ATM Subsystem

Design and Implementation

Nicholas Stewart
Jonathan Halim
Christian Devile

Kenny Lee
Ashley Yu

***Abstract*—We set out to create a tool to facilitate banking service for the clients of a major bank. We created a solution for our problem using Java for our backend. In addition, we utilized *SQLite* as our database solution. Furthermore, we used a Mobile application GUI to create a familiar touch interface for the bank's clients to use. We linked our front end mobile app to our database over the internet through socket programming in Java.**

I. Introduction / Problem

It is in the best interest of many of the world's finest financial institutions to keep their clients happy. The degree to which an institution can provide this happiness often boils down to how well it can address two fundamental problems. The first of those problems regards how well an institution can provide its clients and customers with the tools necessary to provide them with direct access to their personal finances and accounts. The second problem is how efficiently and reliably an institution's management can view transaction history and other data from its ATMs, using it to improve its solution to the first problem. The subsystem's design and implementation which is outlined in this document provides an excellent solution to address these two needs.

II. Analysis

A. Stakeholders

This subsystem's stakeholders encompass three of the four primary categories of stakeholders. A financial institution's clients and customers are one of those stakeholders and are categorized as external operational stakeholders. These clients have a stake in the subsystem due to their expected repeat use of ATMs in order to manage the funds within their accounts. The second of the subsystem's three stakeholders are the financial institution's employees. These employees are categorized as internal operational stakeholders since they will be required to interact with ATMs in order to empty and refill its cash boxes. Finally, the third group of stakeholders for this subsystem are the management staff at the institution. These stakeholders are categorized as internal executive since they hold an interest in the transactional records and other use data that the institution's ATMs will produce.

B. System Requirements

When drafting the system requirements for our subsystem, we employed the popular *FURPS+* technique. With this technique, we were able to break down our system's requirements into five distinct, and equally important, categories: functional, usability, reliability, performance, and security. The majority of our system's requirements fell under the first of those five categories, functional. For the system's functional requirements, we had to ensure the system we developed would enable a financial institution's clients to perform a sundry of standard financial transactions. For example, some functional requirements included allowing clients to withdraw, deposit, transfer, and view the balances of their accounts. Other functional requirements included displaying warnings when certain transactions could not be made due to insufficient funds and other similar constraints. Being able to notify the bank's management when an ATM's withdrawal or deposit boxes needed refilling and or emptying was another key functional requirement of the subsystem. With respect to usability requirements, all our subsystem required was that the ATM's touchscreen interface must be navigable to a degree at which the clients could easily interact with the system without having any prior knowledge of how it functions. Our sole reliability requirement was that the system's software be self-sustaining for the period of time agreed upon between us and the financial institution. Self-sustainment in this context included things such as scheduled downtime to rollout software updates as well as perform maintenance at regularly scheduled intervals. In terms of performance, clients' interactions with the system needed to be fluid and responsive. Security requirements consisted of requiring card PIN numbers as means for accessing clients' accounts as well as also preventing unauthorized access of a clients' accounts after repeated failures to provide valid PINs (locking the cards). These security requirements helped to ensure a financial institution's clients were protected from suspicious or fraudulent activity related to their accounts.

C. Identifying Use Cases

To identify the use cases needed for our subsystem, we used the event decomposition technique. Using our functional requirements, in tandem with respect to what us consumers believed to be standard business practices from the banking industry, we narrowed down our requirements into a list of all of the possible user-system interactions that could take place. The items on this list became our use cases. The Verb-Noun technique also proved useful when naming these use cases. *(See Appendix A, Figure 1)*

III. Design techniques

A. Database

Our subsystem's database had to keep track of all the clients defined within it for access control, keep a record of all of their transactions, and the current state of the ATM systems. Our eventual design process led us to create a relational database consisting of nine distinct tables. *(See Appendix A, Figure 2)* With every table being eventually tied to a bank branch, the relationships split off into two primary segments. Clients have account openings which directly correspond to their debit cards. No matter what, if an account is accessible to a client, they have an account opening association class binding them to it. This is because of the many to many relationship between clients and accounts that had to be resolved. Clients are also presented the option to link a debit card to their accounts. It is through this card that ATM systems can be used to manage their accounts. It is important to note that only one card is ever issued to a client at a time. However, a single card can be used to access multiple accounts (if the financial institution's policies allow it). There is also a many to many relationship between debit card and account which is why there is an association class between them called card activation.

Under bank branches, lie ATM systems and ATM sessions. These systems contain all of the current data regarding the ATMs, while the sessions are running hold a log of the actions taken at an ATM including the corresponding times at which those actions were taken. The data being shared at this step of using the ATM includes the sessionActive variable in both ATM and ATMSession. We decided to resolve this through a database trigger in the ATM session that would update the ATM itself whenever a session was inserted or edited to be terminated. The data access classes were made to only allow editing of the database in a way that made sure the values were always synchronized. Under sessions, there is the transaction table, which is a record of all the money changing ownership in the ATMs or clients' accounts. For the actual server implementation, we combined all three different transaction types into a single table, with a type identifier. Consequently, when read back, the program knows which data in the fields to expect. For the database management system, the best solution we found was to implement it by using *SQLite.* We chose *SQLite* as it provided us with a simple relational database under the full control of our server code which properly catered to the needs of our subsystem.

