Tab 1

PROJECT REPORT

Online Banking Management System

Name:AKANKSHYA PANDA

REGISTRATION NUMBER:25BCE10573

College NAME:VIT BHOPAL UNIVERSITY

SUBJECT:CSE

TOPIC:**Online Banking Management System**

Tab 2

1. Introduction

This report details the design and implementation of a rudimentary **Online Banking Management System (OBMS)**developed in Python. The system is designed to simulate core banking functionalities, allowing users to create accounts, manage deposits and withdrawals, check balances, and close accounts. The project serves as a practical demonstration of **Object-Oriented Programming (OOP)** principles, data structure management, and basic input/output handling.

2.Problem Statement

The primary challenge addressed by this project is the need for a simple, self-contained system to **model the fundamental operations of a bank**. Specifically, the system must securely manage individual account data (holder name, account number, balance) and provide a reliable interface for transactional activities (deposit, withdrawal) while ensuring data integrity and preventing common financial errors like overdrafts or negative transactions.

3.Functional Requirements

Major functional modules used,clear input/output structure,a logical workflow of how the user interacts with the system,user and data management,data input and processing

ID	Requirement	Description
1	Account Creation	Allow a user to create a new account with a name

		and optional initial deposit.
2	Unique ID Generation	Automatically generate a unique account number for every new account.
3	Deposit	Allow users to add funds to an existing account. Must validate that the amount is positive.
4	Withdrawal	Allow users to remove funds from an existing account. Must validate that the amount is positive and that sufficient balance exists (no overdraft).
5	Balance Check	Allow users to view the current balance of a specific account.
6	Display Details	Allow users to view all details (Name, Number,

		Balance) of a specific account.
7	Close Account	Allow users to close an account, provided the balance is zero.
8	Display All	Allow the bank operator to view details of all active accounts.

4.Non Functional Requirements

These define **how well** the system performs.

