task 1 (Testing Requirements) – createAccount(int accountNumber, double initialBalance)

1. **Statement Coverage Requirement:**
   Goal: Ensure every line of code in each method runs at least once
   Test cases must ensure that the following statements are executed at least once

   Statements:

   - **Statement 1:** if(accounts.containsKey(accountNumber) || initialBalance < 0)
   - **Statement 2:** return false;
   - **Statement 3:** accounts.put(accountNumber, initialBalance);
   - **Statement 4:** return true;
2. **Branch Coverage Requirement:**
   Goal: Verify that each possible branch (e.g., every if, else if, else) is tested as both true and false, covering all the different paths the code can take.

   Branches:
   - **Branch 1:** Accounts.containsKey(accountNumber) ll initialBalance<0 is false
   - **Branch 2:** Accounts.containsKey(accountNumber) ll initialBalance<0 is true

3. **Condition Coverage Requirement:**
   Goal: Ensure each individual condition in a combined statement (like amount == 0 || balance == 0) is checked both as true and false.
   Conditions:
   - **Condition 1:** initialBalance < 0

- **Condition2:** accounts.containsKey(accountNumber)

Task 2 (Junit Test Cases) – createAccount(int accountNumber, double amount)

Test method signature: void testCreateAccount(int accountNumber, double initialBalance, boolean expectedResult)

This test method uses parameterized inputs to fully test the createAccount method in the BankAccountManagementSystem class. Each test case runs through different scenarios with different inputs to test the createAccount method. The goal is to cover all statement, branch, and condition requirements and make sure the method works as intended regardless of the input. The inputs, outputs and pre-conditions for the method can be seen in table 1.

Test cases

Test case id	Precondition	accountNumber	initialBalance	Expected output	Actual output
1	accounts={Accounts{1}}	2	1	TRUE	TRUE
2	accounts={Accounts{1}}	1	1	FASLE	FALSE
3	accounts={Accounts{1}}	2	-1	FALSE	FALSE

Table 1. Table of preconditions, inputs and outputs for test cases for the createAccount method.

* accounts={Accounts{1}} denotes a hashmap accounts including the associated values within (accountNumber=1,initialBalance=1).

Test Case 1: Account can be created

Test to see that an account with a positive initial balance were an account in accounts with the same accountNumber doesn't exist is successfully created in the system.

Test Case 2: Duplicate account can not be created

Test that the system prevents the creation of a duplicate account when an account with the same accountNumber exists in accounts.

Test Case 3: Account with negative value cannot be created

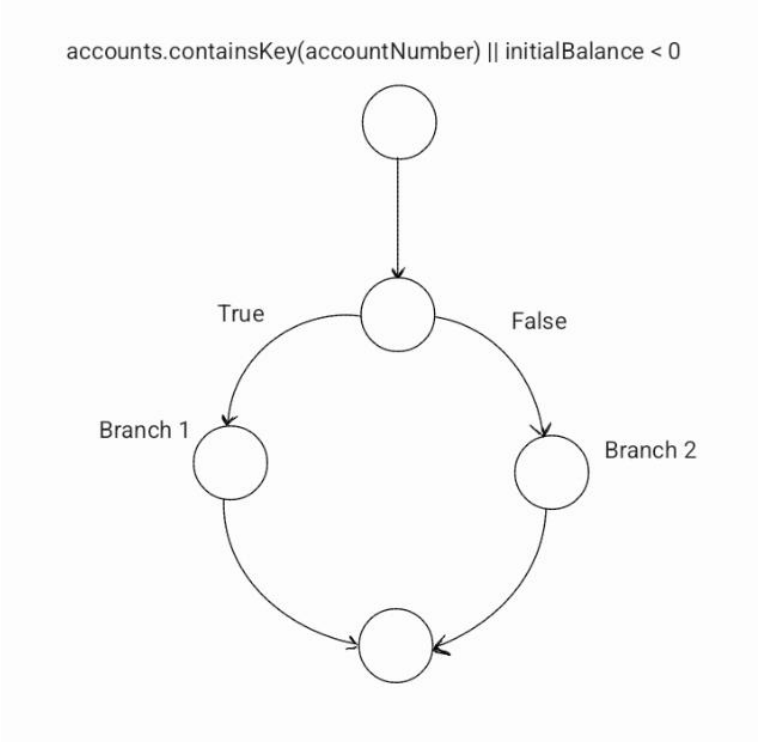
Test that the system prevents to create an account with a negative initialBalance when an account with the same accountNumber doesn't exist in accounts.

