Online Banking System
(T Bank)

Introduction

Project domain

The domain of the project is defined as Online Banking System. This system can be described as an app with a set of tools for both bank clients and bank tellers, to interact with the services that the bank provides. This set of tools includes but is not limited to: manipulation of funds, balance accounts, loans, credits - for clients, and creation of new client accounts and client account look-up – for the bank tellers.

Project objectives

- User-friendly interface
- Convenient interface – user has to take less steps to access certain functionality in comparison to other real banking applications.
- App robustness
- Extensive functionality for bank clients
- Performance
- Easy to maintain code
- Detailed documentation

Modelling tools

We used UML as our modelling language.

To build the diagrams, Lucidchart was used the most, with some diagrams being drawn by hand where it was more convenient.

Languages and the framework

Python was used as the programming language for the classes and test cases.

For the User Interface, we used HTML, JavaScript, CSS and the Flask framework, which is a web framework that works with Python.

Development Methodology

We chose the spiral methodology for the development. We thought that it would be easier to reevaluate our risks multiple times and it would be much easier to make changes to the project. Our risk assessment involved two resources – time and the abilities of developers in our group.

First Prototype

We had our first prototype right after Phase 3. We had the classes code ready and an understanding on how would we want for our app layout to look like - where we would have our UI objects positioned, in what sections of the app we would put any specific functionality, and how would our home page for both client and teller look like. During the risk assessment, we decided that we had enough time to build application in the most customizable way.

After searching through frameworks and libraries, we found out about Flask and decided that we would build a web application using the framework, HTML, CSS and JavaScript. Web applications offer the biggest amount of customization when compared to libraries like Tkinter or PyQT, it would take much less time to learn new tools for our team members who knew how to build web applications and we could share our code easily.

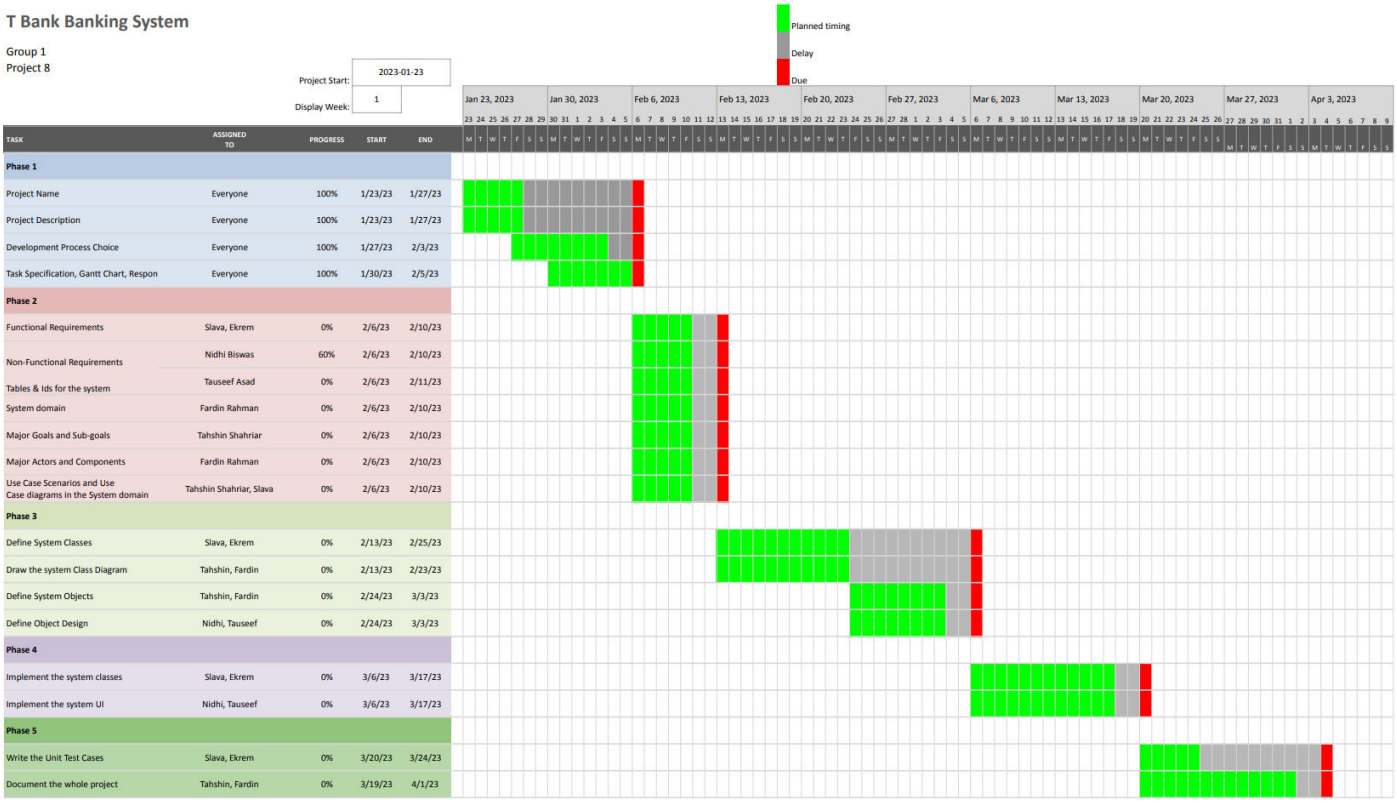
Second Prototype

During the later stages of Phase 4 we had our second prototype. We learnt how to build a web application using Flask, made changes to the structure of the Classes code, implemented additional functionality that we did not think of during the earlier phases and refactored the code. The second prototype had the basic functionality implemented and it was a working web application. After the risk evaluation we decided to redesign our interface and redistribute the responsibilities in the group.

Third Prototype

During Phase 5, we had our third prototype that is contained in the folder with this report. The application has undergone the necessary testing. It was fully finished and working, with some more functionality added to it and some of it removed to a lack of meaning to it.

Gantt Chart with member responsibilities



TASK	ASSIGNED TO	PROGRESS	START	END
Phase 1				
Project Name	Everyone	100%	1/23/23	1/27/23
Project Description	Everyone	100%	1/23/23	1/27/23
Development Process Choice	Everyone	100%	1/27/23	2/3/23
Task Specification, Gantt Chart, Respon	Everyone	100%	1/30/23	2/5/23
Phase 2				
Functional Requirements	Slava, Ekrem	0%	2/6/23	2/10/23
Non-Functional Requirements	Nidhi Biswas	60%	2/6/23	2/10/23
Tables & Ids for the system	Tauseef Asad	0%	2/6/23	2/11/23
System domain	Fardin Rahman	0%	2/6/23	2/10/23
Major Goals and Sub-goals	Tahshin Shahriar	0%	2/6/23	2/10/23
Major Actors and Components	Fardin Rahman	0%	2/6/23	2/10/23
Use Case Scenarios and Use Case diagrams in the System domain	Tahshin Shahriar, Slava	0%	2/6/23	2/10/23
Phase 3				
Define System Classes	Slava, Ekrem	0%	2/13/23	2/25/23
Draw the system Class Diagram	Tahshin, Fardin	0%	2/13/23	2/23/23
Define System Objects	Tahshin, Fardin	0%	2/24/23	3/3/23
Define Object Design	Nidhi, Tauseef	0%	2/24/23	3/3/23
Phase 4				
Implement the system classes	Slava, Ekrem	0%	3/6/23	3/17/23
Implement the system UI	Nidhi, Tauseef	0%	3/6/23	3/17/23
Phase 5				
Write the Unit Test Cases	Slava, Ekrem	0%	3/20/23	3/24/23
Document the whole project	Tahshin, Fardin	0%	3/19/23	4/1/23

Requirements Analysis Phase

Table of Functional Requirements:

- Customer:
    a. Log in to the system
    b. Open a new savings account
    c. Apply for a new credit account
    d. Close an account
    e. Apply for a loan
    f. Send an Interac e-Transfer
    g. Send a wire transfer
    h. Transfer funds between own accounts
    i. Deposit a cheque
    j. Pay a bill
    k. Set automatic payments
    l. Request money
    m. Set a daily / monthly limit on an account
    n. Lock a card
    o. View transaction history
    p. View transaction details
    q. View statements
    r. View reports
    s. Add a payee to the list of payees
    t. Change profile details
    u. Set new password
- Admin (bank teller)
    a. Log in to the system
    b. Register a new bank customer
    c. Manage customer accounts
    d. Manage customer settings (described in the Customer section)
    e. Add funds to one of the customer accounts
    f. View customer information

g. View customer account history
h. Request an investigation on behalf of the customer

ID	Description
FR1	Log in to the system
FR2	Open a new savings account
FR3	Close an account
FR4	Apply for a loan
FR5	Send an Interac e-Transfer
FR6	Send a wire transfer
FR7	Transfer funds between own accounts
FR8	Deposit a cheque
FR9	Pay a bill
FR10	Set automatic payments
FR11	Request money
FR12	Set a daily / monthly limit on an account
FR13	Lock a card
FR14	View transaction history
FR15	View transaction details
FR16	View statements

FR17	View reports
FR18	Add a payee to the list of payees
FR19	Change profile details
FR20	Set new password
FR21	Register a new bank customer
FR22	Manage customer accounts
FR23	Manage customer settings
FR24	Add funds to one of the customer accounts
FR25	View customer information
FR26	View customer account history
FR27	Request an investigation on behalf of the customer

Table of Non-Functional Requirements:

ID	Description
NFR1	The system must have a fast loading time of less than 3 seconds with a total number of concurrent users > 100
NFR2	The system must have a fast response time of less than 3 seconds for actions like logins, transactions, and viewing transactions history.
NFR3	The system must have a response time of less than 30 seconds for locking a card user requested
NFR4	The mobile banking system should maintain a consistent level of performance during peak usage periods, with no more than a 10% decrease in transaction processing time.
NFR5	The system must use multi-factor authentication for all user accounts to prevent unauthorized access
NFR6	The mobile banking system should implement protection against common types of attacks, such as cross-site scripting (XSS), SQL injection, and other web-based attacks.
NFR7	The system must accept only secure passwords that have a minimum length of 8 characters and that is a combination of alphabets, numbers, and symbols.
NFR8	The system should block login trials after five unsuccessful attempts

NFR9	The system should support English and French and have a clear and consistent visual design.
NFR10	Thel system interfaces must be user-friendly and simple to learn, including helping hints and messages and intuitive workflow.
NFR11	The system should be accessible to users with disabilities, conforming to relevant accessibility standards such as WCAG 2.1.
NFR12	The system should be designed to handle potential failures and recover quickly in case of a system crash or cyber attack
NFR13	The system should also be able to maintain data integrity and consistency, with all user details and transactions being accurately recorded and processed without any data loss
NFR14	The system should be able to handle any unexpected errors or exceptions, with clear and informative error messages provided to users in case of any problems
NFR15	The system must be designed with clear documentation provided to developers and system administrators.

Table of Use Cases:

ID	Description
Register a new bank customer	The system must register a new bank customer to the banking system
Log in to the system	The system should allow the user to log in to their account credentials with correct user and password
Apply for a new credit account	The system should check for the customer's financial details to allow them their selected credit limit for their newly registered credit account.
Close an account	The system should allow the user to delete their account permanently if they wish to.
Send an e-Transfer	The system should allow multiple money transfers (to a certain limit) between different registered account holders of the bank.
Transfer funds between own accounts	The system must allow customers to transfer funds between own registered accounts.

Pay a bill	The system must allow the customer to pay bills
Set automatic payments	The system should allow customers to set up autopayments on a scheduled date to pay recurring bills.
Set a daily / monthly limit on an account	The system should allow temporary daily or monthly limit on the number of transactions customers are eligible to.
Block a card	The system must allow operations for blocking all card operations for a selected card incase of emergency or suspense upon the confirmation of the customer.
View transaction history	The system must let customer access their transaction history available to view in their accounts.
View statements	The system must allow customers to access bank statements categorized by their particular account and period of time.
Change profile details	The system should allow customers to make any

	changes to their profile informations provided to the bank.
View customer information	The system should let the bank teller view all the account details of a customer.
Request an investigation on behalf of the customer	The system should allow any types of investigation upon customer's request on any transactions.

Major Goals and Table of Sub-goals

The major goals Performance, Dependability, Fault Tolerance, Cost, Maintenance, End User Criteria can satisfy the needs and requirements of its users and other systems it interacts with. Following these concepts typically enables to develop systems and components that are more maintainable. This is significant since the maintenance stage of software development may be the most expensive.

ID	Description
SG1	Optimizing software's code and algorithms to handle requests more quickly
SG2	Functionality to handle a large number of simultaneous users and transactions
SG3	Implement a backup and recovery strategy to minimize any potential data loss or corruption
SG4	Design the system to handle potential failures and recover quickly in case of any outages or disruptions
SG5	Monitor and optimize system uptime and availability, with regular testing and monitoring

SG6	Use encryption to protect sensitive data in the database
SG7	Utilizing open source software and agile development approaches to save costs
SG8	Standardizing data formats by utilizing common data structures or data interchange protocols lessens the requirement for customized data transfers by making it simpler to move data between systems
SG9	Maintenance Cost: By ensuring that the system is thoroughly tested and verified before it is published, developing with an Agile methodology can help minimize the cost of maintenance
SG10	Implement load balancing and other strategies to distribute load and prevent performance degradation
SG11	Connecting components with APIs to make it simpler to change or add functionality in the future
SG12	Utilizing open standards for data storage can facilitate the system's portability across different application domains

SG13	Creating website that can port on various devices, including desktop and mobile devices, without needing to be recompiled or updated, which can help make it simple to move the system to different platforms
SG14	Develop an intuitive and easy-to-use interface, with clear instructions and feedback
SG15	Creating a user-friendly interface, offering clear and simple documentation, and providing live support to assist users in getting the most out of the system

Table of Actors and Table of Components

Actors:

AC1) Bank Customers

AC2) Bank Tellers

ID	Description
AC1	Bank customers refer to individuals who have an account with the bank and use its services. Customers who use the "T-Banking System" can register, apply for new accounts, close existing ones, transfer money electronically to another bank, pay bills, set up automatic payments, limit the use of an account, block a card, view transaction history and statements, edit and view profile information, and request investigations.
AC2	Bank tellers or bank administrators serve the purpose of serving customers with a wide variety of financial services. They hold access to the banking system administration and have the privilege to start a new account for a customer, close an account, block a card, grant access to change the customer's details, and fulfill investigative requests.

