***Appendix 1:  
   
Assignment submission cover sheet ASSIGNMENT   
  
SUBMISSION COVER SHEET*** *Student Id****: cse24-089****Student names****: Katlego Galetshetse*** *Student email:* ***cse24-089@thuto.bac.ac.bw*** *Cohort:* ***OOAD*** *Assignment title****: JAVA****Date of submission:* ***19 September 2025*** *Programme of Study:* ***OOAD****Year of Study:* ***Year 2****Intellectual property statement By checking the box below, I certify that this assignment is my own work and is free from plagiarism. I understand that the assignment may be checked for plagiarism by electronic or other means and may be transferred and stored in a database for the purposes of data-matching to help detect plagiarism. The assignment has not previously been submitted for assessment in any other unit or to any other institution.* *I have read and understood the Botswana Accountancy College plagiarism guidelines policy. ☐ Agree* ***Signature…K.GALETSHETSE………………………………….   
  
Date…………19/09/2025………………………………***

***Requirements Elicitation***

***Functional Requirements***

* **User Authentication:** Register, login, logout. Passwords stored hashed (controller/DAO responsibility).
* **Customer Profile Management:** Create/update customer details (names, address, contact, national ID).
* **Account Opening:** Create **Savings**, **Investment**, or **Cheque** accounts.
  + **Investment:** requires initial deposit ≥ **BWP 500**.
  + **Cheque:** requires employment *company* and *address*.
  + **Savings:** deposit-only.
* **Deposits:** Any positive amount across all accounts.
* **Withdrawals:** Allowed for Investment and Cheque; **prohibited** for Savings.
* **Interest Payment (Monthly):**
  + **Investment:** 5% per month (0.05).
  + **Savings:** 0.05% per month (0.0005).
  + Applied as a transaction and persisted.
* **Balance & History:** View current balance and transaction history by account.
* **Branch Context:** Accounts carry a branch attribute for reporting.
* **Audit & Logging:** Record every transaction with timestamp and running balance.

***Non-Functional Requirements***

* **Security:** Parameterized SQL (Prepared Statements), hashed passwords, role-based access (Customer/Admin).
* **Performance:** Single-user desktop profile; transaction operations ≤ 200ms on commodity hardware.
* **Reliability:** Atomic deposit/withdraw operations; DB constraints for referential integrity.
* **Usability:** Minimal-click JavaFX screens; clear validation messages.
* **Maintainability:** Layered architecture (View → Controller → Service → DAO → DB); unit tests; Javadoc.
* **Portability:** SQLite/H2 for frictionless local runs; JDBC abstraction for swap.

***An appendicle of the Interview Record***

* The interview between interviewer Katlego Galetshetse and interviewee (client) Kentsenao Baseki was conducted via Microsoft Teams on 18 September 2025 from 10:30 to 11:30 to gather the client’s project requirements.  
   ***Here are the requirements my client needs:*  
  Customers**: can be individuals or companies.
* Individuals can use national ID, date of birth, next of kin to open an account.
* Companies use company number, incorporation date, signatories.

**Accounts**: Savings, Investment, and Cheque.

* Minimum balances: Savings (50 pula), Investment (500 pula), Cheque (0).
* Customers can hold multiple account types, but not multiple of the same type.

**Transactions**: deposits, withdrawals, and viewing transaction history.

* Transaction history must record date, type, amount, credit/debit, and balance.

**Interest**:

* Automated calculation.
* Savings (individuals: 0.025), Companies (0.075).
* Recorded as transactions in history.

**Account Opening**:

* Account opening is done by bank staff (not the customer).
* Opening an account includes capturing customer details (first name, last name, address, ID number.
* Registering a customer and opening an account should not be the same, my client wants before opening an account, the system must register the customers details first.  
    
