**High-Level Design (HLD) for ATM Application**

**1. Introduction**

The ATM application is designed to simulate the functionality of a real-world ATM system. It allows users to log in, create accounts, perform transactions such as deposits, withdrawals, and transfers, and view transaction histories. The system ensures that sensitive data, such as user credentials and transaction details, are handled securely.

The architecture is modular, with separate components managing different responsibilities such as account management, user authentication, and transaction handling. This document provides an overview of the system's architecture, key components, interactions, and data flow.

**2. System Overview**

The system is a console-based Java application built around the following core functionalities:

1. **Login**: Users can log in to their accounts by providing a valid user ID and PIN.
2. **Account Creation**: New users can create accounts by providing basic details and an initial deposit.
3. **Transactions**: Users can deposit or withdraw money, and transfer funds between accounts.
4. **Transaction History**: Users can view their past transactions, including deposits, withdrawals, and transfers.

**3. Architectural Components**

The application follows a layered architecture with distinct roles for each layer:

1. **Presentation Layer**: Interacts with the user through the console interface, providing a simple text-based menu.
2. **Service Layer (ATM)**: Manages business logic, including validating input, managing the menu, and directing commands to the appropriate components.
3. **Data Layer (Bank, Account, Transaction)**: Manages user data, account details, and transaction history.

**4. Key Design Goals**

* **Modularity**: The system is designed to be modular, with different classes responsible for different tasks (e.g., ATM, Bank, Account, AccountHolder).
* **Security**: User authentication is handled by validating user ID and PIN, preventing unauthorized access to accounts.
* **Extensibility**: The system is designed to be easily extensible for future features, such as additional types of transactions or enhanced security features.

**5. Functional Requirements**

**5.1 Login and Authentication**

* **Description**: Users must provide a valid user ID and PIN to log in.
* **Input**: User ID and PIN.
* **Output**: Success or failure message.
* **Flow**:
  + The user is prompted to enter credentials.
  + The system checks the credentials against stored account data.
  + If valid, the user is granted access to their account.
  + If invalid, an error message is displayed.

**5.2 Account Creation**

* **Description**: New users can create an account by providing necessary details.
* **Input**: Name, user ID, PIN, and initial deposit.
* **Output**: Success message and generated account number.
* **Flow**:
  + The user is prompted to enter personal details and initial deposit.
  + A unique account number is generated.
  + The new account is stored in the system.

**5.3 Deposits, Withdrawals, and Transfers**

* **Description**: Users can perform basic financial transactions.
* **Input**: Amount of money to deposit, withdraw, or transfer, and target account for transfers.
* **Output**: Success or failure message.
* **Flow**:
  + The system checks for sufficient balance (in case of withdrawals or transfers).
  + For successful operations, the balance is updated and a transaction record is created.

**5.4 Transaction History**

* **Description**: Users can view a list of their previous transactions.
* **Input**: None.
* **Output**: List of past transactions with details (date, type, amount).
* **Flow**:
  + The system retrieves the transaction history for the user’s account and displays it.

**6. Non-Functional Requirements**

**6.1 Security**

* **Description**: Sensitive data like user credentials must be securely stored and managed.
* **Solution**: Credentials are stored as plain strings but can be hashed/encrypted in the future.

**6.2 Performance**

* **Description**: The system should respond to user input promptly.
* **Solution**: The design ensures that key operations like authentication and transactions are handled efficiently.

**6.3 Scalability**

* **Description**: The system should be able to handle a growing number of users and accounts.
* **Solution**: The system's modular design allows for easy scalability with improved data management techniques.

**7. Technology Stack**

The ATM application is developed using the following technologies:

* **Programming Language**: Java
* **Development Environment**: IDEs like IntelliJ IDEA or Eclipse can be used for development.
* **Data Storage**: In-memory storage using Java's HashMap for account holders and transaction history. Future enhancements can incorporate persistent databases like MySQL or MongoDB.

