**MINISTRY OF EDUCATION AND SCIENCE OF THE KYRGYZ REPUBLIC**

**KYRGYZ STATE TECHNICAL UNIVERSITY named after I. Razzakov**

**Institute of Information Technologies**

**Department of Computer Systems Software**

**Course work**

Discipline: Object-Oriented Programming

on the topic:

«User (Bank) Management System»

Done by:Alshalabi Rami – SE(engl) 1-21

Checked by: Musabaev E. B.

Bishkek 2024

Content:

[1. Analysis and formation of requirements 3](#_Toc167136567)

[1.1. User Requirements 3](#_Toc167136568)

[1.1.1. Admin 3](#_Toc167136569)

[1.2. Functional requirements 3](#_Toc167136570)

[1.3. Non-functional requirements 5](#_Toc167136571)

[2.Use Case Diagram 7](#_Toc167136572)

[3. Class Diagram 8](#_Toc167136573)

[3.1. Description of classes and their purposes 8](#_Toc167136574)

[4. The process of running the program 16](#_Toc167136575)

[Prerequisites 16](#_Toc167136576)

[Installation 16](#_Toc167136577)

[5. Conclusion 17](#_Toc167136578)

[6. Screenshots of the program 17](#_Toc167136579)

# 1. Analysis and formation of requirements

## 1.1. User Requirements

Пользователи система: Administrator, or regular user

### Admin

The Administrator should have the capability to perform the following functions:

1. Adding, deleting, viewing, updating, and searching for client records.
2. Adding, deleting, viewing, updating, and searching for user accounts.
3. Deposit, Withdraw, Viewing balances - logs, and Transferring money.
4. Viewing, Updating, and Calculating Currencies in the currency exchange screen.

For regular users, specific permissions(within the mentioned before) will be assigned based on their roles and responsibilities within the banking system by the admin or the other trusted users.

## Functional requirements

1. **User Authentication:**
   * The system shall require users to authenticate themselves using a username and password.
   * User credentials shall be stored securely and verified during the login process.
   * Upon successful authentication, users shall be granted access to the system functionalities based on their assigned roles and permissions.
2. **Client Management:**
   * The system shall allow administrators to add, delete, view, update, and search for client records.
   * Client records shall include essential information such as account number, PIN code, name, phone number, and account balance.
   * Administrators shall have the ability to perform CRUD operations on client records efficiently.
3. **User Management:**
   * The system shall provide administrators with the capability to add, delete, view, update, and search for user accounts.
   * User accounts shall contain details such as first name, last name, email, phone number, username, password, and permissions.
   * Administrators shall be able to assign specific roles and permissions to users based on their responsibilities within the system.
4. **Transaction Management:**
   * Users shall be able to perform various transactional activities, including deposits, withdrawals, and fund transfers.
   * The system shall maintain transaction logs to record details such as transaction ID, timestamp, type, amount, source account, destination account, and status.
   * Users shall have the ability to view transaction history and track financial activities accurately.
5. **Currency Exchange:**
   * The system shall support currency exchange operations, allowing users to convert between different currencies based on predefined exchange rates.
   * Exchange rates shall be regularly updated to reflect current market values and ensure accurate currency conversions.
   * Users shall be provided with the flexibility to perform currency exchange transactions seamlessly.
6. **Access Control and Permissions:**
   * The system shall implement role-based access control mechanisms to regulate user access and permissions.
   * Administrators shall have full control over system functionalities and user permissions, with the ability to modify roles and permissions as needed.
   * Regular users shall be granted access to specific functionalities based on their assigned roles and permissions, ensuring data security and integrity.
7. **Error Handling and Validation:**
   * The system shall incorporate robust error handling mechanisms to handle exceptions and unexpected behaviors gracefully.
   * Input validation shall be enforced to ensure the integrity and consistency of data entered into the system.
   * Error messages shall be displayed to users in a clear and concise manner, providing guidance on resolving issues encountered during system interactions.
8. **Data Storage and Management:**
   * Client, user, transaction, and other relevant data shall be stored securely in a structured format.
   * The system shall utilize appropriate data storage mechanisms, such as files or databases, to manage and retrieve information efficiently.
   * Data integrity shall be maintained through regular backups and measures to prevent data loss or corruption.
9. **System Navigation and Usability:**
   * The system shall provide users with an intuitive and user-friendly interface for seamless navigation and interaction.
   * Menu options and functionalities shall be clearly labeled and organized to facilitate ease of use.
   * Users shall be guided through the system workflows with prompts and instructions to perform desired actions effectively.
10. **Logging and Auditing:**
    * The system shall maintain detailed logs of user activities, including login attempts, transactional operations, and administrative actions.
    * Audit logs shall be accessible to authorized users for monitoring system usage and detecting any suspicious or unauthorized activities.
    * Logs shall be timestamped and stored securely to ensure accountability and traceability of system events.