Statement coverage:

For test case 1 the PC enters the if statement and executes statements 1 and 2. For test case 3, the if statement is false and thus the PC continues past if statement an account is created the function returns true, successfully executing statements 3 and 4. All statements are now covered.

$$Statement\ coverage\ level = \frac{number\ of\ statements\ covered}{number\ of\ statements} * 100 = \frac{4}{4} * 100 = 100$$

Branch coverage:

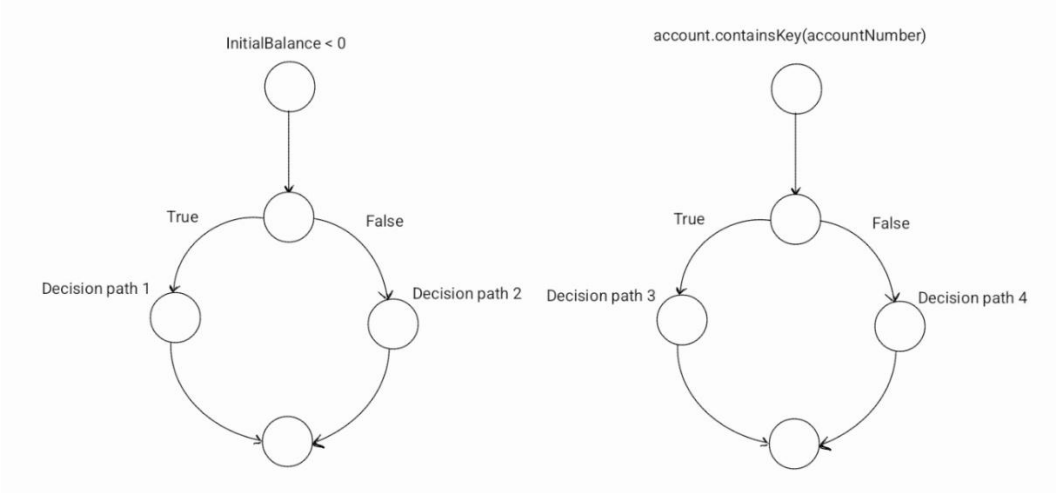


For test case 1, the if statement is deemed false since we have a positive initialBalance and accounts does not contain the accountNumber, therefore Branch 2 is executed.

For test case 2, the if statement is true since accounts contains the new accountNumber, therefore Branch 1 is executed.

$$Branch\ coverage\ level = \frac{number\ of\ branches\ covered}{number\ of\ branches} * 100 = \frac{2}{2} * 100 = 100$$

Condition coverage:



For test case 1, condition 1 is false since we have a positive initialBalance and condition 2 is false since no accounts with the same accountNumber exists in accounts. This means we successfully covers Decision path 2 and Decision path 4.

For test case 2, condition 2 is true since we already have an instance of the accountNumber in accounts, we then successfully cover decision path 3. Condition 1 is not checked in this case since the or statement Is already deemed true by condition 2.

For test case 3, condition 2 is false since no accounts with the same accountNumber exists in accounts however condition 1 is true since we have a negative initialBalance causing the condition to be true, thus we cover Decision path 1.

$$Condition\ coverage\ level = \frac{number\ of\ condition\ paths\ covered}{number\ of\ condition\ paths} * 100 = \frac{4}{4} * 100 = 100$$

Coverage Level

Covered testing requirements:

Requirement 1.1: The system must allow users to create a bank account with a unique account number and an initial balance.

Covered by: Test Case 1

- The test case confirms that an account with positive initial balance can be created and added to accounts providing the same account number does not already exist in accounts.

Requirement 1.2: If an account with the same account number already exists, the system must prevent the creation of a duplicate account

Covered by: Test Case 2

- The test cases validates that the system prevents the creation of an account if there is already the same account number in accounts.

Requirement 1.3: The system must validate that the initial balance is a non-negative value.

Covered by: Test Cases 3

- The test case verify that the system prevents creation of an account if the initial balance is negative.

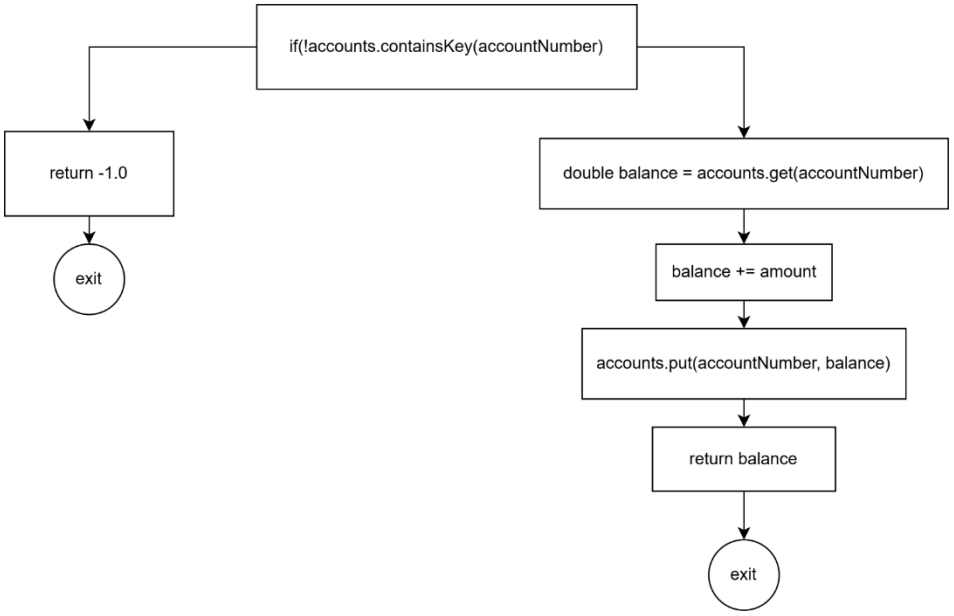
DEPOSIT METHOD:

Task 1 (Testing Requirements) – deposit (int accountNumber, double amount)

1. **Statement Coverage Requirement:**
   Goal: Ensure every line of code in each method runs at least once
   Test cases must ensure that the following statements are executed at least once.

