Implementing an Automated Teller Machine (ATM) System

Melchior Duc - Erasmus Student 0404721

11 novembre 2018

1 Relational Database Backend

You will develop a simple Automated Teller Machine (ATM) System which uses a relational database on AWS as backend and Java clients as ATMs.

We create a simple and sensible database scheme that allows you to keep track of account balances. The account_id is used to identify an account. It has a pinCode because the customer must be able to identify himself. An account has a balance. Finally the locker field allows to know if a client is writing on the account to preserve the atomicity of certain transactions.

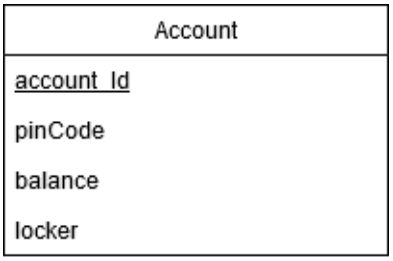

FIGURE 1 – Data Base - AWS

2 ATM Java Client

A customer must be able to login using her account number and corresponding pinCode, deposit and withdraw funds, display her account balance. We must make sure that the operation's results adequately reflect in the database.

The Figure 2 presents the users and the functionalities that must be implemented to answer the problem.

I created a class account.java that implements the different methods used.

The text interface is in the class DBSetup. It allows the user to identify himself with his account number and Pin code. If he is a customer he accesses the other functions. He can withdraw money, deposit or display the threshold of his account.

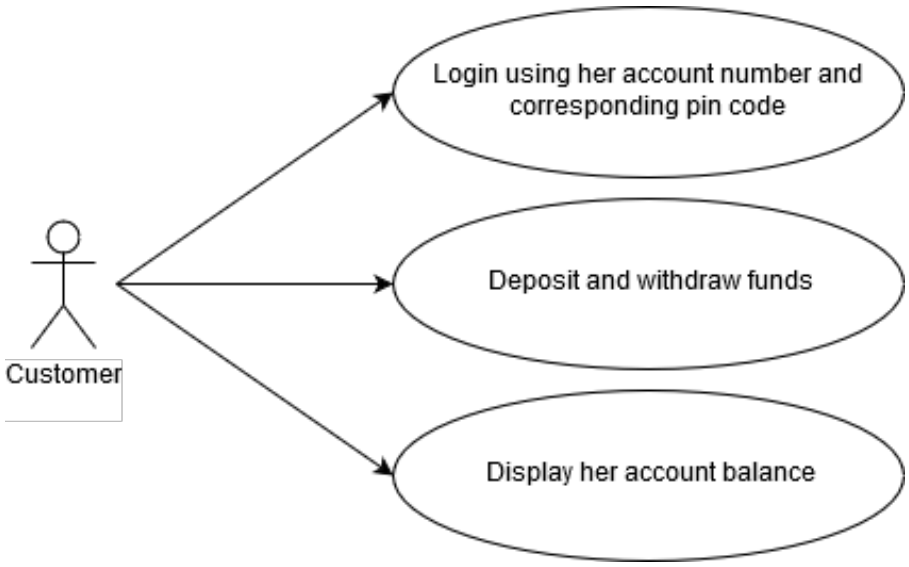

FIGURE 2 – Functions

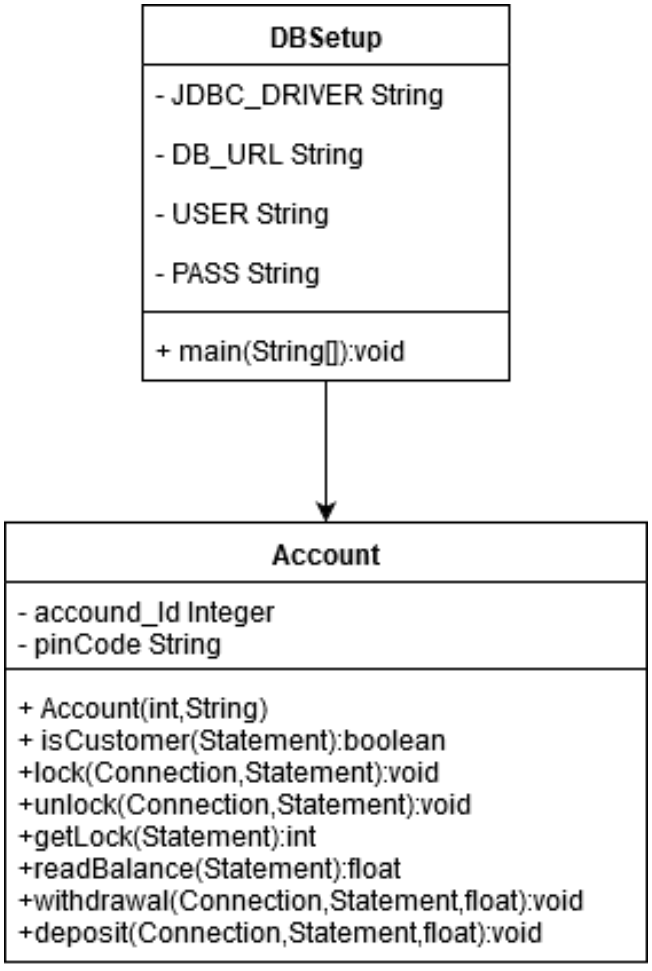

FIGURE 3 – Class Diagram

3 Concurrent Access

1.Conceptually discuss how such an invariant could be guaranteed. Should enforcement happen in the database on within clients ?

The problem lies in the fact that two people identify themselves at the same time on the same account. It could withdraw more money than counting it contains or losing it in a parallel deposit. To avoid this, you must ensure simultaneous single access to an account. To have more security and understand what is written in the database, the enforcement should happen in the database on wthin client.

2. Implement one of the solutions you discussed in 1 within your system. Test your implementation in a sensible way and document your testing procédure and results.

I'm going to implement a mutex. For this I add to my table the locker field that allows you to know if a customer is connected to an account and is writing (withdrawal or deposit). If the locker is activated (locker = 1) you can not remove or drop until the operation is completed. A user is already logged into the account. For this we implement the lock and unlock (figure 4) methods that will allow to lock or unlock an account. This solution is difficult to test as explained in exercise session. It should be possible to run a concurrent access.

```
public void lock(Connection conn, Statement stmt) throws SQLException {
    conn.setAutoCommit(false);
    stmt = conn.createStatement();
    String update_1 = "UPDATE account SET locker=1 WHERE account_id=" +
                      Integer.toString(this.account_Id);
    try {
        stmt.executeUpdate(update_1);
        conn.commit();
    } catch (SQLException e) {
        conn.rollback();
    }
}

public void unlock(Connection conn, Statement stmt) throws SQLException {
    conn.setAutoCommit(false);
    stmt = conn.createStatement();
    String update_1 = "UPDATE account SET locker=0 WHERE account_id=" +
                      Integer.toString(this.account_Id);
    try {
        stmt.executeUpdate(update_1);
        conn.commit();
    } catch (SQLException e) {
        conn.rollback();
    }
}
```

FIGURE 4 – Class Diagram