B. Domain Layer

Our domain layer consisted of our data access classes. Each class in our database had its own respective data access class. Within each of those classes was the logic needed to properly insert and retrieve information from our database. Each of our data access classes contained a function called generateStatements() that contained multiple prepared statements that would either insert or retrieve information from the database. We chose to use prepared statements as they are pre-compiled once, making it faster for a repeated execution of dynamic *SQLite*. In addition, it provided separation between query code and the parameter values which helped our debugging process since it improved the overall readability of the code. Furthermore, the prepared statements also helped to protect against *SQLite* injection. The variation in our classes can be seen with the different number of prepared statements within that function. Moreover, some classes only had two prepared statements, while others had up to four. For every prepared statement we had in a data access class, there was a corresponding function to execute the different queries needed. Many of our queries contained default values such as the values we used for several of our dates and times. Within our get functions, we would create the class's corresponding entity object where we would return the entity object with the information we wanted to receive stored back in it. On the other hand, within our functions to insert data, we would lock the database prior to executing the query and unlocking after the query finished in order to maintain concurrency.

C. View Layer

For our subsystem's visual interface, we decided to use Android Studio to emulate an ATM on an android phone. We wanted our application to be connected to a server which would update a client's account information by sending messages back and forth from the server to the application via a TCP socket. By doing so, we were able to virtualize the process of going to an ATM and making a transaction. The features we included in our ATM subsystem involved actions such as viewing a map of available ATMs, selecting an ATM, logging into an ATM, performing different transactions, logging out, as well as printing a receipt. These were the main features we implemented for our ATM subsystem. In order to achieve the aforementioned functionality in our ATM's graphical user interface, we included various screens for each action in which the client needed to proceed through a series of different prompts in order to perform the desired transaction. *(See Appendix A, Figure 4)* The following numbered process briefly describes this flow of activities between the ATM's interface and the user:

1) The client is shown a map of Cal Poly's ATM's when they open the application
2) Upon selecting an ATM, a login screen is displayed to the user, prompting for a card number and its corresponding four-digit PIN
3) Upon successfully logging in, the client would then be brought to the transaction screen where they can select a transaction to be executed by the system
4) Each transaction has its own set of successful or unsuccessful screens which inform the client of whether or not his or her requested transaction has been successfully processed
5) After finishing all desired transactions, the client can log out of the ATM, which will then automatically save their transaction history as well as other information regarding their session as well as print a receipt containing the details of those transactions.

Throughout the various screens, there are several edge cases that had to be considered and checked by both the client and the database before allowing the user to transition to the application's next screen. For example, when the application initially opens, the user is prompted to select an ATM. However, the ATM must not be currently in use. If the ATM is currently in use, the user is denied access and is prompted to choose a different ATM location. After choosing an available ATM, the client is asked to enter a card number and the PIN number associated with that card number. To gain access to an account, a valid card number and its corresponding valid PIN must be provided. The client has five attempts to input the correct PIN number or else the associated tge card will be locked. The card can neither be locked nor expired in order to successfully login into an ATM. Once logged into the ATM, the client cannot remain idle for longer than two minutes otherwise he or she is logged out by the system for security purposes. When a client is withdrawing from the ATM, he or she may not withdraw more funds than he or she has available. If this is the case, the transaction is cancelled. If the client would like to withdraw more bills than the ATM currently has available, then the transaction is also cancelled. If a client would like to deposit more bills than the ATM can hold, then the transaction is cancelled once again. In all of these cases, the client will be notified that the transaction was unsuccessful and will automatically be returned to the main screen. Lastly, if the client has made any transactions during the current session, a receipt consisting of his or her transaction details will be printed upon logout. If no transactions were made during the current session, no receipt would be printed.

D. Server Connection

Connecting our system's frontend to the backend was made possible through the use of input and output data streams that sent messages via TCP sockets. While TCP cannot keep segment data secure against our messages being intercepted, it was sufficient enough for our initial version. Through a large list of unique flags, ensuring both the integrity and validity the communications between frontend and backend was simple. By designing detailed and thorough controller classes on both the front and backends to facilitate communication between an unrestricted number of clients and our server, their integration with one another was seamless. Without the high quality error catching and handling in our implementation, debugging our system's server communications would not have been possible.

IV. Team Findings / Suggestions

A. Obstacles

Some of the obstacles we encountered in the duration of implementing the ATM subsystem included trying to familiarize ourselves with new software such as Android Studio and *SQLite*. Only one of our members was familiar with databases so it was a bit of a challenge to come up with a fully functioning server which would be able to initialize client account information and successfully link it to our graphical user interface. Android Studio's interface was sometimes difficult to work with as it was very hard to debug our code at times due to errors which seemed to be quite generic and uninformative at times. Not to mention, GitHub also brought some challenges as well when it came to merge conflicts with so many varying branches as some of our members were also using GitHub for the very first time. It was difficult to work together in separate times and not work on the same parts twice. Additionally, making sure that we were able to capture all the possible edge cases in the server as well as successfully displaying alerts in the graphical user interface came to be a bit of a tedious matter. There were so many cases in which we had to take into consideration when it came to implementing an ATM subsystem such as checking whether or not the client's account had enough funds or if the client was trying to request for a withdrawal that was not a multiple of twenty. For this project, there was a lot that needed to be learned in order to successfully implement the ATM subsystem server which held pre-initialized client account information and passed it from the database through the ATM graphical user interface.

