ILLICITATION

Functional Requirements

The banking system is required to provide essential services that allow customers to create and manage their accounts securely and efficiently. One of the primary functions is **customer registration** , where a new customer must be able to open an account with the bank and log in securely before accessing services. Once registered, a customer can **hold multiple accounts**, including savings, investment, and cheque accounts, depending on their financial needs.

The system must also support **account transactions**. Customers should be able to deposit funds into any account they own, while specific account types impose restrictions. For example, the savings account allows deposits but does not permit withdrawals, while the investment account requires an initial deposit of at least BWP500.00 and permits both deposits and withdrawals. Cheque accounts allow free deposits and withdrawals but require proof of employment before opening.

Another critical functionality is the **automatic calculation and application of monthly interest**. Investment accounts should receive 5% monthly interest, while savings accounts accrue 0.05% monthly interest. This ensures that the system reflects realistic banking operations and provides financial benefits to customers based on account type.

Together, these functional requirements define the main operations that the banking system must perform to serve customers effectively.

Non-Functional Requirements

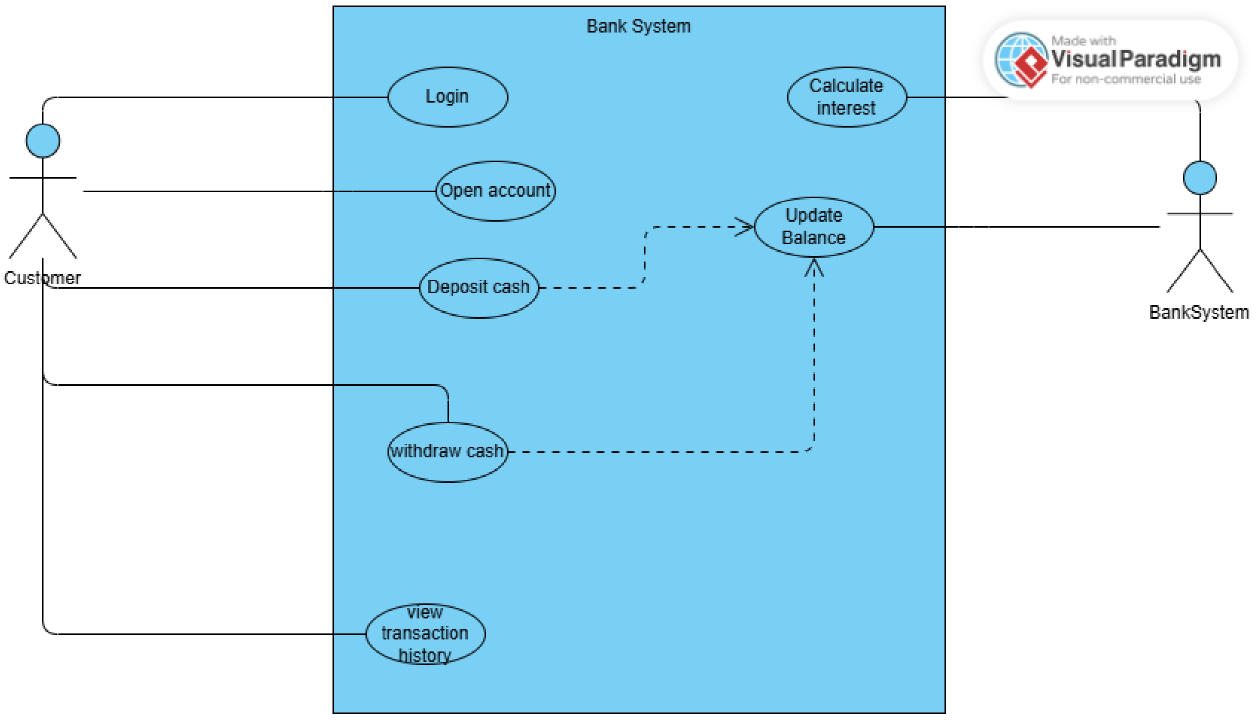
Beyond the core services, the system must also demonstrate important qualities that ensure it is practical, reliable, and user-friendly. First, the system must support **multi-user performance**, being capable of handling at least 100 users by accessing the system simultaneously without degradation in speed or responsiveness. This ensures scalability and efficient delivery service.

