

# Assignment 3

## Quality Control for for MyBankUML

COMP 354

Introduction to Software Engineering

Mykyta Onipchenko (40281057)

Bogdan Ivan (40283137)

Mate Barabas (40285661)

Veronika Pontolillo (40286664)

Kimberley Kam Sing (40319425)

Gorden Quach (40263250)

**Due: November 13th, 2025**

# Table of Contents

(1) Technical Review.....	2
Preparations.....	2
Roles.....	2
Issue #1.....	2
Issue #2.....	3
Issue #3.....	3
Issue #4.....	4
Issue #5.....	4
Issue #6.....	5
Issue #7.....	5
(2) Unit tests.....	6
Transaction.....	6
TransactionInfo.....	7
Account.....	8
Branch.....	11
Customer.....	12
Bank.....	13
BankDB.....	14
(3) Integration tests.....	17
LockOperation Processing.....	17
User Privacy Processing.....	18
Operation Queue.....	19
User Login.....	20
TransactionOperation Processing.....	21
Admin Information View.....	22
<b>(4) Design Revision.....</b>	<b>23</b>
(5) Appendix.....	24
Activity Log.....	24
Work Division.....	25

# (1) Technical Review

## Preparations

In a group call, the team went over the design proposed and Assignment 2. Each team member identified 1-2 issues that are worth addressing. After that, we voted on 7 and assigned them to be reviewed in this document. The issues were assigned in such a way so that the person reviewing the issue was not the one that proposed it.

## Roles

**Producers:** The entire team

**Review Leader/Recorder:** Mykyta Onipchenko

**Reviewers:** The entire team

**Agenda:** Identify and go over the issues

**Time:** Assessment can be done individually asynchronously, about 20 minutes per issue

## Issue #1

**Issue:** The objects proposed in Assignment 2 design lack certain fields necessary for storing bank information

**Reviewed by:** Mykyta Onipchenko

**Severity level:** Medium

**Results:** Inability to store relevant data, need to add these fields at some point (if done much later might require lots of code changes)

**Preparation:** 5 minutes

**Assessment:** 10 minutes

**Rework:** 40 minutes

**Vote:** 4/5. This issue is easier to address before writing DB creation and Java class files, but could also be handled in early stages of development

**How:** Review the components again, determine what other banking relevant information could be added to every component

**What:** Insert added fields to the DB (as columns or as new tables) and as class fields. Update fetching logic if necessary.

## Issue #2

**Issue:** The current system currently does not have a way to track historical informational changes (changes of names, emails, phones, etc)

**Reviewed by:** Gorden Quach

**Severity level:** Medium

**Results:** Unable to view the changes made to an account and to revert the changes if they are made.

**Preparation:** 16 mins

**Assessment:** 10 mins

**Rework:** 50 mins

**Vote:** 5/5. Having a history of changes made to an account will help manage the database better, especially when many changes have been made to an account.

**How:** implement an additional attribute to the account table in the database that stores all changes made to the account whenever changes are made.

**What:** Add an attribute (column) to the Account table named "Changes" which stores all updates made to the account attributes.

## Issue #3

**Issue:** The current system design doesn't accommodate internationalization so there's no translation of text and currency formats.

**Reviewed by:** Kimberley Kam Sing

**Severity level:** Medium

**Results:** The system won't be adaptable for users in different languages or regions which adding later might affect the restructuring of the user interface, database fields and data retrieval logic.

**Preparation:** 15 minutes

**Assessment:** 15 minutes

**Rework:** 45 minutes

**Vote:** 4/5. It is more efficient to incorporate internationalization considerations early.

**How:** Review all components like UI, database and backend logic to identify where to display, store and handle the translation content.

**What:** Add support for translation tables in the database, configure files to support multiple languages, and ensure the UI can load the translations.

## Issue #4

**Issue:** A user is not limited on the number of accounts they can have

**Reviewed by:** Bogdan Ivan

**Severity level:** Major

**Results:** The user can create an unlimited number of accounts, which could lead to spam, abuse, or data inconsistency

**Preparation:** 10 minutes

**Assessment:** 10 minutes

**Rework:** 45 minutes

**Vote:** 4/5. It's an important issue that should be addressed soon

**How:** Implement a validation check during account creation that enforces a maximum number of accounts per user (for example, based on email or SIN)

**What:** Add a constraint in the account creation logic and update the UI to display an appropriate error message when the limit is reached

## Issue #5

**Issue:** Currently, it's possible for an account to be locked for an unlimited number of times without any restriction or consequence.