Components:

C1)The User Interface or UX

C2)Services Backend

C3)Authentication and Security:

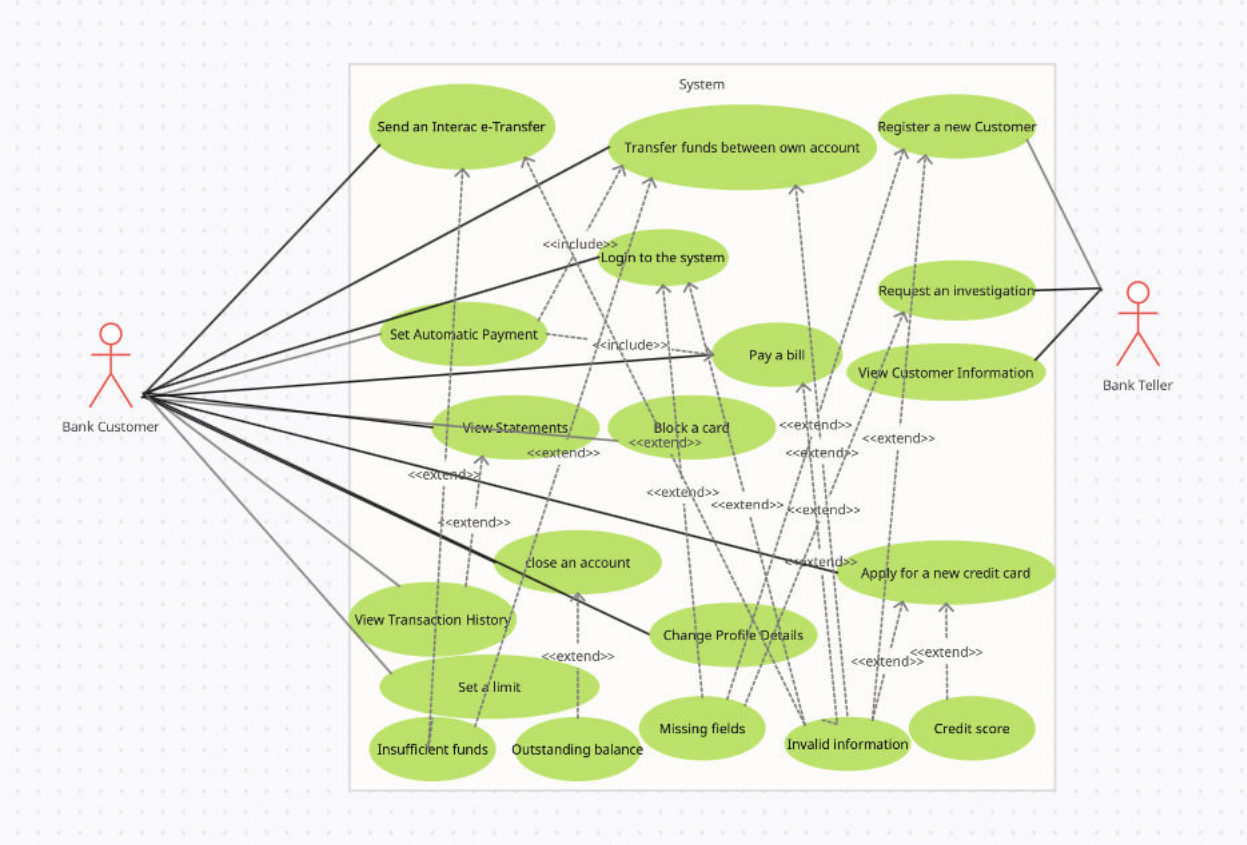
C4)Customer Account Management:

C5) Customer Service and Support

ID	Description
C1	The User interface or User experience provides the customers with overall satisfaction, enjoyment, and usefulness when interacting with the system. The design aims to provide consumers with a seamless and understandable experience by considering their needs, objectives, and expectations.
C2	The Banking System cannot function without its backend services. It offers the customers with the underpinning architecture and support for the frontend elements engaging, including the database system, load balancing, caching, security, etc. This enables developers to build dependable, scalable systems that offer a secure user experience.
C3	Authentication and Security are essential elements of the banking system. Sensitive data are shielded from misuse and unlawful access. Other methods for protecting sensitive data include user authentication like multi factor authentication, data encryption, and routine software patch updates.
C4	Account management is when customers create, maintain, and monitor their bank accounts. The services are accessible through the user's banking interface, which provides access to balances, transactions, bill-paying, and other features. Customers can

	manage their accounts more efficiently, and simple access to financial services is provided.
C5	Customer service and support are essential in any business as it helps in building customer satisfaction. Services can be accessed through emails, phone calls, social media and chatbots. Collecting feedback through surveying customers could steadily enhance the user experience. Calls and emails for support are answered promptly. These could prioritize customer services and support and create a positive impact on the service.

Use Case Diagram:



Use Case Scenarios:

Register a new bank customer

- Brief Description
    - This use case describes the process of registering a new bank customer to the banking system
- Actors
    - Bank Teller
- Triggers
    - System Administrator clicks on "Register Account" button on Bank Admin Interface
- Flow of Events
    - Basic Flow

1. Use case begins when the Bank Teller clicks on "Create Account Button" on the Bank Admin's side on the system.
2. System prompts with a page to fill in details about the new customer such as their name, address, contact information, and other identification information.
3. The Bank teller clicks on 'Register Account' button
4. The system verifies the information, branches to "Data Encryption USe Case" and saves it in the database, and generates a unique account number and a one-time password for the new customer.
5. The system displays a success message on successful registration of the customer.

- Alternative Flows and/or Subflows
    - Use case displays an error message if any required fields are left blanked or invalid characters has been entered and prompt to the user(Bank Admin) to correctly enter the information
- Special requirements
    - A valid Government issued photo ID and other basic information like SIN, TIN, and contact information.
- Preconditions
    - The bank teller is logged into the system
- Postconditions
    -
    - Success Postcondition
        i. The bank account is created successfully and unique account number and one-time password is generated
    - Failure Postcondition
        i. The use case will prompt the user to re-enter the information again
- Extension Points
    - None

Log in to the system

- Brief Description
    - The use case describes the process of logging in to the bank account. This use case appears when the user opens the mobile banking application. The application displays a login screen with the username and password
- Actors
    - Bank Customer
- Triggers
    - Bank Customer clicks on the "Login" button after entering their username and passwords
- Flow of Events
    - Basic Flow

1. Use case begins when customer enters their username and passwords and click on the login button
2. The system verifies the customer's login credentials
3. The system allows access to the customer's account if the credentials are valid
4. The system displays the account dashboard with account types followed by the balance in it along with other available options.

- Alternative Flows and/or Subflows
    - Invalid credentials
        i. If the customer's login credentials are incorrect, the system displays an error message and prompts the customer to re-enter their credentials
        ii. If the customer enters incorrect username or password multiple times, the system blocks the log in trials for 5 minutes and prompts the customer to click on 'forget passwords' to set a new password or contact the bank's customer service for assistance
    - Insufficient funds
        i. The use case ends with a pop up stating that the customer does not have enough funds.
- Special requirements
    - Username and passwords
- Preconditions
    - The customer has an active bank account
- Postconditions
    -
    - Success Postcondition
        i. The system displays the account dashboard with account balance in each account and other options.
    - Failure Postcondition
        i. The system prompts the user to enter login credentials again up to a certain time.
- Extension Points
    - None

Apply for a new credit account

- Brief Description
    - This use case describes customers filling out an online application form with their personal information such as name, address, date of birth and also other financial and employment necessity information while applying for a new credit account.
- Actors
    - Bank Customer
- Triggers
    - Customers click on the “submit application” button with their consent approved in the registration page for opening a credit account.
- Flow of Events
    - The user selects "Open a new credit account" from the account actions menu
    - The user enters their personal and financial details as well as the desired credit limit into an online application form.
    - The system verifies the customer's information and runs a credit check to check the user’s financial strength for the desired credit limit.
    - Once approved, a credit account is set and the user receives a confirmation email followed by security encryption and privacy protocols in securing the account.
    -
- Alternative Flows and/or Subflows
    - Low credit score
        i. Use case ends with the system displaying a message if the user’s financial strength is not strong enough for the credit limit they have chosen.
        ii.
    - Use case displays an error message if the information provided by the user to fill up the registration form seems incorrect(fake) or similar to another user.
- Special requirements
    - Amount of credit required
- Preconditions
    - The customer has enough funds in their account to set up a credit account for the particular chosen credit limit.
- Postconditions
    - The customer is maintaining active credit transactions
- Success Postcondition
        i. The credit account is created successfully and is linked to their online banking account
        ii. Has a unique credit card is assigned to the account to be delivered to them in person.
- Failure Postcondition
        i. The customer has been rejected to create a credit account due to their poor credit history,financial strength, or other factors.

- Extension Points
    - None

Close an account

- Brief Description
    - The use case describes the process of canceling the bank account. The use case will appear when the customer requests for the account closure through the banking application.
- Actors
    - Bank teller
    - Bank customer
- Triggers
    - The customer clicks on "Close account" button on the account dashboard and the system administrator clicks on "Close User account" on Bank Admin Interface
- Flow of Events
    - Basic Flow

1. Use case begins when the Customer applies for "Close Account" on the account dashboard.
2. The bank teller will access the Bank Admin Interface to initiate the closure of the account
3. The "Close User Account" button on the bank admin interface will let the bank teller delete all of the saved transactions and the client private information
4. Once the account is closed an automated email will be sent out to the customer
5. The customer will lose all access to the bank account after the closure

- Alternative Flows and/or Subflows
    - Alternative flow (account has funds)

1. The use cases branch to the alternative flow when the user has funds in the account during the closing request
2. Request will not be completed if there is a negative balance in the chequing/savings account
3. The system will deny the request if there is an overdue balance on the user's credit card
4. All of the failures will pop up a "Request Denied" message which will lead to the issues
5. The admin interface will provide the user with the ability to view the customer's details and obtain a report
6. The account closure will be successful once the customer has resolved the objections

- Special requirements
    - Double authorization from the system administrator
- Preconditions
    - The client has to request closure
    - Account needs to have zero balance and the overdue bills need to be paid in full
- Postconditions
    - Success Postcondition

1. Client will get a full account report to their designated email

- Failure Postcondition

    - An unsuccessful message will be sent to the client's preferred email and phone number via the banking application.
- Extension Points
    - None

Send an e-Transfer

- Brief Description
    - The use case describes the process of sending a money from the user's account to an account that is saved (or will be saved) in the contact list of the user.This use case appears when the user presses e-Transfer button on the account actions menu. The action asks for an amount of money to send and a contact registered with an email or a/ phone number to send the money to
- Actors
    - Bank Customer
    - Contact that receives the money
- Triggers
    - The user clicks on “Send Money” button on the account actions menu
- Flow of Events
    - User case begins when the customer clicks on “e-Transfer ” button on the dashboard in their account
    - System prompts with a page to fill in fields including the amount of transactions, account type to send money from, and name and contact information of the receiver if the receiver is not a saved contact. If the receiver is in the saved contact list, it will ask the customer to select the receiver from the saved contact list.
    - Customer clicks on the “Complete Transfer” button
    - The system verifies that the amount entered, and receiver’s information is valid and the user has sufficient funds to make the transfer
    - If the transfer is successful, the system deducts the transferred amount from the sending account and sends it to the receiving account.
    - The system displays a success message and shows new account balances.
- Alternative Flows and/or Subflows
    - Incorrect information
        i. The use case ends with the system displaying an error message if the amount is entered incorrectly or the receiver’s information is incorrect.
    - Insufficient Funds
        i. The use case ends with a message stating that there’s not enough funds and prompts the user to re-enter the amount.
- Special requirements
    - Amount of money and email or phone number of the contact
- Preconditions
    - The customer must be logged in

- Postconditions
    -
- Success Postcondition
    - The system displays a brief report of the transfer statement including the information of the receiver and the sent amount of money
    -
- Failure Postcondition
    - the system prompts the user to re-enter the information
- Extension Points

Transfer funds between own accounts

- Brief Description
    - This use case describes the process of transferring funds between own accounts
- Actors
    - Bank Customer
- Triggers
    - Customer Clicks on the "Complete Transfer" button on the "Transfer Fund" page of their account
- Flow of Events
    - User case begins when the customer clicks on "Transfer Funds" button on the dashboard in their account
    - System prompts with a page to fill in multiple fields including account types from which the fund would be transferred to the account type receiving the fund and the amount of funds.
    - Customer clicks on the "Complete Transfer" button
    - The system verifies that the amount entered is valid and the user has sufficient funds to make the transfer
    - If the transfer is successful, the system deducts the transferred amount from the sending account and adds it to the receiving account.
    - The system displays a success message and shows new account balances.
- Alternative Flows and/or Subflows
    - Use case displays an error message if the amount is entered incorrectly or if there's not enough fund and prompts the user to re-enter the amount
- Special requirements
    - Amount of money and account types to transfer funds from and to
- Preconditions
    - The customer must be logged in
- Postconditions
    -
    - Success Postcondition
        i. The system displays a success message and new account balances.

    - Failure Postcondition
        i. The system prompts the user to enter the amount again
- Extension Points

Pay a bill

- Brief Description
    - This use case describes the process of sending money from the user's account to the billing company. This use case appears on clicking "Pay Bill" button on the dashboard
- Actors
    - Bank Customer
    - Billing Company
- Triggers
    - Customer clicks on the "Complete Payment" button in the "Pay Bill" page of their account
- Flow of Events
    - User case begins when the customer clicks on "Pay Bill" button on the dashboard in their account
    - System prompts with a page to fill in multiple fields including name of the billing company, account number of the billing company, payment amount, and user's account type for sending the money.
    - Customer clicks on the "Complete Payment" button.
    - The system verifies that the bill information and amount entered is valid, the user has sufficient funds to make the payment
    - If the transfer is successful, the system deducts the payment amount from the user's account and send it to the billing company
    - The system displays a success message and shows the new account balance.
- Alternative Flows and/or Subflows
    - Use case displays an error message if any required field is left blank, bill information or amount is entered incorrectly or if there's not enough fund and prompts the user to fill in the fields correctly
- Special requirements
    - Name and account no of the billing company
- Preconditions
    - The customer must be logged in the system
- Postconditions
    -
- Success Postcondition
    - The system displays a brief report of the transfer statement including the information of the billing company and the sent amount of money
- Failure Postcondition

    - The system prompts the user to re-enter the information
- Extension Points

Set automatic payments

- Brief Description
    - This use case describes the process of setting autopayments on a scheduled date to pay a recurring bill such as a credit card, utility bill or mortgage
- Actors
    - Bank Customer
- Triggers
    - The customer clicks on "Scheduled payment" on user interface
- Flow of Events
    - Case starts when the client logs into the account and accesses the menu from the dashboard
    - Menu will have the option to pay bills and it will require billing information
    - Client will have to add payee by entering company name, account number and zip code
    - After adding the payee to the list the option for making the recurring payment will appear
    - The payment will require the amount, the account the money will be deducted from and setting up the delivery dates
- Alternative Flows and/or Subflows (Incorrect information)
    - The use case branches to an alternative flow when the user inputs incorrect information (Wrong payee name or account number)
    - The input box will pop up a message saying incorrect information and it will eventually fail the transaction
    - Use case will have an "OK" button which will lead back to the payee information page
- Special requirements
    - None
- Preconditions
    - Need to add the payee information
- Postconditions
    - Success Postcondition

        1. The application will show a message if the payment is successful

- Failure Postcondition
    - It will take back to the system which will require the user input after it shows an error
- Extension Points
    - None

Set a daily / monthly limit on an account

- Brief Description
    - This use case indicates a temporary daily or monthly limit on the number of transactions clients can perform on their savings or checking accounts.
- Actors
    - Bank customers
- Triggers
    - The customer selects the "Daily Transaction Limits" on the account summary page
- Flow of Events
    - Customer clicks on the Summary page from the user dashboard
    - A user prompt will ask to "Choose Limits" for the account
    - The option will contain ATM withdrawal, purchase or transfer limits for the account
    - The prompt will ask for the expiry date
    - "Done" button will succeed the process
- Alternative Flows and/or Subflows
    - None
- Special requirements
    - None
- Preconditions
    - The client is logged in
    - Putting proper information
- Postconditions
    - Success Postcondition

        1. The application will show a message if the limit has been set

- Failure Postcondition
    - It will keep showing error messages if the information is incorrect
    - The prompt page will require correct information afterwards
- Extension Points

    - None

Block a card

- Brief Description
    - This use case describes the operation of blocking all card operations for a selected card.
- Actors
    - The client
- Triggers

    - The client selects a specific card in the provided menu and confirms the blockage of the card.
- Flow of Events
    - Normal flow
        i. The client clicks on the "Privacy and Security" button, located next to the client's name on top of the screen.
        ii. From the provided page with Privacy and Security options, the client selects "Block a card".
        iii. The system shows a list of cards that the client has.
        iv. The client selects a card they want to block.
        v. The system provides a message with a list of consequences this will have, with buttons "Lock the card", "Select a different card" and "Cancel".
        vi. The client clicks on "Block the card".
        vii. The banking system blocks the corresponding card.
        viii. The use case ends with a pop up message indicating that the card has been successfully blocked, which also provides bank contact information to further investigate the issue leading to the card blocking (stealing, etc.). The client is returned to the "Privacy and Security" section.
    - Alternative Flow (Cancel the operation)
        i. The use case branches to the alternative flow when the client selects "Cancel" (Normal flow, v).
        ii. The use case ends with the client returned to the "Privacy and Security" section of the system.
- Special requirements
    - None
- Preconditions
    - The client is logged in.
    - The client has to have an existing physical or virtual card linked to one of the accounts.
- Postconditions
    - Success Postcondition
        i. xviii. The card selected before is locked.
        ii. The client is shown the "Privacy and Security" section of the banking system.
    - Failure Postcondition
        i. The client is returned to the "Privacy and Security" section of the banking system.
- Extension Points
    - None

View transaction history

- Brief Description

    - This use case describes the process of a client accessing transaction history.
- Actors
    - The client
- Triggers
    - The client clicks on the corresponding account.
- Flow of Events
    - Normal flow
        i. The use case starts with the client being on the main page, where he clicks on one of the accounts they want to see the history of.
        ii. The use case ends with the page with the corresponding account and its history shown.
    - Alternative Flows and/or Subflows
        i. xxiii.None
- Special requirements
    - None
- Preconditions
    - The client is logged in.
    - The client has an account (chequing, savings, credit).
- Postconditions
    - Success Postcondition
        i. xxiv.The transaction history is shown to the client.
    - Failure Postcondition
        i.
- Extension Points
    - The client can select a transaction to see detailed information about it, such as exact time of the transaction, location, etc.

