**Module Code: CSE202- OBJECT ORIENTED ANALYSIS & DESIGN WITH JAVA**

**Title of Paper: Object-Oriented Analysis and Design: Java-Based System Architecture**

**Name: Kgosi Phaphane**

**Student Registration Number: Cse24-099**

**Requirements Elicitation**

**1. Introduction**

This section presents the requirements elicitation for the Banking System. Requirements were identified through a mock interview with the bank manager and refined into functional and non-functional requirements. The aim was to capture both what the system must do and the qualities it must demonstrate.

**2. Mock Interview Summary**

A mock interview was conducted between the developer and the client (bank manager). The goal was to gather user expectations, business rules, and constraints for the banking system. The full transcript is provided in **Appendix A**.

**Key points gathered:**

* The system must support customer registration, account opening, deposits, withdrawals, and balance viewing.
* Business rules:
  + *SavingsAccount*: earns small monthly interest but withdrawals are not allowed.
  + *InvestmentAccount*: requires a minimum initial deposit of BWP500.00 and earns 5% monthly interest.
  + *ChequeAccount*: only available for employed customers, requiring employer details.
* The system should automatically pay monthly interest to eligible accounts.
* Secure login, fast performance, and ease of maintenance are important requirements.

**3. Functional Requirements**

The following functional requirements (FRs) describe the system’s essential services.

| **ID** | **Requirement** | |  | **Priority** | **Acceptance Criteria** |
| --- | --- | --- | --- | --- | --- |
| FR1 | Register Customer | |  | High | A new customer record is created with a unique ID. |
| FR2 | Open Account | |  | High | Accounts follow rules: Savings = no withdrawals, Investment = min BWP500, Cheque = employer required. |
| FR3 | Login / Authentication | |  | High | Only valid credentials grant access; invalid attempts are logged. |
| FR4 | Deposit Funds | |  | High | Balance is updated and a transaction record is created. |
| FR5 | Withdraw Funds | |  | Medium | Withdrawals are only permitted from Cheque and Investment accounts; overdraws are prevented. |
| FR6 | View Balance & Transaction History | |  | High | System displays the correct balance and recent transactions. |
| FR7 | Pay Monthly Interest | |  | High | Monthly interest is applied correctly and recorded in the system. |
|  | |

**4. Non-Functional Requirements**

The non-functional requirements (NFRs) define system qualities and constraints.

| **ID** | **Requirement** | **Description** |
| --- | --- | --- |
| NFR1 | Security | Passwords must be securely hashed; audit logs maintained; role-based access enforced. |
| NFR2 | Performance | Normal operations (e.g., deposit, balance check) must complete within 2 seconds. |
| NFR3 | Reliability | The monthly interest calculation process must succeed at least 99% of scheduled runs. |
| NFR4 | Usability | Forms must validate input and provide clear error messages. |
| NFR5 | Maintainability | The code should be modular, well-documented, and easy to extend. |

**5. Conclusion**

The elicited requirements provide a clear foundation for the design of the Banking System. They describe both the functional capabilities of the system and the non-functional qualities it must satisfy. These requirements will guide the development of UML models in subsequent sections of this assignment.

**Appendix A: Mock Interview Transcript**

**Interviewer (Developer):** What basic services should the system provide?  
**Client (Bank Manager):** The system must register customers, open accounts, accept deposits, allow withdrawals where permitted, show balances and transactions, and pay monthly interest.

**Interviewer:** What are the rules for the different accounts?  
**Client:**

* Savings accounts: pay small monthly interest but do not allow withdrawals.
* Investment accounts: require a BWP500.00 minimum deposit and pay 5% monthly interest.
* Cheque accounts: only for employed customers, employer details must be recorded.

**Interviewer:** How should interest be handled?  
**Client:** The system must automatically calculate and pay interest monthly.

**Interviewer:** What about security and performance?  
**Client:** Customers must log in securely. Passwords must be stored safely, and the system should respond quickly.

