Project 1: My Banking System – Advanced Requirements

Objectives

The advanced requirements of the Project 1 aim to:

- Allow students to innovate and improvise existing code or design into a better shape.
- Apply the concept of SOLID principles in developing software applications.
- Implement threading and concurrency to handle simultaneous and mass transactions.
- Save data permanently so that it persists on future execution of the program.
- Document changes that have been applied to the software application.

Required Improvements

1. SOLID Principles

Apply the concept of SOLID principles. They include but are not limited to what is stipulated in the document.

1.1 Single Responsibility Principle (SRP)

- Move transaction-related logic such as fund transfers, deposits, withdrawals, credits, and any other out of `Account` or `Bank`.
- This new class can handle validation such as having enough balance, update of balances for the correct account/

1.2 Open-Closed Principle (OCP)

- Inclusion of additional accounts such as `StudentAccount` and `BusinessAccount` without modifying the existing Account subclasses heavily.
- There should be minimal (or no) changes when these new account types are added.

1.3 Interface Segregation Principle (ISP)

- Specialized interfaces that allow accounts to define the behavior they need such as transfers for accounts that can fund transfer, and a creditable interface for accounts that can do credits.

1.4 Dependency Inversion Principle (DIP)

- Should you insist to include File I/O or data storage functionalities, define other interface(s) to handle that logic so that multiple implementations such as a mix of File I/O and database can be included in the application without breaking the system.

2. Data Persistence and File I/O – Choose only one

A. Option A: File-Based Storage (+3 points)

- CSV or JSON for readability and easier implementation/debugging.
- When the program exits or a transaction is made, update the current state of the `Bank`, affected `Accounts`, and log new `Transactions`.
- On program startup, parse those files to reconstruct objects.
- For a more advanced feature and security especially if threading is involved, you may want to include a temporary file where it will replace the main file after a transaction is made.

B. Option B: Database (+5 points)

- Use SQLite Database to create tables for `Account`, `Transactions`, and maybe `Banks` if you want.
- It should include CRUD operations such as:
  i. CREATE: Insertion of new accounts/banks.
  ii. READ: Fetch account details on login.
  iii. UPDATE: Adjust balances or credits after transactions.
  iv. DELETE: Optional but used for closing accounts.
- Utilize `java.sql` for this one.

3. Mass Transaction Handling

3.1 Thread Setup

- Multiple clients can be simulated executing random instructions.
- One might do repeated deposits.
- Another might do repeated transfers.
- Another might do repeated withdrawals.
- Utilize synchronized methods to prevent conditions where the update of balance, transaction logs, and other actions may result to a conflict.

3.2 Stops

- When exiting the simulation, the process must not immediately stop.
- Each thread or process must finish their actions first before it exits.

3.3 Logs

- For every thread or process, transactions must be logged, and a console output tracks the progress of every thread.

4. Documentation and UML Updates

4.1 UML Changes

- Update class diagrams to reflect any changes committed to the project.

4.2 JavaDoc & Comments

- Document any exceptions, interfaces, or classes.

4.3 Changelogs through a README.MD file

- Provide a concise summary of the changes such as the inclusion of additional methods, or the isolation of responsibilities (if any).