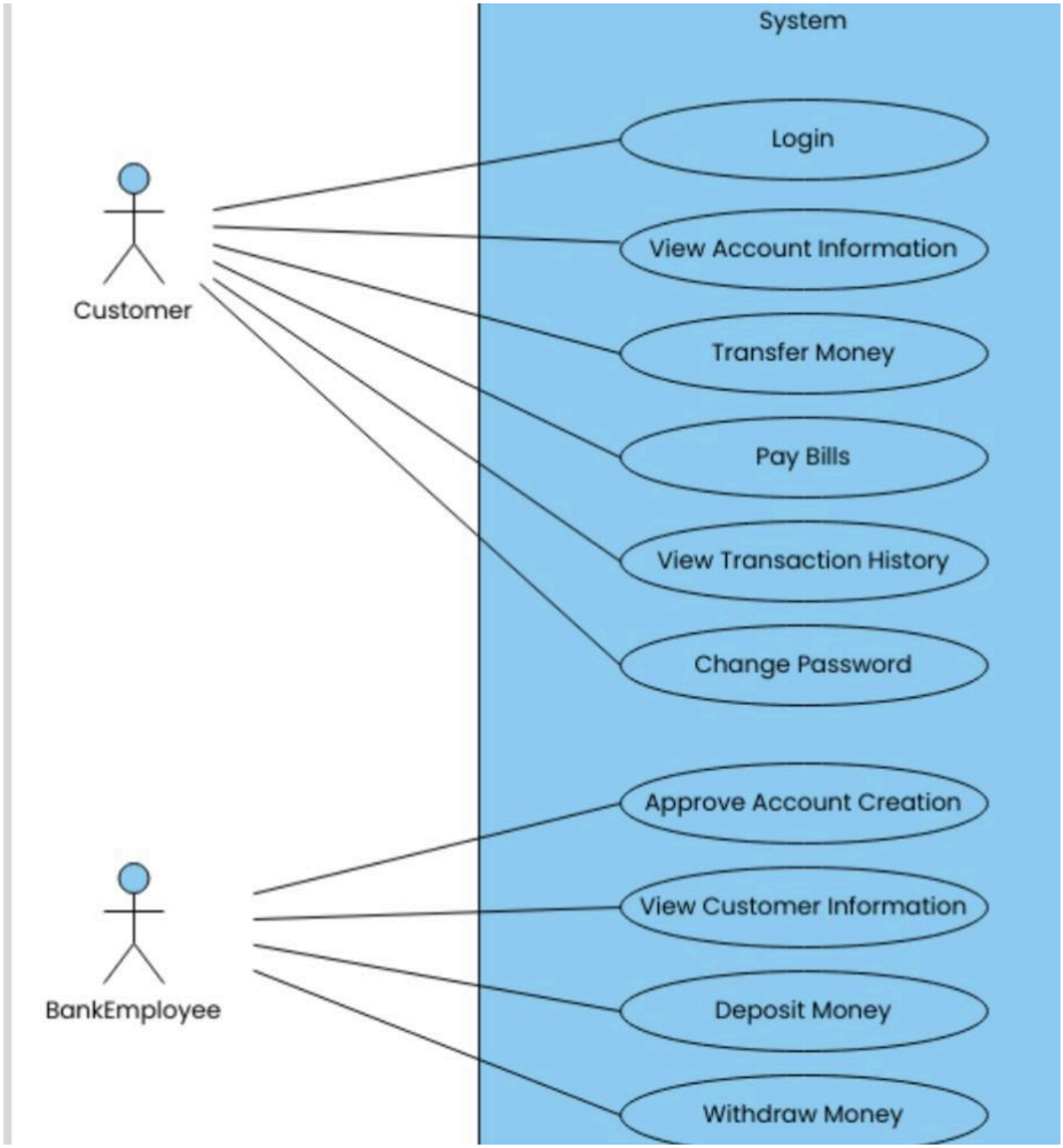
- **Security:** Account numbers must be unique, and transactions must be validated against business rules (e.g., preventing negative deposits and overdrafts).
- **Usability:** The system must provide a clear, menu-driven command-line interface (CLI) for ease of use.
- **Maintainability:** The code should be well-structured using OOP principles for easy modification and scaling.
- **Performance:** All core operations (lookup, deposit, withdrawal) should execute instantly, given the small scale and in-memory nature of the database.
- **Reliability:** The system must handle non-numeric inputs gracefully using exception handling (try/except).