View statements

- Brief Description
    - This use case describes the process of a client accessing a specific bank statement. The statements are categorized by account (chequing, credit, savings) and period of time.
- Actors
    - Client
- Triggers
    - The client clicks on the "View Statement" button after specifying the account and period.
- Flow of Events
    - Normal Flow
        i. xxvi.The client clicks on the corresponding account on the main page of the banking system.

ii. xxvii.The client clicks on the “View Statements” button located above the history of transactions.
iii. xxviii.The system provides a list of periods for which the bank statements exist.
iv. xxix.The user clicks on the “View Statement” button next to the necessary period.
v. The use case ends with the statement being shown.

- Alternative Flows and/or Subflows
  i. xxxi.None

- Special requirements
  - None
- Preconditions
  - The client is logged in
- Postconditions
  - Success Postcondition
    i. The bank statement is shown to the client.
- Extension Points
  - The statement is available for download.

Change profile details

- Brief Description
  - The use case describes the process of a client accessing the profile information section and changing the section corresponding to the needs.
- Actors
  - Client
- Triggers
  - The client clicks on the “pencil” icon next to one of the sections of personal information, while being inside the “Personal Information” section of the banking system.
- Flow of Events
  - Normal flow
    i. xxxiii.The use case starts, with the client logged in to the system, clicking on the “Personal Information” button located next to the client’s name at the top of the page.
    ii. xxxiv.The client clicks on the pencil icon next to the information they would like to change (e-mail, address).
    iii. xxxv.The system asks to input the updated information.
    iv. xxxvi.The client inputs the information and clicks on “Update”.
    v. xxxvii.The system makes updates to the information in the database. If the updated information is to be manually checked, it is sent to the corresponding department (address or name change).

vi. xxxviii.The use case ends with a return to the page with personal information and a pop up saying that the information has been updated.

- Alternative Flows and/or Subflows
  - Alternative flow (incorrect information provided)
    i. xxxix.The use case branches to the alternative flow when the user provides incorrect information (the information doesn't match the regex, etc.) (Normal flow, iv).
    ii. The system shows a pop up saying that the information provided is incorrect and gives a reason (doesn't match the format, too many symbols, etc.)
    iii. The use case ends with the user clicking on "Okay" on the pop up and being returned to the form to input information again.
- Special requirements
  - None
- Preconditions
  - The client is logged in.
- Postconditions
  -
  - Success Postcondition
    i. The personal information page is shown. The information is either immediately updated, or sent to the bank for a manual check.
  - Failure Postcondition
    i. The system is waiting for user interaction. The form to provide updated information is shown.
- Extension Points
  - None

View customer information

- Brief Description
  - This use case describes the process of checking a client account information.
- Actors
  - Bank teller
- Triggers
  - Bank teller clicks on the "Find a Client" button on the main page of the visual interface.
- Flow of Events
  - Normal flow
    i. Bank teller clicks on the corresponding button on the main page of the visual interface.
    ii. A prompt asks to input the search query. The options are: a) search by account number (unique customer ID); b) search by personal information (client name, phone number, e-mail address).

iii. Bank teller inputs the information (either a or b).
iv. xlvii.The system shows the list of the corresponding to the search query accounts.
v. xlviii.The bank teller selects the account from the search results and clicks on “Manage Client”.
vi. The use case ends with the client account management page being shown, with the system adding the action of access to the account to the access history.

- Alternative Flows and/or Subflows
  - Alternative flow (client not found)
    i. The use case branches to the alternative flow when there are no search results (normal flow, iii).
    ii. A pop up dialog appears saying “No search results”.
    iii. The use case ends with the bank teller clicking on “Okay” on the pop up. The system returns the bank teller to the search form (Normal flow, ii).
- Special requirements
  - None
- Preconditions
  - The bank teller is logged in.
  - The customer exists in the banking system.
- Postconditions
  -
  - Success Postcondition
    i. System waiting for interaction. The client management page shown to the bank teller and a corresponding element added to the client account access history.
  - Failure Postcondition
    i. System waiting for interaction. The search form is shown.
- Extension Points
  - None

Request an investigation on behalf of the customer

- Brief Description
  - This use case describes the process of requesting an investigation on one of the customer’s transactions. This use case is called from the use case “View Customer Information”
- Actors
  - Bank Teller
- Triggers

        - Bank Teller clicks on the "Submit Investigation Form" button on one of the transactions in the Transaction History page of a customer.
- Flow of Events
    - Basic flow
        i. Use case begins when the Bank Teller clicks on "Request Investigation".
        ii. System prompts with a page with multiple fields to fill in details regarding the investigation request.
        iii. Bank teller clicks on "Submit Investigation Request".
        iv. System saves the request and sends it to the investigation department.
        v. Use case ends with a pop-up dialog showing the request reference number and a success message, with the application form being closed. Return to the transaction history of the client.
    - Alternative Flow (Insufficient Information)
        i. Use case branches to the "Insufficient Information" use case when at least one of the mandatory fields in the form was not filled in and the bank teller clicks on "Submit Request" (Basic flow, iii).
        ii. A pop up dialog appears with a message saying "Insufficient Information. Please fill in all the missing fields marked red.
        iii. The use case ends with clicking "Okay" on the pop up, and returns to the application, with fields required now marked red.
- Special requirements
    - None
- Preconditions
    - The bank teller is logged in.
    - The customer has a transaction recent enough to do the investigation.
- Postconditions
    - Success Postcondition
        i. System waiting for user interaction. Transaction history shown. The application is sent to the department.
    - Failure Postcondition
        i. Transaction history shown. No changes done to the system or reports sent.
- Extension Points
    - None

Trigger: what will start the use case

Actors: that participate in this use case

Entry conditions: that must hold in order for the use case to begin at all

Success guarantee: if the use case falls all the way to the end through the main flow of events, it is a success – the postconditions of which define

what it actually means

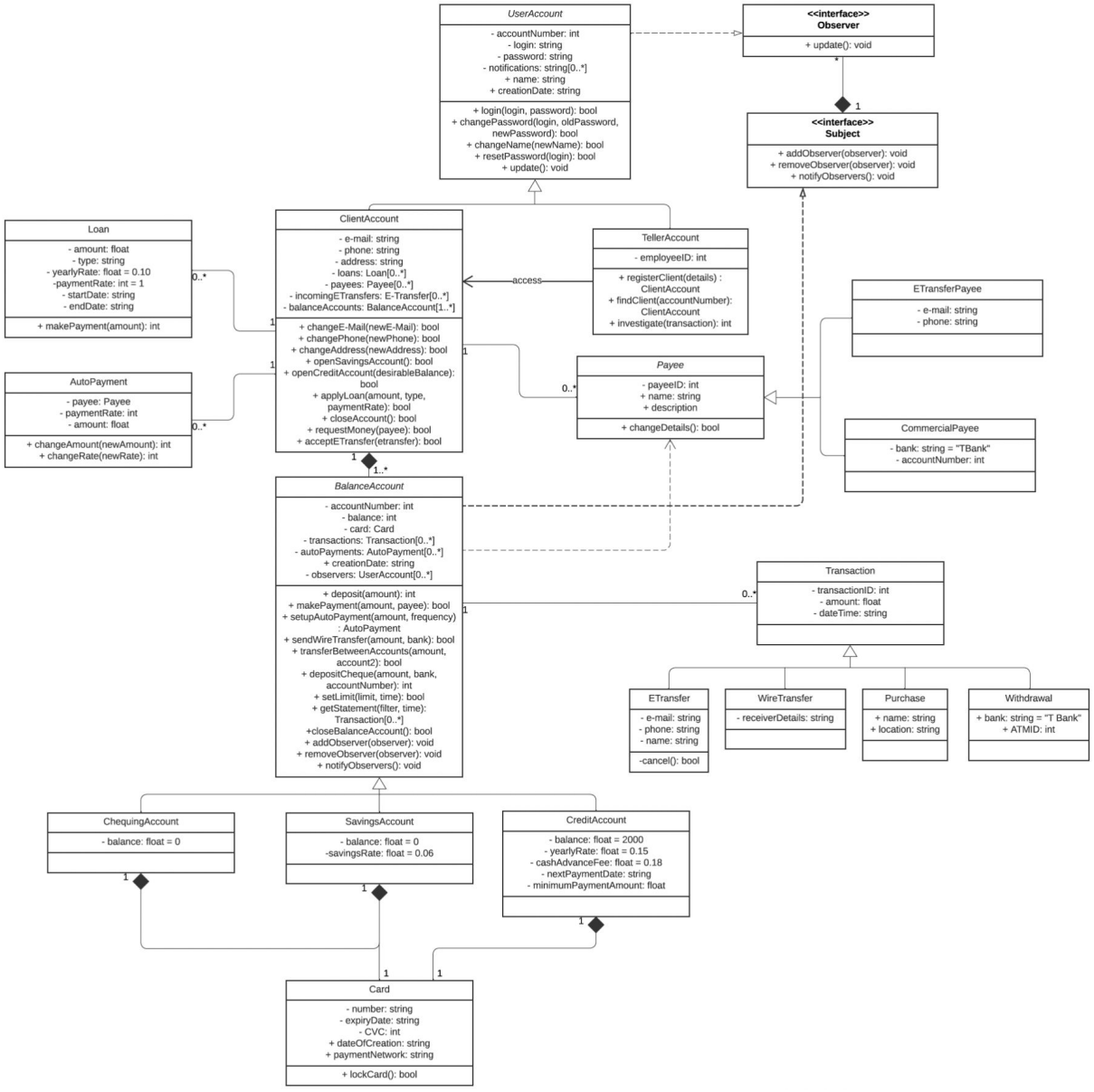
Minimum guarantee: if the use case ends via ANY of the alternative flows, it did not succeed, and failure postconditions define what should hold in that case (at least)

Steps in the main (basic) and alternative (exceptional) flows are often numbered to facilitate searching and cross-referencing

System Design Phase

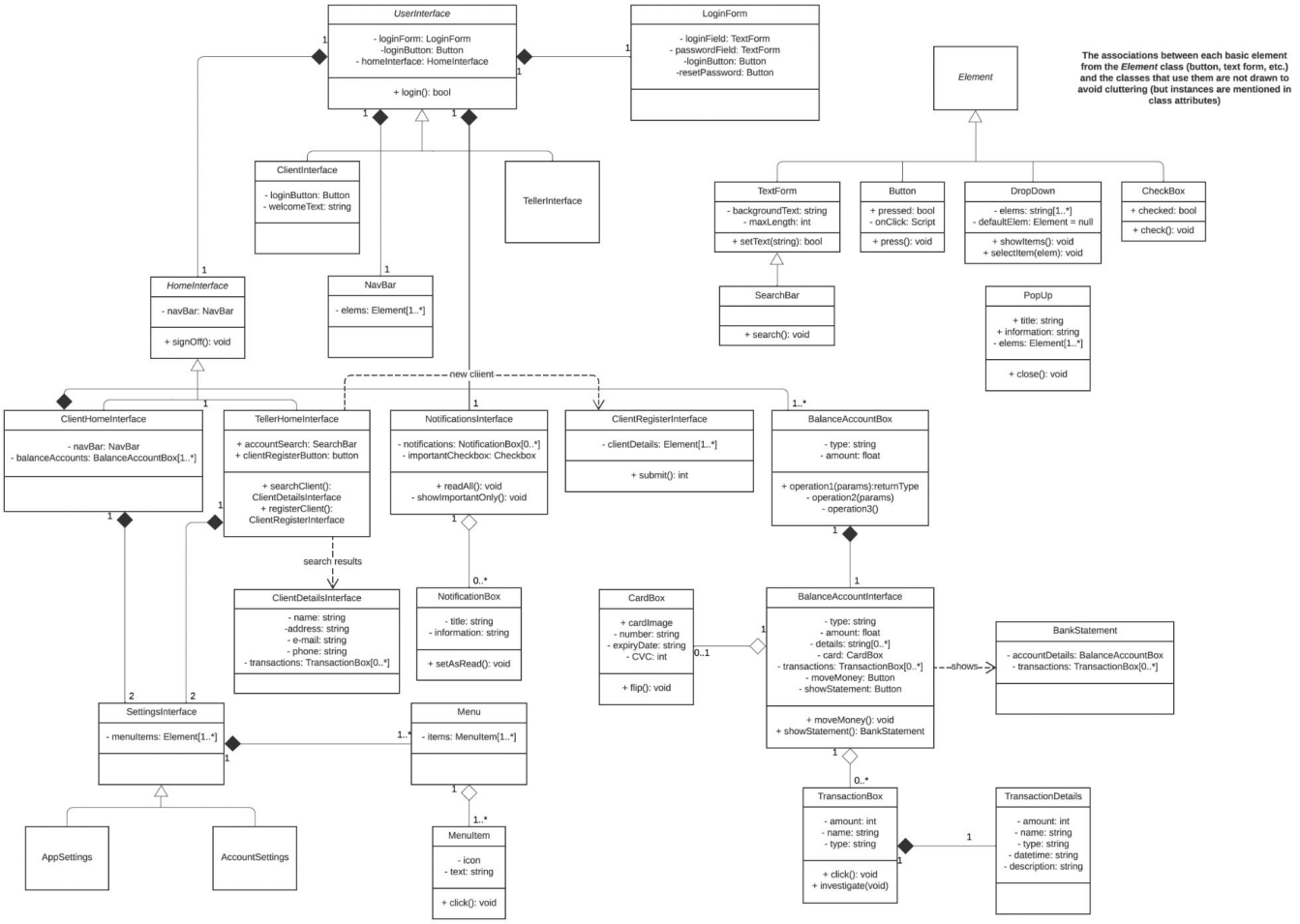
Class Diagrams:

UML Class Diagram



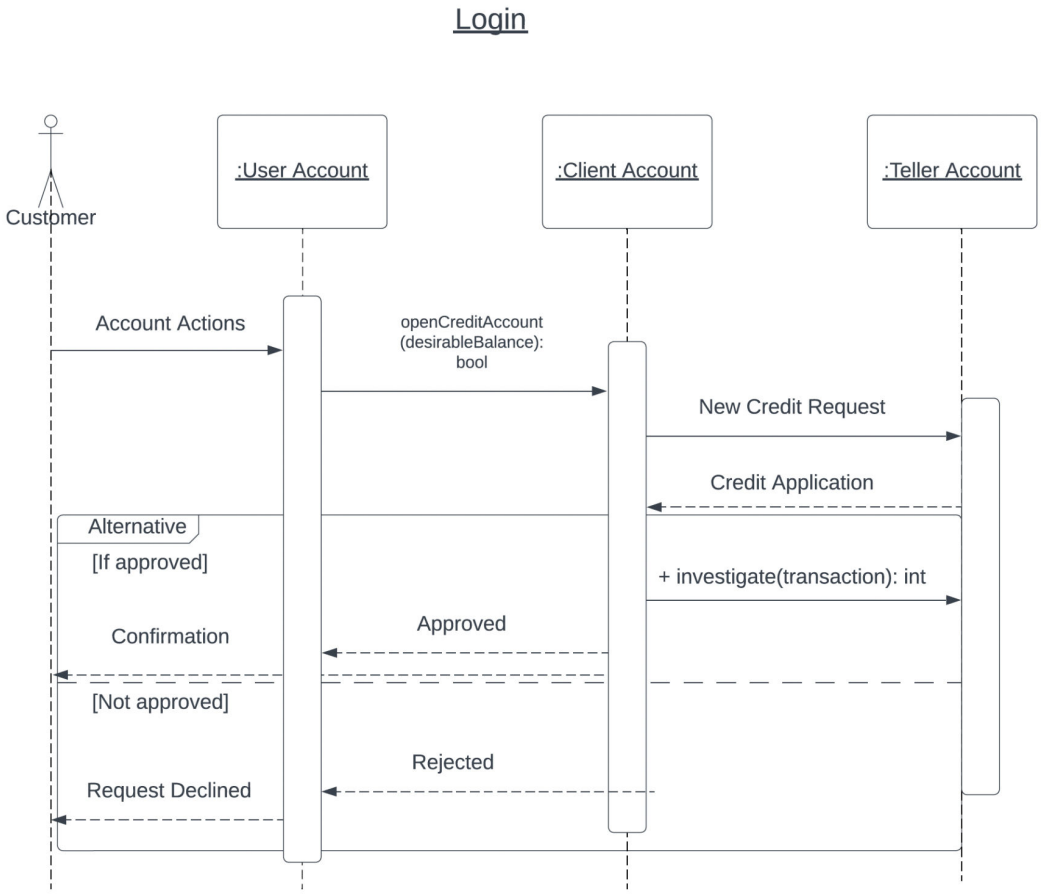
UML UI Class Diagram

***Something*Interface stands for a "user interface" (not related to the OOP concept of an interface)**

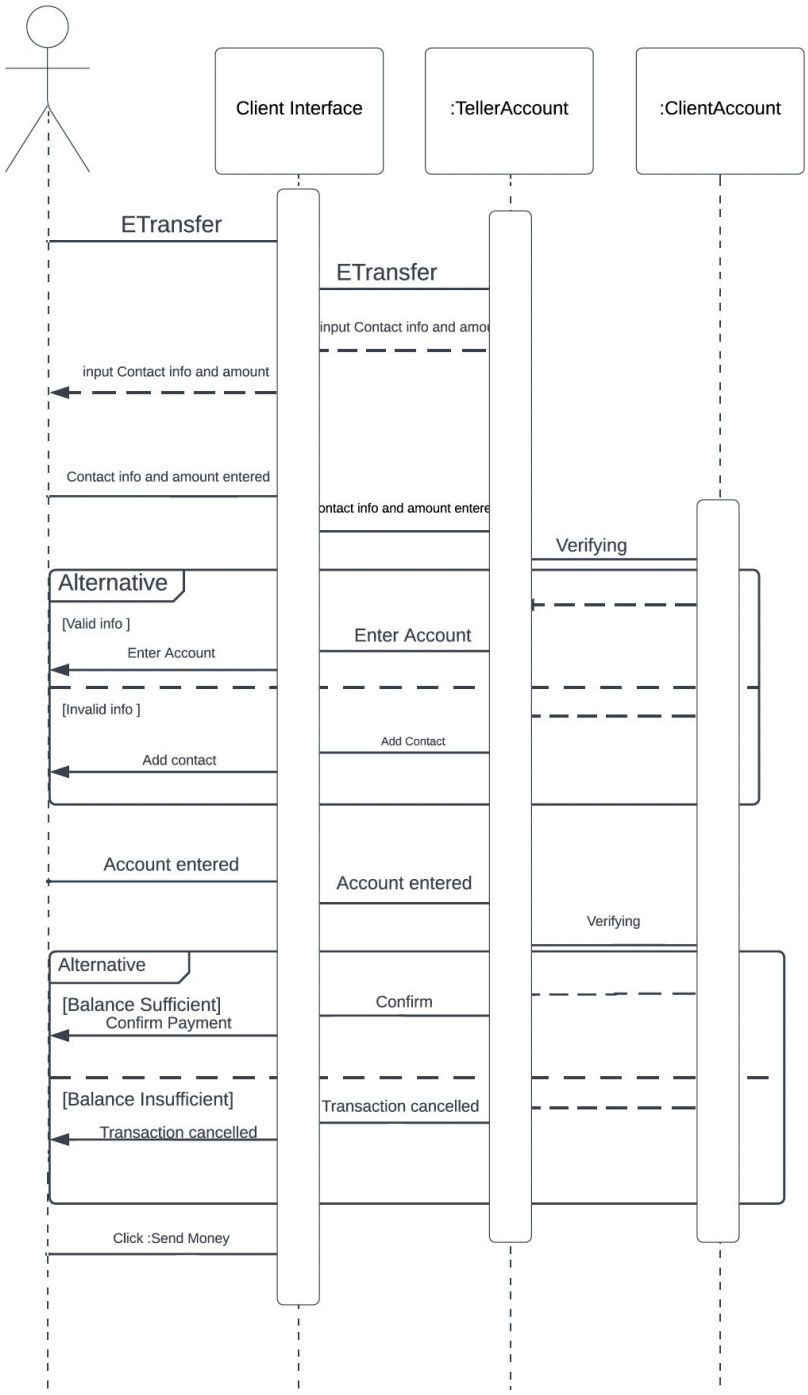


Sequence Diagrams:

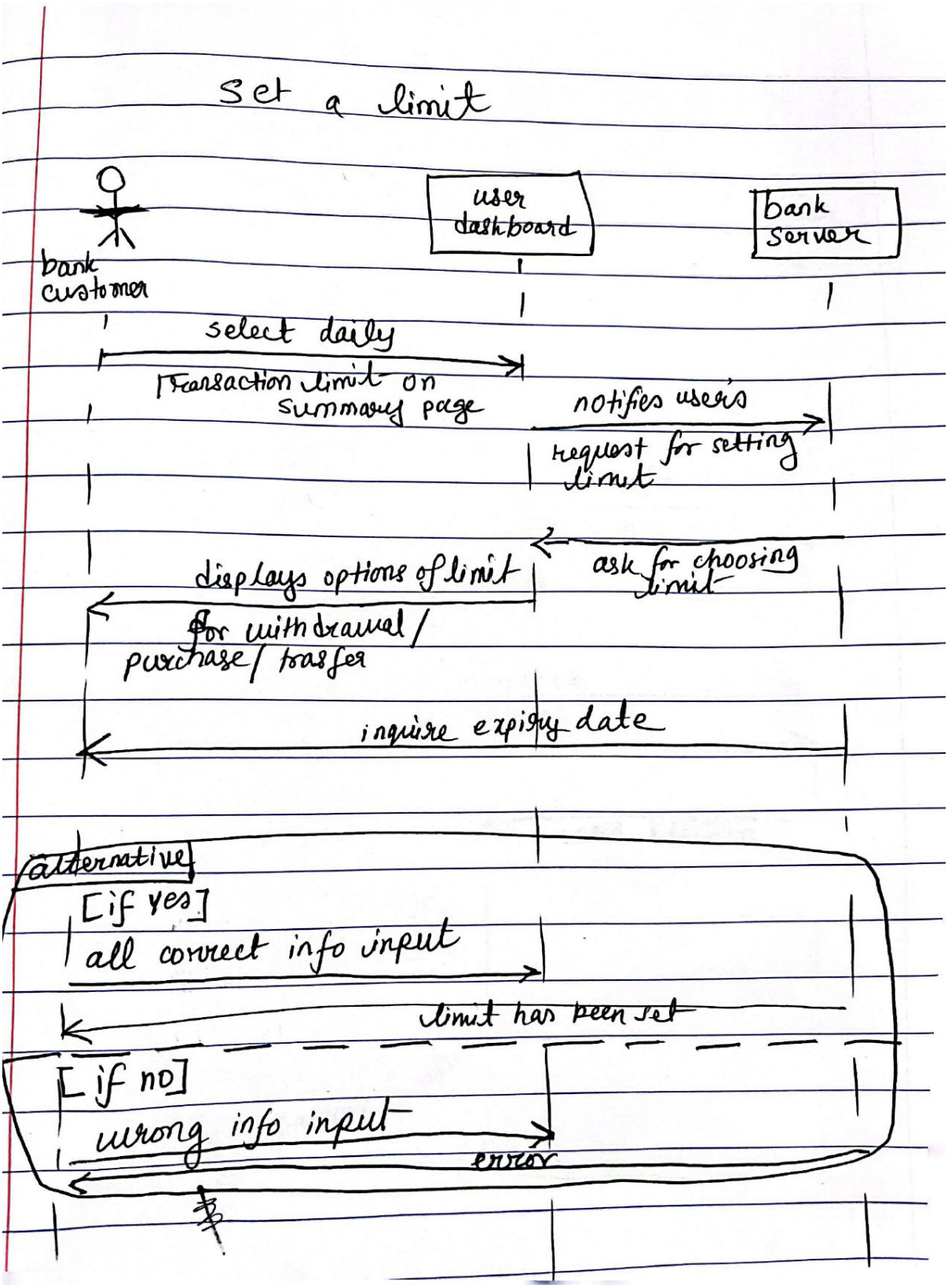
Apply for a Credit Card



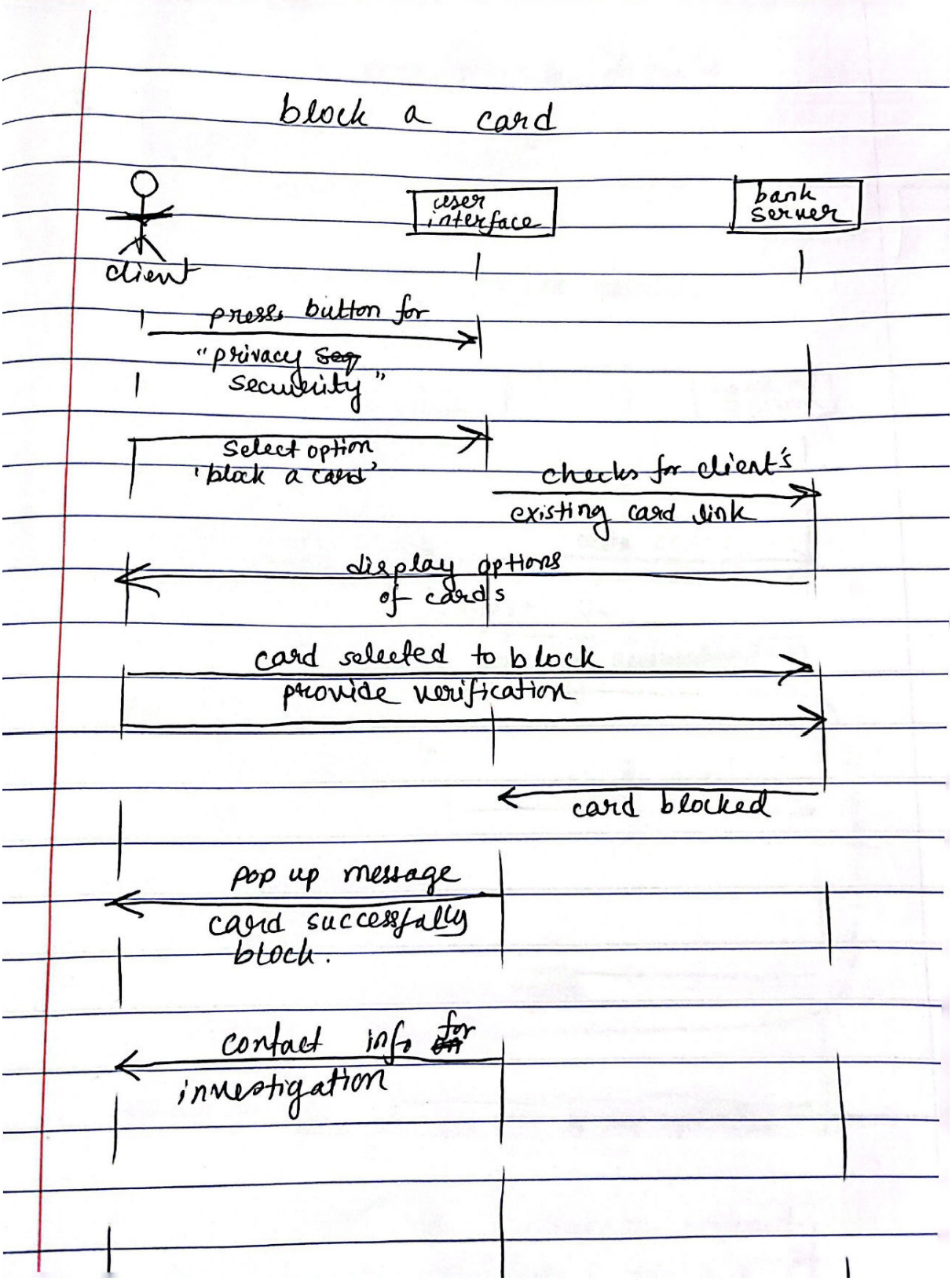
E-Transfer


Client Interface
:TellerAccount
:ClientAccount
ETransfer
ETransfer
input Contact info and amount
input Contact info and amount
Contact info and amount entered
Verifying
Alternative
[Valid info]
Enter Account
Enter Account
[Invalid info]
Add Contact
Add contact
Account entered
Account entered
Verifying
Alternative
[Balance Sufficient]
Confirm Payment
Confirm
[Balance Insufficient]
Transaction cancelled
Transaction cancelled
Click :Send Money

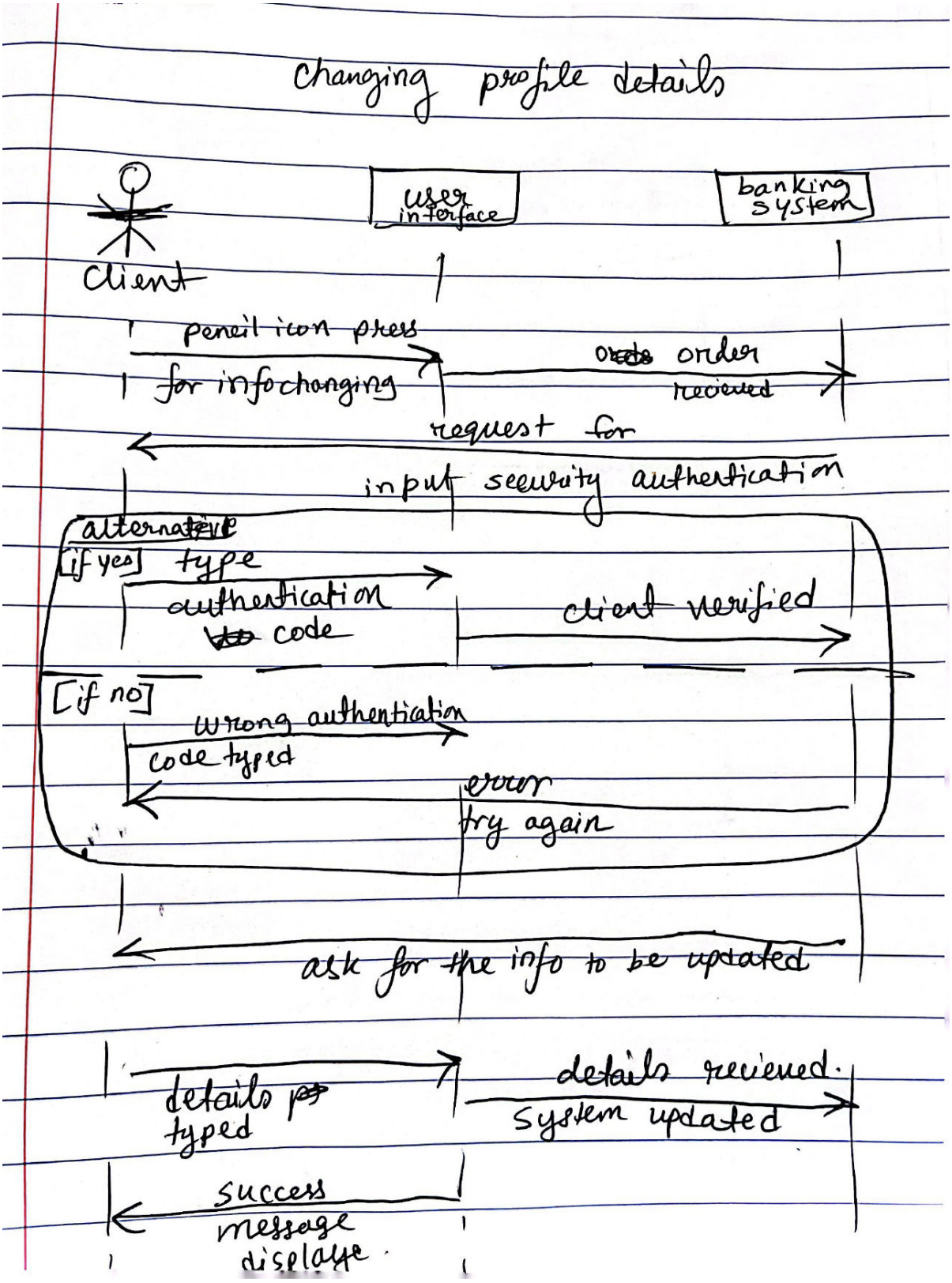
Set a Limit


Set a limit
bank customer
user dashboard
bank server
select daily transaction limit on summary page
notifies users request for setting limit
ask for choosing limit
displays options of limit for withdrawal / purchase / trasfer
inquire expiry date
alternative
[if yes]
all correct info input
limit has been set
[if no]
wrong info input
error