## 1.3. Non-functional requirements

1. **Security:**
   * The system shall implement robust encryption techniques to protect sensitive data stored in files, ensuring user credentials, transaction details, and personal information are securely encrypted.
   * Access control mechanisms shall be enforced within the application to restrict unauthorized access to system functionalities and data.
   * Measures shall be taken to prevent buffer overflows, SQL injection, and other common security vulnerabilities in C++ code.
2. **Performance:**
   * The system shall be optimized for efficient memory usage and processing speed to provide a responsive user experience within the console environment.
   * Response times for user inputs and system operations shall be minimized to ensure smooth interaction and timely execution of banking tasks.
   * Code optimization techniques shall be employed to enhance performance and minimize computational overhead in critical system functions.
3. **Reliability:**
   * The application shall demonstrate high reliability and stability, minimizing the occurrence of crashes, errors, or unexpected behavior during normal operation.
   * Error handling mechanisms shall be implemented to gracefully handle exceptions and recover from unexpected failures, ensuring uninterrupted service for users.
   * Automated testing procedures shall be established to validate system reliability and identify potential issues before deployment.
4. **Scalability:**
   * The system architecture shall be designed to accommodate potential scalability requirements, allowing for the addition of new features or expansion of user base in the future.
   * Scalability testing shall be conducted to assess the system's ability to handle increased workload and user concurrency without sacrificing performance or stability.
   * Code modularization and abstraction shall be employed to facilitate future enhancements and modifications to the system's functionality.
5. **Usability:**
   * The console interface shall be designed with simplicity and clarity in mind, providing intuitive navigation and clear prompts for user input.
   * User input validation shall be implemented to guide users through the application flow and prevent input errors or invalid commands.
   * Help documentation or inline instructions shall be provided within the application to assist users in understanding available functionalities and using the system effectively.
6. **Maintainability:**
   * The codebase shall be well-structured and documented to facilitate ease of maintenance and future development by other programmers.
   * Version control systems such as Git shall be utilized to track changes and manage code revisions effectively, enabling collaboration among developers.
   * Code refactoring and cleanup procedures shall be performed regularly to improve readability, reduce technical debt, and enhance maintainability.
7. **Interoperability:**
   * The application shall be compatible with standard input/output streams and system libraries available in the C++ programming language, ensuring interoperability with various platforms and environments.
   * External libraries or APIs may be integrated into the system as needed to extend functionality or facilitate interaction with external systems or services.
8. **Data Integrity and Consistency:**
   * Data validation and error-checking mechanisms shall be implemented to ensure the integrity and consistency of data stored and processed within the application.
   * File handling operations shall be performed securely to prevent data corruption or loss, with appropriate error handling and recovery mechanisms in place.
9. **User Privacy:**
   * User privacy shall be respected and protected through adherence to privacy policies and data protection regulations governing the collection, storage, and processing of user information.
   * Personal data shall be handled with care and stored securely, with access restricted to authorized personnel only.
   * Consent mechanisms may be implemented to allow users to control the use of their personal data within the application, in compliance with privacy laws and regulations.

# 2.Use Case Diagram

The use case diagram is shown in *Figure 1*

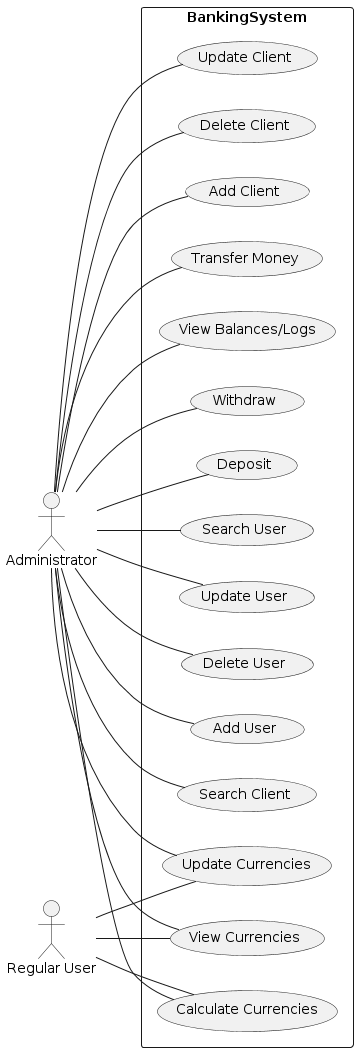


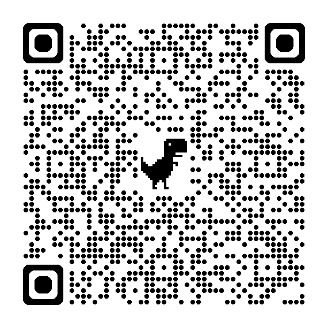
Figure 1–Use Case Diagram

# 3. Class Diagram

 The class diagram is shown in Figure 2*.*

Рисунок 2 Диаграмма классов

System has 8 (main) classes: clsInputValidate, clsString, clsBankClient, clsPerson, clsCurrency, clsDate, util, clsUser

