Group 6: Console-based Bank System

# Student Name:

1. Nattanicha Nilsriphaiwan
2. Ying Su

# Course:

COMP 4603 – Advanced C++

# Instructor:

Frederic Guo

# Introduction:

A simple command-line application is designed to simulate a basic banking system. It allows users to manage accounts, perform transactions, and track account history. This project aims to provide a foundational understanding of banking systems while implementing object-oriented programming principles.

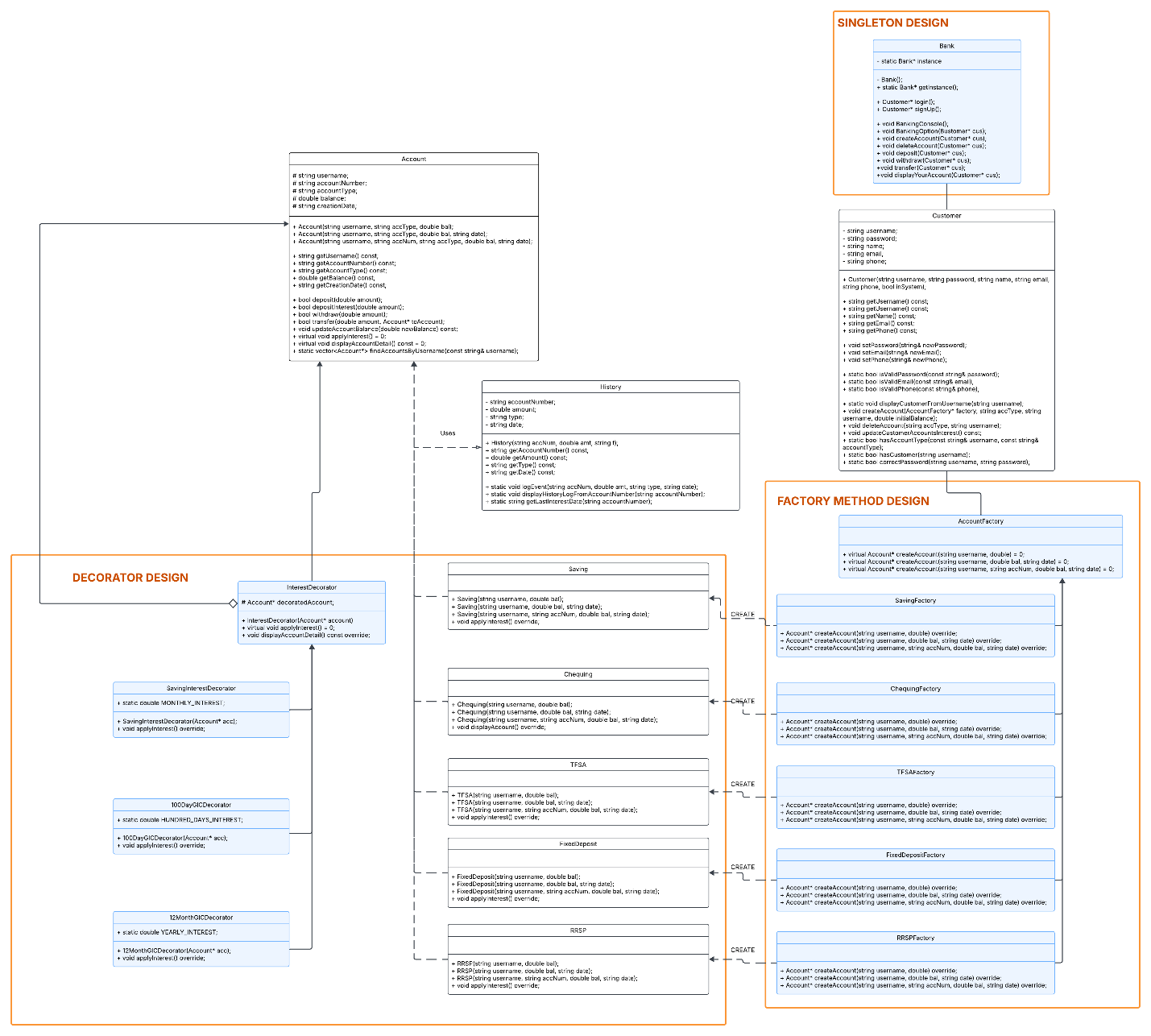
# Objectives:

* **Account Management**: Create, delete, and manage multiple account types (e.g., Savings, Chequing, Fixed Deposit, TFSA, RRSP).
* **Transactions:** Perform deposit, withdrawal, and transfer operations between accounts.
* **Transaction History:** Track and store transaction history for each account.
* **Balance Inquiry:** Allow users to check account balances.
* **Interest Calculation:** Apply interest on relevant account types.
* **Security:** Basic password protection for accounts.