**Interviewer:** Any other expectations?  
**Client:** The system should keep transaction records and be easy to maintain.

**Structural UML Modelling**

**1. Introduction**

Structural modelling describes the static structure of the Banking System. It includes the **Use Case Diagram**, which shows the external interactions between actors and system functions, and the **Class Diagram**, which captures the internal structure of the system using object-oriented concepts.

**2. Use Case Diagram**

**2.1 Actors**

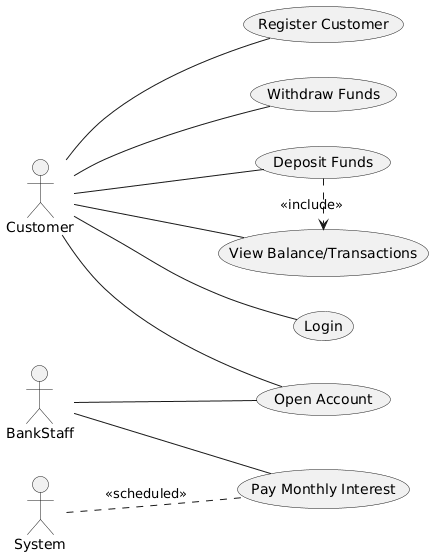
* **Customer**: Registers, logs in, opens accounts, deposits, withdraws, and views balances/transactions.
* **BankStaff**: May assist with opening accounts and managing interest payments.
* **System**: Performs automated tasks such as paying monthly interest.

**2.2 Use Cases**

* Register Customer
* Login
* Open Account
* Deposit Funds
* Withdraw Funds
* View Balance & Transaction History
* Pay Monthly Interest

**2.3 Relationships**

* Deposit Funds → includes → View Balance (after deposit, updated balance is shown).
* Pay Monthly Interest is triggered automatically by the System.



This diagram shows how **customers** interact with the system for everyday banking functions, while **bank staff** and the **system itself** handle special processes such as account creation and monthly interest. The include relationship ensures deposits always update the balance view.