**Reviewed by:** Veronika Pontolillo

**Severity level:** Medium

**Results:** Users can repeatedly request to lock their account without a limit, which is inefficient and consumes time and resources. This process should ideally be reserved for urgent or emergency situations rather than being used freely.

**Preparation:** 15 minutes

**Assessment:** 15 minutes

**Rework:** 45 minutes

**Vote:** 4/5. This is an issue that should be addressed so that locking a user's account does not become a frequent occurrence.

**How:** Introduce a validation check in the account locking process that enforces a maximum number of times that an account can be locked per user.

**What:** Add a constraint in the account locking implementation and display an error message to the UI when the user has reached the lock limit. An additional possibility could also be to have a warning error display to the UI when the user is getting close to the maximum number of times their account can be locked.

## Issue #6

**Issue:** There is a lack of clear design patterns on permissions

**Reviewed by:** Mate Barabas

**Severity level:** Medium

**Results:** The lack of standardized patterns for the permissions will result in inconsistent implementation across the application. Which would create difficulties between the workers who develop the software and a higher risk of security breaches if there is a lack of clear and concise direction.

**Preparation:** 15 minutes

**Assessment:** 15 minutes

**Rework:** 1 hour

**Vote:** 3/5. This issue should be addressed but can be delayed until the problem becomes relevant and then we can solve it from there.

**How:** Create and establish the necessary design pattern that establishes these permissions. These design patterns should establish how we check permissions, assign and remove them and how we edit them.

**What:** Define a permission checking interface or pattern such as a decorator pattern. Document how role-based or attribute-based access control functions for our application. Create reusable validation components. Possible to implement a type of registry or database that holds all users and permission levels and if a user requests to do something that is not possible for them it will be refused. As the request will check their permission level in the registry/database.

## Issue #7

**Issue:** DB integrity constraints are not clearly established

**Reviewed by:** Mykyta Onipchenko

**Severity level:** High

**Results:** Improperly validated data allows actors to create inconsistent data, posing a security risk to the customers.

**Preparation:** 10 minutes

**Assessment:** 15 minutes

**Rework:** 3 hours

**Vote:** 5/5. Not fixing this issue early will introduce severe amounts of technical debt. Plus, it involves customer security.

**How:** Create a set of guidelines on how the data should be validated.

**What:** Define reusable row datatypes (like address, SIN, phone number, etc) and corresponding CHECK and NOT NULL constraints. Establish an E/R diagram showing FOREIGN and PRIMARY KEYS. Define fields that should be UNIQUE.

## (2) Unit tests

### Transaction

Criteria	Description
Component	Transaction
Name	Test valid constructor and getters
Objective(s)	<ul style="list-style-type: none"><li>• Ensure that the constructor initializes all the fields</li><li>• Ensure getters provide access to the fields</li></ul>
Steps	<ul style="list-style-type: none"><li>• Optionally, use <code>@ParameterizedTest</code> to test at least 2 possibilities for every constructor argument</li><li>• Create an instance of the class with supplied arguments</li><li>• Check the return value of all getters</li></ul>
Expected outcome	<ul style="list-style-type: none"><li>• All getters should return the same values as they were supplied in the constructor</li></ul>
Expected exceptions	<ul style="list-style-type: none"><li>• None, only valid arguments are tested in this test</li></ul>

Criteria	Description
Component	Transaction
Name	Test invalid constructor
Objective(s)	<ul style="list-style-type: none"><li>• Ensure that the constructor throws an exception if arguments are invalid</li></ul>
Steps	<ul style="list-style-type: none"><li>• Optionally, use <code>@ParameterizedTest</code> to test each argument being invalid</li><li>• Create an instance of the class with supplied arguments</li><li>• Check if the constructor throws an exception</li></ul>
Expected outcome	<ul style="list-style-type: none"><li>• The test should throw an exception</li></ul>
Expected exceptions	<ul style="list-style-type: none"><li>• info = null ► <code>IllegalArgumentException</code></li></ul>