5.System Architecture

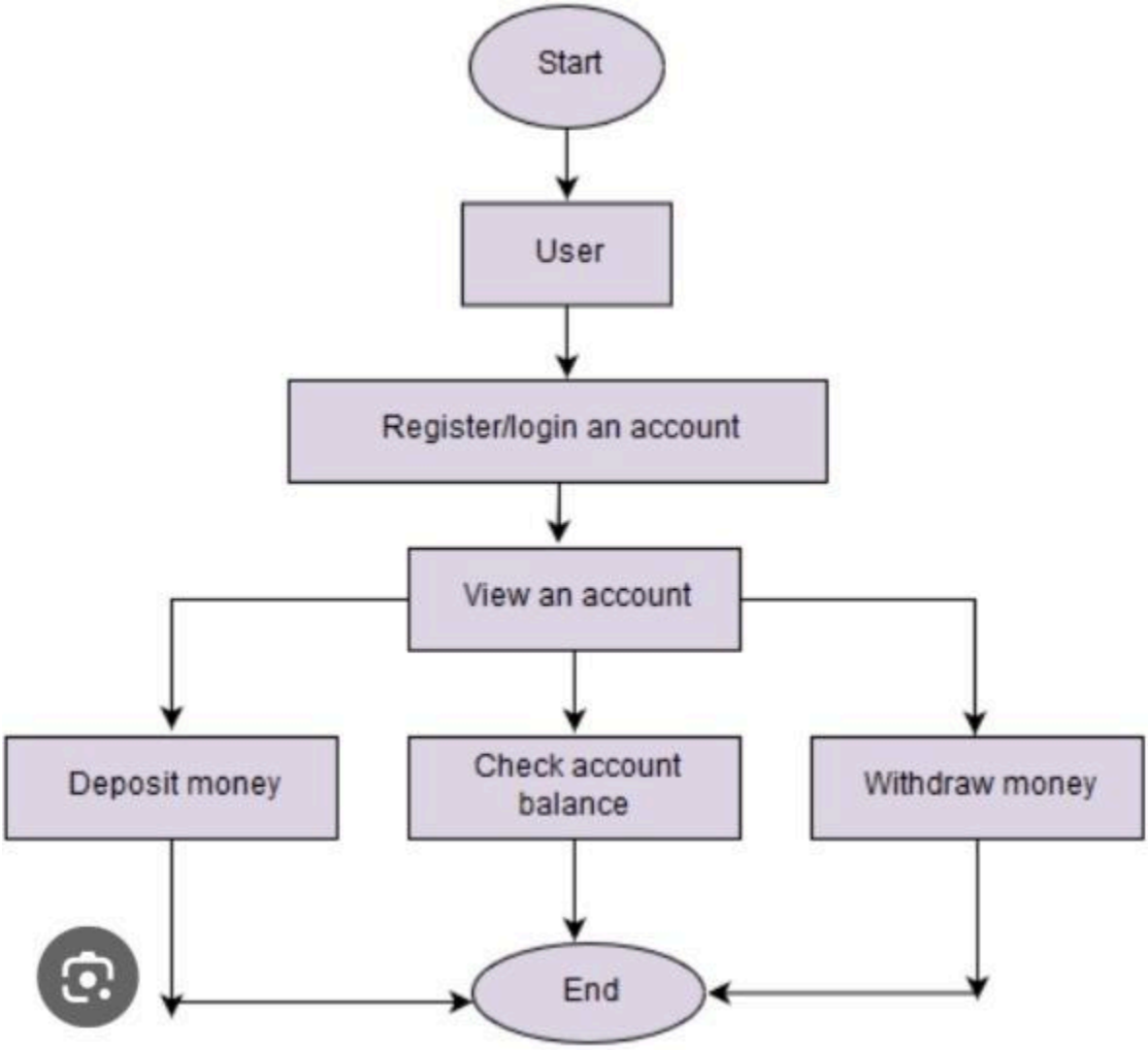

User (Human)
(Interacts with)
main() Function
(CLI Interface)
(Controls / Orchestrates calls to)
Bank Class
(Account Manager)
<>-- (Manages/Contains multiple)
Account Class
(Individual Account)
(Bank uses for IDs)
random Module
(Python Standard)