full resolution photo:

## 3.1. Description of classes and their purposes

1. **clsInputValidate**:
   * Functions:
     1. **isInRange(int value, int lowerBound, int upperBound)**: Checks if a given integer value is within the specified range.
     2. **isInRange(float value, float lowerBound, float upperBound)**: Checks if a given float value is within the specified range.
     3. **isInRange(long value, long lowerBound, long upperBound)**: Checks if a given long value is within the specified range.
     4. **isInRange(double value, double lowerBound, double upperBound)**: Checks if a given double value is within the specified range.
     5. **isInRange(char value, char lowerBound, char upperBound, bool matchCase = true)**: Checks if a given character value is within the specified range, optionally ignoring case.
     6. **readPositiveNumber(const std::string& message = "")**: Prompts the user to enter a positive integer and returns it.
     7. **get\_int(const std::string& message = "")**: Prompts the user to enter an integer and returns it.
     8. **get\_float(std::string message = "")**: Prompts the user to enter a float and returns it.
     9. **get\_double(std::string message = "")**: Prompts the user to enter a double and returns it.
     10. **get\_Long(const std::string& message = "")**: Prompts the user to enter a long integer and returns it.
     11. **getLongLongInt(const std::string& message = "")**: Prompts the user to enter a long long integer and returns it.
     12. **get\_string(std::string message = "")**: Prompts the user to enter a string and returns it.
     13. **get\_char(std::string message = "")**: Prompts the user to enter a single character and returns it.
     14. **getInputInRange(const int& from, const int& to, const std::string& message = "")**: Prompts the user to enter an integer within a specified range and returns it.
     15. **get\_Positive\_double(const std::string& message)**: Prompts the user to enter a positive double number and returns it.
     16. **readPinCode(const std::string& message = "Please enter Pin Code: ")**: Prompts the user to enter a PIN code and returns it as a string.
     17. **readPhoneNumber(const std::string& message = "Please enter phone number: ")**: Prompts the user to enter a phone number and returns it as a string.
     18. **readPassword(const std::string& message = "Please enter a password (at least 4 characters, no spaces): ")**: Prompts the user to enter a password and returns it as a string.
     19. **trimLeft(const std::string& str)**: Trims leading spaces from a string.
     20. **trimRight(const std::string& str)**: Trims trailing spaces from a string.
     21. **trim(const std::string& str)**: Trims leading and trailing spaces from a string.
     22. **lowerAllStringLetters(const std::string& originalStr)**: Converts all letters in a string to lowercase.
     23. **removeSpaces(const std::string& originalStr)**: Removes all spaces from a string.

