Introduction

Object-Oriented Programming (OOP) represents a fundamental paradigm shift in software development, moving away from a purely procedural approach to one centered on "objects." These objects, which are instances of classes, serve as models for real-world entities by encapsulating both data (attributes) and the behaviors that operate on that data (methods). The primary goal of OOP is to manage the complexity inherent in software by organizing it into logical, self-contained, and reusable units. This approach leads to a more intuitive design process and results in code that is significantly more modular, maintainable, and scalable over time.

The banking domain provides a classic and highly effective use case for applying the OOP methodology. A financial system is naturally composed of distinct entities with specific properties and functions, such as different types of accounts, customers, and transactions. Attempting to model such a system using a procedural approach would likely lead to a rigid and convoluted codebase, where data and functions are loosely connected, making future modifications or extensions difficult and error-prone. By contrast, OOP allows for the creation of a clean and logical architecture where an `Account` can be defined as a base class, establishing a common blueprint for all account types.

This project leverages the power of inheritance to extend this base `Account` class into more specialized subclasses: `SavingsAccount` and `CheckingAccount`. These child classes inherit the common attributes and methods—like an account number and the ability to deposit funds—while also introducing their own unique characteristics, such as an interest rate for savings or an overdraft limit for checking. This hierarchical structure not only promotes code reuse but also demonstrates polymorphism, as common methods like `display_details()` are overridden to provide specific outputs for each account type. Furthermore, concepts like encapsulation are used to protect sensitive data like the account balance, allowing access only through controlled public methods. The project also showcases operator overloading to create more intuitive syntax for transactions, solidifying the practical benefits of a well-designed object-oriented system.

Problem Statement

The challenge is to design and build a simple yet robust banking application capable of managing multiple types of bank accounts, such as savings and checking. A traditional procedural approach to this problem can lead to disorganized and tangled code that is difficult to maintain, modify, and extend without introducing errors.

Therefore, the problem is to create a functional banking system that effectively utilizes Object-Oriented Programming principles. The system must have a logical structure that mirrors the real-world domain, ensuring that the code is modular, reusable, and easily extensible for future enhancements, such as adding new account types or features.

Objectives

- To design and implement a console-based application for basic banking operations.
- To correctly implement **encapsulation** by bundling data and methods into classes and controlling access to the data.
- To use **inheritance** to create specialized classes (`SavingsAccount`, `CheckingAccount`) from a general base class (`Account`).
- To demonstrate **abstraction** by using an **abstract class** to define a common, mandatory interface for all account types.
- To implement **polymorphism** through **method overriding**, allowing the same method to behave differently for different subclasses.
- To showcase **operator overloading** to provide a more intuitive syntax for common operations like deposits and withdrawals.

Methodology

The project is developed using Python, a high-level programming language well-suited for Object-Oriented Programming (OOP). The methodology is centered on designing a system of classes that effectively model a real-world banking scenario while demonstrating key OOP concepts.