B. What Was Learned

This project was a significant learning experience for all of us as we worked on it for many hours consistently for over a month. Some of the things we learned about included learning how to use

SQLite and Android Studio for the very first time. Implementing this subsystem seemed like a large task at first, but we pieced it together bit by bit and learned a lot on how to use different softwares to best show how a virtual ATM would function. We learned how to initialize the database for our server using *SQLite* as well as how to use Android Studio to emulate our application showing various screens in which our client would be taken through. GitHub was also a part of our learning experience as we encountered conflicts which we sometimes were not sure of how to properly fix. Moreover, making sure that we were all on the same page when it came to implementing this subsystem was a large part of our project.

C. Future Development

If we were given more time to add onto our ATM subsystem, we would have additional tweaks to the placement of buttons and other visual elements on our user interface. There are most likely other edge cases we have forgotten to consider and would most likely be able to identify and address if we were given more time to work on it. Another addition to this project we would've also liked to include was adding a credit card as a new card type as opposed to only being able to read debit cards. However, we did a decent job of including a majority of a typical ATM's features into this version of our mobile application. For this application to be used, one important feature would be to encrypt the traffic between the client programs and the server program. Without this, financial data is visible to other people on the internet. One final thing we would likely want to work on in a future version would be displaying the date and time on a majority of the screens of the ATM. We were unable to reliably display live dates and times across all views in our application.

V. Conclusion

While the degree to which the problems that warranted the existence of our subsystem will forever remain constant, the solution to those problems that has been outlined above irrefutably reduces the impact those hurdles will have on any financial institution. By using a solution like the one we proposed, any financial institution can more easily achieve their goals with respect to customer satisfaction. All in all, we acquired a better overall understanding of how this specific subsystem functioned internally and externally.

Appendix A

Figure 1.

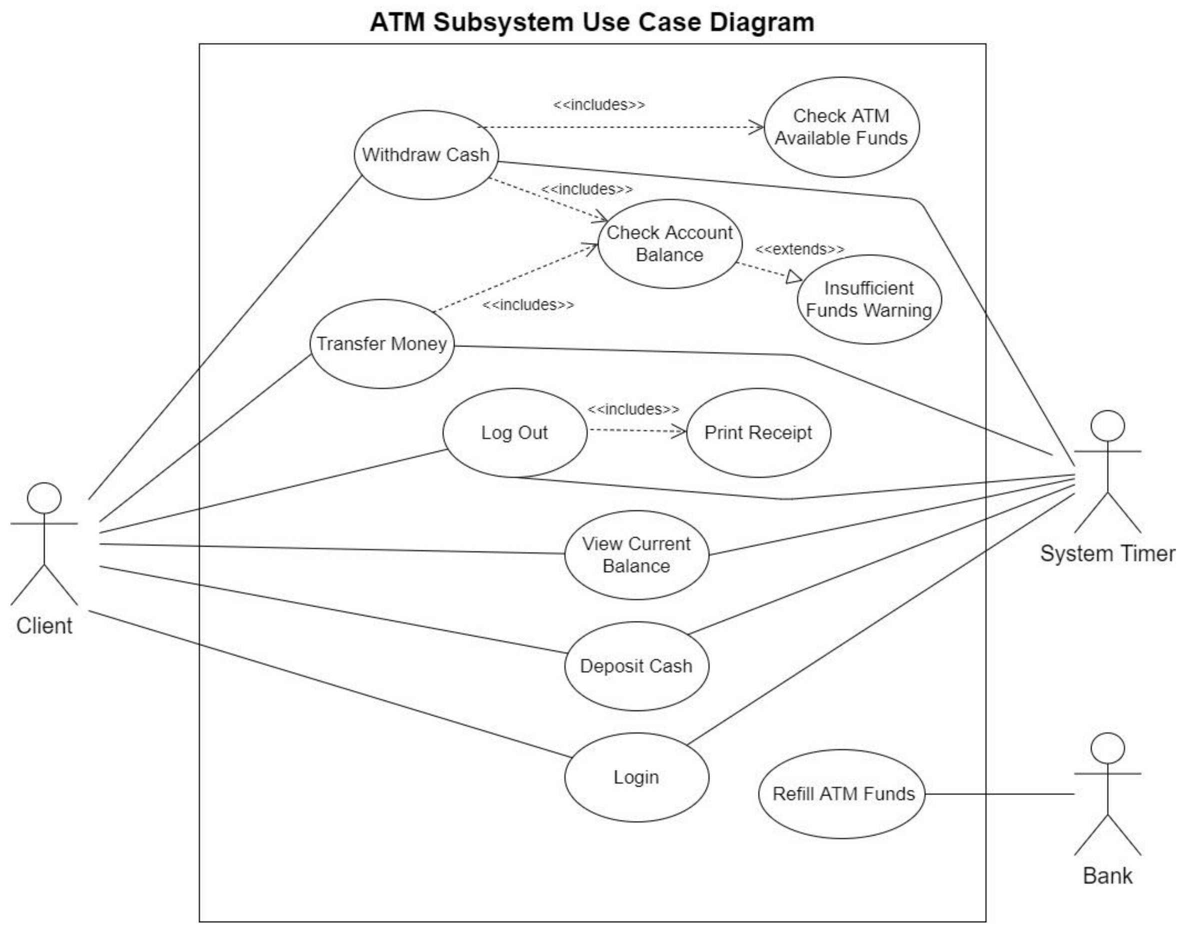


Figure 2.