Since the system is developed in **Java**, it should run seamlessly on all devices and operating systems that support the Java Runtime Environment (JRE). This makes the system **portable** and widely accessible across desktops, laptops, and potentially mobile devices.

In terms of **usability**, the system must be designed in such a way that customers and bank staff can use it without the need for intensive training. The graphical user interface should therefore be intuitive and straightforward, reducing the learning curve for end-users.

Although audit logs were considered, the client confirmed they will not be implemented in the current scope. Security is still maintained through authentication, ensuring that only authorized users can access their accounts and perform transactions.

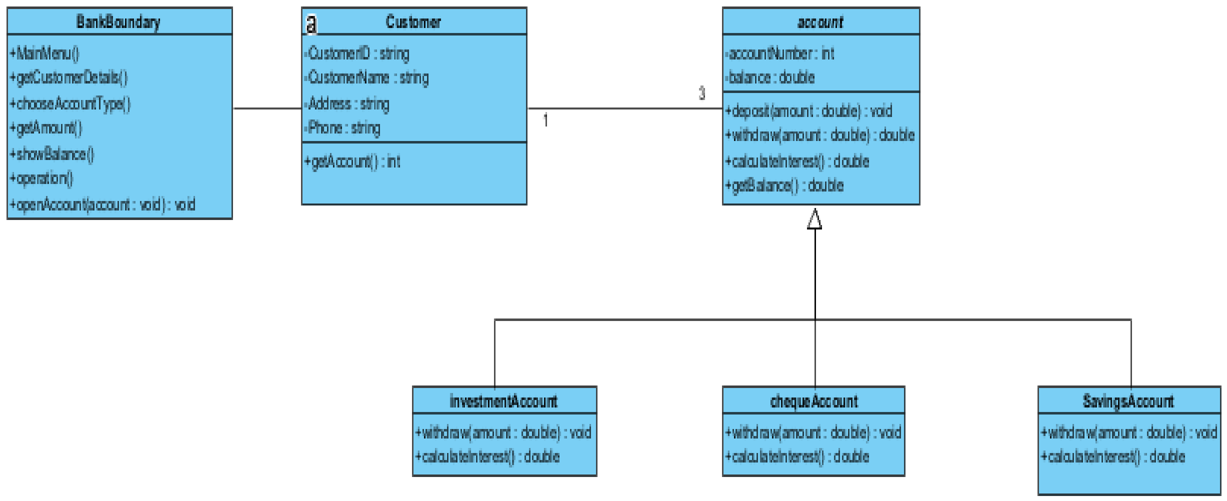
Use case diagram for the system



The Use Case Diagram shows how a **Customer** interacts with the Banking System. The main actions include **Login**, **Open Account**, **Deposit Funds**, **Withdraw Funds**, and **View transaction history**.

The diagram highlights the relationships between the customer and these key system functions