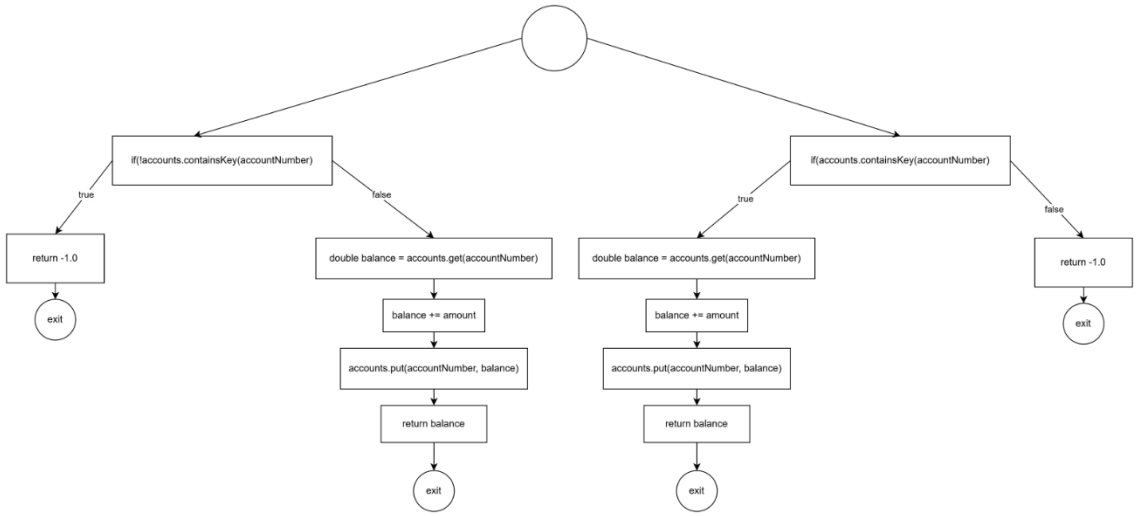
   Statement 1: Return –1.0;
   Statement 2: double balance = accounts.get(accountNumber);
   Statement 3: balance =+ amount;
   Statement 4: accounts.put(accountNumber);
   Statement 5: return balance;

2. **Branch Coverage Requirement:**
   Goal: Verify that each possible branch (e.g., every if, else if, else) is tested as both true and false, covering all the different paths the code can take.



   Branch 1: (!accounts.containsKey(accountNumber)) is true, the following should execute:
   return –1.0;
   Branch 2: (!accounts.containsKey(accountNumber)) is false, the following should execute:
   double balance = accounts.get(accountNumber);
   balance+=amount;
   accounts.put(accountNumber);
   return balance;

3. **Condition Coverage Requirement:**
   Goal: Ensure each individual condition in a combined statement (like amount == 0 || balance == 0) is checked both as true and false.



Condition 1: (!accounts.containsKey(accountNumber)) is true, the following should execute:
return –1.0;

Condition 2: (!accounts.containsKey(accountNumber)) is false, the following should execute:
double balance = accounts.get(accountNumber);
balance+=amount;
accounts.put(accountNumber);
return balance;

Condition 3: (accounts.containsKey(accountNumber)) is true, the following should execute:
double balance = accounts.get(accountNumber);
balance += amount;
accounts.put(accountNumber);
return balance;

Condition 4: (accounts.containsKey(accountNumber)) is false, the following should execute:
return –1.0;

Task 2 (Junit Test Cases) – deposit (int accountNumber, double amount)

Test method signature: void testDeposit(int accountNumber, double amount, double expectedResult)

This test method takes a set of parameterized inputs to test for 2 correct outputs

Test Cases:

Test No.	accountNumber	amount	expectedValue	Expected Output	Actual Output
1	12345	2000	-1.0	True	True
2	54321	2500.0	-1.0	True	True
3	12345	2000.0	2100.0	True	True
4	54321	2500.0	2600.0	True	True

Test Case 1: Account does not exist

The test input creates a scenario where a user attempts to deposit into an account using an account number that does not exist within the system. The deposit method should return the value –1.0, therefore the expectedValue variable is set to –1.0. Within the test method, an if statement tests for whether the expectedValue is equal to –1.0 and if so, asserts whether the actual output of the deposit method is equal to –1.0 or not.