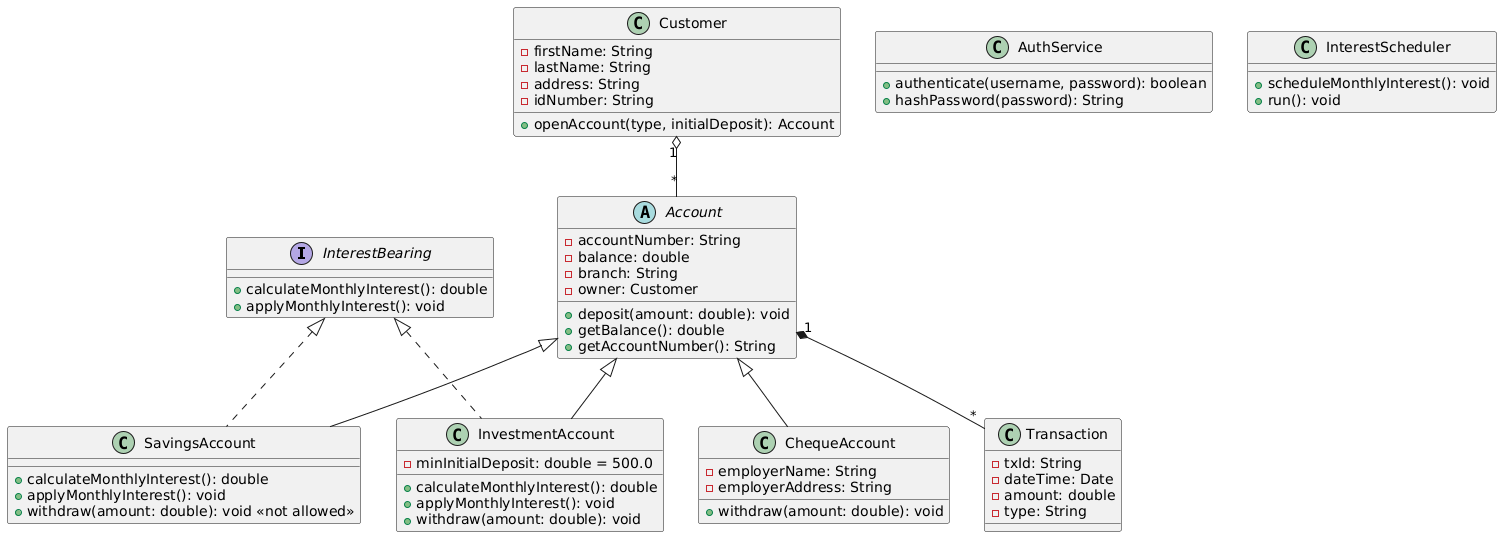
**3. Class Diagram**

**3.1 Classes and Responsibilities**

* **Customer**: Stores personal information and allows customers to open accounts.
* **Account (abstract class)**: Base for all account types; manages deposits, balances, and account details.
* **SavingsAccount**: Extends Account, prevents withdrawals, pays 0.05% monthly interest.
* **InvestmentAccount**: Extends Account, requires BWP500 minimum deposit, pays 5% monthly interest.
* **ChequeAccount**: Extends Account, requires employer details; allows withdrawals.
* **Transaction**: Records deposits, withdrawals, and interest applications.
* **InterestBearing (interface)**: Defines methods for calculating and applying interest.
* **AuthService**: Handles secure login and password hashing.
* **InterestScheduler**: Applies monthly interest automatically.

**3.2 OOP Principles Demonstrated**

* **Abstraction**: Account is an abstract class.
* **Inheritance**: SavingsAccount, InvestmentAccount, and ChequeAccount extend Account.
* **Polymorphism**: Interest calculation handled through the InterestBearing interface.
* **Encapsulation**: Private attributes with public methods (deposit(), getBalance()).
* **Overriding**: Subclasses override interest calculation methods.



**3.4 Explanation**

The class diagram demonstrates a clear object-oriented design for the banking system. Accounts are abstracted into a common superclass, specialised through inheritance, and extended with specific rules. The use of the InterestBearing interface provides polymorphism for interest calculation, while encapsulation ensures data is managed securely.

**4. Conclusion**

The structural UML modelling defines the system’s static structure. The **Use Case Diagram** shows how external actors interact with system functions, while the **Class Diagram** demonstrates how the system is organised internally using object-oriented design principles. These models provide the blueprint for system implementation.

**Behavioural UML Modelling**

**1. Introduction**

Behavioural modelling describes the **dynamic behaviour** of the system, focusing on how objects and processes interact over time. This section presents **two sequence diagrams** and **one state diagram** for the Banking System.

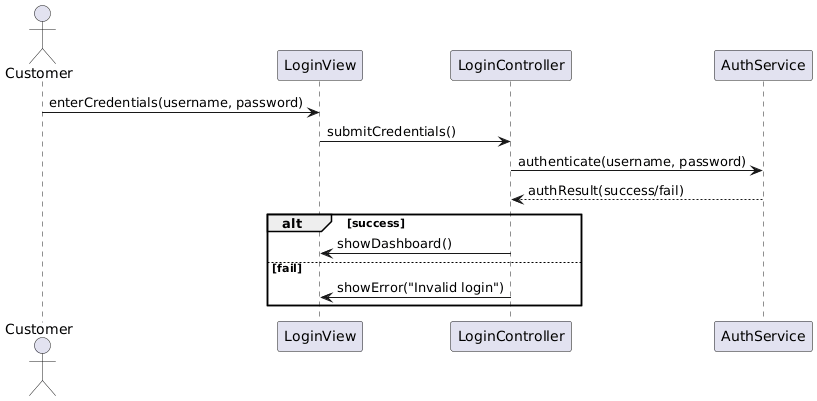
**2. Sequence Diagram 1: Login Process**

**2.1 Description**

This sequence diagram illustrates how a customer logs into the system. It shows the interaction between the Customer, LoginView, LoginController, and AuthService objects.