***Purpose: Provides methods for input validation and string manipulation.***

1. **clsString**:
   * Functions:
     + Constructors: Initialize the class instance with a string value.
     + **SetValue**, **GetValue**: Set and get the string value.
     + **Length**, **CountWords**, **UpperFirstLetterOfEachWord**, **LowerFirstLetterOfEachWord**, **UpperAllString**, **LowerAllString**, **InvertAllLettersCase**, **CountLetters**, **CountCapitalLetters**, **CountSmallLetters**, **CountSpecificLetter**, **CountVowels**: String manipulation and counting functions.
     + **Split**: Splits a string into substrings based on a delimiter.
     + **TrimLeft**, **TrimRight**, **Trim**: Trim whitespace from the beginning, end, or both ends of a string.
     + **JoinString**, **ReverseWordsInString**, **ReplaceWord**, **RemovePunctuations**: More advanced string manipulation functions.
   * Purpose: Provides a range of string manipulation and analysis functions.
2. **clsBankClient**:
   * Class Structure:
     + Create New Client: Assign a new generated acc number then append other data entered by the user.
     + Inherits from **clsPerson**, indicating a relationship between a bank client and a person.
     + Contains private member variables \_Mode, \_**AccountNumber**, \_**PinCode**, \_**AccountBalance**, and \_**MarkedForDelete** to manage the client's data.
     + Defines an enumeration **enMode** to represent different modes of the client object (Empty, Update, AddNew).
     + Declares a nested struct **stTrnsferLogRecord** to represent transfer log records.
   * Constructor:
     + Initializes the **clsBankClient** object with provided parameters such as mode, name, email, phone, account number, pin code, and account balance.
   * Private Methods:
     + \_**GetEmptyClientObject**: Returns an empty clsBankClient object.
     + \_**ConvertTransferLogLineToRecord**: Converts a line of transfer log text to a **stTrnsferLogRecord** struct.
     + \_**PrepareTransferLogRecord**: Prepares a transfer log record for logging transactions.
     + \_**ConvertLineToClientObj**: Converts a line of client data text to a clsBankClient object.
     + \_**ConvertClientObjToLine**: Converts a clsBankClient object to a line of client data text.
     + \_**LoadClientsDataFromFile**: Loads client data from a file.
     + \_**SaveClientsDataToFile**: Saves client data to a file.
     + \_**Update**: Updates client data.
     + \_**AddNew**: Adds new client data.
     + \_**AddDataLineToFile**: Adds a data line to a file.
     + \_**RegisterTransferLog**: Registers a transfer log.
   * Public Methods:
     + **isEmpty**: Checks if the client object is empty.
     + **accountNumber**: Gets the account number associated with the client.
     + **getPinCode**/**setPinCode**: Getter and setter methods for modifying and accessing the pin code.
     + **getAccountBalance**/**setAccountBalance**: Getter and setter methods for modifying and accessing the account balance.
     + **find**: Finds a client by account number and optional pin code.
     + **Save**: Saves the client data.
     + **isClientExist**: Checks if a client with a specific account number exists.
     + **createNewClient**: Creates a new client object with AddNew mode.
     + **Delete**: Deletes a client.
     + **getClientsList**: Gets a list of clients.
     + **getTotalBalances**: Gets the total balance of all clients.
     + **Deposit**: Deposits money into the client's account.
     + **Withdraw**: Withdraws money from the client's account.
     + **Transfer**: Transfers money between clients' accounts.
     + **GetTransfersLogList**: Gets a list of transfer log records.
   * File Handling:
     + Reads and writes client data and transfer log records to/from text files.
3. **clsPerson**:
   * Functions:
     + Constructor: Initialize the class instance with person details.
     + Getter and setter functions for person attributes.
   * Purpose: Represents a person entity with methods for managing personal information.
4. **clsCurrency**:
   * This class represents a currency entity, storing information such as country, currency code, currency name, and exchange rate.
   * Functions:
     + Constructors: Initializes the class instance with currency details.
     + **IsEmpty**: Checks if the currency object is empty.
     + Getter functions: Country, **CurrencyCode**, **CurrencyName**, Rate to retrieve currency attributes.
     + **UpdateRate**: Updates the exchange rate for a currency and saves it to a file.
     + **FindCurrencyByCodeOrCountry**: Finds a currency object by its code or country name.
     + **IsCurrencyExist**: Checks if a currency exists based on its code or country name.
     + **GetCurrenciesList**: Retrieves a list of currency objects from a file.
     + **ConvertTo**\_USD: Converts an amount from the currency to USD.
     + **ConvertTo\_otherCurrency**: Converts an amount from the currency to another currency.
   * File I/O functions: \_**LoadCurrencyDataFromFile**, \_**SaveCurrencyDataToFile**.
   * Private utility functions: \_**ConvertLineToCurrencyObject**, \_**ConvertCurrencyObjectToLine**, \_**UpdateDatabase**, \_**GetEmptyCurrencyObj**.
5. **clsDate**:

**Constructors**:

Default constructor initializes the date to the current system date.

Constructor with a string parameter allows initialization from a date string in the format "DD/MM/YYYY".

Constructors with separate day, month, and year parameters.

**Accessor and Mutator Methods**:

Getters and setters for day, month, and year properties.

Properties defined using **\_\_declspec(property)**.

**Date Validation**:

Methods to check the validity of a date.

Leap year detection.

**Date Arithmetic**:

Addition and subtraction of days, weeks, months, years, decades, and centuries.

Methods for incrementing and decrementing dates.

Calculation of the difference between two dates in days.

Handling end-of-week, end-of-month, and end-of-year scenarios.

**Date Comparison**:

Comparison of two dates, including before, after, and equality checks.

Enumeration for date comparison results.

**Date Formatting**:

Conversion of date objects to string representations.

Formatting options for printing month calendars.

**Utility Methods**:

Retrieval of the system date and time.

Calculation of age in days.

Methods for checking weekends, business days, etc.