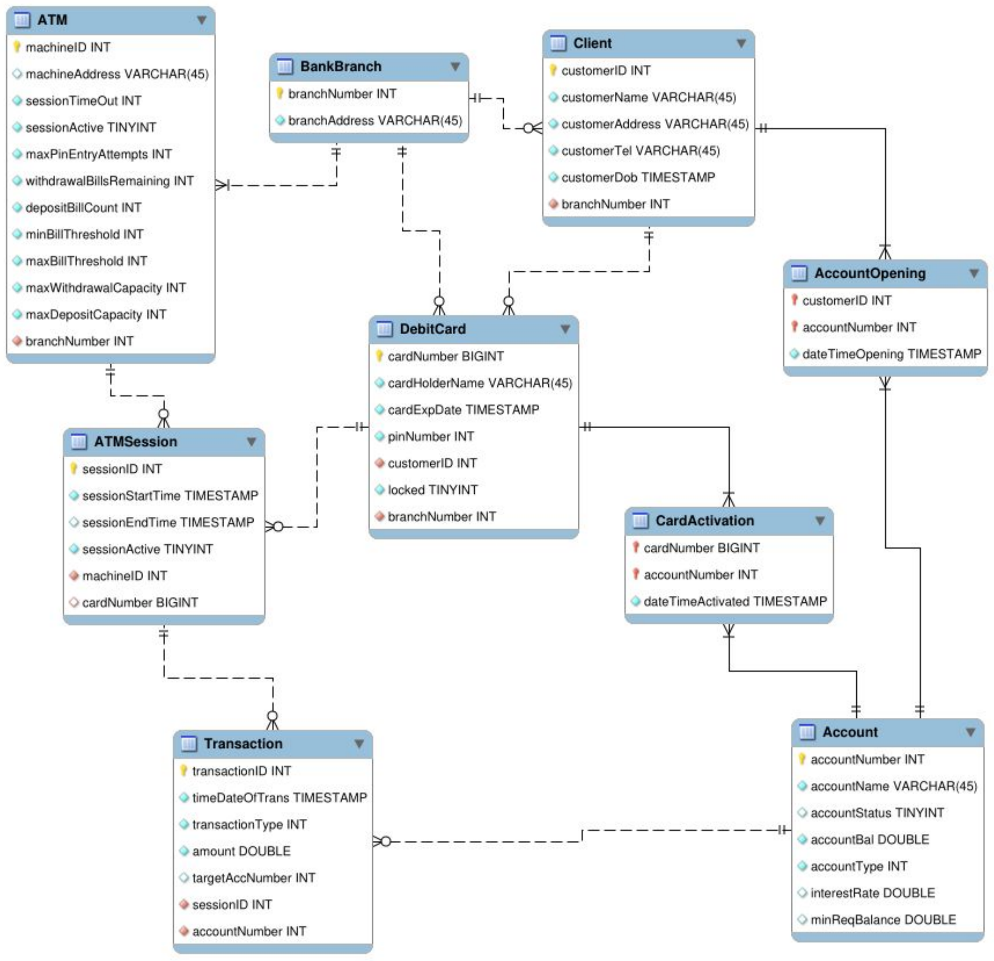


Figure 3.