Class diagram for the system

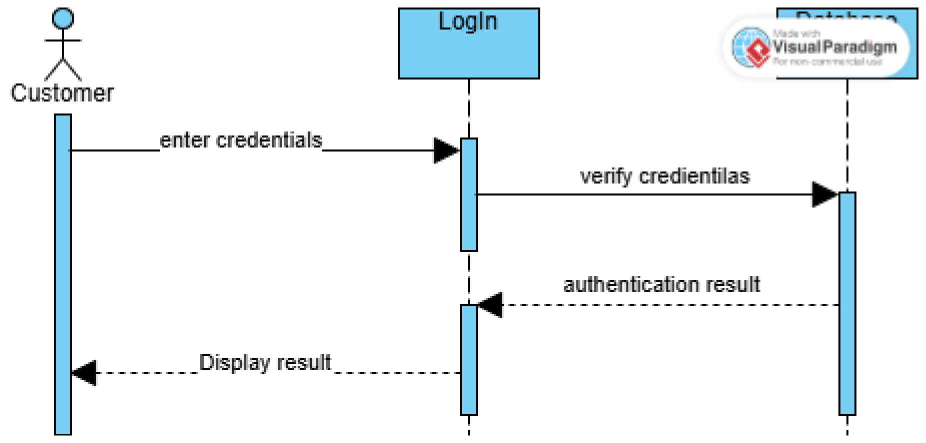

BankBoundary
+MainMenu()
+getCustomerDetails()
+chooseAccountType()
+getAmount()
+showBalance()
+operation()
+openAccount(account : void) : void
a
Customer
-CustomerID : string
-CustomerName : string
-Address : string
-Phone : string
+getAccount() : int
1
3
account
-accountNumber : int
-balance : double
+deposit(amount : double) : void
+withdraw(amount : double) : double
+calculateInterest() : double
+getBalance() : double
investmentAccount
+withdraw(amount : double) : void
+calculateInterest() : double
chequeAccount
+withdraw(amount : double) : void
+calculateInterest() : double
SavingsAccount
+withdraw(amount : double) : void
+calculateInterest() : double

The Class Diagram represents the static structure of the Banking System. The main classes include an **abstract Account** class, which defines common attributes (e.g., account number, balance, branch) and methods (e.g., deposit, withdraw). Specific account types—**Savings Account**, **Investment Account**, and **ChequeAccount**—inherit from the abstract Account class and implement or override behavior as required.

The **Customer** class holds one or more Account objects, illustrating a one-to-many relationship. The diagram demonstrates **inheritance**, **abstraction**, and **encapsulation**, providing a clear blueprint for implementing the system in Java while adhering to object-oriented principles.