## TransactionInfo

Criteria	Description
Component	TransactionInfo
Name	Test valid constructor and getters
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that the constructor initializes all the fields</li> <li>• Ensure getters provide access to the fields</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use <code>@ParameterizedTest</code> to test at least 2 possibilities for every constructor argument</li> <li>• Create an instance of the class with supplied arguments</li> <li>• Check the return value of all getters</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• All getters should return the same values as were supplied in the constructor</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• None, only valid arguments are tested in this test</li> </ul>

Criteria	Description
Component	TransactionInfo
Name	Test invalid constructor
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that the constructor throws an exception if arguments are invalid</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use <code>@ParameterizedTest</code> to test each argument being invalid</li> <li>• Create an instance of the class with supplied arguments</li> <li>• Check if the constructor throws an exception</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• The test should throw an exception</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• <code>id &lt;= 0</code> ► <code>IllegalArgumentException</code></li> <li>• <code>amount = null</code> ► <code>IllegalArgumentException</code></li> <li>• <code>amount &lt;= 0</code> ► <code>IllegalArgumentException</code></li> <li>• <code>source = null</code> ► <code>IllegalArgumentException</code></li> <li>• <code>destination = null</code> ► <code>IllegalArgumentException</code></li> </ul>

## Account

Criteria	Description
Component	Account
Name	Test working constructor
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that the constructor initializes all the fields/attributes</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use <code>@ParameterizedTest</code> to test at least 2 possibilities for every constructor argument</li> <li>• Create an instance of the class with supplied arguments and parameters</li> <li>• Print the object in the console using <code>toString()</code>;</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• All constructors should store the passed parameters and all parameters should be correct when printing to the console.</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• None, testing will be done with valid parameters.</li> </ul>

Criteria	Description
Component	Account
Name	Test invalid constructor
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that the constructor throws an exception if arguments are invalid</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use <code>@ParameterizedTest</code> to test each argument being invalid</li> <li>• Create an instance of the class with supplied wrong arguments and parameters</li> <li>• Check if the constructor throws an exception</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• The test should throw an exception when calling the constructor with invalid parameters</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• <code>id &lt;= 0</code> ► <code>IllegalArgumentException</code></li> <li>• <code>name = null</code> ► <code>IllegalArgumentException</code></li> <li>• <code>customer = null</code> ► <code>IllegalArgumentException</code></li> <li>• <code>transactions= null</code> ► <code>IllegalArgumentException</code></li> </ul>

Criteria	Description
Component	Account
Name	Test getters
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure getters provide access to the fields</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Create an instance of the class with proper arguments and parameters</li> <li>• Make sure it doesn't throw an exception when creating the instance</li> <li>• Call the getters associated with the Account class</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• All getters should return the same values as the ones passed in the constructor:</li> <li>• getID(): return this.id</li> <li>• getName(): return this.name</li> <li>• getCustomer(): return all customer attributes associated to this account object</li> <li>• getTransactions(): return all transaction objects associated to this account object</li> <li>• getIsLocked():return this.isLocked (bool)</li> <li>• getBalance(): return information of the balance of the account</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• None, the getters called are only for valid instances of the Account class.</li> </ul>

Criteria	Description
Component	Account
Name	Test adding transactions to the account
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure the transaction is properly added if valid and throw an exception otherwise</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Create an instance of a Transaction object</li> <li>• Create an instance of an Account object</li> <li>• Call the addTransaction method</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• Message stating that the transaction has successfully been added to the Account if it is valid</li> <li>• Exception is thrown if invalid</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• Invalid transaction = IllegalArgumentException (or InvalidTransactionException if custom)</li> </ul>

Criteria	Description
Component	Account
Name	Setting the lock on the account
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure the account is locked when setLocked is called.</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Create an instance of an Account object with isLocked = locked/unlocked</li> <li>• Call the setLocked method to lock/unlock account</li> <li>• Call the getLocked getter to verify the changes</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• Returns the new isLocked status for the account(Lock -&gt; Unlocked/ Unlocked -&gt; Locked)</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• None, all arguments are valid in all the steps.</li> </ul>