  ***STRUCTURAL UML MODELLING***

***System Use Case Diagram*Actors**

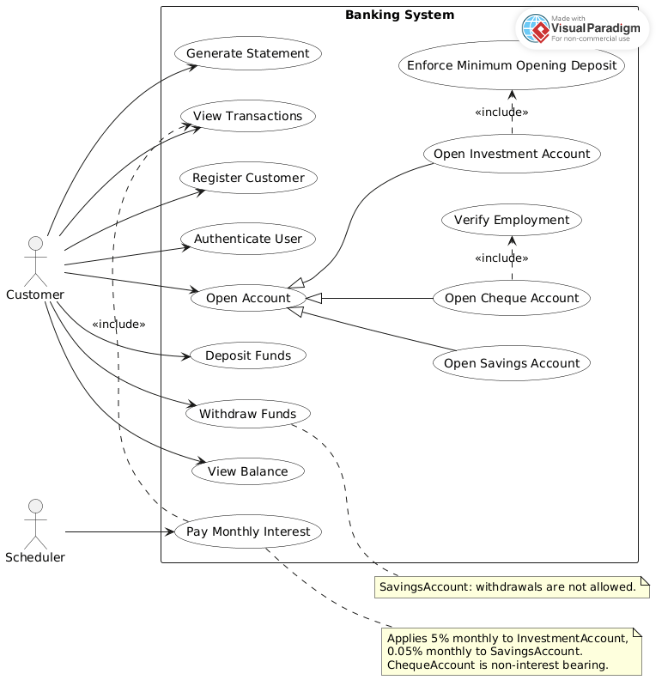
* Customer: initiates registration, authentication, account opening, deposits/withdrawals, balance/transactions, statements.
* Scheduler/Admin: a non-human actor that triggers monthly interest posting.

**Key use cases**

* Open Account specialized into Savings, Investment, Cheque (generalization).
* Employment from Open Cheque Account: Cheque requires employment metadata.
* Enforce Minimum Opening Deposit from Open Investment Account: codifies the min-capital gate.
* Pay Monthly Interest (Scheduler-driven): includes View Transactions to guarantee an auditable ledger trail.

**Business rules annotated as notes**

* Savings: withdrawals are disallowed (policy-level constraint).
* Interest policy: Investment = 5% monthly, Savings = 0.05% monthly, Cheque = non-interest-bearing.

***  
  
  
  
CLASS DIAGRAM*Core entities**

* Customer owns Account(s) (composition): an account cannot exist without a customer—good lifecycle control.
* Account (abstract): common state (account Number, branch, balance, status) and core behaviour.
* Savings Account, Investment Account, Cheque Account (inheritance): specialize withdrawal/interest behaviour.
* Transaction: immutable ledger entries linked to Account (aggregation), carrying type, amount, running Balance, timestamp, reference.

**Policy scaffolding**

* Interest-bearing (interface) implemented by Savings/Investment.
  + Enables Bank.payMonthlyInterest() to iterate polymorphically over only interest-eligible accounts.
* Employment Info composed into Cheque Account: models a hard dependency at creation.

**Encapsulation & method design**

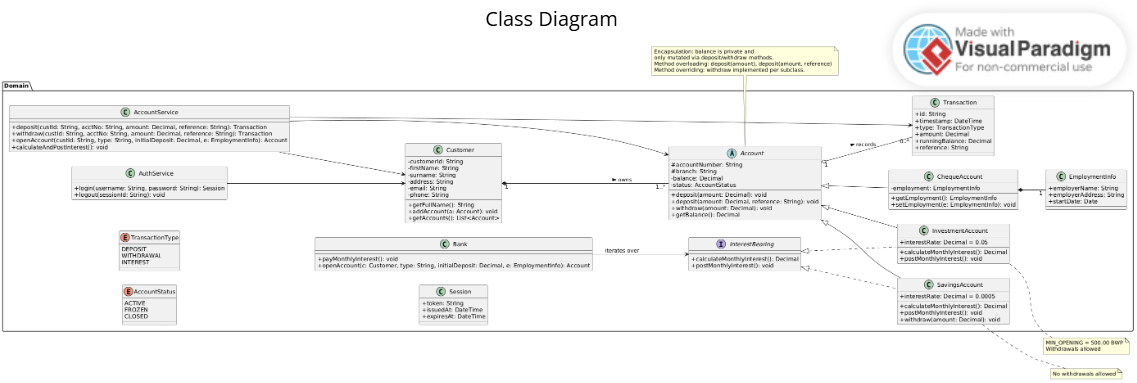
* balance is private; changes only via deposit/withdraw.
* Overloading: deposit(amount) vs deposit (amount, reference) supports richer audit trails without bloating the base signature.
* Overriding: subclasses tailor withdraws (Savings forbids; others allow per policy).