Test Case 2: Account does not exist

The test input creates the same scenario as test case 1, using different values, where the user account does not exist and tests for whether the actual output of the deposit method is equal to –1.0 or not.

Test Case 3: Account does exist

The test input creates a scenario where a user attempts to deposit into an account that does exist within the system. The deposit method should update the balance of the account and return the updated balance, therefore the expectedValue variable is set to 2100.0 (the balance expected in the account, given that the balance is currently 100.0). Within the test method, the if statement test for whether the expectedValue is set to –1.0 and if not, creates an account with a balance of 100.0 to test the deposit method on an account which exists. The variable balanceAfterDeposit stores the result of the deposit method, which is then asserted to be equal to the expectedResult variable.

Test Case 4: Account does exist

The test input creates the same scenario as test case 3, using different values, where the user account does exist. An account is created with a balance of 100.0, the deposit method is invoked on this account, and the return value is compared with the value of expectedResult.

Coverage Level

Covered testing requirements:

Requirement 2.1: Users must be able to deposit funds into their existing accounts by providing their account number and the amount to deposit.

Covered by: Test case 3 & 4.

Requirement 2.2: Depositing funds should increase the amount balance by the specified amount

Covered by: Test case 3 & 4.

Statement coverage level:

$$Statement\ Coverage\ Level\ =\ \frac{5}{5} = 100\%$$

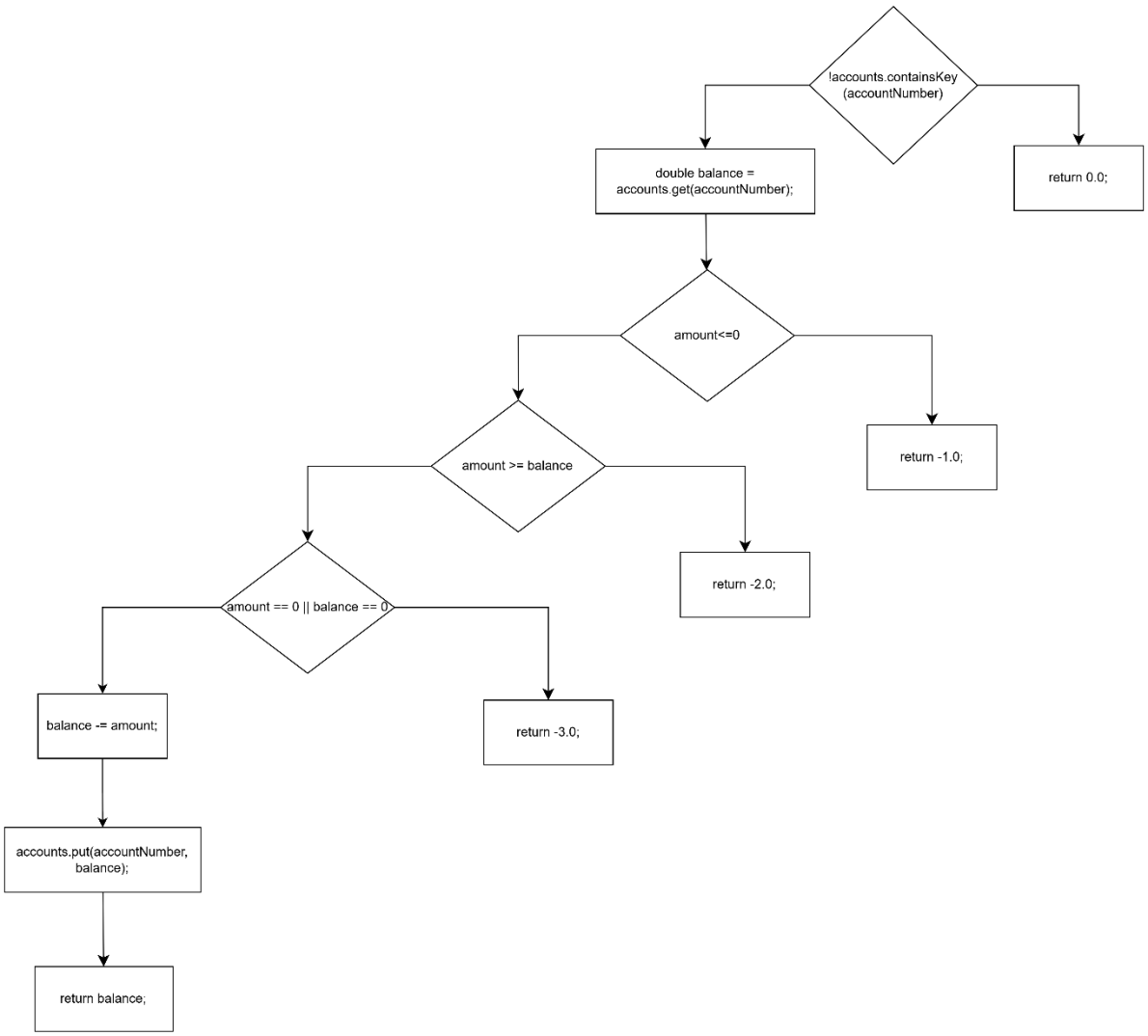
Branch coverage level:

$$Branch\ Coverage\ Level\ =\ \frac{2}{2} = 100\%$$

Condition coverage level:

$$Condition\ Coverage\ Level\ = 2\ \times \frac{2}{4} = \frac{4}{4} = 100\%$$

WITHDRAW METHOD:



Task 1 (Testing Requirements) – Withdraw (int accountNumber, double amount)

Statement Coverage Requirement:

Goal: Ensure every line of code in each method runs at least once

Test cases must ensure that the following statements are executed at least once

Statement 1: !accounts.containsKey(accountNumber)

- This statement will execute every time the method is called, determining if the account exists and whether the withdrawal process should proceed.