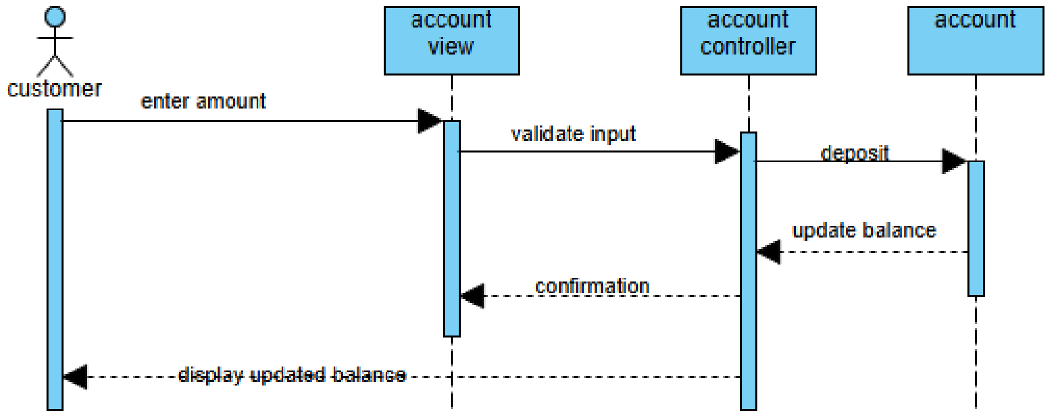
Behavioral diagrams

Sequence diagram for login feature



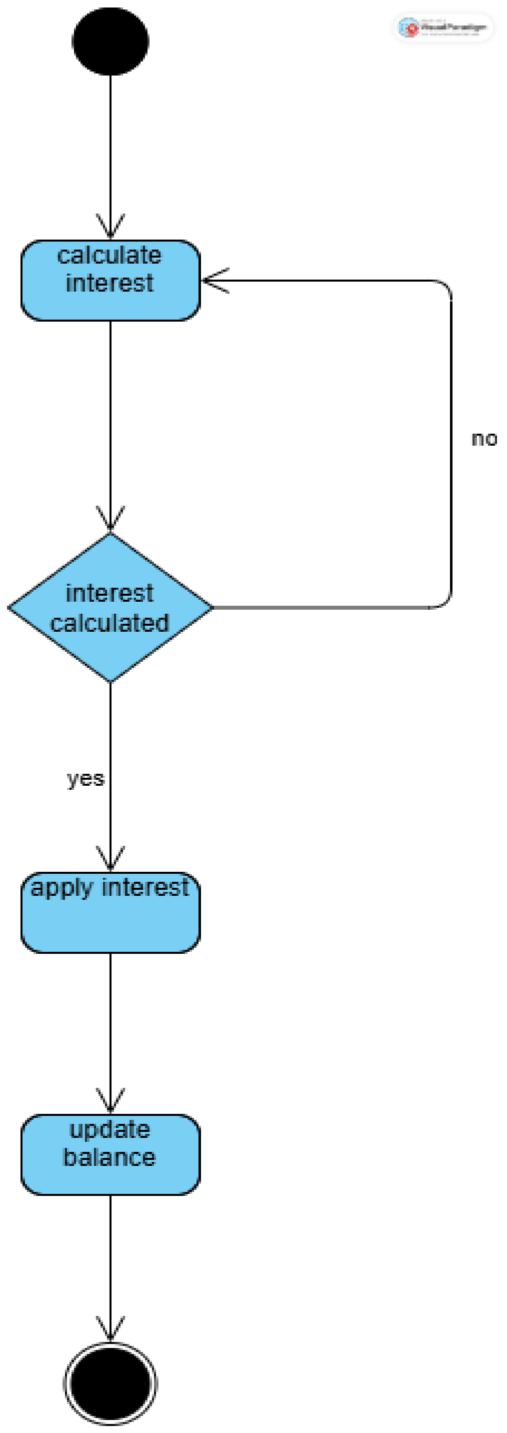
Login Sequence Diagram: This diagram shows the flow of messages when a Customer attempts to log in. The Customer provides credentials, which are validated by the **controller**. The controller interacts with the **Customer** class to verify details and returns a success or failure.

Sequence diagram for depositing funds



This diagram demonstrates how a Customer deposits money into an account. The Customer initiates a deposit via the GUI, which sends the request to the **Account Controller**. The controller then updates the relevant **Account** object by calling the deposit method and confirms the updated balance back to the GUI.

State diagram for pay interest



The State Diagram represents the lifecycle of an **Account** object when paying interest. The key states include **Interest Calculation**, **Balance Updated**, and **Interest Paid**. Transitions occur due to events like calculate Interest() or update Balance(). This diagram captures all possible states and transitions of the account during the interest payment process, showing the dynamic behavior of the system.

Appendix: Interview Record

Date and Time: 18 September 2025
Place: Microsoft Teams (Online)
Name and Role of Interviewee: (Mr. Themba Moeng)
Name of Interviewer: [Mosupi Mothibedi]
Purpose of Interview: Gather requirements for the Banking System.

Questions & Responses

Q1: Should there be an authentication process/login?
A1: Yes, this is required as a functional requirement.

Q2: How many users can the system handle at once?
A2: The system should support at least 100 users simultaneously.

Q3: Should audit logs be included?
A3: No, audit logs will not be necessary.

Q4: Should the system run on all devices?
A4: Yes, since it is Java-based, it should be portable.

Q5: Will training be required to use the system?
A5: No intensive training will be needed.

This documentation has outlined the requirements and design of the Banking System, including functional and non-functional requirements, UML diagrams, and the interview record. The diagrams demonstrate key object-oriented principles, while the requirements provide a clear foundation for development. This documentation serves as a blueprint for implementing the system in Java