Case Study 1 – Banking System

The goal of this case study is to familiarize the developers in designing a C++ stand-alone application. As part of this case study, developers need to implement a banking application that will help the users in creating customers, accounts, depositing and withdrawing money, checking their balance, sorting, persisting and searching Customer Data.

Business Requirements

1. The standalone program upon executing need to show below Menu:
    1. **Create New Customer Data**
    2. **Assign a Bank Account to a Customer**
    3. **Display balance or interest earned of a Customer**
    4. **Sort Customer Data**
    5. **Persist Customer Data**
    6. **Show All Customers**
    7. **Search Customers by Name**
    8. **Exit**
2. Upon entering option 1 from the Menu, the program needs to accept the below attribute values of a Customer
   Customer Id: int (automatic generation starts from 100)
   Customer Name: String
   Age: int

Mobile Number: int
Passport Number: String
Customer Id should be generated automatically and must get incremented by 1 each time object of Customer is created.
[Hint: Make use of ‘static’ keyword in C++ to generate and assign customer ids sequentially]

[Hint: Use parametrized constructors to create instances of

Customers in the system]

3. Upon entering option 2 from the Menu, the program needs to accept the Customer Id to whom new Bank Account would be assigned. A customer in a system needs to be associated with only one Bank Account. The Account class consists of
   Account Number : long
   BSB Code : long
   Bank Name : String
   Balance : double
   Opening Date : String DD/MM/YYYY

[Hint: Use parametrized constructors to create instances of Bank

Accounts. Create one instance of Bank Account per Customer using the concept of Composition. Make use of an appropriate ‘setter’ or Mutator method to assign a Bank Account to an existing Customer]

4. Modify Customer class to add one more attribute DOB.
   (mark as String DD/MM/YYYY format)
   Validation should be performed on the day, month and year of the date to check if it is a valid date. If validation fails, ask user to re-enter the DOB. (Date of Birth).
   [Hint: Use methods of string to extract day, month and year. Use string’s utility method to convert String value to int value.]

5. Modify the system so that Bank Accounts would be created either as i. Savings Account or ii. Fixed Account.
   [Hint: Make use of C++ Inheritance to derive 2 sub (child) classes SavingAccount and FixedDeposit from parent class Account]

   SavingAccount sub-class:-

It should have following 2 additional attributes

1. Is Salary Account (True/False)
2. Minimum Balance (default value is 5000)

A SavingAccount can be a Customer's Salary Account too
and User should specify it when creating a Saving Account.

If account is not a Salary Account, and if actual balance is lesser than minimum balance i.e. 5000, then account should not be created in this case. [Hint: Custom Exception needs to be thrown and User should receive an Error Message – "Insufficient balance for Account creation, Minimum balance is 5000"]
If account is a Salary Account, then even Zero balance is allowed.
On creating a Saving Account, User would be able to deposit, withdraw money and check their balance.
If account is Non-Salary Account, then minimum balance of 5000 has to be there and only excess amount could be withdrawn.

FixedDeposit sub-class: -
It should have following 3 additional attributes

1. Deposit Amount. (The minimum is 1000)
2. Tenure (in years) (The minimum is 1year max is 7 years)

3. InterestEarned (double)

6. The SavingAccount and FixedDeposit, both would earn interest, but interest calculation would vary in both.
   [Hint: Mark CalculateInterest method as pure virtual function in Account and override in sub classes SavingAccount and FixedDeposit].
   For SavingAccount, Interest = balance * years * 0.04.
   For FixedDeposit, Interest = deposit amount * years * 0.08.

7. Upon entering option 4 from the Menu, the system needs to ask User whether sorting needs to be done on Customer Names or Bank Balance.
   Modify the system to store all the Customers along with their Bank Accounts in an appropriate collection framework, so that
   Sorting would be easy on Customers.
   The system should support sorting of customers based on
   1. Customer Names
   2. Bank Balance.

   [Hint: Use appropriate container/collection framework class supporting Sorting.]

8. Upon entering option 5 from the Menu, the system needs to ask User whether persistence needs to be achieved using File System or RDMBS.
   Enhance the system so that it would be able to persist the Customer data into external storage. The system needs to support 2 types of storages namely

   1. File System
   2. Relational Database (RDBMS)

[Hint: Design “pure abstract class” for DAO layer namely “Idao”.]

Provide below abstract (pure virtual) methods in the pure abstract class “Idao”.

1. saveAllCustomers
2. retrieveAllCustomers

Derive 2 concrete classes from the parent class Idao.

1. FileStorageDao :- This class would serialize and deserialize all the Customer details along with Account information, except the balance, into a flat file namely – customers.txt.
   [Hint: Make use of streams in C++]
2. DatabaseStorageDao :- This class would persist and retrieve all the Customer details along with Account information into RDBMS. E.g. MySQL.

[Hint: This feature can be implemented later]

9. Upon entering option 6 from the Menu, the system needs to fetch the All Customer Data from the Database and would be displayed on a Console.

10. Upon entering option 7 from the Menu, the User would be asked to enter the name of Customer to be searched. The relevant data would be fetched from flat file and displayed to the User, supporting the Search mechanism.
[Hint: If no record found matching customer to be searched in the file, then Custom Exception namely – CustomerNotFoundException needs to be thrown and appropriate Error Message needs to be displayed to the End User.]