Statement 2: return 0.0;

- This statement executes when the account does not exist, terminating the method early and returning 0.0 to indicate no transaction occurred.

Statement 3: double balance = accounts.get(accountNumber);

- This statement retrieves the current balance of the specified account. It only executes if the account exists in the system.

Statement 4: amount <= 0

- This statement evaluates whether the withdrawal amount is less than or equal to zero, determining if the withdrawal request is invalid.

Statement 5: return -1.0;

- This statement executes when the withdrawal amount is zero or negative, signaling an invalid transaction.

Statement 6: amount >= balance

- This statement evaluates whether the withdrawal amount exceeds or equals the current balance, determining if the account has insufficient funds.

Statement 7: return -2.0;

- This statement executes when the withdrawal amount is greater than or equal to the balance, indicating that overdraft is not permitted.

Statement 8: amount == 0 || balance == 0

- This statement checks if the withdrawal amount is zero or the account balance is zero, determining if the transaction is impossible to fulfill.

Statement 9: return -3.0;

- This statement executes if either the withdrawal amount is zero or the balance is zero, signaling that the withdrawal cannot proceed.

Statement 10: balance -= amount;

- This statement deducts the withdrawal amount from the account balance. It only executes for valid withdrawals.

Statement 11: accounts.put(accountNumber, balance);

- This statement updates the account's balance in the system after a successful withdrawal.

Statement 12: return balance;

- This statement executes at the end of the method, returning the updated balance after a successful withdrawal.

Branch Coverage Requirement:

Goal: Verify that each possible branch (e.g., every if, else if, else) is tested as both true and false, covering all the different paths the code can take.

Branch 1: if (!accounts.containsKey(accountNumber))

- Branch 1a: The condition evaluates as true, meaning the account does not exist.
- Branch 1b: The condition evaluates as false, meaning the account exists.

Branch 2: if (amount <= 0)

- Branch 2a: The condition evaluates as true, meaning the withdrawal amount is zero or negative.
- Branch 2b: The condition evaluates as false, meaning the withdrawal amount is positive.

Branch 3: if (amount >= balance)

- Branch 3a: The condition evaluates as true, meaning the withdrawal amount exceeds or equals the balance.
- Branch 3b: The condition evaluates as false, meaning the withdrawal amount is less than the balance.

Branch 4: if (amount == 0 || balance == 0)

- Branch 4a: The condition evaluates as true, meaning either the withdrawal amount is zero, or the account balance is zero.
- Branch 4b: The condition evaluates as false, meaning neither the withdrawal amount nor the balance is zero.

Condition Coverage Requirement:
Goal: Ensure each individual condition in a combined statement (like amount == 0 || balance == 0) is checked both as true and false.

Condition 1: if (!accounts.containsKey(accountNumber))

- **Condition 1a**: if (!accounts.containsKey(accountNumber)) = true
  The account does not exist.
- **Condition 1b**: if (!accounts.containsKey(accountNumber)) = false
  The account exists.

Condition 2: if (amount <= 0)

- **Condition 2a**: amount = 0
  The withdrawal amount is exactly zero.
- **Condition 2b**: amount < 0
  The withdrawal amount is negative.
- **Condition 2c**: amount > 0
  The withdrawal amount is positive.

Condition 3: if (amount >= balance)

- **Condition 3a:** amount = balance
  The withdrawal amount equals the balance.
- **Condition 3b**: amount > balance
  The withdrawal amount exceeds the balance.
- **Condition 3c:** amount < balance
  The withdrawal amount is less than the balance.

Condition 4: if (amount == 0 || balance == 0)

- **Condition 4a**: amount == 0 and balance != 0
  The withdrawal amount is zero, and the balance is non-zero.
- **Condition 4b:** balance == 0 and amount != 0
  The balance is zero, and the withdrawal amount is non-zero.
- **Condition 4c:** amount != 0 and balance != 0
  Neither the amount nor the balance is zero.
- **Condition 4d**: amount == 0 and balance == 0
  Both the withdrawal amount and the balance are zero.

Task 2 (Junit Test Cases) – Withdraw (int accountNumber, double amount)

In this report, I have numbered each statement, branch, and condition in the withdraw method to make things easier to trace without repeating same lines of code. For example, Statement 1 refers to

!accounts.containsKey(accountNumber), Branch 1a tests when this is true, and Condition 2a covers amount = 0 in if (amount <= 0). This helps me keep the coverage analysis organized.