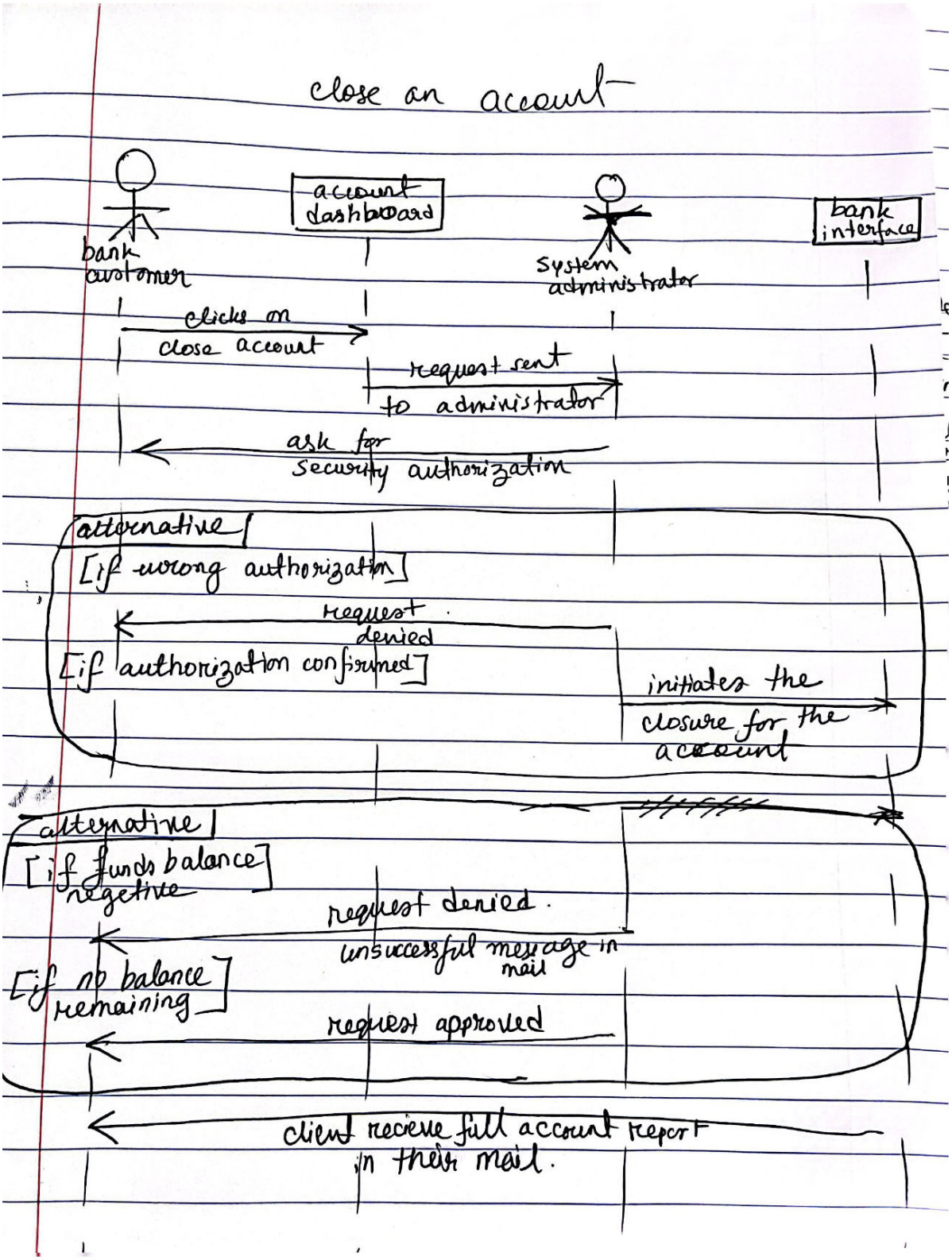
Block a Card


block a card
client
user interface
bank server
press button for "privacy security"
Select option 'block a card'
checks for client's existing card link
display options of cards
card selected to block
provide verification
card blocked
pop up message
card successfully block.
contact info for investigation

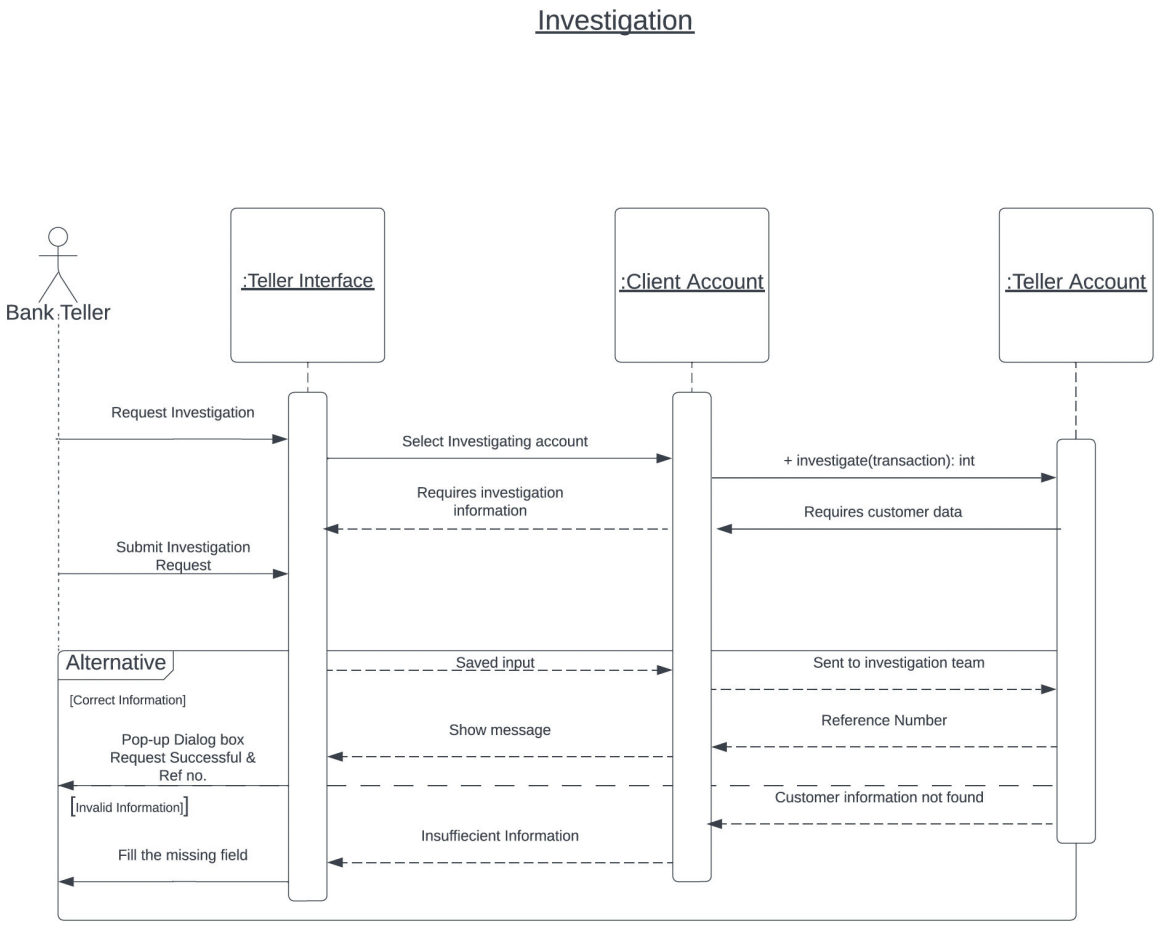
Changing Profile Details



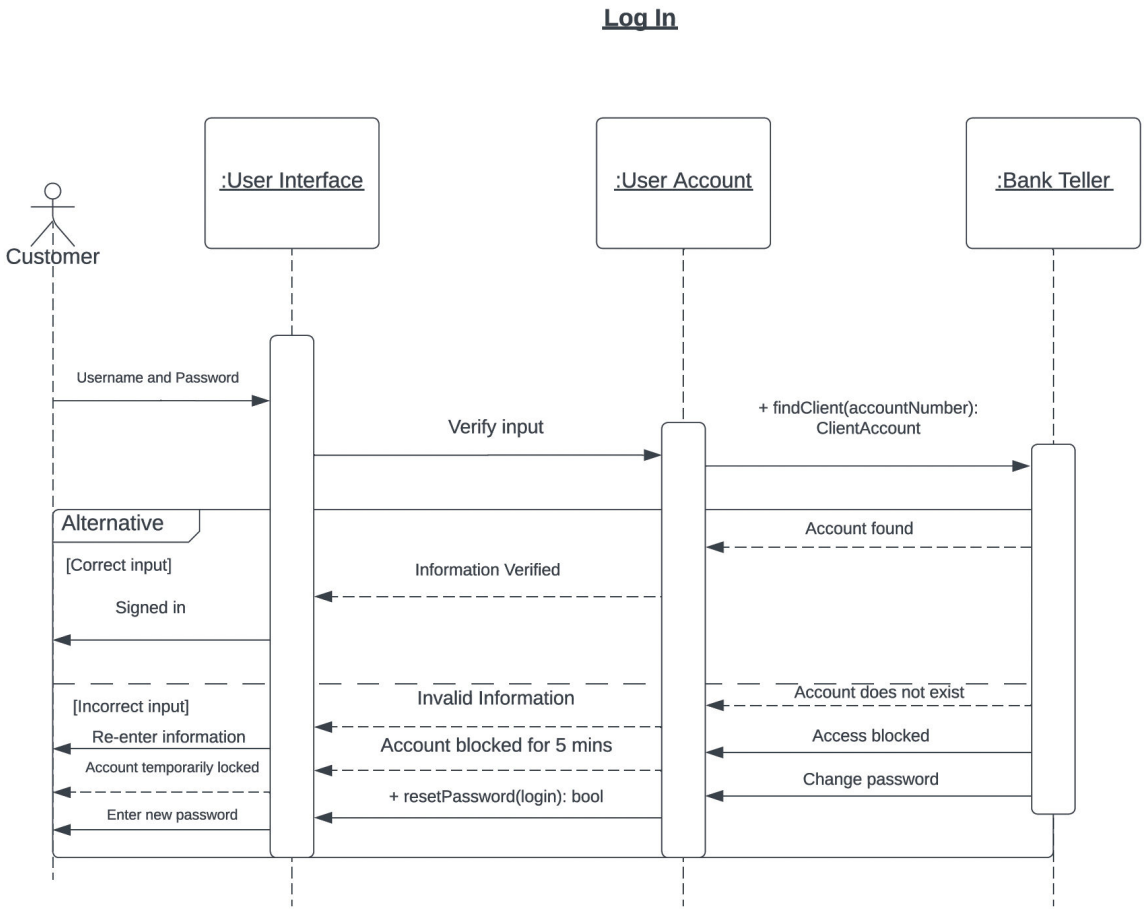
Close an Account



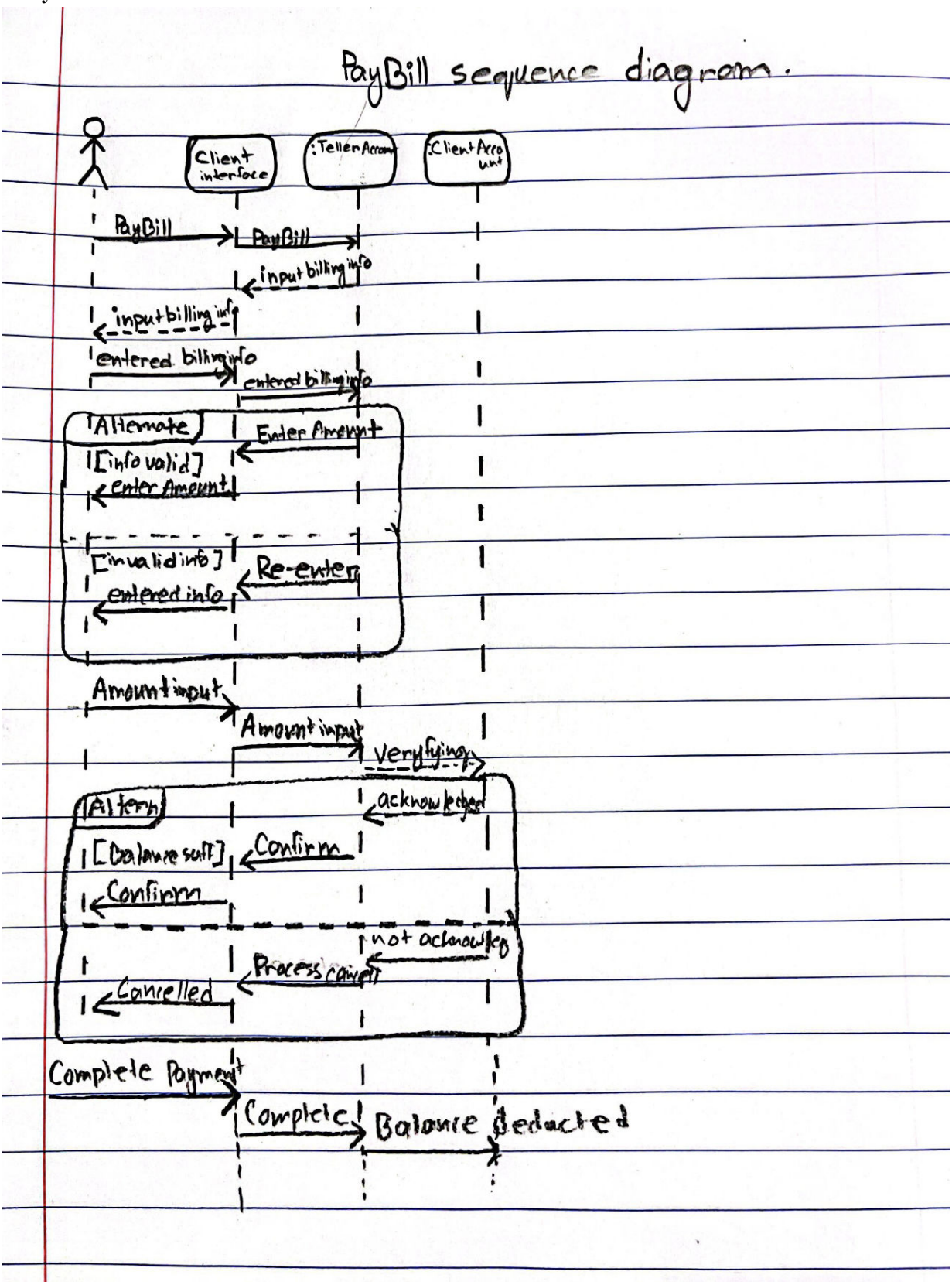
Investigation



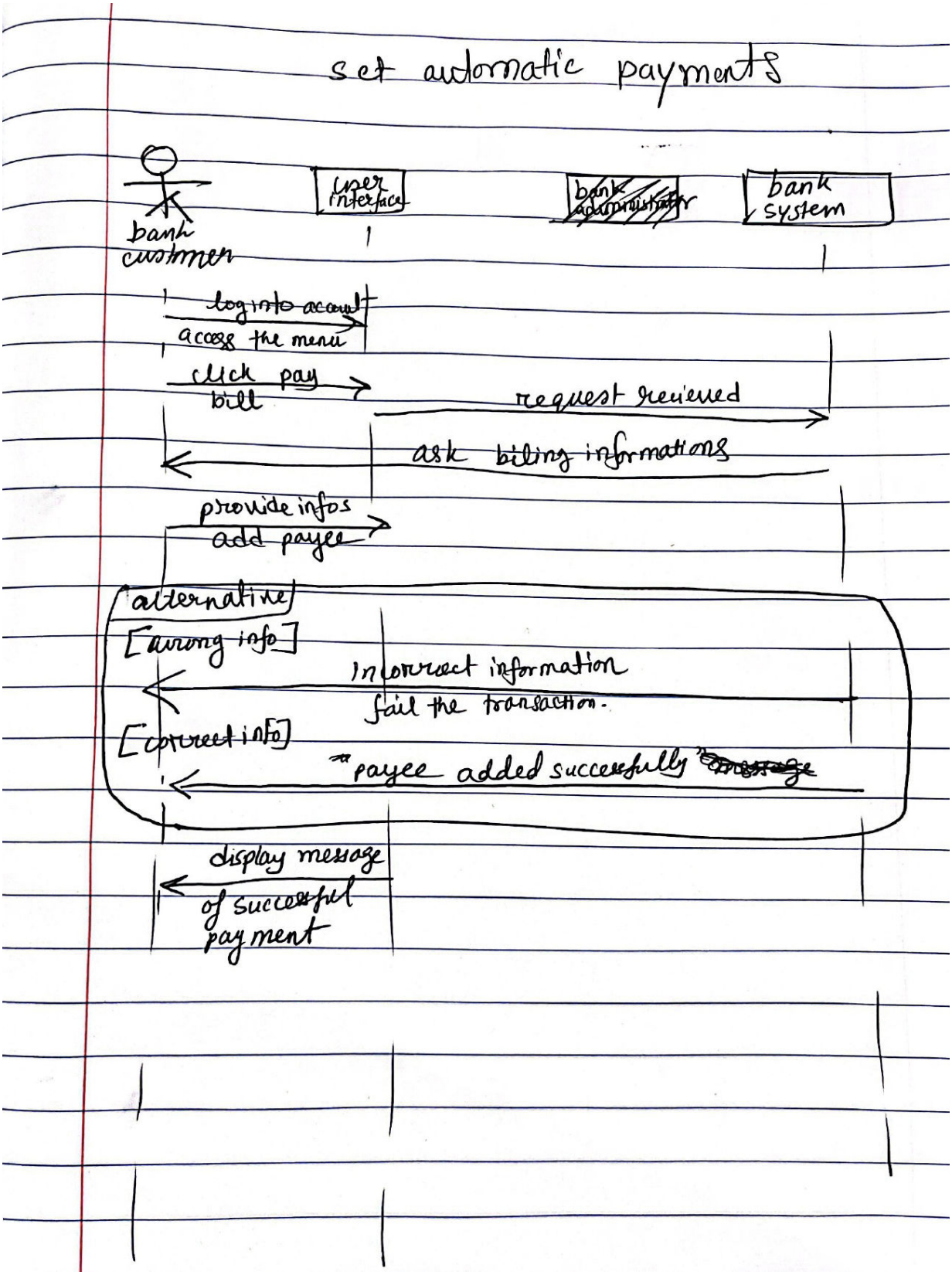
Log in


Log In
:User Interface
:User Account
:Bank Teller
Customer
Username and Password
Verify input
+ findClient(accountNumber):
ClientAccount
Alternative
Account found
[Correct input]
Information Verified
Signed in
[Incorrect input]
Invalid Information
Account does not exist
Re-enter information
Account blocked for 5 mins
Access blocked
Account temporarily locked
Change password
+ resetPassword(login): bool
Enter new password