Generation of month and year calendars.

* + Purpose: Represents a date entity with methods for date manipulation and validation.

1. **util**:
   * This class provides a collection of utility functions for various purposes.
   * Functions:
     + **isLeapYear**: Checks if a given year is a leap year.
     + **numberOfDaysInAYear**: Returns the number of days in a year.
     + **encryptText**, **decryptText**: Encrypts and decrypts text using a specified encryption key.
     + **convertNumberToText**: Converts a number to its textual representation.
     + **randomNumber**: Generates a random number within a specified range.
     + **getRandomCharacter**, **getRandomWord**: Generates random characters or words.
     + **generatePassword**, **generateKey**: Generates strong passwords or keys.
     + **isPalindromeArray**: Checks if an array or vector is a palindrome.
     + **swap**: Swaps the values of two variables.
     + **shuffleArray**: Shuffles the elements of an array or vector.
     + **fillArrayWithRandomNumbers**: Fills an array or vector with random numbers.
     + **GetSystemDateString**: Gets the current system date in string format.
   * Private functions for number-to-text conversion: **numberToText**, **toText**.

Начало формы

1. **clsUser**:
   1. **Class Structure**:
      1. Inherits from **clsPerson**, indicating a relationship between a user and a person.
      2. Contains private member variables **\_Mode**, **\_UserName**, **\_Password**, **\_Permissions**, and **\_MarkedForDelete** to manage user data.
      3. Defines an enumeration **enMode** to represent different modes of the user object (Empty, Update, AddNew).
      4. Declares a nested struct **stLoginRegisterRecord** to represent login/register records.
   2. **Constructor**:
      1. Initializes the **clsUser** object with provided parameters such as mode, name, email, phone, username, password, and permissions.
   3. **Private Methods**:
      1. **\_ConvertLoginRegisterLineToRecord**: Converts a line of login/register text to a **stLoginRegisterRecord** struct.
      2. **\_PrepareLogInRecord**: Prepares a login record for logging user login events.
      3. **\_GetEmptyUserObject**: Returns an empty **clsUser** object.
      4. **\_ConvertLineToUserObj**: Converts a line of user data text to a **clsUser** object.
      5. **\_ConvertUserObjToLine**: Converts a **clsUser** object to a line of user data text.
      6. **\_AddDataLineToFile**: Adds a data line to a file.
      7. **\_LoadUsersDataFromFile**: Loads user data from a file.
      8. **\_SaveUsersDataToFile**: Saves user data to a file.
      9. **\_Update**: Updates user data.
      10. **\_AddNew**: Adds new user data.
   4. **Public Methods**:
      1. **IsEmpty**: Checks if the user object is empty.
      2. **ISMarkedForDeleted**: Checks if the user is marked for deletion.
      3. **GetUserName/SetUserName**: Getter and setter methods for modifying and accessing the username.
      4. **GetPassword/SetPassword**: Getter and setter methods for modifying and accessing the password.
      5. **GetPermissions/SetPermissions**: Getter and setter methods for modifying and accessing the permissions.
      6. **Find**: Finds a user by username and optional password.
      7. **Save**: Saves the user data.
      8. **ISUserExist**: Checks if a user with a specific username exists.
      9. **CreateNewUser**: Creates a new user object with AddNew mode.
      10. **Delete**: Deletes a user.
      11. **GetUsersList**: Gets a list of users.
      12. **CheckAccessPermission**: Checks if the user has specific access permissions.
      13. **RegisterLogin**: Registers a login event for the user.
      14. **GetLoginRegisterList**: Gets a list of login/register records.
   5. **File Handling**:
      1. Reads and writes user data and login/register records to/from text files.

# 4. The process of running the program

### Prerequisites

* C++ Compiler (e.g., g++)
* Git (optional)
* Visual Studio (optional)

### Installation

1. Clone the repository:

git clone https://github.com/mattar740/bank-management-system.git

* Alternatively, you can download the project as a ZIP file and extract it.

1. Navigate to the project directory:

cd bank-management-system

1. Compile the source code using your C++ compiler:

g++ main.cpp -o BankManagementSystem

or use the CMake configs that i already added in the root folder, using visual studio and compile the program using it!

Move the .txt files to the

./BankManagementSystem

Where the .exe file is located, and the run the .exe file!

1. Log in using the provided credentials:

Username: admin Password: 1234

1. Follow the on-screen instructions to navigate through the various menu options and perform banking operations.

# 5. Conclusion

In conclusion, the development of the User (bank) management system utilized object-oriented programming (OOP) principles to create a well-structured and modular application. Classes were used to encapsulate data and behavior, promoting code reusability and maintainability. The system also employed screen classes to manage the user interface effectively, enhancing the user experience.