Test method signature: void testWithdraw(int accountNumber, double amount, double expectedResult, String description)

This test method uses parameterized inputs to fully test the withdraw method in the BankAccountManagementSystem class. Each test case runs through different scenarios with different input to test the withdraw method. The goal is to cover all statement, branch, and condition requirements and make sure the method works as intended regardless of the input.

Test Case 1: Account does not exist

This test checks that the withdraw method correctly handles a case where the account doesn't exist in the system. It should return 0.0, indicating that no transaction occurred.

- Input: (100, 50.0) where accountNumber = 100 (an account that doesn't exist) and amount = 50.0
- Expected Output: 0.0
- Actual Output: 0.0
- Result: Passed
- Statement Coverage: Executes statement 1 and reaches 2
- Branch Coverage: Covers branch 1a
- Condition Coverage: Satisfies condition 1a

Test Case 2: Account exists and valid withdrawal is made

This test checks that the method correctly processes a valid withdrawal when the account exists and has a sufficient balance. The withdrawal amount is less than the current balance, so the method should deduct the specified amount and return the updated balance, confirming the transaction was successful.

- Input: (101, 50.0) where accountNumber = 101 (an existing account) and amount = 50.0 (less than the balance)
- Expected Output: 4950.0
- Actual Output: 4950.0
- Result: Passed
- Statement Coverage: Executes statements 1, 3, 4, 6, 8, 10, 11, 12.
- Branch Coverage: Covers Branch 1b, 2b, 3b, 4b.
- Condition Coverage: Satisfies Condition 1b, 2c, 3c, 4c.

Test Case 3: Amount is zero

This test checks that trying to withdraw 0.0 is handled as invalid. The method should return -1.0 to show that zero isn't a valid withdrawal amount.

- Input: (101, 0.0) where accountNumber = 101 and amount = 0.0
- Expected Output: -1.0
- Actual Output: -1.0
- Result: Passed
- Statement Coverage: Executes Statement 1, 2, 4 and reaches 5.
- Branch Coverage: Covers Branch 1b, 2a.
- Condition Coverage: Satisfies Condition 1b, 2a

Test Case 4: Amount is less than balance

This test checks that a normal withdrawal, where the amount is less than the balance, works as expected. The method should deduct the withdrawal amount from the balance and return the updated balance.

- Input: (101, 1000.0) where accountNumber = 101 and amount = 1000.0
- Expected Output: 4000.0
- Actual Output: 4000.0
- Result: Passed
- Statement Coverage: Executes statements 1, 3, 4, 6, 8, 10, 11, 12.
- Branch Coverage: Covers Branch 1b, 2b, 3b, 4b.
- Condition Coverage: Satisfies Condition 1b, 2c, 3c, 4c

Test Case 5: Amount is negative

This test checks that the method rejects a withdrawal with a negative amount. Negative values should be invalid, and the method should return -1.0.

- Input: (101, -50.0) where accountNumber = 101 and amount = -50.0
- Expected Output: -1.0
- Actual Output: -1.0
- Result: Passed

- Statement Coverage: Executes statements 1, 3, 4 and reaches 5.
- Branch Coverage: Covers Branch 1b, 2a.
- Condition Coverage: Satisfies Condition 1b, 2b

Test Case 6: Amount greater than balance

This test checks that a withdrawal request larger than the current balance is not allowed as the account does not permit overdrafts. The method should return -2.0 to indicate insufficient funds.

- Input: (101, 10000.0) where accountNumber = 101 and amount = 10000.0
- Expected Output: -2.0
- Actual Output: -2.0
- Result: Passed
- Statement Coverage: Executes Statement1, 3, 4, 6 and reaches 7.
- Branch Coverage: Covers Branch 1a, 2b, 3a.
- Condition Coverage: Satisfies Condition 1b, 2c, 3b.

Test Case 7: Amount equals balance

This test checks that a withdrawal equal to the full balance is treated as an overdraft attempt. The method should return -2.0, preventing a zero balance.

- Input: (101, 5000.0) where accountNumber = 101 and amount = 5000.0
- Expected Output: -2.0
- Actual Output: -2.0
- Result: Passed
- Statement Coverage: Executes Statement 1, 3, 4, 6 and reaches 7.
- Branch Coverage: Covers Branch 1b, 2b, 3a.
- Condition Coverage: Satisfies Condition 1b, 2c, 3a

Test Case 8: Zero balance, non-zero amount

This test checks that attempting to withdraw from an account with a zero balance returns -3.0, indicating that there aren’t enough funds for the transaction.

- Input: (999, 50.0) where accountNumber = 999 (new account with zero balance) and amount = 50.0

- Expected Output: -3.0
- Actual Output: -2.0
- Result: Failed
- Statement Coverage: Executes Statement 1, 3, 4, 6, and reaches 7
- Branch Coverage: Covers Branch 1b, 2b, 3a.
- Condition Coverage: Satisfies Condition 1b, 2c, 3b

Test Case 9: Non-zero balance, amount zero

This test checks that trying to withdraw 0.0 from an account with funds is flagged as invalid. The method should return -1.0 because zero is not a valid withdrawal amount.