## Branch

Criteria	Description
Component	Branch
Name	Test valid constructor and getters
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that the constructor initializes all the fields</li> <li>• Ensure getters provide access to the fields</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use <code>@ParameterizedTest</code> to test at least 2 possibilities for every constructor argument</li> <li>• Create an instance of the class with supplied arguments</li> <li>• Check the return value of all getters</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• All getters should return the same values as were supplied in the constructor</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• None, only valid arguments are tested in this test</li> </ul>

Criteria	Description
Component	Branch
Name	Test invalid constructor
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that the constructor throws an exception if arguments are invalid</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use <code>@ParameterizedTest</code> to test each argument being invalid</li> <li>• Create an instance of the class with supplied arguments</li> <li>• Check if the constructor throws an exception</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• The test should throw an exception</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• <code>id &lt;= 0</code> ► <code>IllegalArgumentException</code></li> <li>• <code>address = null</code> ► <code>IllegalArgumentException</code></li> <li>• <code>address = ""</code> (empty) ► <code>IllegalArgumentException</code></li> <li>• <code>bank = null</code> ► <code>IllegalArgumentException</code></li> </ul>

## Customer

Criteria	Description
Component	Customer
Name	Test invalid constructor
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that the constructor initializes all the fields</li> <li>• Ensure getters provide access to the fields</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use <code>@ParameterizedTest</code> to test at least 2 possibilities for every constructor argument</li> <li>• Create an instance of the class with supplied arguments</li> <li>• Check the return value of all getters</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• All getters should return the same values as were supplied in the constructor</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• None, only valid arguments are tested in this test</li> </ul>

Criteria	Description
Component	Customer
Name	Test invalid constructor
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that the constructor throws an exception if arguments are invalid</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use <code>@ParameterizedTest</code> to test each argument being invalid</li> <li>• Create an instance of the class with supplied arguments</li> <li>• Check if the constructor throws an exception</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• The test should throw an exception</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• <code>id &lt;= 0</code> ► <code>IllegalArgumentException</code></li> <li>• <code>firstName = null</code> ► <code>IllegalArgumentException</code></li> <li>• <code>firstName = ""</code> (empty) ► <code>IllegalArgumentException</code></li> <li>• <code>lastName = null</code> ► <code>IllegalArgumentException</code></li> <li>• <code>lastName = ""</code> (empty) ► <code>IllegalArgumentException</code></li> <li>• <code>dateOfBirth = null</code> ► <code>IllegalArgumentException</code></li> <li>• <code>sin = null</code> ► <code>IllegalArgumentException</code></li> <li>• <code>sin</code> does not follow <code>^\d{3}-\d{3}-\d{3}\$</code> ► <code>IllegalArgumentException</code></li> <li>• <code>accounts = null</code> ► <code>IllegalArgumentException</code></li> <li>• <code>bankBranch = null</code> ► <code>IllegalArgumentException</code></li> </ul>

## Bank

Criteria	Description
Component	Bank
Name	Test valid constructor and getters
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that the constructor initializes all the fields</li> <li>• Ensure getters provide access to the fields</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use <code>@ParameterizedTest</code> to test at least 2 possibilities for every constructor argument</li> <li>• Create an instance of the class with supplied arguments</li> <li>• Check the return value of all getters</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• All getters should return the same values as were supplied in the constructor</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• None, only valid arguments are tested in this test</li> </ul>

Criteria	Description
Component	Bank
Name	Test invalid constructor
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that the constructor throws an exception if arguments are invalid</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use <code>@ParameterizedTest</code> to test each argument being invalid</li> <li>• Create an instance of the class with supplied arguments</li> <li>• Check if the constructor throws an exception</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• The test should throw an exception</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• <code>id &lt;= 0</code> ► <code>IllegalArgumentException</code></li> <li>• <code>name = null</code> ► <code>IllegalArgumentException</code></li> <li>• <code>name = ""</code> (empty) ► <code>IllegalArgumentException</code></li> <li>• <code>branches = null</code> ► <code>IllegalArgumentException</code></li> </ul>