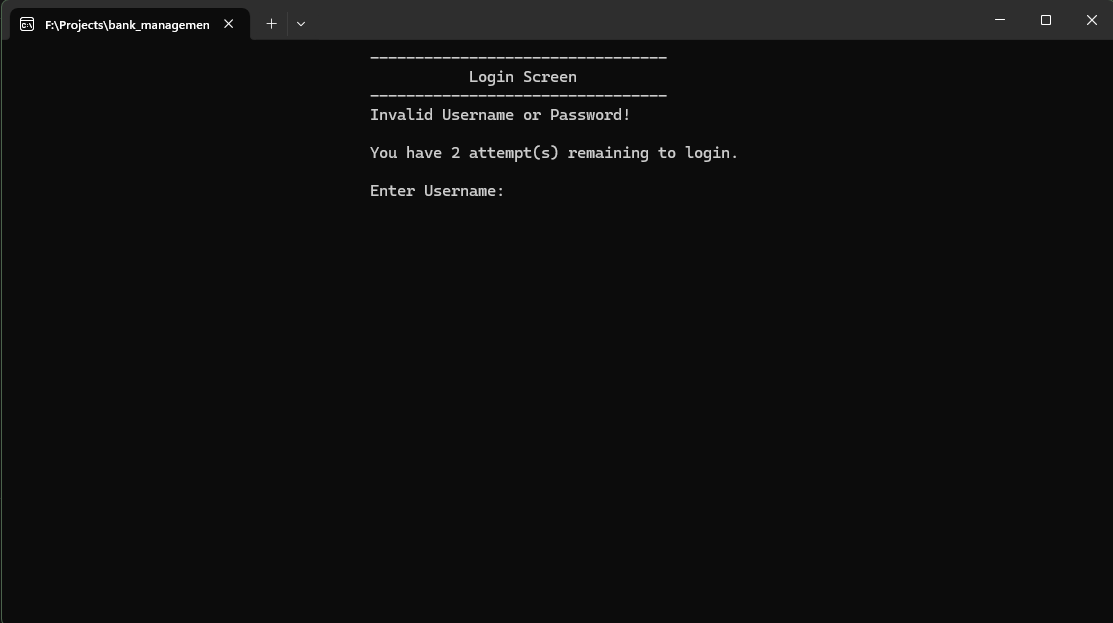
Furthermore, text files were utilized as a simple and efficient means of data storage, allowing for easy retrieval and manipulation of information. This approach ensured data persistence while keeping the implementation straightforward and accessible.

Overall, the project showcases the effective use of OOP concepts, modular design, and thoughtful consideration of user interface and data management. The GitHub repository for the bank management system can be found at: <https://github.com/Rami-0/bank_management_system>.

# 6. Screenshots of the program



Pic 1 Login screen



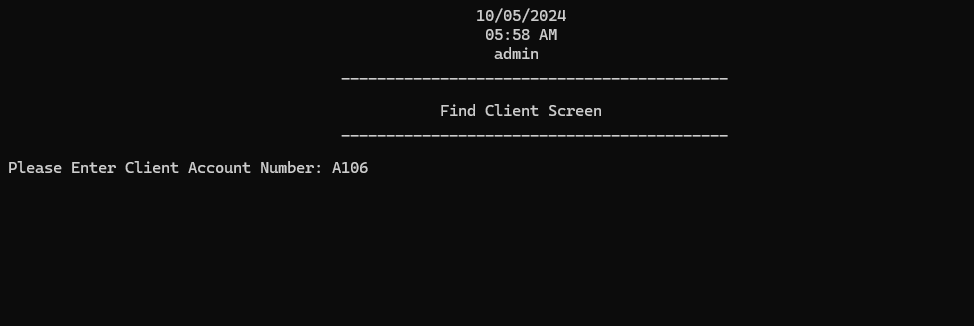
Pic 2 Login screen (error)