- Input: (101, 0.0) where accountNumber = 101 and amount = 0.0
- Expected Output: -1.0
- Actual Output: -1.0
- Result: Passed
- Statement Coverage: Executes statements 1, 3, 4 and reaches 5.
- Branch Coverage: Covers Branch 1b, 2a
- Condition Coverage: Satisfies Condition 1b, 2a

Test Case 10: Both balance and amount zero

This test checks that trying to withdraw 0.0 from an account with a zero balance also returns -3.0, as there are no funds to withdraw and the amount is invalid.

- Input: (999, 0.0) where accountNumber = 999 and amount = 0.0
- Expected Output: -3.0
- Actual Output: -1.0
- Result: Failed
- Statement Coverage: Executes statements 1, 3, 4 and reaches 5.
- Branch Coverage: Covers Branch 1b, 2a
- Condition Coverage: Satisfies Condition 1b, 2a

Test Case 11: Valid withdrawal, both amount and balance non-zero

This test checks that a withdrawal with a non-zero balance and non-zero amount works as expected. The method should deduct the amount from the balance and return the new balance, confirming a successful transaction.

- Input: (101, 500.0) where accountNumber = 101 and amount = 500.0
- Expected Output: 4500.0
- Actual Output: 4500.0
- Result: Passed
- Statement Coverage: Executes statements 1, 3, 4, 6, 8, 10, 11, 12.
- Branch Coverage: Covers Branch 1b, 2b, 3b, 4b.
- Condition Coverage: Satisfies Condition 1b, 2c, 3c, 4c.

Coverage Level – void testWithdraw(int accountNumber, double amount, double expectedResult, String description)

Covered testing requirements:

Requirement 3.1: Users must be able to withdraw funds from their existing accounts by providing the account number and the amount to withdraw.

Covered by: Test Cases 2, 4, 11

- These test cases confirm that valid withdrawals can be made from existing accounts by providing the account number and amount.

Requirement 3.2: The system should validate that the account exists before allowing a withdrawal.

Covered by: Test Cases 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

- All test cases confirm that the system checks for the existence of an account before processing a withdrawal, as they involve accounts either existing or not existing in the system.

Requirement 3.3: The system should not permit overdraft; a withdrawal amount must not exceed the current account balance.

Covered by: Test Cases 6, 7, 8

- These test cases verify that the system prevents withdrawals greater than the account balance, with the method returning a specific value to indicate insufficient funds.

Requirement 3.4: Withdrawal operations should reduce the account balance by the specified amount.

Covered by: Test Cases 2, 4, 11

- These test cases ensure that, for valid withdrawals, the account balance is reduced by the correct amount and the updated balance is returned.

Statement coverage level:

$$Statement\ coverage\ level = \frac{number\ of\ statements\ covered}{number\ of\ statements} * 100 = \frac{11}{12} * 100 = 91.67\%$$

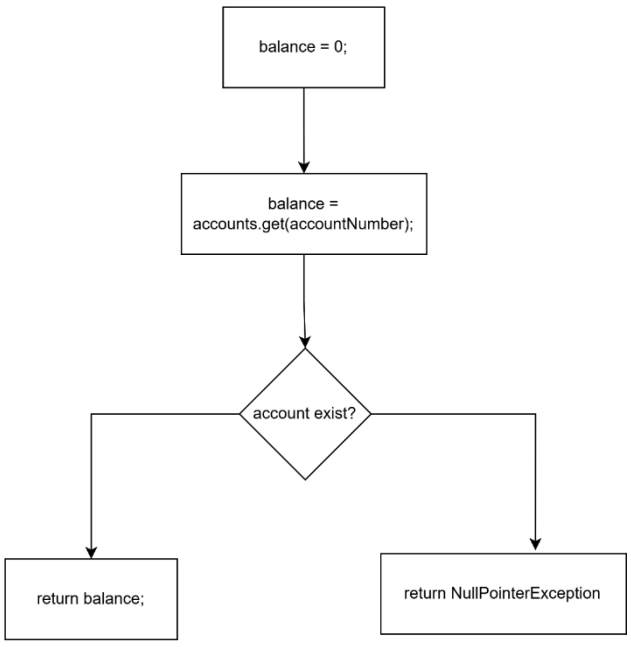
Branch coverage level:

$$Branch\ coverage\ level = \frac{number\ of\ branches\ covered}{number\ of\ branches} * 100 = \frac{7}{8} * 100 = 87.5\%$$

Condition coverage level:

$$Condition\ coverage\ level = \frac{number\ of\ condition\ paths\ covered}{number\ of\ condition\ paths} * 100 = \frac{9}{12} * 100 = 75\%$$

GET ACCOUNT BALANCE METHOD:



Task 1 (Testing Requirements) – getAccountBalance (int accountNumber)

Statement Coverage Requirement:

Goal: Ensure every line of code in each method runs at least once
Test cases must ensure that the following statements are executed at least once

1. **double balance = 0;**

This line is executed every time the getAccountBalance method is called, regardless of whether the account exists.