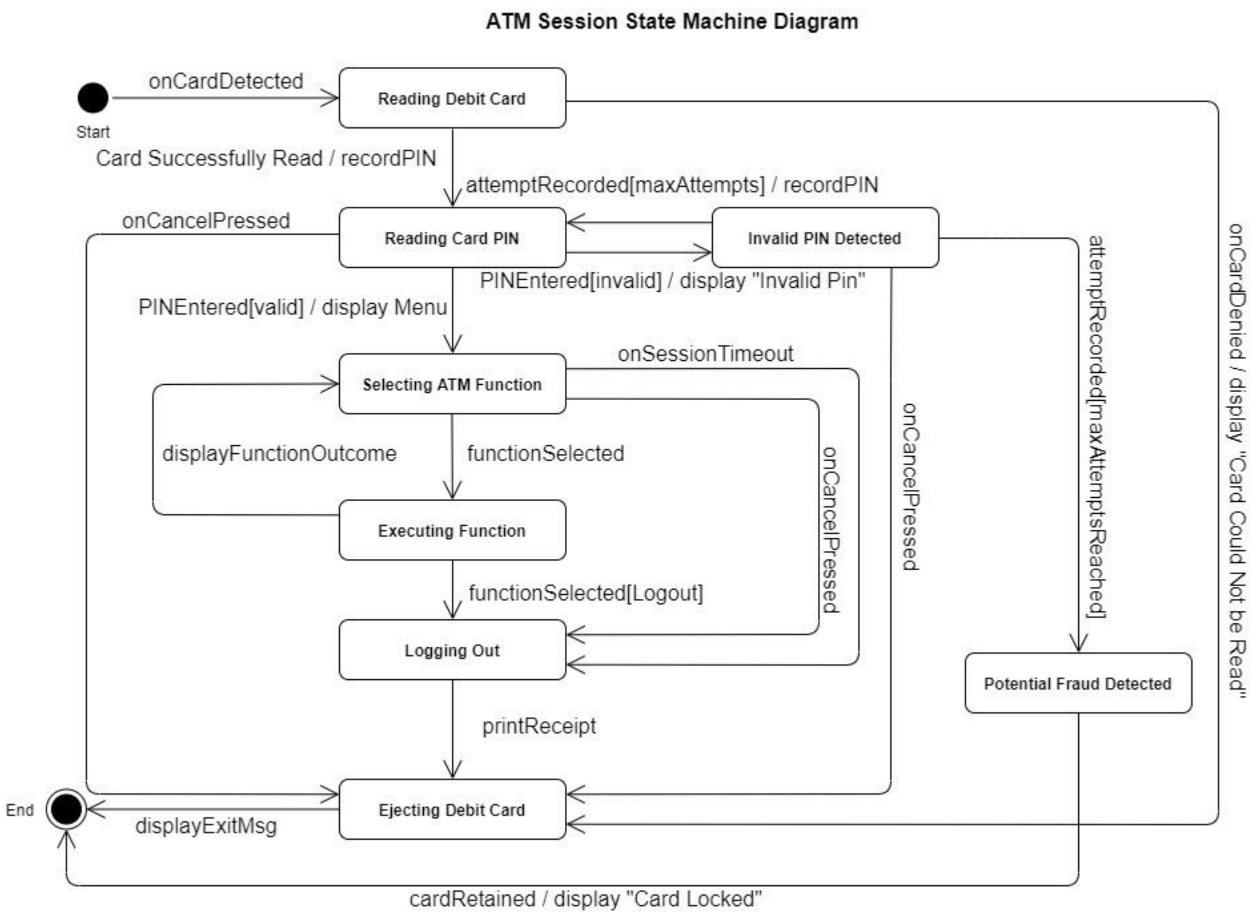
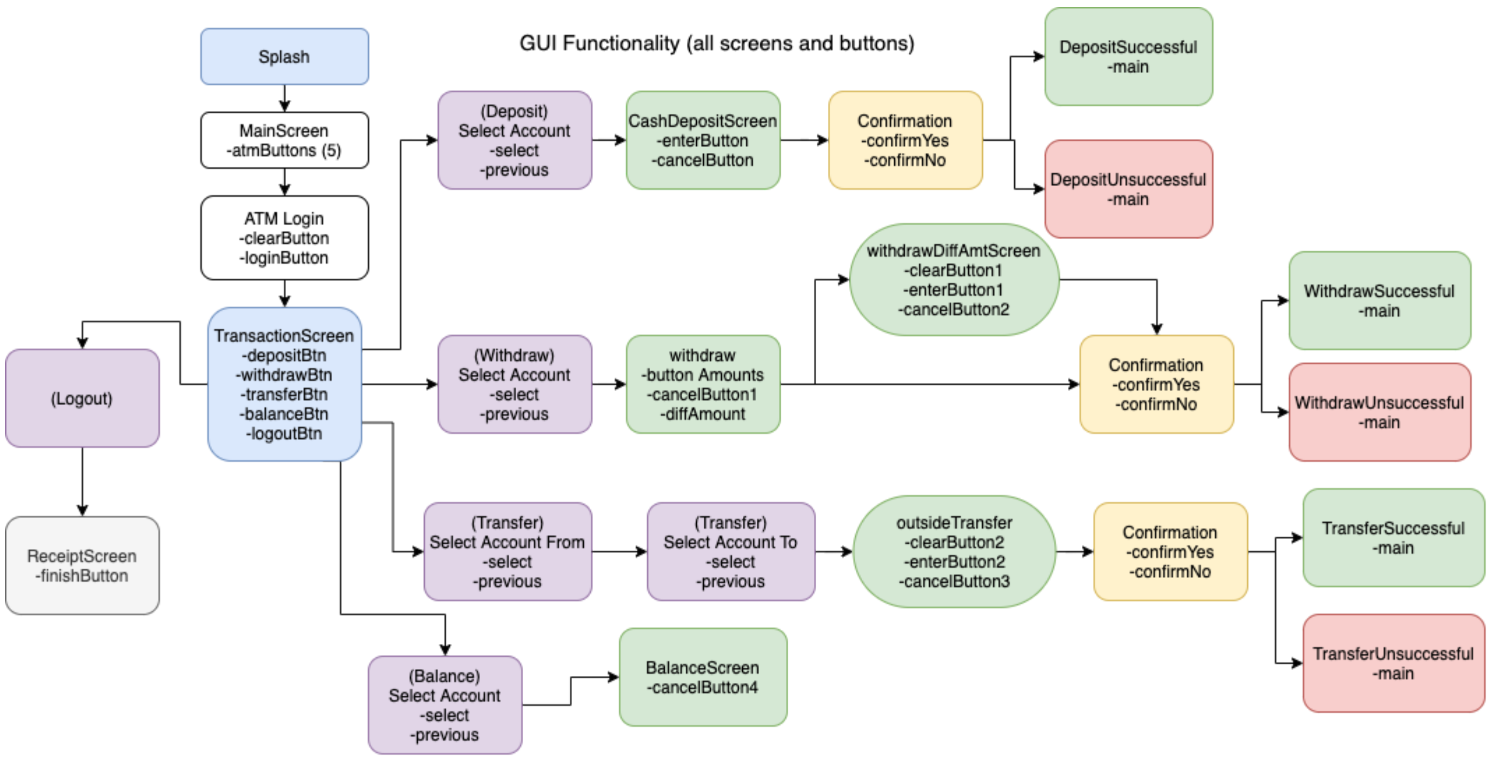


Figure 4.