## BankDB

Criteria	Description
Component	BankDB
Name	Test valid constructor and getters
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that the constructor initializes all the fields (banks, connection, operationQueue)</li> <li>• Ensure the getter provides access to the field</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use <code>@ParameterizedTest</code> to test at least 2 possibilities for every constructor argument</li> <li>• Create an instance of the class with supplied arguments</li> <li>• Check the return value of the <code>getBanks()</code> getter</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• This getter should return the same values for the list of banks as what was supplied in the constructor</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• None, only valid arguments are tested in this test</li> </ul>

Criteria	Description
Component	BankDB
Name	Test invalid constructor
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that the constructor throws an exception if arguments are invalid</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use <code>@ParameterizedTest</code> to test each argument being invalid</li> <li>• Create an instance of the class with supplied arguments</li> <li>• Check if the constructor throws an exception</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• The test should throw an exception</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• banks = null ► <code>NullPointerException</code></li> <li>• connection = null ► <code>NullPointerException</code></li> <li>• connection = unusable or closed connection ► <code>IllegalStateException</code></li> <li>• operationQueue = null ► <code>NullPointerException</code></li> </ul>

Criteria	Description
Component	BankDB
Name	Test adding a new account
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that a new account is successfully added to the database</li> <li>• Verify that the account data is saved to the database and is retrievable</li> <li>• Ensure that duplicate or invalid accounts do not get stored and are rejected</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Initialize a valid BankDB instance with a mock or test database connection</li> <li>• Create a new Account object with corresponding details (id, name, customer, transactions, isLocked)</li> <li>• Add this new Account object to the database by calling the appropriate method</li> <li>• Retrieve the same Account object using the Account getter function and verify that it matches the input</li> <li>• Optionally, trigger exception handling by attempting to add a duplicate account or an invalid account object</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• A new account is successfully saved to the database and is retrievable</li> <li>• The returned account details are consistent with the input values</li> <li>• Duplicate or invalid accounts are rejected with appropriate exceptions</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• account = null ➤ NullPointerException</li> <li>• account id already exists ➤ IllegalStateException</li> <li>• account object initialized with invalid or missing fields ➤ IllegalArgumentException</li> <li>• database connection unavailable ➤ IllegalStateException</li> </ul>

Criteria	Description
Component	BankDB
Name	Test updating account balance after transfer
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that transferring funds between two accounts correctly updates both account balances</li> <li>• Verify that if one balance update fails, then both are rolled back</li> <li>• Confirm that negative balances or invalid transfer amounts are prevented</li> <li>• Validate that both source and destination accounts exist before processing</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Initialize BankDB with two valid accounts, let's say AccountA with a balance of \$1000, and AccountB with a balance of \$500</li> <li>• Execute a transfer operation from AccountA to AccountB for a valid amount of money like \$200</li> <li>• Retrieve both accounts and verify that AccountA's balance decrease to \$800 and AccountB's balance increased by that amount so it is now \$700</li> <li>• Optionally, test invalid scenarios such as invalid or negative transfer amounts to trigger exceptions</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• AccountA's balance is reduced by the transferred amount and AccountB's balance is increased by that same amount</li> <li>• The total combined balance remains constant</li> <li>• If an error occurs, no changes are applied</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• Source or destination account is not found ➤ <code>IllegalArgumentException</code></li> <li>• Insufficient funds in source account ➤ <code>IllegalStateException</code></li> <li>• Invalid transfer amount ➤ <code>IllegalArgumentException</code></li> <li>• Database connection failure ➤ <code>IllegalStateException</code></li> </ul>

### (3) Integration tests

#### LockOperation Processing

Criteria	Description
Components	BankDB, Operation, LockOperation, Account
Name	Test if issuing a LockOperation in the operation queue locks the specified account (sets the field and updates the DB)
Objective(s)	<ul style="list-style-type: none"><li>• Ensure operation queue can be written to</li><li>• Ensure BankDB processes operations from the queue</li><li>• Ensure LockOperation updates given Account's locked state</li><li>• Ensure LockOperation updates the account's row in the DB</li></ul>
Steps	<ul style="list-style-type: none"><li>• Optionally, use @ParameterizedTest to test at least 2 possibilities (locking and unlocking an account)</li><li>• Initialize a sample DB with 2 accounts</li><li>• Connect BankDB to the sample DB</li><li>• Create a new LockOperation to update the account state</li><li>• Request operation processing</li><li>• Check that the operation is no longer in the queue</li><li>• Check that given Account's locked state was updated</li><li>• Check that other Account locked state did not change</li><li>• Check that the DB row associated with a given account was updated</li><li>• Check that the DB row associated with the other account was not updated</li></ul>
Expected outcome	<ul style="list-style-type: none"><li>• Operation queue is empty</li><li>• Given account state (both in the object and in the DB row) is set to the locked state specified in LockOperation object</li><li>• Other account state (both in the object and in the DB row) did not change</li><li>• Setting an account state to the same as it has now should be a NOOP</li></ul>
Expected exceptions	<ul style="list-style-type: none"><li>• Account in LockOperation does not exist in the DB ► UnsupportedOperationException</li></ul>