2. **balance = accounts.get(accountNumber);**

This statement is executed when an account number is passed to the method. It will either retrieve a value if the account exists or return null (causing a NullPointerException) if the account does not exist.

3. **return balance;**

This line executes at the end of the method, either returning the correct balance for an existing account or potentially causing a NullPointerException if the balance retrieval fails.

Branch Coverage Requirement:

Goal: Verify that each possible branch (e.g., every if, else if, else) is tested as both true and false, covering all the different paths the code can take.

Branch 1: accounts.get(accountNumber) returns a value (account exists)

- When the account number exists in the accounts HashMap, it retrieves the stored balance and sets it to balance.

Branch 2: accounts.get(accountNumber) returns null (account does not exist)

- When the account number does not exist in the accounts HashMap, accounts.get(accountNumber) returns null, which could cause a NullPointerException when trying to assign it to a double.

Condition Coverage Requirement:

Goal: Ensure each individual condition in a combined statement (like amount == 0 || balance == 0) is checked both as true and false.

Condition 1: Checking whether the account number exists in the accounts HashMap.

- a)If the account exists, accounts.get(accountNumber) returns the associated balance.
- b)If the account does not exist, accounts.get(accountNumber) returns null.

Task 2 (Junit Test Cases) – getAccountBalance (int accountNumber)

Test method signature: void testGetAccountBalanceForExistingAccount();

Test Case 1: Gets Balance for an Existing Account

This test checks that the getAccountBalance method correctly handles a case where the account exists in the system. It should verify that the account exists before returning the correct balance for that account.

In this test, the method createAccount is called to create an account with a specific account number (12121) and an initial balance of 300.0. The test then invokes the getAccountBalance method with the same account number to retrieve the balance. The returned balance is compared against the expected value (300), ensuring that the system correctly gets and displays the balance for an existing account.

- Input: (12121) where accountNumber = 12121 (an account that exists) and has an existing balance of amount = 300.0
- Expected Output: 300.0 – which is the balance of account number 12121
- Actual Output: 300.0
- Result: Passed
- Statement Coverage: Executes statement 1 and 2 reaches 3
- Branch Coverage: Covers branch 1
- Condition Coverage: Satisfies condition 1a

Test Case 2: Gets Balance for an account that does not exist

This test checks that the getAccountBalance method correctly handles a case where the account does not exist in the system. It should verify that the account does not exist then returns a null pointer exception since the account is not in the system.

- Input: (778899) where accountNumber = 778899 (an account that does not exist)
- Expected Output: a null pointer exception because this particular is not in the account hash map
- Actual Output: A null pointer exception
- Result: Passed
- Statement Coverage: Executes statements 1 and 2
- Branch Coverage: Covers branch 2
- Condition Coverage: Satisfies condition 1b

Coverage Level

Covered testing requirements:

Requirement 4.1: Users should be able to check their account balance by providing the account number.

Covered by: Test Case 1

- Test Case 1 allows users to check their account balance by providing the account number

Requirement 4.2: The system must validate that the account exists before displaying the account balance.

Covered by: Test Cases 1,2

- These test cases validate the account number exists in the accounts HashMap before trying to retrieve the account balance.

Requirement 4.3: The system should display a message if the account does not exist

Covered by: none

- From a white-box testing perspective, we can see that the current implementation of the getAccountBalance method does not contain any print statements that would display a message when an account does not exist. While the test cases validate that the system checks whether the account exists (either by returning a balance or throwing an exception), they do not capture or verify whether a message is displayed.

Statement coverage level:

$$Statement\ coverage\ level = \frac{number\ of\ statements\ covered}{number\ of\ statements} * 100 = \frac{3}{3} * 100 = 100$$

Branch coverage level:

$$Branch\ coverage\ level = \frac{number\ of\ branchs\ covered}{number\ of\ branches} * 100 = \frac{2}{2} * 100 = 100$$

Condition coverage level:

$$Condition\ coverage\ level = \frac{number\ of\ condition\ paths\ covered}{number\ of\ condition\ paths} * 100 = \frac{2}{2} * 100 = 100$$

Overall Coverage level

These percentages show the overall coverage levels for statement, branch, and condition coverage. They were calculated by adding the coverage from each method and dividing by the total possible coverage to get the final percentages.

Statement Coverage level:

$$Statement\ coverage\ level \frac{4+5+11+3}{4+5+12+3} * 100 = 95.83\%$$

Branch Coverage level:

$$Branch\ Coverage\ Level\ = \frac{2+2+7+2}{2+2+8+2} * 100 = 92.86\%$$

Condition Coverage level:

$$Condition\ Coverage\ Level\ = \frac{4+4+9+2}{4+4+12+2} * 100 = 86.36\%$$