GUI Functionality (all screens and buttons)
Splash
MainScreen -atmButtons (5)
ATM Login -clearButton -loginButton
TransactionScreen -depositBtn -withdrawBtn -transferBtn -balanceBtn -logoutBtn
(Logout)
ReceiptScreen -finishButton
(Deposit) Select Account -select -previous
CashDepositScreen -enterButton -cancelButton
Confirmation -confirmYes -confirmNo
DepositSuccessful -main
DepositUnsuccessful -main
(Withdraw) Select Account -select -previous
withdraw -button Amounts -cancelButton1 -diffAmount
withdrawDiffAmtScreen -clearButton1 -enterButton1 -cancelButton2
Confirmation -confirmYes -confirmNo
WithdrawSuccessful -main
WithdrawUnsuccessful -main
(Transfer) Select Account From -select -previous
(Transfer) Select Account To -select -previous
outsideTransfer -clearButton2 -enterButton2 -cancelButton3
Confirmation -confirmYes -confirmNo
TransferSuccessful -main
TransferUnsuccessful -main
(Balance) Select Account -select -previous
BalanceScreen -cancelButton4

Figure 5.

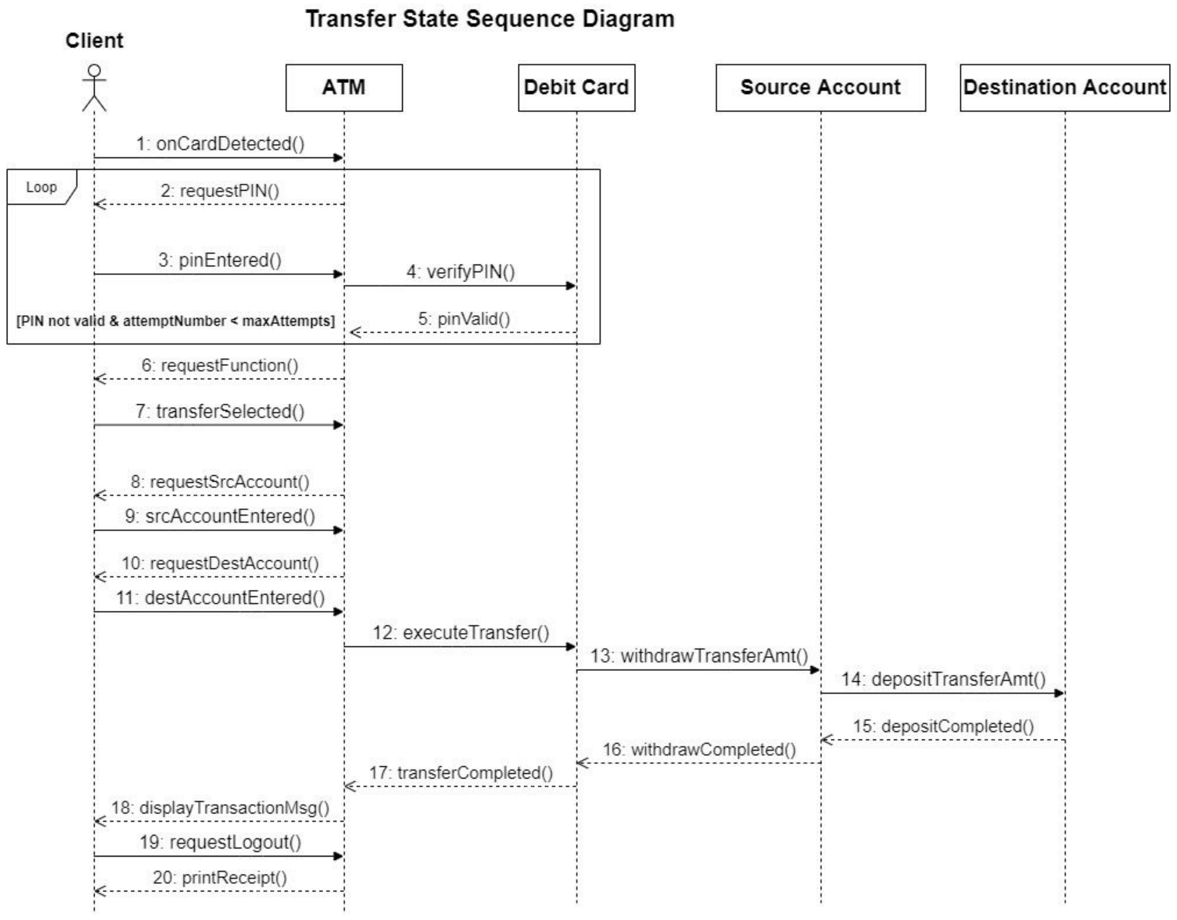


Figure 6.