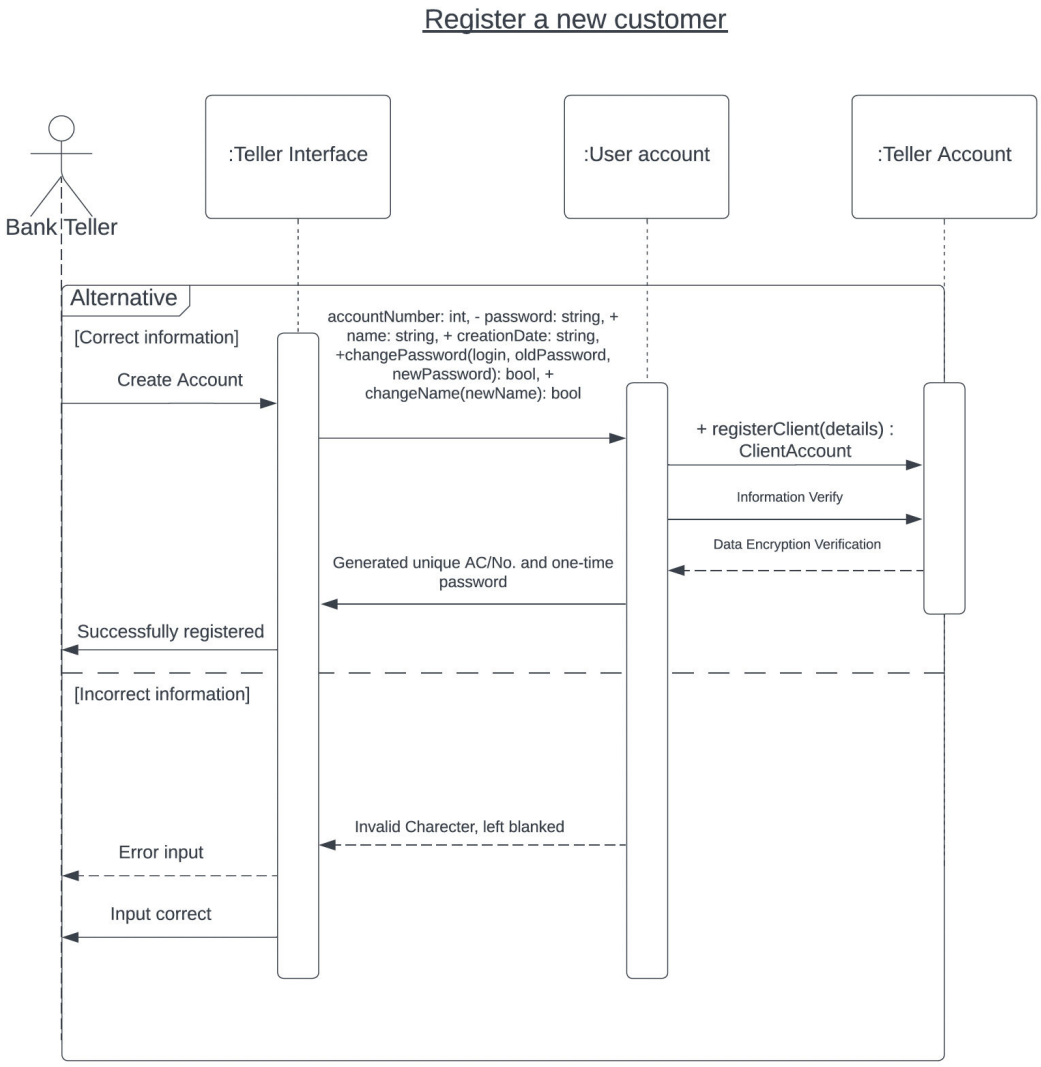
Pay Bill


PayBill sequence diagram.
Client interface
:TellerAcco
:ClientAcco unt
PayBill
PayBill
input billing info
input billing info
entered billing info
entered billing info
Alternate
Enter Amount
[info valid]
enter Amount
[invalid info]
Re-enter
entered info
Amount input
Amount input
verifying
Altern
acknowledged
[Balance suff]
Confirm
Confirm
not acknowled
Process cancel
Cancelled
Complete Payment
Complete
Balance deducted