**Steps:**

1. The Customer enters login credentials.
2. The LoginView passes the credentials to the LoginController.
3. The LoginController calls the AuthService to authenticate.
4. AuthService validates the credentials and returns the result.
5. If successful, the customer is shown the dashboard; otherwise, an error is displayed.



**3. Sequence Diagram 2: Deposit Funds**

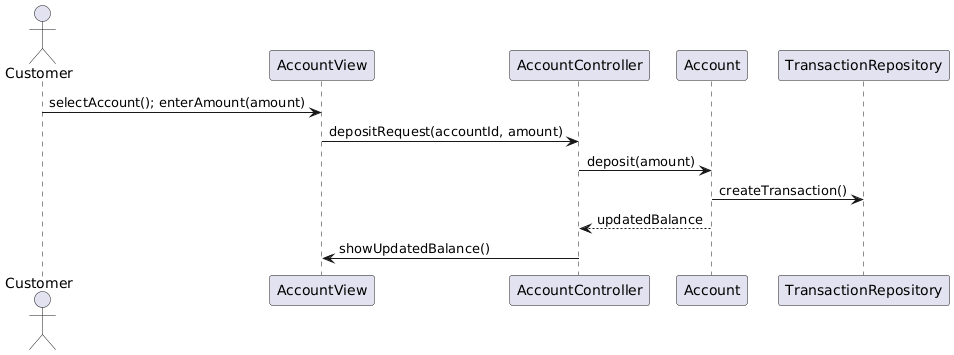
**3.1 Description**

This diagram shows how a customer deposits money into an account. It captures the flow from initiating the deposit to updating the balance and recording the transaction.

**Steps:**

1. Customer selects account and enters amount.
2. AccountView sends a deposit request to the AccountController.
3. AccountController validates and calls the deposit method in the Account.
4. Account updates balance and records the transaction.
5. Updated balance is shown to the customer.

**3.2 Diagram**

****

**4. State Diagram: Pay Interest Process**

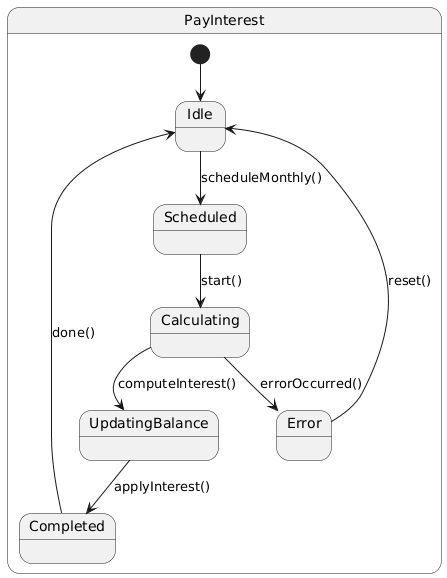
**4.1 Description**

This state diagram represents the lifecycle of the **Pay Interest** process. It shows the different states from idle to completion and how the system handles errors.

**States:**

* **Idle**: Waiting for the scheduled trigger.
* **Scheduled**: Process is queued to start.
* **Calculating**: Interest is being computed.
* **UpdatingBalance**: Account balances are updated.
* **Completed**: Process ends successfully.
* **Error**: Failure occurred; returns to Idle after reset.

4.2 Diagram



**5. Conclusion**

The behavioural models capture how the Banking System responds to dynamic events. The **Login** and **Deposit Funds** sequence diagrams show object interactions for two critical scenarios, while the **Pay Interest state diagram** describes the system’s state transitions during the automated monthly interest process. These diagrams demonstrate how the system functions in real-world use cases and provide a foundation for implementation.