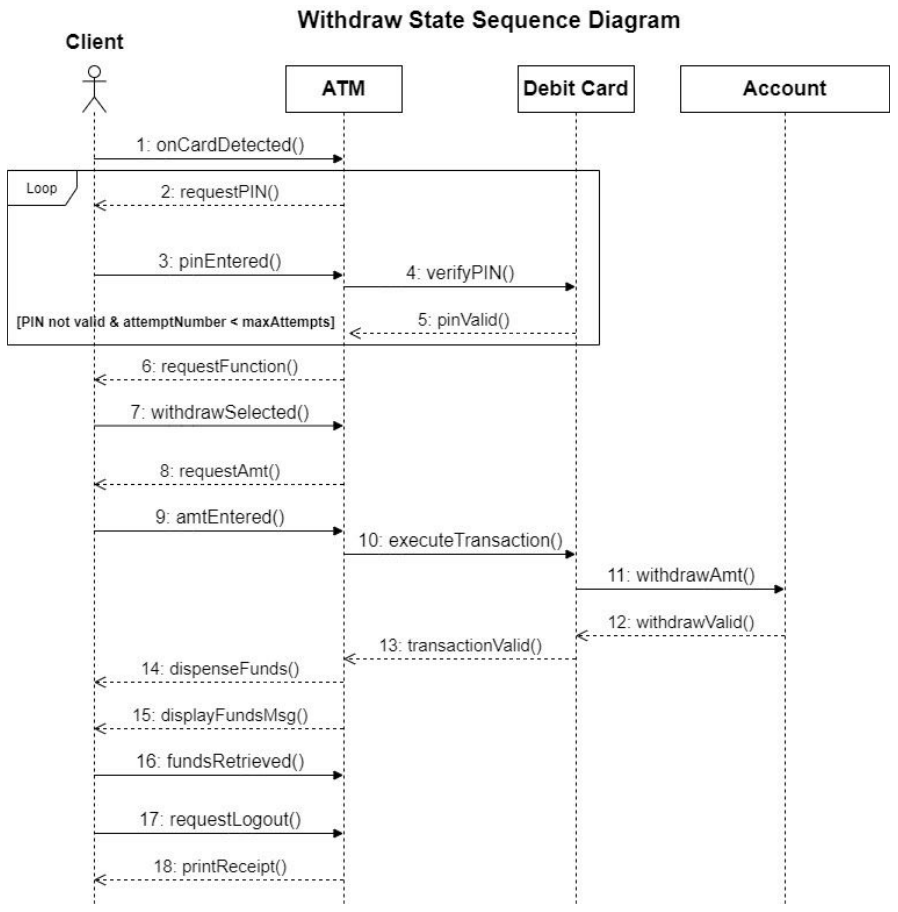


Figure 7.