## User Privacy Processing

Criteria	Description
Components	BankDB, Customer, Account
Name	Test if regular logged in users can see their own private information and cannot see the private information of other users
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure BankDB verifies session ownership before returning Customer or Account data</li> <li>• Ensure a Customer can access their own data fields</li> <li>• Ensure access to another Customer's private data is denied</li> <li>• Ensure no sensitive data leaks through DB requests</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use @ParameterizedTest to test multiple scenarios (valid session, invalid session, accessing another user's data)</li> <li>• Initialize a sample BankDB with 2 Customers, CustomerA and CustomerB, each with their own Accounts</li> <li>• Connect BankDB to the sample DB</li> <li>• "Log in" as CustomerA</li> <li>• Request CustomerA's private data</li> <li>• Request CustomerB's private data using CustomerA's session</li> <li>• Logout CustomerA, and perform a request without any active session</li> <li>• Optionally, log all request results for audit validation</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• Requests for the logged-in Customer's own data succeed</li> <li>• Requests for another Customer's data using the same Customer are rejected</li> <li>• Access without a valid session is denied</li> <li>• BankDB's Customer table and Account table remain unchanged</li> <li>• Repeated authorized reads result in consistent data with no side effects</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• Session expired or invalid ► IllegalArgumentException</li> <li>• Customer not found ► IllegalArgumentException</li> <li>• Attempt to access another Customer's private data ► IllegalArgumentException</li> </ul>

## Operation Queue

Criteria	Description
Components	BankDB, Operation, Bank
Name	Test if the operations are in first-come-first-serve order, and that they are successfully added and removed upon completion.
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure that all queued operations are executed in order and fully without interruptions</li> <li>• Validate that operations interact correctly with BankDB and Bank components</li> <li>• Verify that valid operations are successfully saved to the database</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Initialize a DB with 2 accounts and an empty operation queue</li> <li>• Connect BankDB to the sample DB with implementations of the Operation interface, such as with TransactionOperation being valid and LockOperation failing</li> <li>• Add multiple operations to the operationQueue with addOperation()</li> <li>• Call process() to simulate the queue processor running to apply changes to the database, for each operation</li> <li>• Verify that the valid operations are saved successfully and that failed changes are either removed from the queue or rolled back to not corrupt any data</li> <li>• Confirm that the data was modified only by successful operations</li> <li>• Test concurrency by adding operations from different threads to ensure thread safety is respected</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• Operations in the queue are processed in the correct order, which is the order they were added in (FIFO)</li> <li>• The database updates happen automatically, where either all changes succeed, or none are applied</li> <li>• The operations that succeed should be saved to BankDB</li> <li>• The operations that do not succeed should not be partially saved or should not save corrupted data to the database</li> <li>• The queue removes operations once completed and the queue is cleared after all operations are processed</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• Invalid operation object passed to addOperation() ► IllegalArgumentException</li> <li>• Connection is invalid or is lost mid-operation ► IllegalStateException</li> <li>• Operation execution failed unexpectedly ► RuntimeException</li> <li>• Two threads modifying the queue simultaneously ► ConcurrentModificationException</li> </ul>