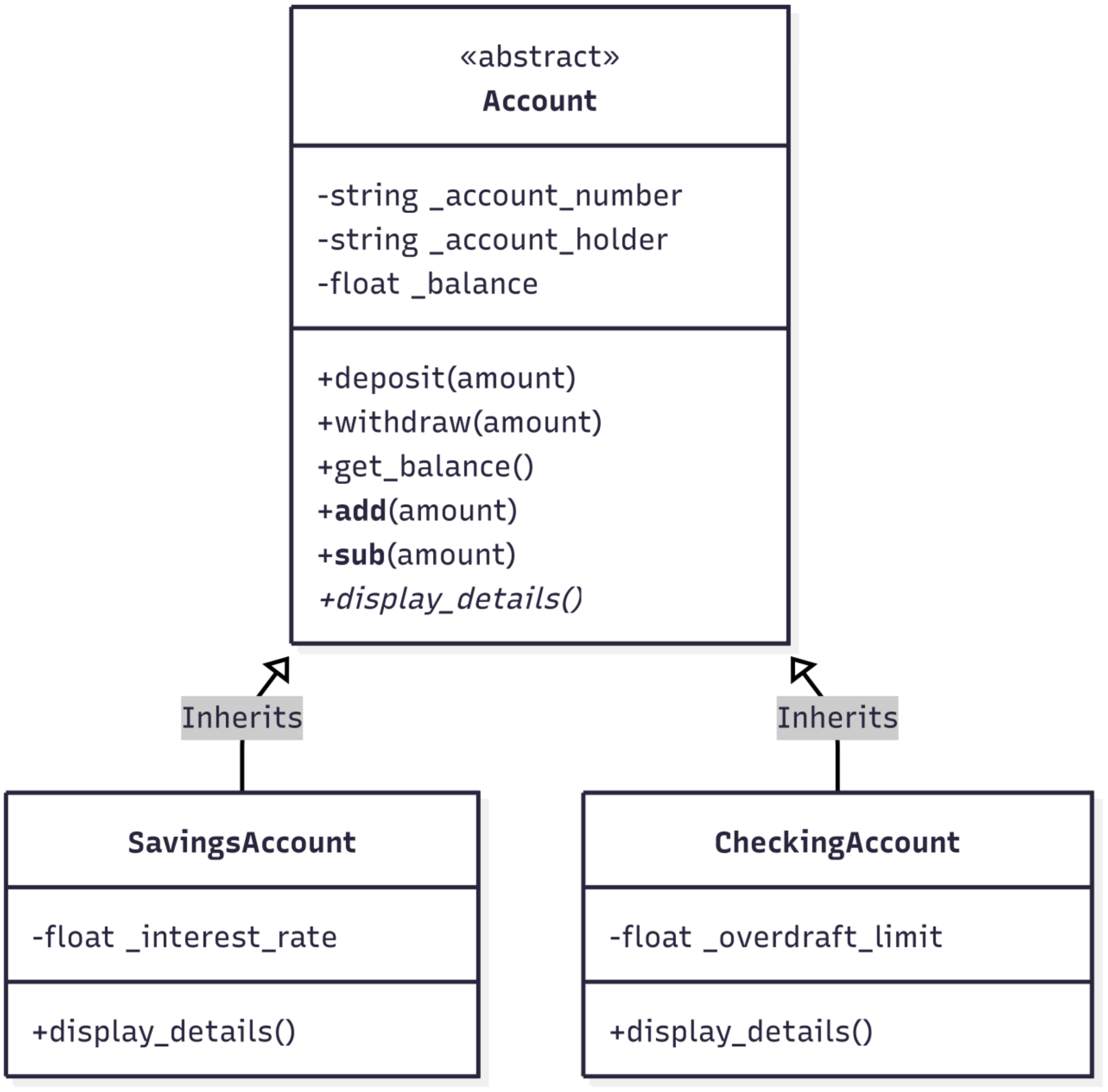


Fig : ***<u>Class Diagram Code</u>***

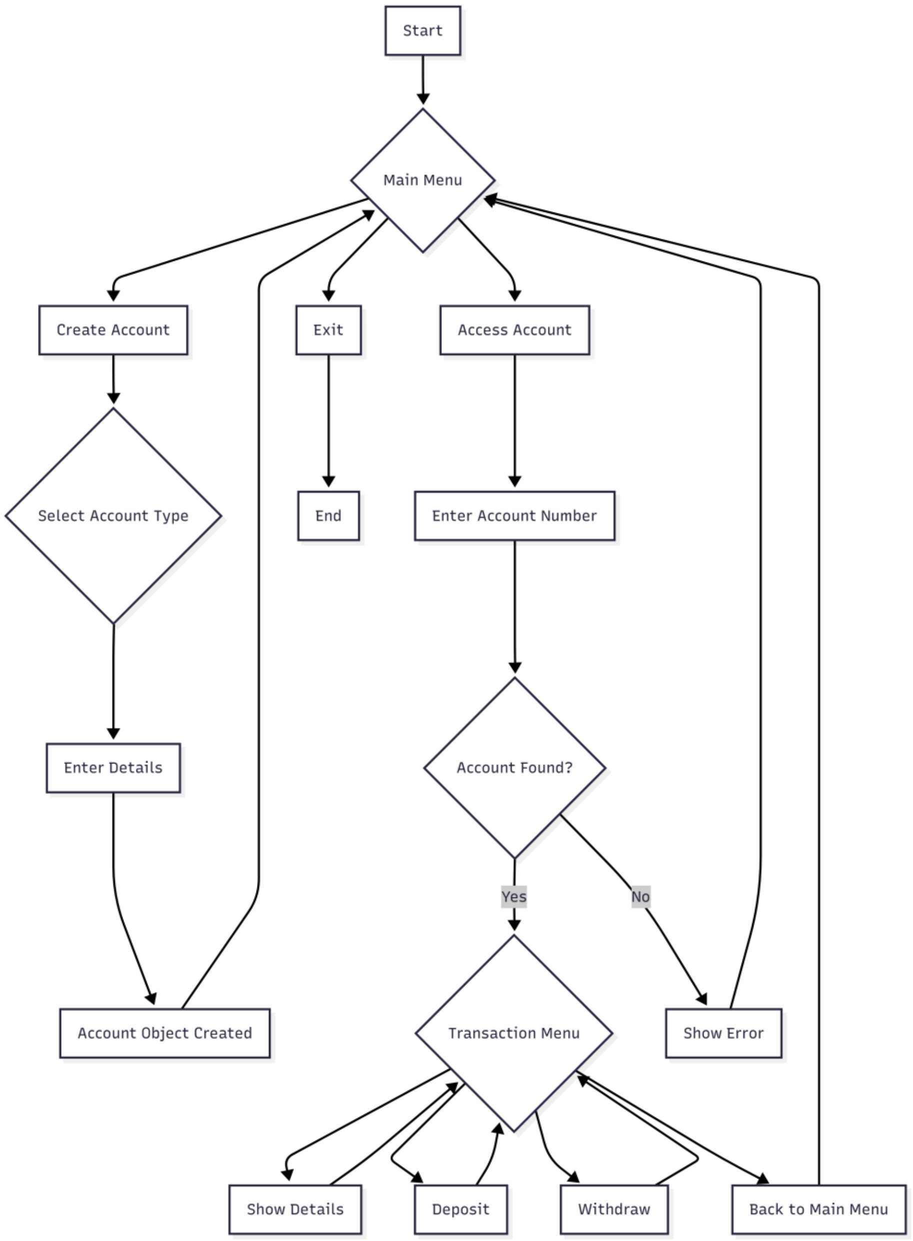


Fig : System Flow Diagram

Class Design and OOP Implementation

The core of the methodology is the implementation of the following OOP principles:

- **Encapsulation**: Data security and integrity are achieved by bundling data (attributes) and methods that operate on that data within a class. In this project, account attributes like `_account_number`, `_account_holder`, and `_balance` are declared as protected (by convention, using a single underscore _). Direct access is restricted; all interactions, such as deposits and withdrawals, are handled through public methods (`deposit()`, `withdraw()`, `get_balance()`), preventing unauthorized or invalid modifications.
- **Abstraction and Abstract Class**: To create a robust and scalable structure, an abstract base class named `Account` is defined using Python's `abc` module. This class serves as a blueprint, hiding the complex internal workings and exposing only the essential functionalities common to all account types. It includes an abstract method, `display_details()`, which forces any subclass that inherits from `Account` to provide its own specific implementation, ensuring a consistent interface across all parts of the program.
- **Inheritance**: To promote code reusability and establish a logical hierarchy, specialized classes `SavingsAccount` and `CheckingAccount` are created. These classes **inherit** from the parent `Account` class, automatically acquiring all its common attributes and methods. They also extend the base functionality by adding their own unique attributes (`_interest_rate` for `SavingsAccount` and `_overdraft_limit` for `CheckingAccount`).
- **Polymorphism through Method Overriding**: Polymorphism, the ability for an object to take on many forms, is implemented via method overriding. The `display_details()` method, declared as abstract in the `Account` class, is implemented differently in the `SavingsAccount` and `CheckingAccount` subclasses. This allows a single method call (`account.display_details()`) to produce different outputs depending on the specific type of account object, making the code more flexible and dynamic.
- **Operator Overloading**: To make the code more intuitive and readable, Python's special "dunder" (double underscore) methods are used to overload standard operators. Specifically, the addition (+) and subtraction (`-`) operators are overloaded for `Account` objects to handle deposits and withdrawals, respectively. This allows for natural-feeling operations like `my_account = my_account + 500`.