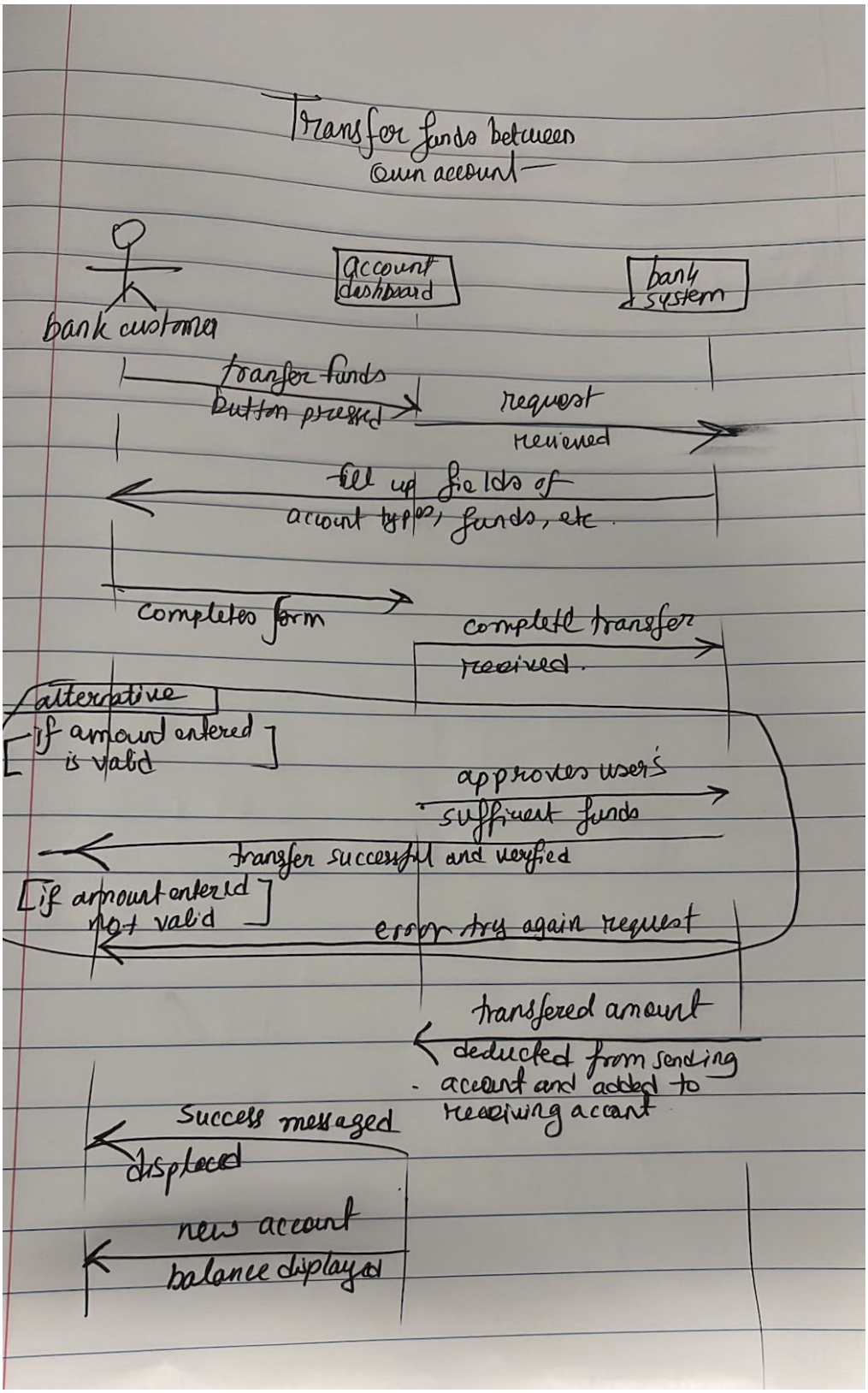
Set Autopayments



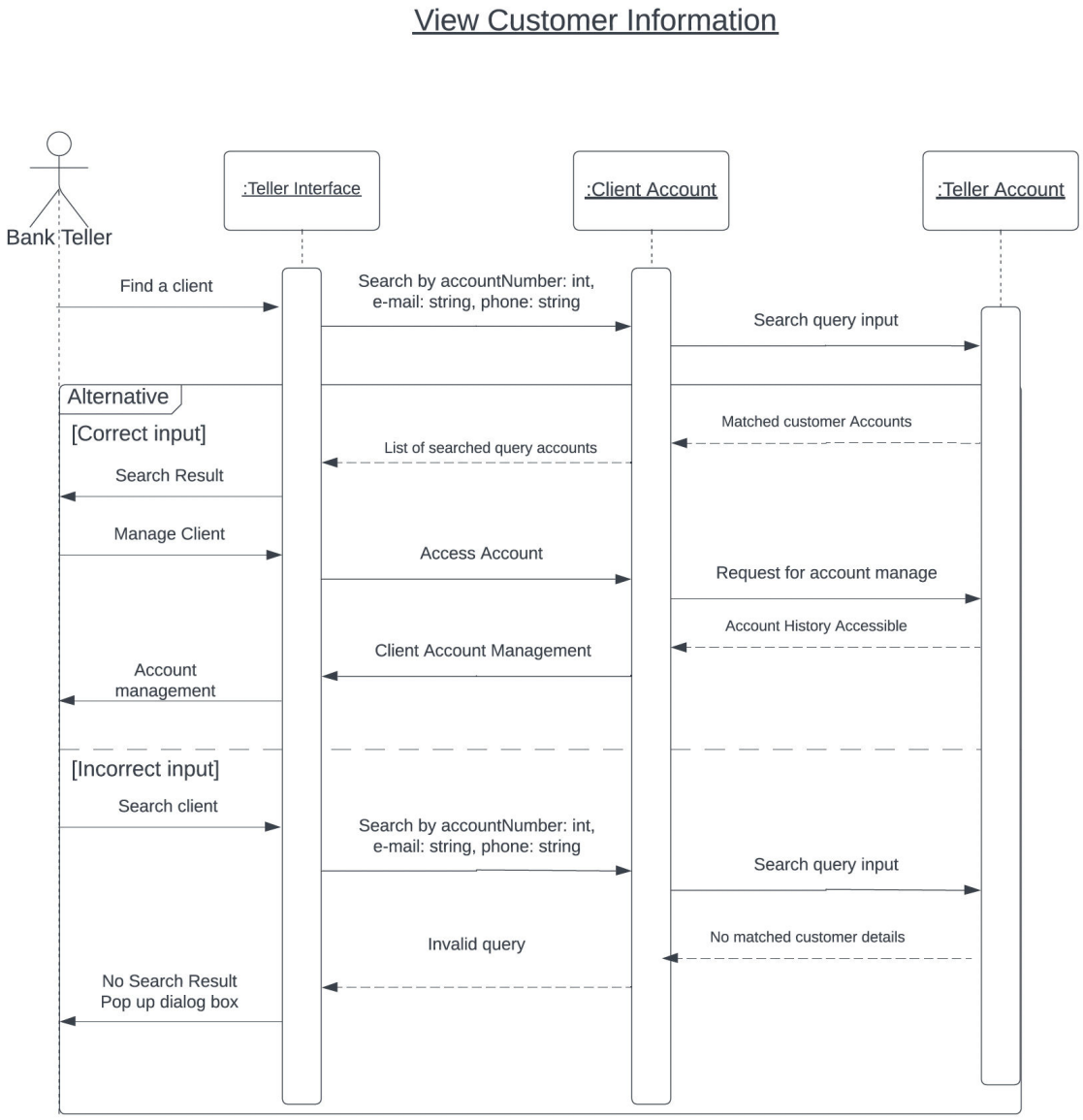
Register a New Customer



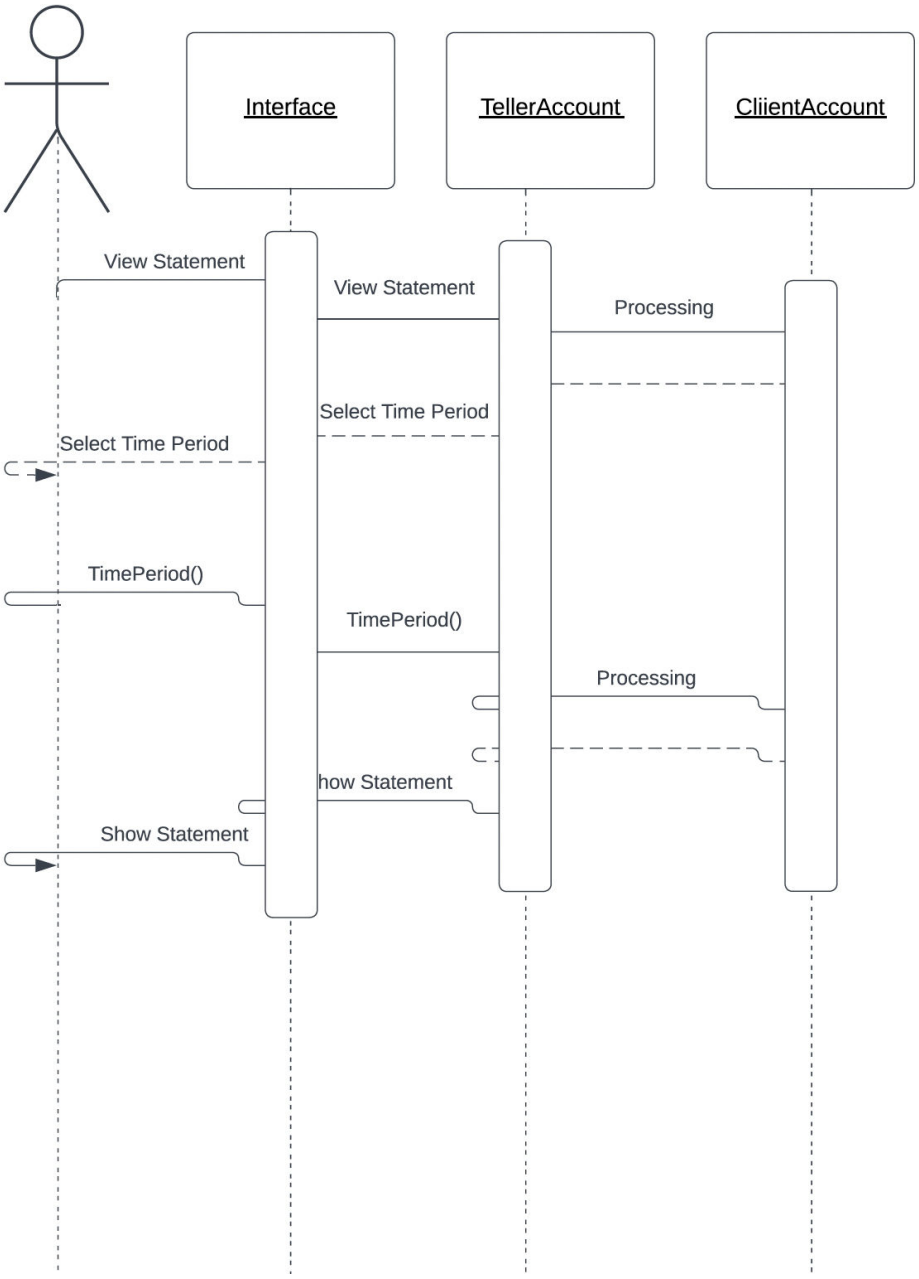
Transfer Between Accounts



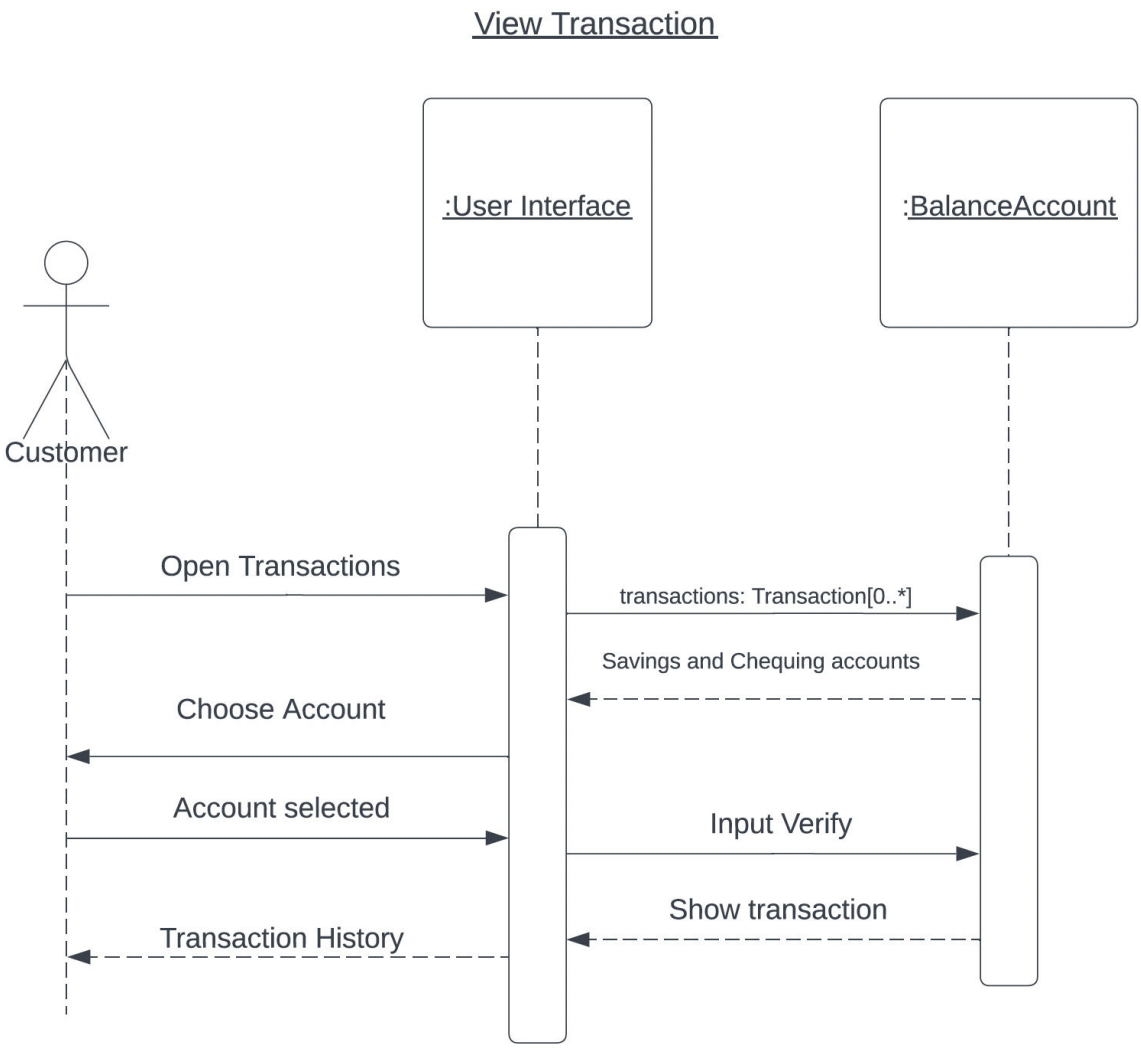
View Customer Information



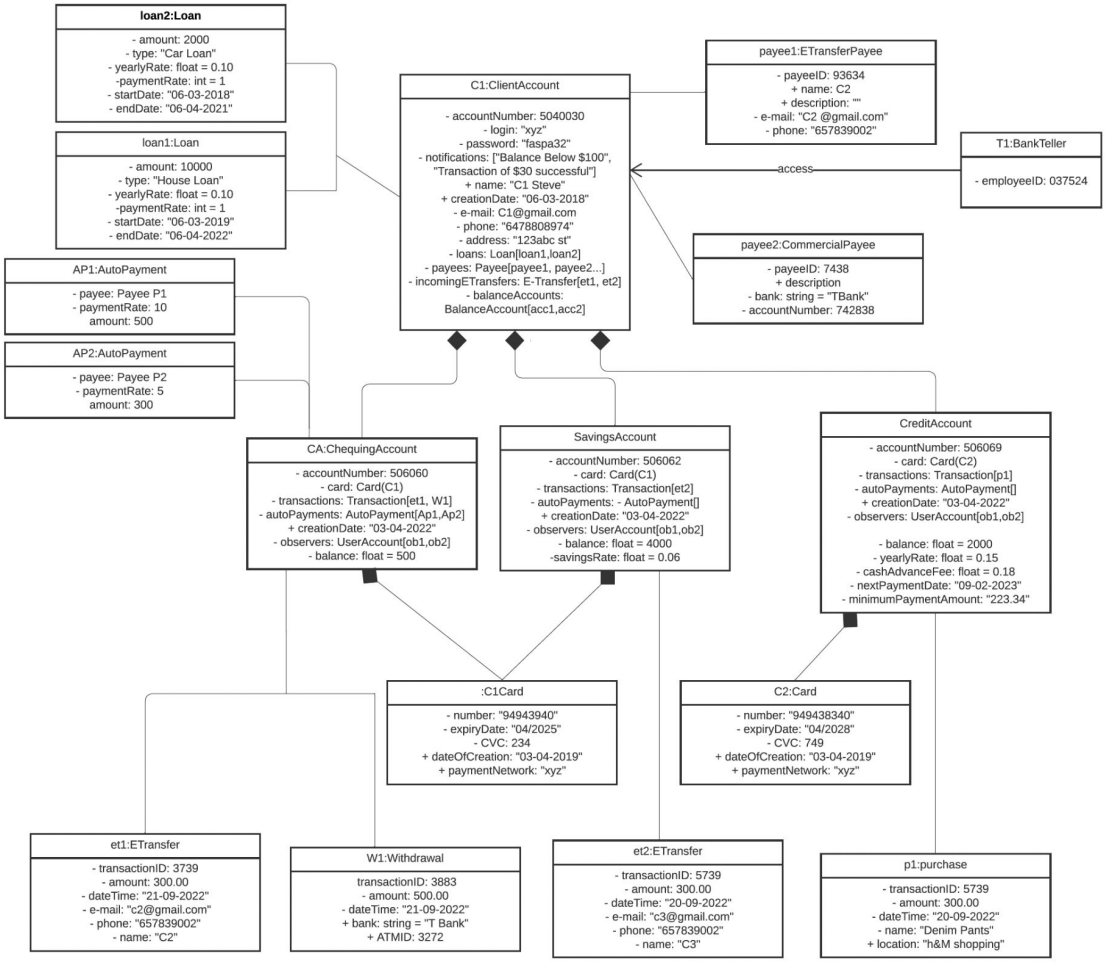
View Statement



View Transaction History

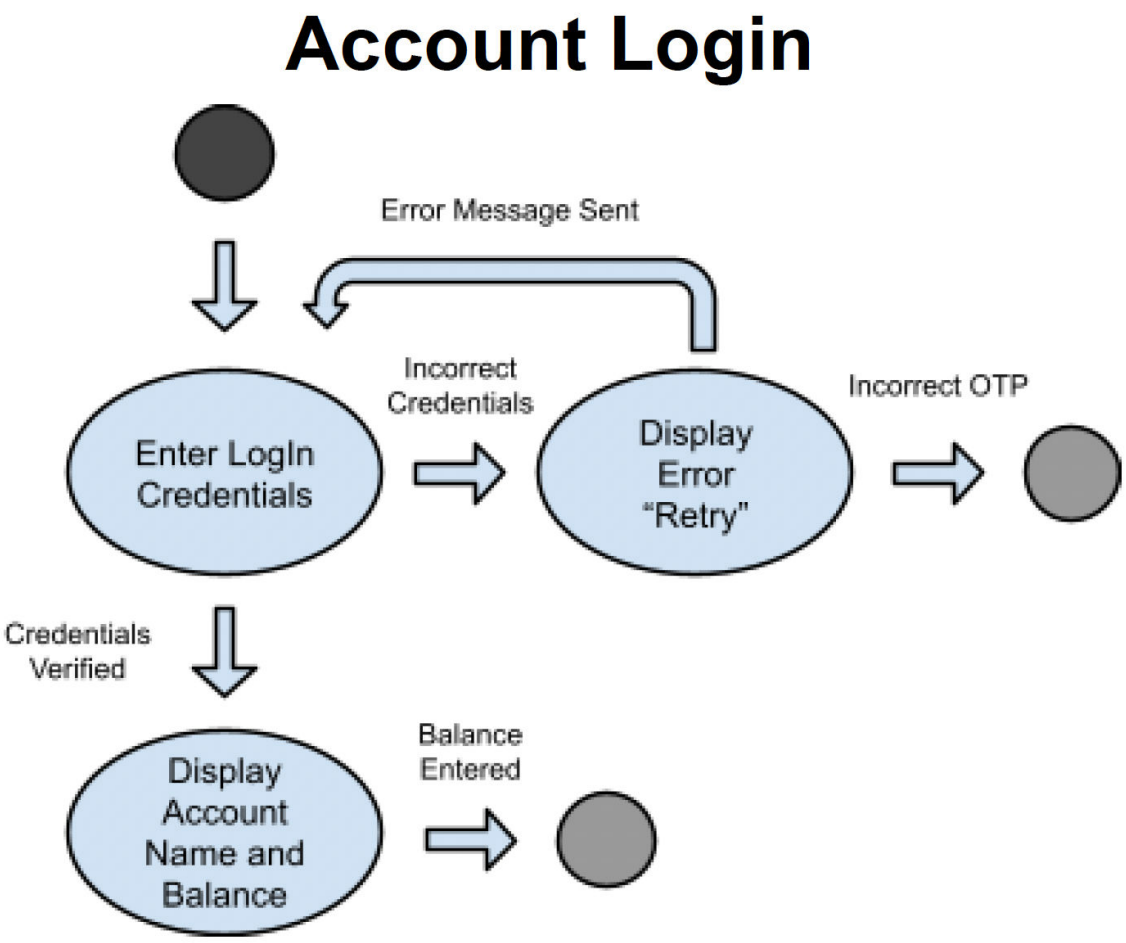


Object Design Diagram:

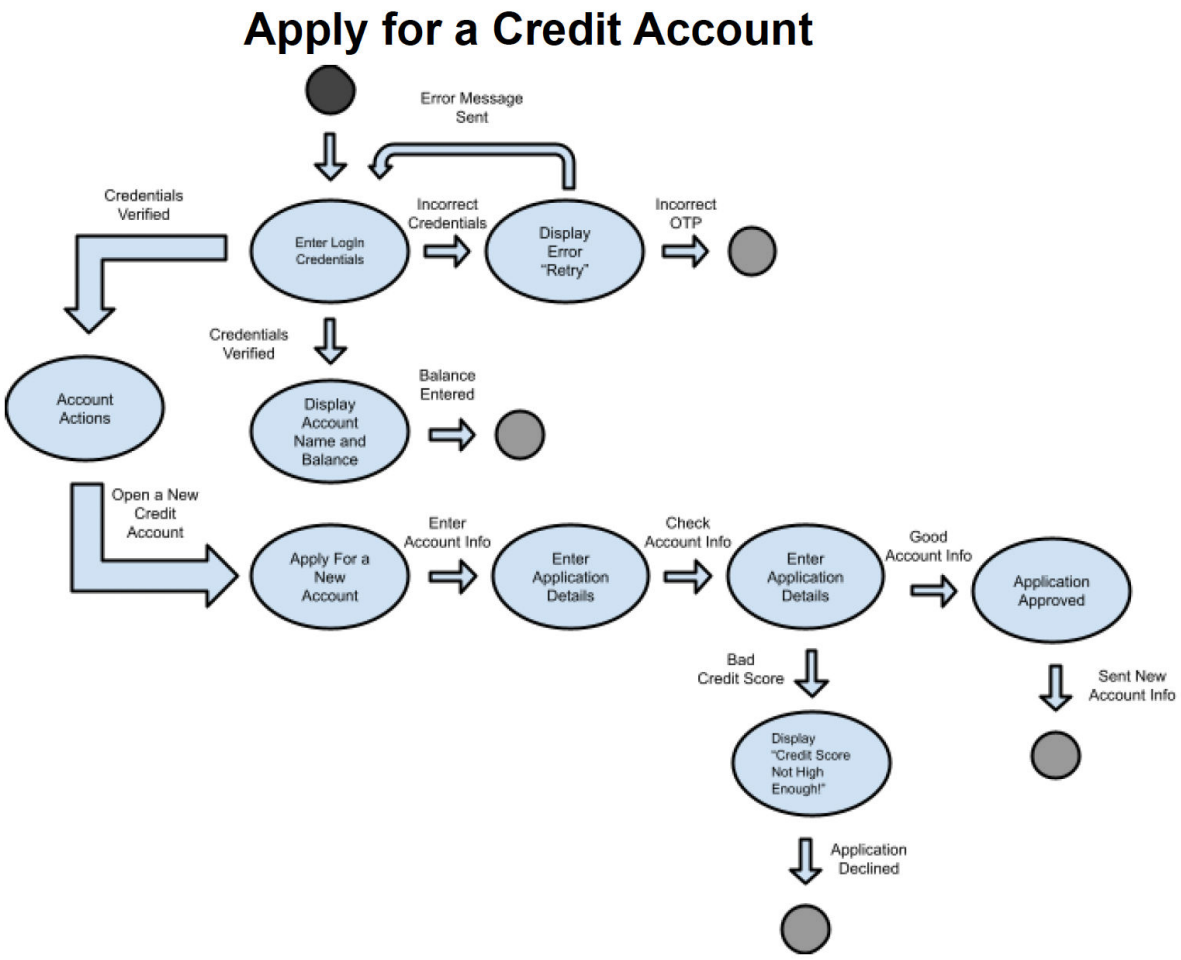


State Diagrams:

Account Login



Apply for a Credit Account



System Implementation Phase

For the implementation of the system classes we used Python. Python was a straight choice for us, as it supports Object Oriented Programming and every team member is knowledgeable in it.

Example of our class code:

```python
class BalanceAccount:
    def __init__(self, master: ClientAccount.ClientAccount, balance: float, paymentNetwork: str="Visa"):
        self.master = master
        self.accountNumber = IDGenerator.IDGenerator.generateBalanceAccountID()
        self.balance = balance
        self.card = Card.Card(paymentNetwork=paymentNetwork)
        self.transactions = []
        self.autoPayments = []
        self.observers = [master] #add the client account of this balance account to observe by default
        self.notifOnAmount = 500 #notify the subscribers on this amount (the observer pattern)
        self.creationDate = datetime.date.today()

    def addObserver(self, observer: UserAccount.UserAccount):
        import UserAccount
        if isinstance(observer, UserAccount.UserAccount):
            self.observers.append(observer)

    def removeObserver(self, observer):
        if observer in self.observers:
            self.observers.remove(observer)
```

For the user interface, we used Flask together with HTML, CSS and JavaScript. Flask is a modern web framework written in Python. It lets us pass information from the program to the HTML templates, and from the input forms on the webpage back to Python, and that is the way we connect our UI with the system.