## User Login

Criteria	Description
Components	BankDB, Customer
Name	Test if a user can successfully login with valid credentials and is denied access with invalid ones
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure BankDB correctly validates login credentials for stored Customer data</li> <li>• Ensure valid sin and password pairs result in successful login</li> <li>• Ensure invalid credentials like wrong sin, password or missing fields are rejected</li> <li>• Ensure login process allows access to only the matching Customer record</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use @ParametizedTest to test multiple login credential combinations (valid and invalid)</li> <li>• Initialize sample BankDB containing Customer records with login fields (sin, password)</li> <li>• Insert at least 2 customers into the database, one with valid sin and password (customerA), the other with different credentials (customerB)</li> <li>• Attempt to login using <ul style="list-style-type: none"> <li>○ 1: valid credentials for customerA</li> <li>○ 2: invalid credentials (wrong sin or password)</li> <li>○ 3: Null or empty fields for sin or password</li> </ul> </li> <li>• Check the result of the login operation (if Customer object is returned or null)</li> <li>• Verify that successful login retrieves correct Customer data from db</li> <li>• Verify invalid login do not return any Customer and throw appropriate exception</li> <li>• Verify that bankDB remains unchanged after login attempts</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• Valid credentials ► successful login, correct Customer object retrieved</li> <li>• Invalid credentials ► login rejected, no Customer object returned</li> <li>• Null/empty fields ► exception thrown</li> <li>• Database content remains unchanged</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• sin = null ► IllegalArgumentException</li> <li>• password = null ► IllegalArgumentException</li> <li>• Invalid/unregistered sin/password ► IllegalArgumentException</li> </ul>

## TransactionOperation Processing

Criteria	Description
Components	BankDB, Operation, TransactionOperation, Account, TransactionInfo
Name	Test if issuing a transaction from the transaction info class to the operation interface functions properly
Objective(s)	<ul style="list-style-type: none"> <li>• Ensure operation queue can be written to</li> <li>• Ensure BankDB processes operations from the queue</li> <li>• Ensure TransactionOperation updates given transaction info from transactionInfo which gets it from account.</li> <li>• Ensure TransactionOperation updates the account's row in the DB</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use @ParameterizedTest to test at least 2 possibilities (locking and unlocking an account)</li> <li>• Initialize a sample DB with 2 accounts</li> <li>• Connect BankDB to the sample DB</li> <li>• Create a new TransactionOperation to update the account state</li> <li>• Request operation processing</li> <li>• Check that the operation is no longer in the queue</li> <li>• Check that given Account's balance was updated</li> <li>• Check that other Account balance was increased by exact amount</li> <li>• Check that the DB row associated with a given account was updated</li> <li>• Check that the DB row associated with the other account was updated</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• Operation queue is empty</li> <li>• Given account state (both in the object and in the DB row) balance is decreased by transaction amount</li> <li>• Other account state (both in the object and in the DB row) is updated by transaction amount</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• Account in TransactionOperation does not exist in the DB ► UnsupportedOperationException</li> <li>• Amount in TransactionOperation does not exist in Account ► UnsupportedOperationException</li> </ul>

## Admin Information View

Criteria	Description
Components	BankDB, Bank, Branch, Customer, Transaction, Operation, TransactionOperation, Account, TransactionInfo
Name	Test if Admins can successfully view all private information of all classes through the admin view interface
Objective(s)	<ul style="list-style-type: none"> <li>• Authenticate user with role Admin</li> <li>• Ensure Admins can access and view the database</li> <li>• Ensure Admins can access Transaction history of customers</li> <li>• Ensure Admins can access bank information</li> <li>• Ensure Admins can access branch information</li> <li>• Ensure Admin view is not filtered by Customer/Branch</li> </ul>
Steps	<ul style="list-style-type: none"> <li>• Optionally, use @ParameterizedTest to test at least 2 possibilities (locking and unlocking an account)</li> <li>• Initialize samples for every class into the DB</li> <li>• Create and AdminUser</li> <li>• Connect BankDB to the sample DB</li> <li>• Log in through the admin interface with valid admin credentials</li> <li>• Request list of all banks and branches</li> <li>• Request list of all customers</li> <li>• Request all accounts from selected customer</li> <li>• Request full transaction history from selected customer</li> <li>• Check each list is populated and that the size of the tables match the rows in the DB</li> </ul>
Expected outcome	<ul style="list-style-type: none"> <li>• Login as Admin succeeds</li> <li>• Admin view shows all banks, branches, customers, accounts, and transactions.</li> <li>• When requesting information from the database, the counts match.</li> <li>• Field values match for all requested information.</li> <li>• No permission errors are thrown</li> </ul>
Expected exceptions	<ul style="list-style-type: none"> <li>• If BankDB connection not established ► database-related exception</li> <li>• If user without Admin role logs in ► UnauthorizedAccessException</li> <li>• If admin not found in database ► UserNotFoundException</li> </ul>