Result and Conclusion

The project's final result is a fully functional, interactive console-based banking application. The program successfully executes and demonstrates all the required Object-Oriented Programming principles. Upon running the script, the user is presented with a clear menu to navigate the application's features. The system effectively handles the creation of different account types, processes transactions, and displays account information as expected.

Expected Output

The application's output is purely text-based, appearing in the user's command line or terminal. The interaction flow is as follows:

The user is greeted with a main menu.

They can create either a Savings or Checking account, and the system confirms the creation and provides a unique account number.

Using the account number, the user can access a specific account's sub-menu.

From the sub-menu, they can perform operations like depositing, withdrawing, and displaying account details. The system provides clear feedback for each action.

Sample Console Interaction

Below are examples of what the console output looks like during a typical user session.

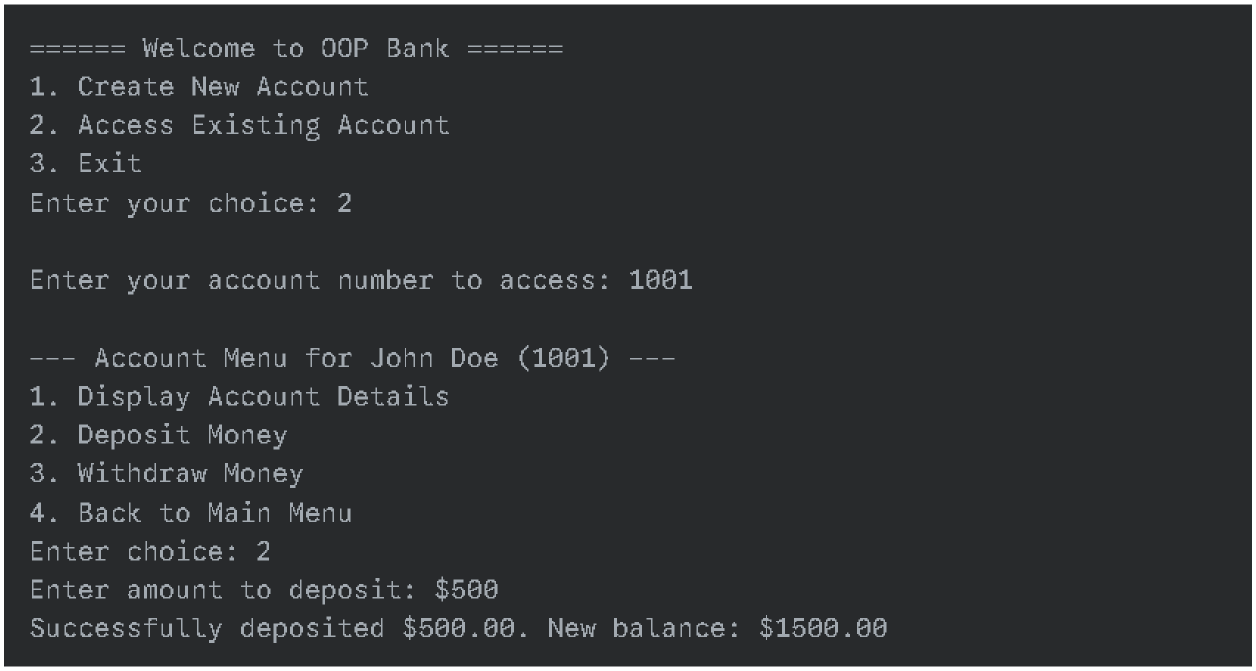
Main Menu and Account Creation:

```
====== Welcome to OOP Bank ======
1. Create New Account
2. Access Existing Account
3. Exit
Enter your choice: 1

--- Create a New Bank Account ---
Enter account holder's name: John Doe
Enter initial balance: $1000
Select account type:
  1. Savings Account
  2. Checking Account
Enter choice (1 or 2): 1

Account created successfully!
Your new account number is: 1001
```