The way Flask works:

```python
@app.route('/client')
def clientInterface(clientObj):
    name = clientObj.username
    checkingsBalance = str(clientObj.chequingAccount.balance)
    checkingsAccno = str(clientObj.chequingAccount.accountNumber)

    return render_template('clientInt.html', name=name, cbalance = checkingsBalance, caccno = checkingsAccno)
```

```html
<div class="credit">
    <h2><a href="">Credit Account</a></h2>
    <div class="ba">
        {{crbalance}}
        {{craccno}}
    </div>
</div>
```

System Testing Phase

This section describes every test case for every class that we have. The big titles indicate the class name being tested, and the smaller ones indicate the test method name.

ClientAccount

test_changeEmail

This test case, Test_changeEmail, is for the changeEmail method of the ClientAccount class.

Check to verify if emails can be altered properly:

The email address "tahshin89@gmail.com" should be changed.

Output: The method changes the email address associated with the client account to "tahshin89@gmail.com" and outputs True.

Verify that email cannot be converted to an empty string:

Type a blank string "" to get False as the method's output.

test_changePhone

The account class's changePhone() method is tested by this function. This techn ique's goal is to update the phone number linked to the account.

Base case :
When a legitimate phone number is input, the base case determines if the proce dure can successfully alter the phone number linked to the account.
The method is first used with the argument "6471239999", and the returned res ult is then saved in the result variable. In order to confirm that the outcome of t he method call is True, an assertion is then made using the assertTrue() method. The following action verifies that the account's phone number has been correctl

y updated. The newPhone variable is given the account object's phone_no attri bute. The assertEqual() method is used to make a claim about whether the new Phone value corresponds to the given phone number, "6471239999".

Edge case:
The edge case determines whether the method may accept an invalid phone nu mber as input, more precisely None.
Initialization begins with calling the method with the argument None, and the r esulting result is then saved in the result variable. In order to confirm that the o utcome of the method call is False, an assertion is then made using the assertFal se() method.

test_changeAddress

Functionality tested: Changing the address of an account

When the changeAddress() method is invoked with a new address as input, this test case determines whether the account's address has been appropriately updated. Additionally, it examines if the function returns True following an address update. Additionally, it checks an edge case when the function is given an empty string as an input to determine if it returns False.

Base case:

When a valid address is input, the base case evaluates if the method can successfully change the account's associated address.

First, the method is invoked with the '"60 Shuter St"' parameter, and the 'result' variable is used to record the method's return value. The 'assertTrue()' method is then used to make an assertion to confirm that the result of the method call is 'True'.

The next action verifies that the account's linked address has been correctly updated. The 'newAddr' variable is given the 'address' attribute of the 'account'

object. The 'assertEqual()' method is used to make a claim about whether the specified address **"60 Shuter St""" and the 'newAddr' value match.

Edge case:

The edge case examines how well the technique can handle an invalid address as an input, more precisely an empty string.

Prior to storing the returned result in the result variable, the procedure is first invoked with the input "". In order to confirm that the outcome of the method call is False, an assertion is then made using the assertFalse() method.

test_applyLoan

The applyLoan method of the ClientAccount class is tested using the test_applyLoan function. This method is in charge of making a new loan object and adding it to the client's account.

Base case:

"Car Loan" is the loan type and a loan amount of 1000 are specified.

The present date and a future date are used as the start and end dates, respectively.

The required arguments are passed when the applyLoan method is called.

To confirm that a new loan object has been created and added to the account, the total number of loans in the client's account are checked.

The newly formed loan object's amount, type, start date, and end date are examined to make sure they correspond to the provided values.

Edge case:

"Car Loan" is the loan type and "0.01" is the loan amount.

The present date and the following day are set as the start and end dates, respectively.

The required arguments are passed when the applyLoan method is called.

To confirm that a new loan object has been created and added to the account, the total number of loans in the client's account are checked.

The newly formed loan object's amount, type, start date, and end date are examined to make sure they correspond to the provided values.

test_openSavingsAccount

Checks that the Account class's openSavingsAccount method operates as intended. By using this method, a fresh savings account should be created and added to the list of savings accounts connected to the account.

Base case:

The list of savings accounts' starting length is kept in a variable. The length of the list of savings accounts is then checked to see if it has grown by two before calling the openSavingsAccount function twice. Both calls to openSavingsAccount are guaranteed to return True by the test.

Edge case:

The list of savings accounts' starting length is kept in a variable. The openSavingsAccount method is used five times, and each time, it is verified that the number of savings accounts has grown by five..

test_openCreditAccount

The procedure opens a new credit account with a predetermined initial balance for the account holder.

Process:

A fresh credit account is opened with a 1,000 (base case) opening balance.

The approach gives True results.

An update is made to the account's list of credit accounts.

The new credit account's balance is confirmed.

A new credit account is attempted to be opened with a 0 initial amount (edge case).

The technique gives a False result.The list of credit accounts on the account stays the same.

Base case:

the method must return True, adding the new credit account with the right balance to the account's list of credit accounts.

Edge case:

The method must return False, and the list of credit accounts for the account must not change.

test_addPayee

The addPayee() method of the account class is examined by this function. A payee (a person or a company) is added to the list of payees connected to the account using the addPayee() method.

Base case:

The payee1 variable is initially assigned a Payee object with the name "Slava" and an empty string for the account number. The assertIn() method is used to make an assertion that payee1 has been added successfully to the list of payees

connected to the account when the addPayee() method is used with payee1 as a parameter.

Edge case:

The names "Fardin" and "Whatever" are used to generate a tuple payee2, which contains the payment information. The outcome of the addPayee() function call is saved in the result variable when payee2 is sent as an argument. The assertFalse() function is used to make an assertion that the method call's outcome is False.

test_acceptEtransfer

Tests the Account class's acceptEtransfer function.

Produces a sample Etransfer object and adds it to the list of incomingEtransfers for the Account.

Uses the example Etransfer object as input when using the acceptEtransfer method of the Account class.

Makes the claim that the method returns True and deletes the Etransfer object from the list of incomingEtransfers.

test_requestMoney

The requestMoney() and fulfillRequest() methods of the account class are tested by this function. The fulfillRequest() function carries out the request by sending the desired sum from the second account to the first account.

The requestMoney() method makes a request for money to another account.

First, a second ClientAccount object is created and filled out with information for the second account.

The second account is then added to the AccountInterface database using the main.AccountInterface.clientAcc.append(account2).

In the most extreme scenario, the requestMoney() function is used to send a request for $500 to account2 from the self.account, and the resulting request object is sent to request1. The assertIn() function is used to make a claim and confirm that'request1' is correct.

test_makeLoanPayment

This test case examines the operation of the ClientAccount class's makeLoanPayment method. It determines if a loan payment from the chequing account is possible, and if so, updates the amount in the account as necessary.

-Setup the test data first.

-Use the parameters listed to call the makeLoanPayment method.

- Verify whether the method call's outcome is True.

-Verify that the chequing account balance has been adjusted to $500.

Test results:

The amount in the checking account is $1,000. "Education Loan" is the loan kind, with a 500 dollar loan amount, a start date of today, and an end date of April 3, 2024.

The loan, the chequing account, and a payment of 500 are passed to the makeLoanPayment method.

Expected result:

The outcome of the call to the makeLoanPayment method must be True.

The chequing account balance must be changed to 500.

test_cancel_balance_account

The cancelBalanceAccount() method of the ClientAccount class is being tested as part of this unit test.

Base case:

The client has a credit account in this test case, and the method is called with the credit account as a parameter. The test determines whether the credit account is effectively cancelled and returns True.

Edge case:

A BalanceAccount class object is passed as an argument to the method in this test case, which is an invalid parameter. The method's return value is verified by the test.

Test result:

The method must return True in the base case and False in the edge case for the test to succeed.

AutoPayment

testChangeAmount

The changeAmount method in the AutoPayment class is tested using this test case to determine whether it is functioning properly. The function is used to modify the payment amount.

Base case: The amount is changed using a valid integer value of 500.

Edge case: The amount is changed by using the integer value 0.

Test Procedure:

A new AutoPayment object is made with a payee and payment rate that are already set.

To modify the payment amount, the changeAmount function is used with a valid integer value of 500.

Using the assertEqual method, the current payment amount is contrasted with the anticipated value of 500.

To modify the payment amount, the changeAmount method is once more invoked with the integer value 0.

To confirm that the payment amount stayed the same, the new payment amount is compared to the first payment amount using the assertNotEqual method.

Expected Outcome:

The first assertEqual function ought to succeed, signifying that the payment sum was updated to the anticipated value of 500.

The second assertNotEqual function ought to succeed, proving that the payment sum remained constant even though an improper value of 0 was entered.

testChangeRate

The changeRate method in the AutoPayment class is tested using this test case to determine whether it is functioning properly. The function is used to modify the auto-payment's payment rate.

Base case:

The payment rate is changed using a legitimate payment rate of 50 days.

Edge case:

The payment rate is changed using an incorrect payment rate of 0 days.

Test Procedure:

A fresh AutoPayment object with a predetermined payee and payment sum is generated.

To modify the payment rate, the changeRate function is used with a 50-day valid payment rate.

Using the assertEqual method, the current payment rate is contrasted with the anticipated value of 50 days.

To modify the payment rate, the changeRate function is called once again with an incorrect payment rate of 0 days.

To confirm that the payment rate stayed the same, the new payment rate is compared to the first payment rate using the assertNotEqual method.

Expected Outcome:

The first assertEqual function ought to succeed, signifying that the payment rate was modified to take the anticipated value of 50 days.

The second assertNotEqual function ought to succeed, proving that the payment rate remained constant even when a false payment rate of 0 days was applied.

Card

testLock

The lockCard and unlockCard functionalities of the Card class are verified using this test case. The card object can be locked or unlocked using these operations, accordingly.

Test Technique:

The setUp function is used to generate a fresh Card object.

To lock the card object, the lockCard method is used.

To ensure that the locked property of the card object is set to True, use the assertTrue function.

To unlock the card object, the unlockCard method is used.

The locked attribute of the card object is checked to see if it is set to False using the assertFalse function.

Expected Outcome:

The lockCard method should successfully set the locked property of the card object to True, as indicated by the first assertTrue function passing.

The unlockCard method should successfully set the locked property of the card object to False, as seen by the second assertFalse function passing.

Loan

testPay

The pay function in the Loan class is tested using this test case to determine whether it is functioning properly. The feature is utilised to make loan repayment instalments.

Test Results:

The Loan object is created with a loan amount of $10,000, a car loan type, and dates for the beginning and conclusion of the loan.

Base-case 1: A legitimate 2000 loan payment is made.

Base-case 2 : The loan receives a proper 8,000 payment.

Edge case: A loan payment of $11,000 is made in error.

Test Procedure:

Using the setUp function and the supplied data, a new Loan object is created.

With a payment value of 11,000, which is more than the loan balance, the pay function is invoked.

To determine if the return result of the pay function matches the anticipated value of -1, which denotes that the payment amount is illegal, the assertEqual function is utilised.

A valid payment value of 2000 is sent to the pay function, which is invoked.

The assertEqual method is used to confirm that the final payment amount, 8000, is correct.

With 8000 as the payment value—another legal payment amount—the pay function is once more invoked.

The assertEqual function is used to verify that the remaining payment amount after making the payment is 0.

Expected Outcome:

Indicating that the pay function returns -1 for an incorrect payment amount, the initial assertEqual function ought to succeed.

The second assertEqual function should succeed, proving that 8000 is the balance due following a successful payment of 2000.

The third assertEqual function should succeed, proving that there is nothing left to pay after a genuine payment of 8000 has been made.

TellerAccount

testRegisterClient

The registerClient method in the TellerAccount class is tested using this test case to determine whether it is functioning properly. The feature is used to add new customers to the bank's database.

Test Results:

The teller username "iamateller", teller name "Nikita", and teller password "teller123" are initialised in the TellerAccount object.

Base case:

The bank registers a new customer named "john98" with a special username.

Edge case:

A new client registration attempt is performed using the same login as the outgoing client.

Test Technique:

The setUp function is used to generate a new TellerAccount object with the supplied information.

The information for a new client with the distinctive username "john98" is passed to the registerClient method.

To confirm that the new client is present in the clientAcc list of registered clients, the assertIn method is utilised.

With the same username "john98" as the last client, the registerClient method is called once more.

To confirm that the return value of the registerClient method is None, indicating that the registration was unsuccessful, use the assertIsNone function.

Expected Outcome:

The initial assertIn function ought to succeed, signifying that the registration of the new client was successful.

The second assertIsNone function should succeed, indicating that a username conflict prevented the registration from being successful.

testFindClient

The findClient method in the TellerAccount class is tested using this test case to determine whether it is functioning properly. A registered client may be located using the function and their account number.

Test Results

The teller username "iamateller", teller name "Nikita", and teller password "teller123" are initialised in the TellerAccount object.

Base case:

A new customer registers with the bank using the login "john98," and the bank uses the customer's account number to identify them.

Edge case:

An effort is made to track down a customer who has an incorrect account number.

Test Technique:

The setUp function is used to generate a new TellerAccount object with the supplied information.

The registerClient method is used to register a new customer with the bank.

The account number of the registered client is used to invoke the findClient method.

To ensure that the delivered client object and the registered client object are same, use the assertEqual method.

With an incorrect account number, the findClient method is performed once more.

To confirm that the returned result is None, indicating that the client was not located, the assertIsNone method is used.

Expected Outcome:

The first assertEqual function should succeed, proving that the findClient method uses the account number of the client to return the right client object.

The second assertIsNone function should succeed, proving that a bad account number causes the findClient method to return None.

UserAccount

testUpdate

The testUpdate() method checks whether the update() method adds a new notification to the notifications list of the UserAccount object.

testLogin

The testLogin() method checks the login() method's functionality by testing four different scenarios: wrong login and password, wrong login, wrong password, and successful login. The test asserts that the login() method returns False for the first three scenarios and True for the successful login scenario.

testChangePassword

The testChangePassword() method tests whether the changePassword() method changes the password of the UserAccount object. It checks for three scenarios: wrong password, wrong username, and successful password change. The test asserts that the password has not been changed for the first two scenarios and that the password has been successfully changed for the third scenario.

BalanceAccount

setUp

Base case: Creates a new instance of a client account and teller account

Edge case: The account setup will fail if any of the cases are invalid or incorrect, missing an attribute or instances are not set properly

testAddObserver

Base case: Assertion checks if the observer is added to the chequing account

Edge case: Make sure if the string has correctly input

testRemoveObserver

Base case: Removes a client account from the list of chequing account

Edge case: Account only can be removed if it is on the list of clients

testNotifyObservers

Base case: Runs the notification functionality for the clients

Edge case: Clients are required to be registered to the chequing account beforehand

testSetNotifAmount

Base case: Sets the amount for notifying the clients for the chequing account

Edge case: Not going to work if the account balance goes below negative 10

testDeposit

Base case: It runs the functionality of depositing to the chequing account

Edge case: Check whether the balance of the chequing account is still old balance

testWithdraw

Base case: It stores the balance of the chequing account in the variable old balance

Edge case: Withdrawing a higher value then the balance will fail the withdraw transaction

testNewTransaction

Base case: It updates the chequing account depending on the amount of wire transfer

Edge case:

testTransferBetweenAccount

Base case: It transfers between a chequing and a saving account of a single client

Edge case: If the balance is lower than the transfer amount then it will not complete the transaction

testSetupAutoPayment

Base case: It requires choosing a payee and the payment will be generated

Edge case: The payee or account name requires to be valid and the amount has to be more than the balance .

testSendWireTransfer

Base case: Transferring the amount from one chequing account to another client's chequing account

Edge case: Transfer will be unsuccessful if the amount is higher than the balance

testSendEtransfer

Base case: Processes an electronic transfer from one registered chequing account

Edge case: The transfer amount needs to be greater than the available balance and payee information has to be completed correctly.

testMakePurchase

Base case: Can process a purchase according to the spending from the chequing account and will be added to the transaction list

Edge case: Amount of purchase has to be sufficient to cover the account balance and the inputs of the vendor's name and amount has to be accurate

testWithdrawalATM

Base case: Clients can use their accounts to withdraw portion of their balance from the chequing account

Edge case: If the account balance is zero and the amount are invalid then the withdraw will fail