7.Design Diagrams

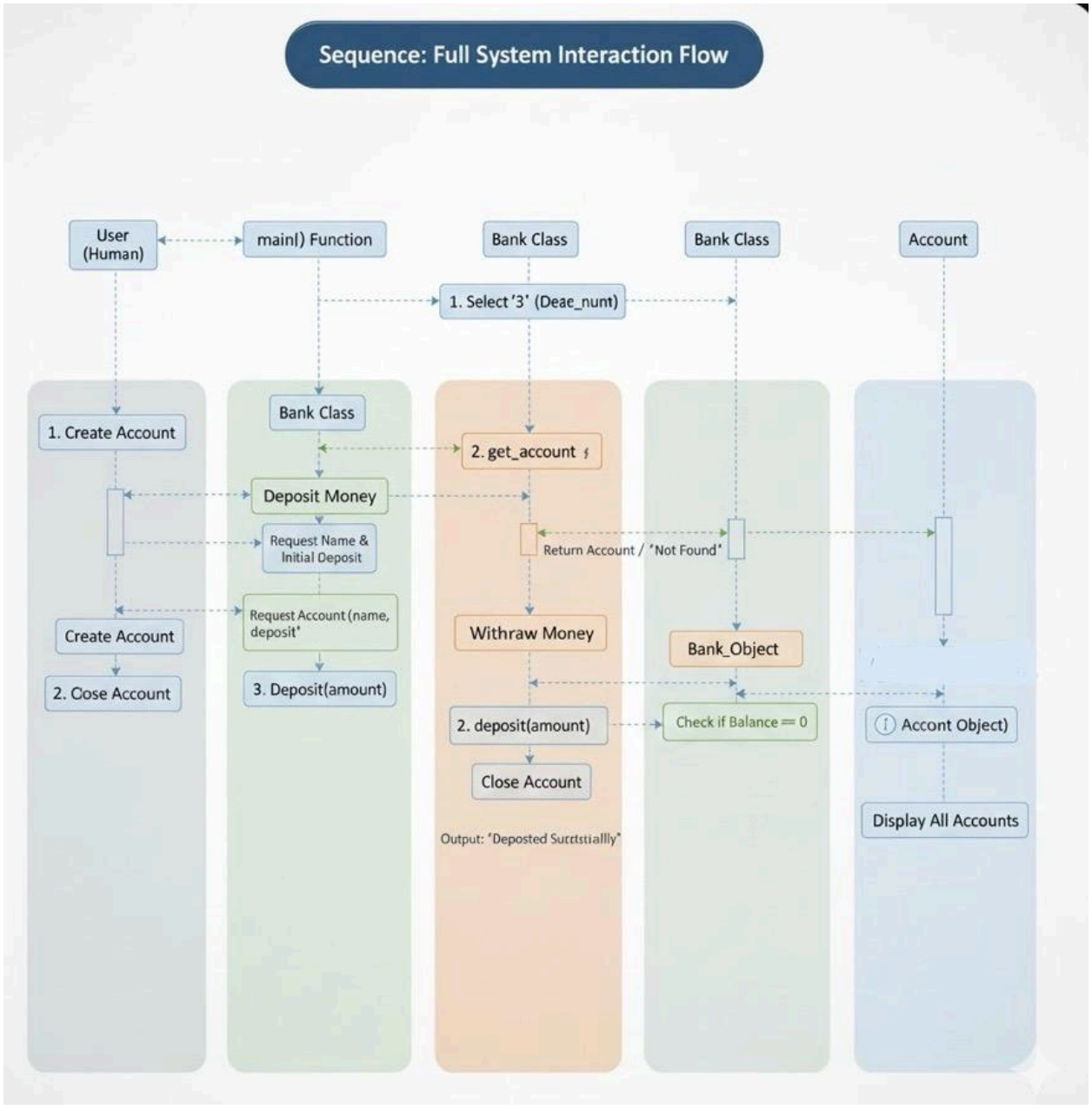
Case diagram



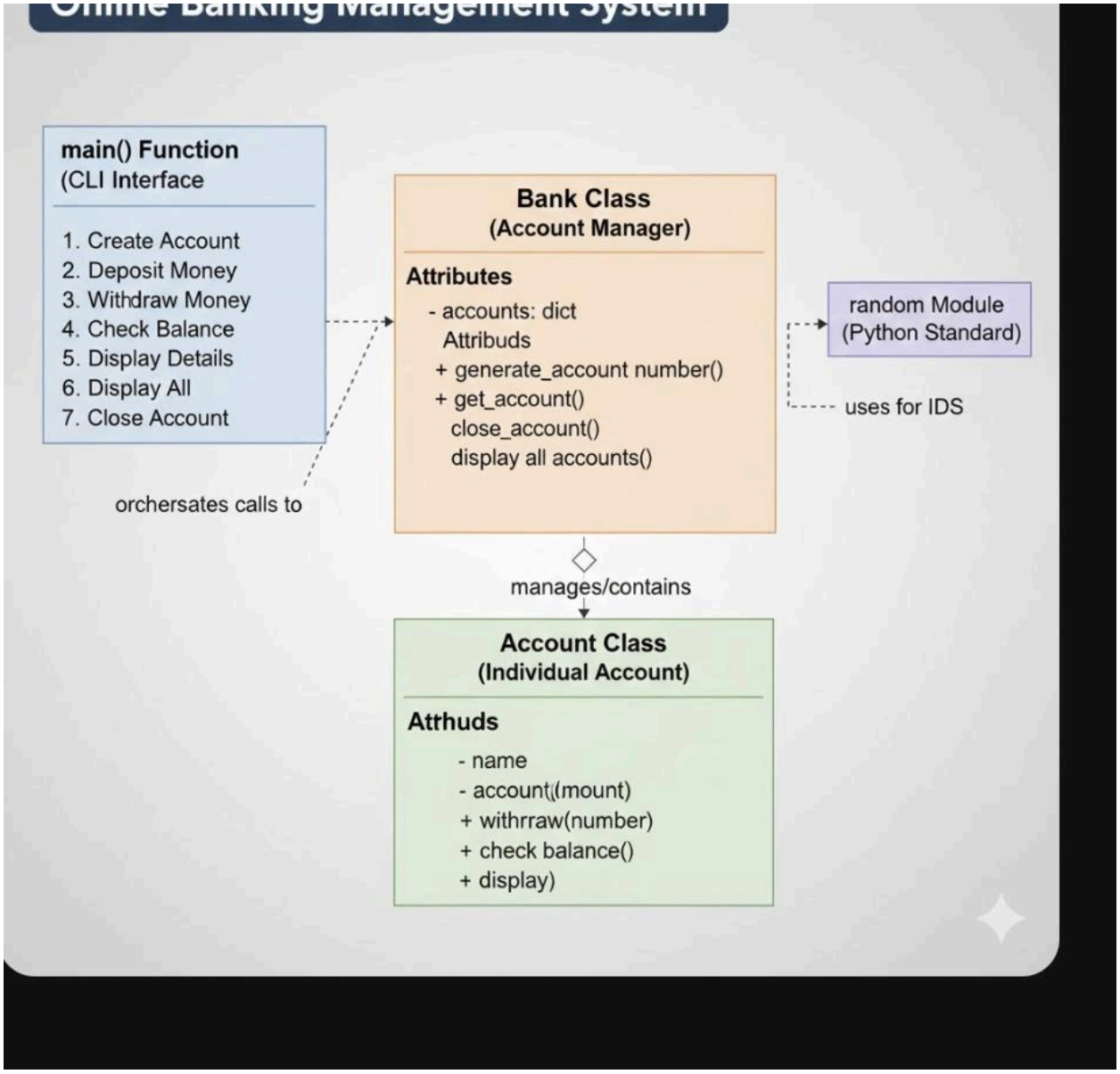
Workflow diagram

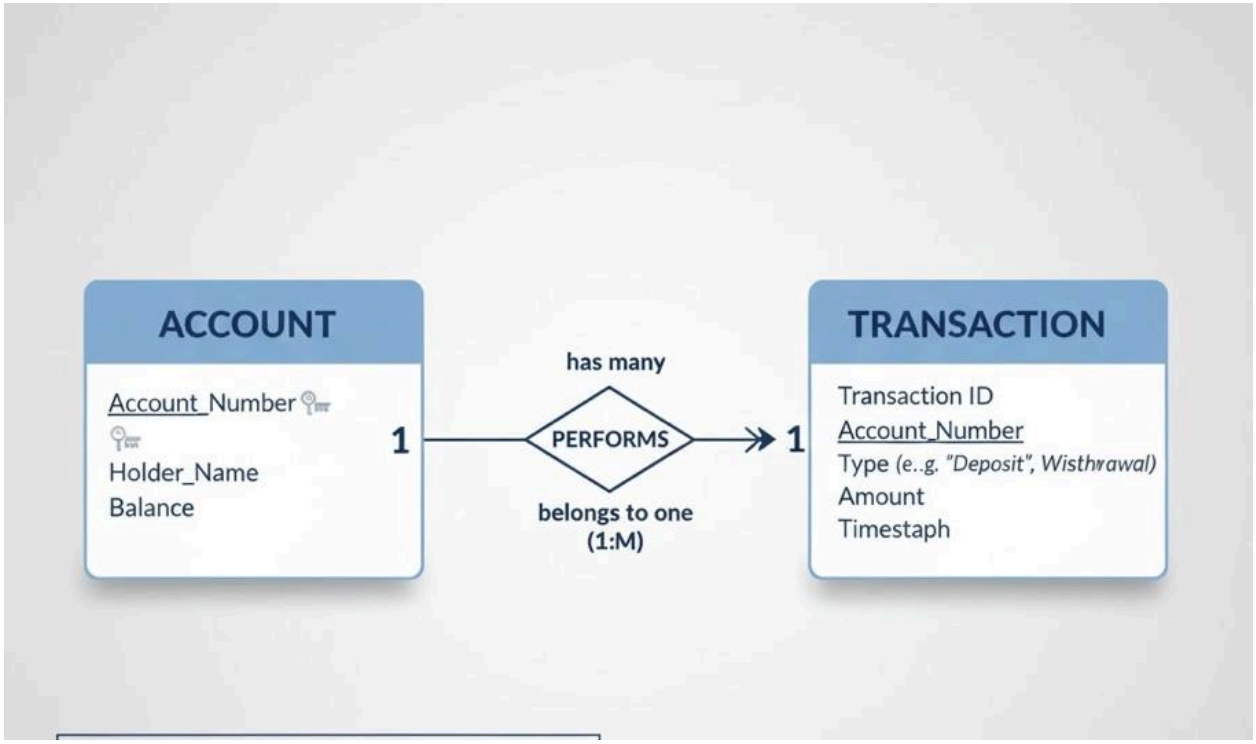


Sequence diagram



Class component diagram




ACCOUNT
Account_Number
Holder_Name
Balance
1
has many
PERFORMS
belongs to one
(1:M)
1
TRANSACTION
Transaction ID
Account_Number
Type (e..g. "Deposit", Wisthrawal)
Amount
Timestaph

Design decisions and rationale

Design Decision	**Rationale**
OOP Structure (Classes)	Encapsulates data (e.g., balance) and behavior (e.g., deposit) together, improving modularity and maintainability.
Bankuses a Dictionary	Using $\text{self.accounts = \{acc\_num: account\_object\}}$ provides **O(1)** (constant time) lookup efficiency when retrieving an account, which is crucial for quick transactions.
In-Memory Storage	Appropriate for a simple demonstration/prototype. It keeps the project self-contained and avoids dependency on external files or databases.
Transaction Validation Logic	Placing validation logic (amount > 0, amount <= balance) directly inside the deposit and withdraw methods ensures that the integrity of the

	Account object can never be compromised by bad data.
Command-Line Interface (CLI)	Simplest implementation for basic input and output, suitable for a Python script focus.

Implementation Details

The system was implemented using **Python 3.x**.

- **Modules Used:** random for generating unique account IDs.
- **Data Structures:** The Bank class uses a **dictionary** to store accounts, mapping the unique account number (string key) to the Account object (value).
- **Key Methods:**
    - **generate_account_number:** Uses a while True loop and random.randint to ensure the generated ID is unique before returning it.
    - **withdraw:** Includes a crucial if/elif/else block to check for amount > 0 and amount <= self.balance before updating the balance.
    - **main loop:** Handles user interaction and calls the appropriate method on the Bank or Account object.