Accessing an Account and Performing Transactions:

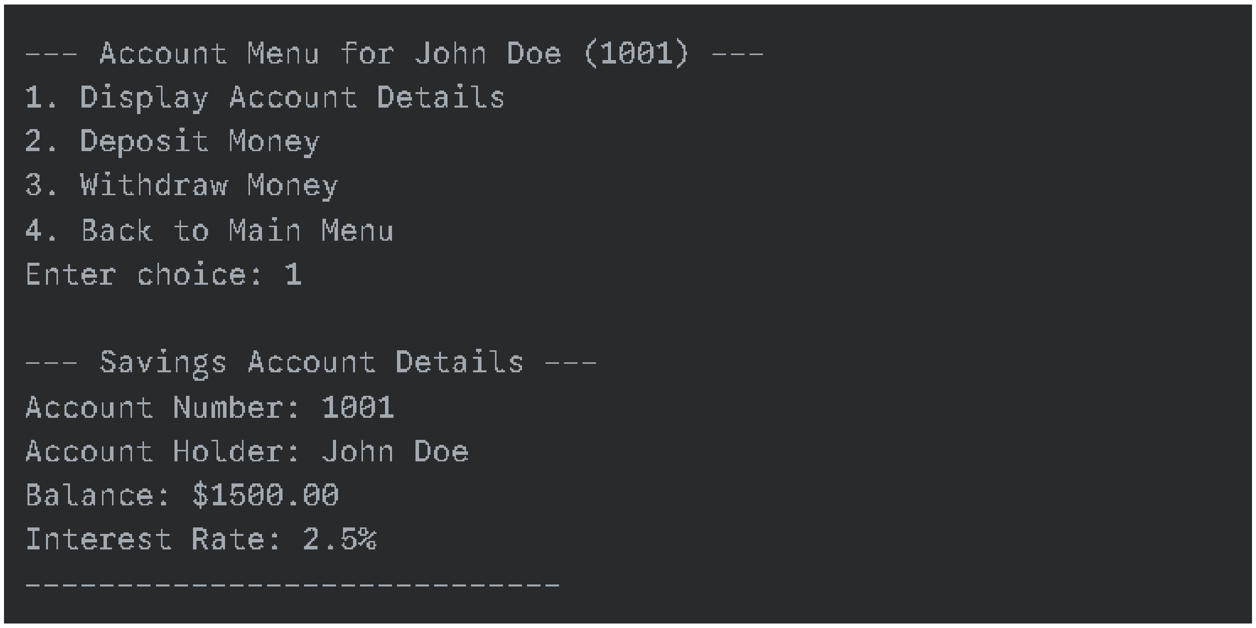


```
====== Welcome to OOP Bank ======
1. Create New Account
2. Access Existing Account
3. Exit
Enter your choice: 2

Enter your account number to access: 1001

--- Account Menu for John Doe (1001) ---
1. Display Account Details
2. Deposit Money
3. Withdraw Money
4. Back to Main Menu
Enter choice: 2
Enter amount to deposit: $500
Successfully deposited $500.00. New balance: $1500.00
```

Displaying Account Details (Polymorphism in Action):



```
--- Account Menu for John Doe (1001) ---
1. Display Account Details
2. Deposit Money
3. Withdraw Money
4. Back to Main Menu
Enter choice: 1

--- Savings Account Details ---
Account Number: 1001
Account Holder: John Doe
Balance: $1500.00
Interest Rate: 2.5%
-----------------------------
```