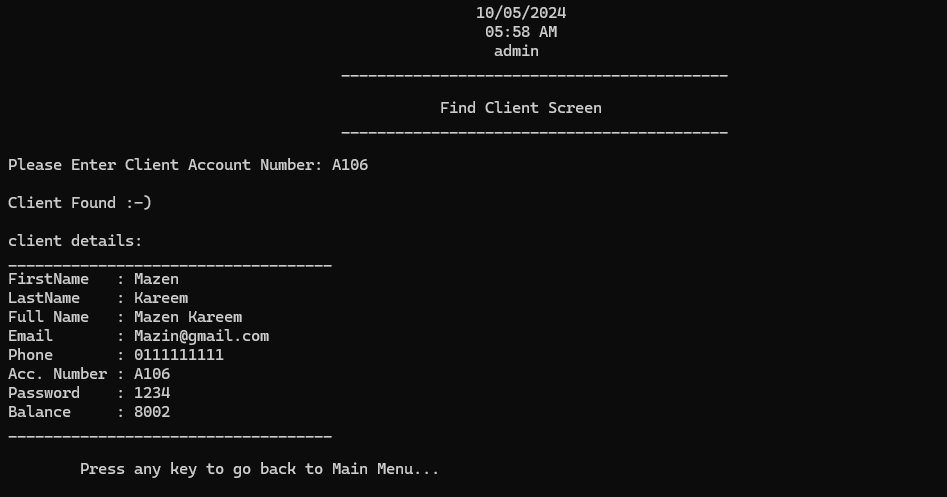
Pic 3 Client List



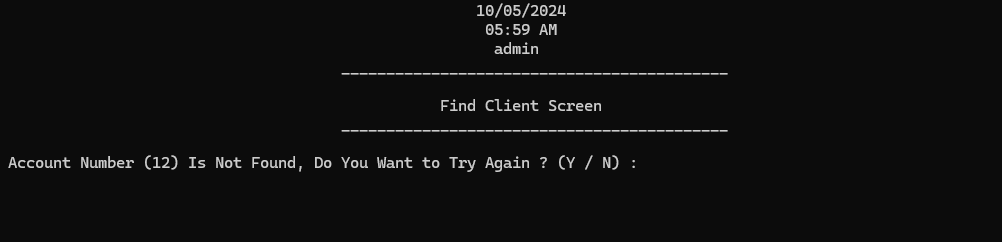
Pic 4 Main Menu



Pic 5 Search Client



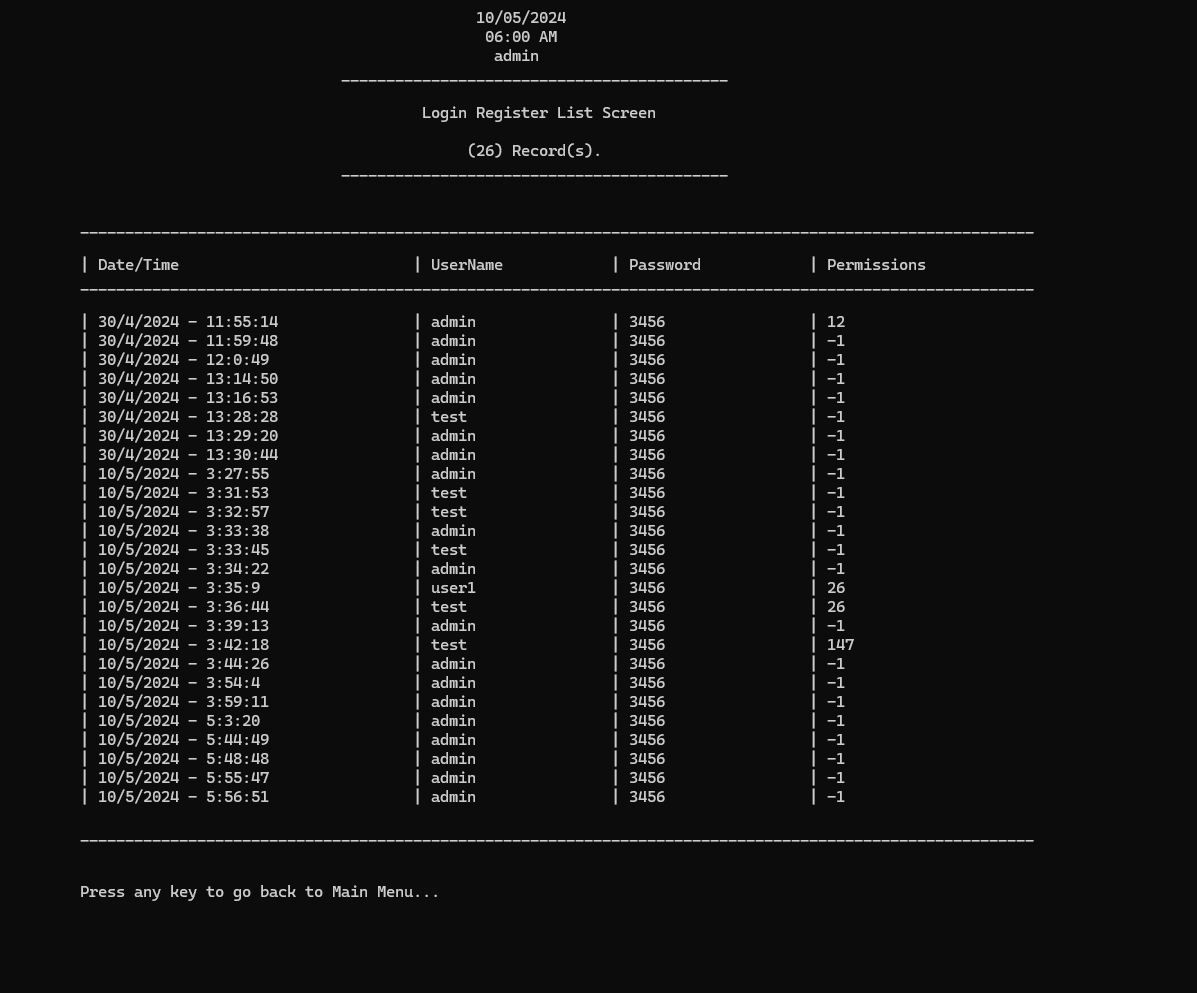
Pic 6 Search Client (found)