Screenshots/results

```
--WELCOME TO THE BANK MANAGEMENT SYSTEM -
1. CREATE NEW ACCOUNT
2. DEPOSIT MONEY
3. WITHDRAW MONEY
4. CHECK BALANCE
5. DISPLAY ACCOUNT DETAILS
6. DISPLAY ALL ACCOUNTS
7. CLOSE AN ACCOUNT
8. EXIT
-----------------------------------------
Enter your choice (1-8): 1
Enter Account Holder Name: sara
Enter Initial Deposit Amount (min 0.0): 300

 ACCOUNT CREATED SUCCESSFULLY.
Assigned Account Number: 229472
Initial Deposit: ₹300.00
```

```
--WELCOME TO THE BANK MANAGEMENT SYSTEM -
1. CREATE NEW ACCOUNT
2. DEPOSIT MONEY
3. WITHDRAW MONEY
4. CHECK BALANCE
5. DISPLAY ACCOUNT DETAILS
6. DISPLAY ALL ACCOUNTS
7. CLOSE AN ACCOUNT
8. EXIT
----------------------------------------
Enter your choice (1-8):
============================================================


--WELCOME TO THE BANK MANAGEMENT SYSTEM -
1. CREATE NEW ACCOUNT
2. DEPOSIT MONEY
3. WITHDRAW MONEY
4. CHECK BALANCE
5. DISPLAY ACCOUNT DETAILS
6. DISPLAY ALL ACCOUNTS
7. CLOSE AN ACCOUNT
8. EXIT
----------------------------------------
Enter your choice (1-8): 1
Enter Account Holder Name: NANDINI SINGH
Enter Initial Deposit Amount (min 0.0): 20000

 ACCOUNT CREATED SUCCESSFULLY.
Assigned Account Number: 700191
Initial Deposit: ₹20000.00
```

```
--WELCOME TO THE BANK MANAGEMENT SYSTEM -
1. CREATE NEW ACCOUNT
2. DEPOSIT MONEY
3. WITHDRAW MONEY
4. CHECK BALANCE
5. DISPLAY ACCOUNT DETAILS
6. DISPLAY ALL ACCOUNTS
7. CLOSE AN ACCOUNT
8. EXIT
-----------------------------------------------
Enter your choice (1-8): 3
Enter Account Number: 700191
Enter withdrawal amount: 2800
 ₹2800.00 withdrawn successfully.


--WELCOME TO THE BANK MANAGEMENT SYSTEM -
1. CREATE NEW ACCOUNT
2. DEPOSIT MONEY
3. WITHDRAW MONEY
4. CHECK BALANCE
5. DISPLAY ACCOUNT DETAILS
6. DISPLAY ALL ACCOUNTS
7. CLOSE AN ACCOUNT
8. EXIT
-----------------------------------------------
Enter your choice (1-8): 4
Enter Account Number: 700191
Current Balance: ₹17200.00
```

Testing Approach

1	create new account with initial deposit	account “Sara” gets created with mentioned deposit
2	withdraw amount	amount 2800 gets withdrawn
3	viewing current balance	current balance is displayed.

Challenges Faced, Learnings, and Key Takeaways

Challenges Faced

- **Handling Floating-Point Precision:** When dealing with currency (float), precision issues can arise (though less prominent in this simple model). The use of **:.2f formatting** in all print statements was critical to display the currency correctly to two decimal places.
- **Input Handling (Robustness):** Ensuring the system didn't crash when a user entered text instead of a number for a monetary amount required careful implementation of the try...except ValueError block in the main function.

Learnings and Key Takeaways

- **OOP is Crucial for Modeling:** The distinct Account and Bank classes clearly demonstrated the power of **encapsulation** and **separation of concerns** in software design.
- **Data Structure Choice:** The efficiency of the system is heavily reliant on choosing the dictionary (dict) for account storage, showcasing the importance of **algorithmic thinking**.

- **Defensive Programming:** Writing code that anticipates user error (like the checks in withdraw and the try/exceptblocks) is essential for building reliable systems.

Future Enhancements

Enhancement	**Description**
Data Persistence	Implement saving account data to a file (.csv or .json) so accounts are not lost when the program closes.
GUI Development	Replace the CLI with a graphical user interface (GUI) using libraries like Tkinter or PyQt for a more modern user experience.
User Authentication	Implement a simple PIN/Password check for withdraw and close_accountmethods to enhance security.

Transaction History	Add a list to the Account class to log all deposit and withdrawactions.

REFERENCES

This project was developed using:

- **Python 3.10+ documentation**
- **Official Python random module documentation**
- **Object-Oriented Programming (OOP) design patterns**