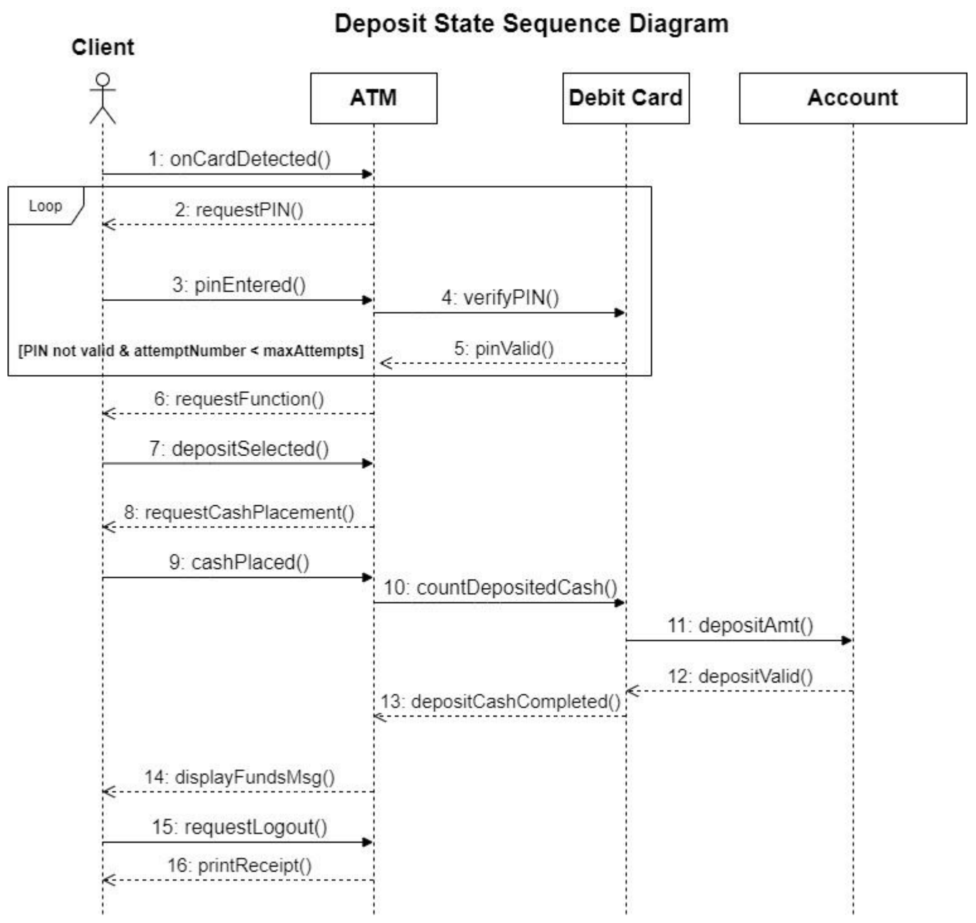
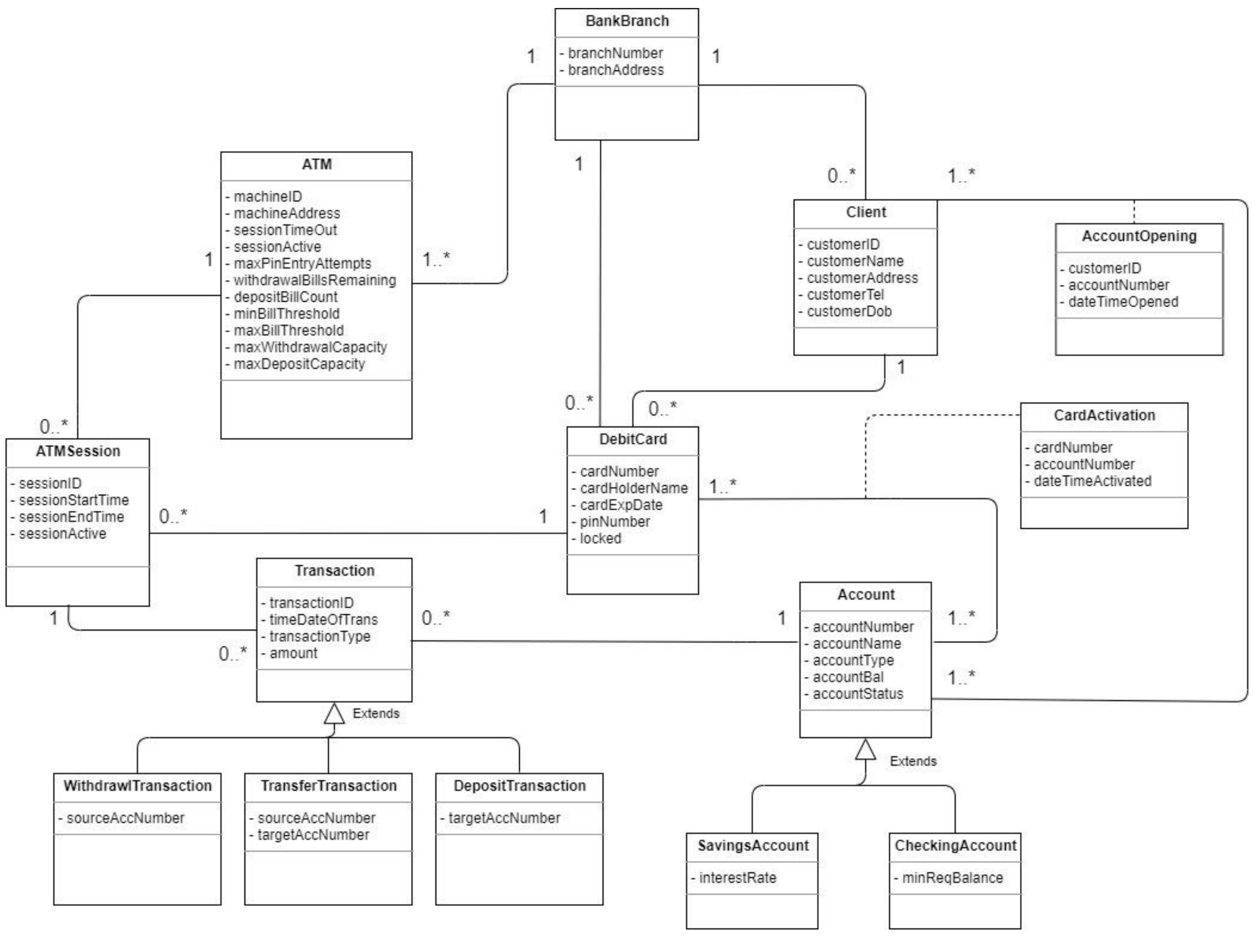


Figure. 8


BankBranch
- branchNumber
- branchAddress
ATM
- machineID
- machineAddress
- sessionTimeOut
- sessionActive
- maxPinEntryAttempts
- withdrawalBillsRemaining
- depositBillCount
- minBillThreshold
- maxBillThreshold
- maxWithdrawalCapacity
- maxDepositCapacity
Client
- customerID
- customerName
- customerAddress
- customerTel
- customerDob
AccountOpening
- customerID
- accountNumber
- dateTimeOpened
ATMSession
- sessionID
- sessionStartTime
- sessionEndTime
- sessionActive
DebitCard
- cardNumber
- cardHolderName
- cardExpDate
- pinNumber
- locked
CardActivation
- cardNumber
- accountNumber
- dateTimeActivated
Transaction
- transactionID
- timeDateOfTrans
- transactionType
- amount
Account
- accountNumber
- accountName
- accountType
- accountBal
- accountStatus
Extends
WithdrawlTransaction
- sourceAccNumber
TransferTransaction
- sourceAccNumber
- targetAccNumber
DepositTransaction
- targetAccNumber
Extends
SavingsAccount
- interestRate
CheckingAccount
- minReqBalance
1
1
1
1..*
0..*
1..*
1
1
0..*
0..*
0..*
0..*
1
1..*
1
0..*
0..*
1
1..*
1..*