# Scope:

This project focuses on implementing core banking features like account management, transactions, balance inquiries, and transaction history. It supports multiple account types and basic security features. The system is command-line based and designed for learning and simulation purposes.

# UML Diagram:

****

<https://lucid.app/lucidchart/947c72c2-5939-4e61-be81-3bb8e94eb0c8/edit?viewport_loc=-5249%2C-636%2C6026%2C2847%2C0_0&invitationId=inv_7db3b363-fb8a-4eb9-93ba-1a63c268eb7d>

# Design Patterns:

## Singleton Design Pattern

* Singleton is applied in Bank class.
* This ensures that only one instance of the Bank class exists.
* It provides a global access point to manage customers and accounts.
* It is implemented by making the constructor private and using a static method to get the instance.

A screenshot of a computer screen

AI-generated content may be incorrect.

## Factory Method Pattern

* AccountFactory class is an interface that defines a method to create accounts.
* Each account type implements the factory to create specific accounts. (SavingFactory, ChequingFactory, FixedDopositFactory, TFSAFactory, and RRSPFactory)
* This pattern allows flexibility when adding new account types without modifying existing code.

A diagram of a factory method

AI-generated content may be incorrect.

## Decorator Pattern

* InterestDecorator allows adding dynamic interest calculation to accounts without modifying the base Account class.
* InterestDecorator is an abstract class that wraps around a concrete Account object.
* Concrete decorators override InterestDecorator methods, and each cconcrete decorator class has a different way of interest calculation.

(SavingInterestDecorator, 100DayGICDecorator, and 12MonthGICDecorator)

A diagram of a computer

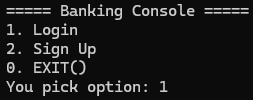
AI-generated content may be incorrect.

# Command Line User Interface:

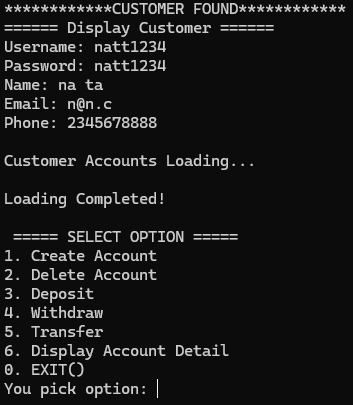
The interface uses a do-while loop to continuously display the menu options until the user chooses to exit. The process works as follows:

* Show menu options (e.g., Login, Sign Up, etc.).
* Get user input and execute the corresponding function using a switch statement.
* Repeat the process until the user selects the exit option, which terminates the loop.
* A user can create accounts by providing their name, a username, a valid password, email, and phone number. User accounts are protected by passwords.
* A user must login to the system with verified username and password first before performing various transactions, like deposit, withdrawal, and transferring money.
* Method to apply interest to eligible accounts is called when the user login to their account, whether the interest is applied will go through the validation process in the method.

## Login or Sign-up UI



## Banking option UI (once an account is created or login):

****

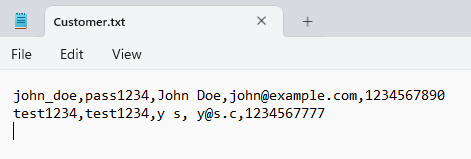
### Interest UI

## 

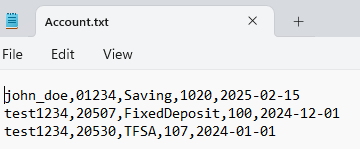
# Data Persistence:

* There are 3 text files used to record program data and history, which mimics 3 tables in a database:
  + Customer.txt for customer records
  + Account.txt for account records
  + History.txt for transaction history
* The Bank class is responsible for reading, writing, and updating these text files.
* Whenever a new object such as a Customer, Account, or transaction History is created, the data will be added to or updated in the respective text files.
* Since all data is stored in these text files, we can always look up and retrieve information directly from the files whenever needed.
* This ensures data persistence and consistency even if the program is closed and restarted in future.

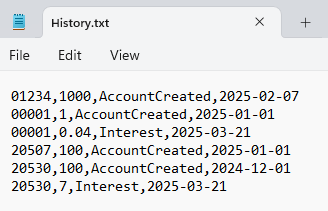
### Customer.txt



### Account.txt

****

### History.txt



# Future Improvements or Extensions

* Hashing the passwords before storing them for data persistence.
* Storing data in a database instead of in text files.
* Different levels of users and privilege.
* More convenient methods in different classes for better user experience.