**Enumerations**

* Account Status (ACTIVE/FROZEN/CLOSED) and Transaction Type (DEPOSIT/WITHDRAWAL/INTEREST) standardize state and events for consistent downstream analytics.

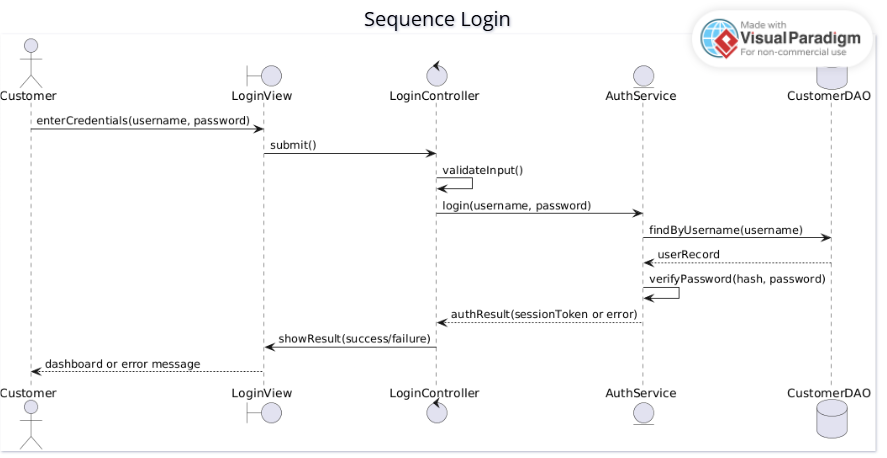
**Strong stance**

* Putting rules like “no withdrawals on Savings” directly in the subclass is the right move. Don’t outsource core invariants to controllers or DAOs—domain code must be the single source of truth.

***  
  
Behavioural UML Modelling***

***Login Sequence Diagram*Layers**

* Boundary (Login View) → Controller (Login Controller) → Service (AUTH Service) → DAO (CustomerDAO).
* Input validation upfront; password verification in the service; session token returned.

***  
  
Deposit Sequence Diagram*Flow**

1. Customer provides account + amount + reference.
2. Controller validates amount > 0.
3. Account Service loads the account, invokes domain-level deposit, persists a Transaction, returns new balance + txn id.

**Risk management**

* Single transactional unit—if persistence fails, the domain state isn’t partially mutated (your implementation should wrap this in a DB transaction). Reference strings future-proof reconciliation.

***A diagram of a sequence of information

AI-generated content may be incorrect.  
  
  
State Diagram***

**Lifecycle**

* Idle → Scheduled → Calculating → Posting → Notifying → Completed → Idle with an explicit Failed → RolledBack branch.

**Inner mechanics**

* Calculating is a sub-state that iterates through InterestBearing accounts.
* Posting creates INTEREST transactions and updates runningBalance.
* Notifying is abstracted to decouple comms (email/SMS/events) from posting.

**Operating model**

* This is production-grade: schedulable, auditable, and roll-backable. It’s exactly what ops teams want when something goes bump at 02:00.

***A diagram of a flowchart

AI-generated content may be incorrect.***