## **(4) Design Revision**

1. Missing AdminView interface ► Implement AdminView interface.
2. Lack of historical information tracking for Transactions ► Implement History attribute to Transactions inside Database.
3. Added additional component attributes to store relevant attributes.
4. Added translation content to the GUI to accommodate customers from different backgrounds.
5. Added limits to the number of opened accounts per User.
6. Introduced a validation check in the account locking process that enforces a maximum number of times that an account can be locked per user.
7. Added new design patterns to help with managing permissions.
8. Added guidelines for information validation to improve security.

## (5) Appendix

### Activity Log

Date	Name(s)	Task(s)	Time
2025-11-07	Entire team	<ul style="list-style-type: none"> <li>• Sync between team members</li> <li>• Clarified work division</li> <li>• Determined issues/tests that will be written about</li> </ul>	1 h
2025-11-08	Mykyta Onipchenko	<ul style="list-style-type: none"> <li>• Initial formatting of the document</li> <li>• “Missing fields” issue</li> <li>• “Transaction” and “Transaction Info” unit tests</li> </ul>	1 h
2025-11-09	Bogdan Ivan	<ul style="list-style-type: none"> <li>• “No number of accounts limit” issue</li> <li>• “Branch” unit tests</li> </ul>	1 h
2025-11-09	Mykyta Onipchenko	<ul style="list-style-type: none"> <li>• “LockOperation” integration test</li> </ul>	20 mins
2025-11-9	Kimberley Kam Sing	<ul style="list-style-type: none"> <li>• “No Internationalization” issue</li> </ul>	20 mins
2025-11-10	Bogdan Ivan	<ul style="list-style-type: none"> <li>• “User Privacy” integration test</li> </ul>	30 mins
2025-11-10	Veronika Pontolillo	<ul style="list-style-type: none"> <li>• “Unlimited account locking” issue</li> <li>• “BankDB” unit tests</li> </ul>	45 mins
2025-11-10	Kimberley Kam Sing	<ul style="list-style-type: none"> <li>• “Customer” unit tests</li> <li>• Start “user login” integration test</li> </ul>	30 mins
2025-11-11	Mate Barabas	<ul style="list-style-type: none"> <li>• “Lack of clear design patter for permissions” issue</li> <li>• “Branch” Unit test</li> <li>• “Transaction” integration test</li> </ul>	1 h
2025-11-11	Veronika Pontolillo	<ul style="list-style-type: none"> <li>• “Operation Queue” integration test</li> </ul>	20 mins
2025-11-11	Gorden Quach	<ul style="list-style-type: none"> <li>• Account Unit Test</li> </ul>	40 mins
2025-11-11	Kimberley Kam Sing	<ul style="list-style-type: none"> <li>• “User login” integration test</li> </ul>	30 mins
2025-11-12	Gorden Quach	<ul style="list-style-type: none"> <li>• “Admin Information View” integration test</li> <li>• Report review</li> <li>• Design Revision</li> </ul>	50 mins
2025-11-12	Mykyta Onipchenko	<ul style="list-style-type: none"> <li>• Report review and final formatting</li> </ul>	10 mins

## Work Division

- **Mykyta Onipchenko**
  - Issues: #1, #7
  - Unit tests: Transaction and TransactionInfo
  - Integration tests: LockOperation Processing
- **Bogdan Ivan**
  - Issues: #4
  - Unit tests: Branch
  - Integration tests: User Privacy Processing
- **Mate Barabas**
  - Issues: #6
  - Unit tests: Bank
  - Integration tests: TransactionOperation Processing
- **Veronika Pontolillo**
  - Issues: #5
  - Unit tests: BankDB
  - Integration tests: Operation Queue
- **Kimberley Kam Sing**
  - Issues: #3
  - Unit tests: Customer
  - Integration tests: User Login
- **Gorden Quach**
  - Issues: #2
  - Unit tests: Account
  - Integration tests: Admin Information View
  - Design Revision