**8. System Architecture**

The system follows a client-server-like architecture where the client interacts with the ATM class, and the server-side logic is handled by the Bank, Account, and Transaction classes. The interaction between these classes ensures smooth flow of data and business logic.

**Key Components**:

1. **ATM Class**:
   * Handles user input and manages high-level operations like login, account creation, and transactions.
   * Directs user commands to the appropriate business logic (e.g., Bank and Account).
2. **Bank Class**:
   * Responsible for managing user accounts and authentication.
   * Acts as the central store for user data and accounts.
3. **Account Class**:
   * Manages individual user accounts, including balance and transaction history.
   * Handles deposits, withdrawals, and transfers between accounts.
4. **Transaction Class**:
   * Stores transaction details (e.g., amount, type, date).
   * Provides a record of all financial activities.

**9. Component Interaction**

The interaction between key components of the system is described below:

1. **User Interface and ATM**:
   * The user interacts with the ATM class through a console interface. The ATM class collects user input and directs the appropriate operations.
2. **ATM and Bank**:
   * The ATM class communicates with the Bank class to authenticate users and retrieve account holder data. For account creation, the ATM adds new account holders to the Bank.
3. **Bank and Account**:
   * The Bank manages a collection of AccountHolder objects. Each AccountHolder has an associated Account, which stores the user's balance and transaction history.
4. **Account and Transaction**:
   * Each transaction is recorded as a BankTransaction object, which is stored in the account’s transaction history. This ensures that all user actions are properly logged.

**10. Data Flow**

The following outlines the general flow of data through the system:

1. **Login**:
   * The ATM prompts for user credentials.
   * The Bank checks the credentials and returns an AccountHolder object if valid.
   * The ATM grants access to the user’s account and shows the menu.
2. **Account Creation**:
   * The ATM collects user details and creates a new AccountHolder object.
   * The new account is added to the Bank, and the user is notified of success.
3. **Transaction Management**:
   * The user selects a transaction type (deposit, withdrawal, transfer).
   * The ATM directs the operation to the user’s Account, which updates the balance and logs the transaction.
4. **Transaction History**:
   * The ATM retrieves the list of BankTransaction objects from the user’s Account and displays them to the user.

**11. Data Model**

The primary data entities in the system are:

1. **AccountHolder**:
   * Stores user information (user ID, PIN, name) and the associated Account.
2. **Account**:
   * Stores account-specific data, such as account number, balance, and a list of BankTransaction objects.
3. **BankTransaction**:
   * Represents individual transactions, storing data like transaction ID, type, amount, and date.
4. **Bank**:
   * Acts as a central repository for all AccountHolder objects.

**12. Error Handling**

The system incorporates basic error handling to ensure robustness:

* Invalid user credentials result in an error message and the option to retry login.
* Insufficient balance during withdrawals or transfers results in a failure message.
* Invalid inputs, such as selecting a non-existent option in the menu, are handled gracefully with appropriate error messages.

**13. Security Considerations**

* **Authentication**: The system ensures that only authenticated users can access account details or perform transactions. User IDs and PINs are used for authentication.
* **Future Enhancements**: To improve security, the system could implement hashed passwords and secure communication protocols.

**14. Limitations and Future Enhancements**

**14.1 Limitations**

* **In-Memory Storage**: Currently, all account and transaction data are stored in-memory using HashMap. This means data is lost when the system shuts down.
* **Limited Transaction Types**: Only basic transactions (deposit, withdrawal, transfer) are supported.

**14.2 Future Enhancements**

* **Persistent Data Storage**: The system can be extended to store data in a relational database like MySQL, allowing data to persist across sessions.
* **Enhanced Security**: The addition of password hashing, encryption, and multi-factor authentication could improve security.
* **Extended Transaction Types**

: Features like bill payments, account statements, and recurring transactions could be added.

**15. Conclusion**

This ATM application provides a functional simulation of an ATM system with basic account management and transaction features. Its modular design ensures it is extensible, maintainable, and secure for future enhancements