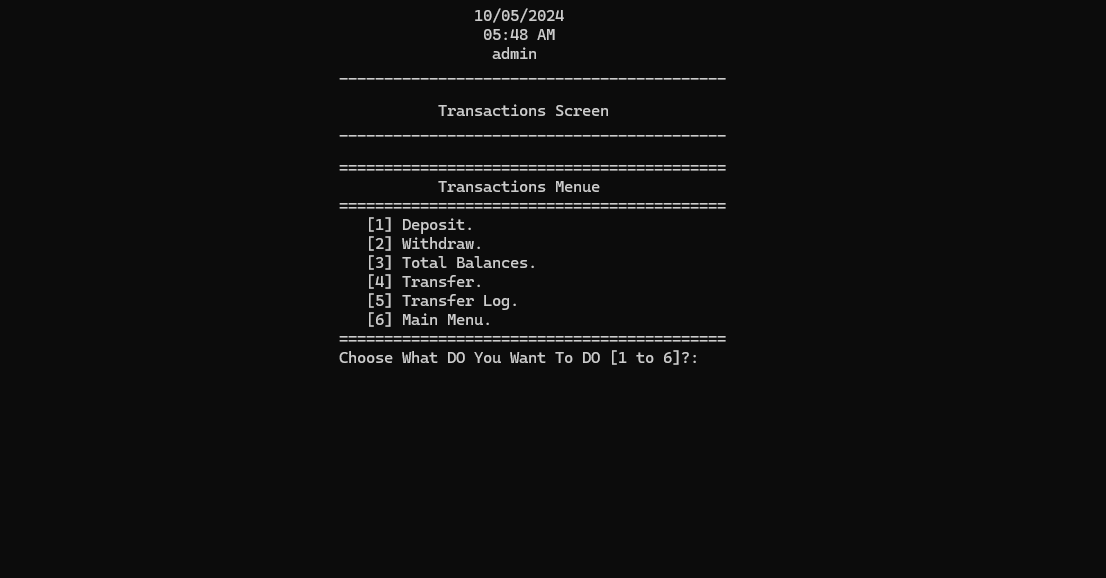
Pic 7 Search Client (not-found)



Pic 8 Manage users



Pic 9 Login Register



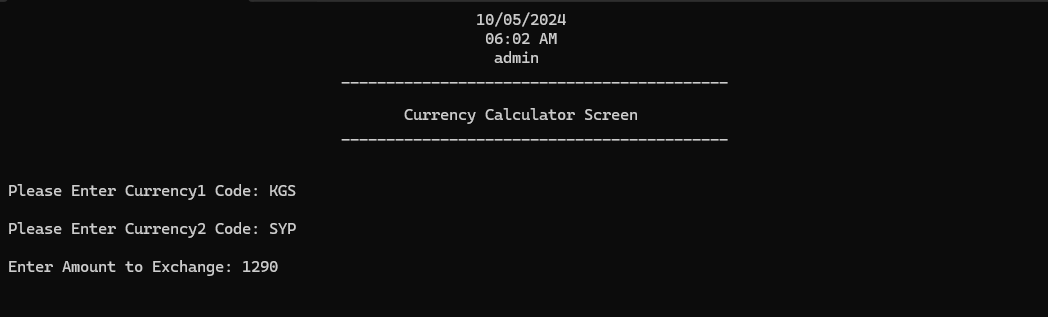
Pic 10 Transactions Menu



Pic 11 Currency Exchange



Pic 12 Currency List



Pic 13 Currency Calculator



Pic 14 Currency Calculator Results