Jordan Bassett

Tayler Caufield

Austin Tesch

CS162 Steve Smiley

Oct 28, 2024

Phase 1

1. **Appropriate primitive data types needed for a bank account system**
2. **Account Holder Name: string.**

Probably two using two string variables for first\_name and last\_name to keep names structured.

1. **Account Balance: double**

Provides the necessary precision to handle financial data accurately.

1. **Account Type: string**

Allows flexibility to store different account types as text (e.g., "Checking", "Savings", “Credit”).

1. **Interest Rate: double**

Precision is important for interest rate calculations, especially when compounding.

1. **Transaction Amount: double**

Precision is needed for accurate transaction handling.

1. **Transaction Date: time\_t**

Typically, a time\_t, often represented as a long int.

1. **Customer ID: int**

A unique integer identifier for each customer.

1. **PIN/Password: string**

Although stored as a string, this value should be hashed or encrypted for security.

1. **Static member variables**
   1. **The bank name:** Since all accounts belong to the same bank, the bank name can be defined as a static member variable. This allows all instances of the class to share the same bank name without duplicating data in each account object. Additionally, if the bank name ever needs to be updated, it can be changed in one place for all accounts.
   2. **Interest Rate for Savings Accounts**: All savings accounts share the same interest rate. This makes it easy to update the rate across all accounts if needed.
   3. **Transaction Fee**: a static member could store this value to apply consistently across accounts.
   4. **Total Number of Accounts**: Useful for generating unique account numbers or for tracking growth.
2. **Constructors needed**
3. **Default constructor**

A default constructor is useful for creating an empty account with default values. This could be helpful when initializing an account object before setting its specific details.

1. **Overloaded Constructor**

An overloaded constructor would allow the creation of an account with specific details provided at the time of creation. This constructor could take parameters like the account number, initial balance, account type, and customer ID to fully initialize an account when it is created.

1. **Seven Classes to be designed:** BankAccount, SavingsAccount, CheckingAccount. CreditAccount, Transaction, Customer, BankAccountContainer
2. **Purpose of each class:**
   * 1. **BankAccount (Base Class)**

Base class for different account types, including attributes and methods common to all accounts, like account\_number, balance, and basic deposit/withdraw methods.

* + 1. **SavingsAccount (Derived from BankAccount)**

Adds features specific to savings accounts, such as an interest\_rate attribute and methods for interest calculation. Inherits core functionality from BankAccount.

* + 1. **CheckingAccount (Derived from BankAccount)**

Adds checking-specific attributes and methods, such as an overdraft\_limit and potentially check-writing functions. Also inherits common functionality from BankAccount.

* + 1. **CreditAccount (Derived from BankAccount)**

Represents credit card accounts with features such as a credit\_limit, APR (annual percentage rate), and possibly reward points. This class can handle credit-specific methods, such as calculating monthly payments or interest on outstanding balances.

* + 1. **Transaction**

Manages the processing of various transaction types—deposits, withdrawals, and transfers. It could also store transaction-specific details, like transaction\_id, transaction\_date, transaction\_amount, and transaction\_type. As well as having methods to reverse or cancel any transactions.

* + 1. **Customer**

Stores customer details, such as name, address, contact\_info, and a list of associated BankAccount objects. This class could also include methods for managing and accessing the customer’s accounts as well as updating contact info.

* + 1. **BankAccount Container**

Acts as a manager for all BankAccount objects within the system. This container would oversee the creation, storage, and retrieval of accounts, allowing the system to efficiently manage multiple accounts across multiple customers. It could include methods to find accounts by account\_number or customer\_id and to handle bulk transactions if needed.

1. **Class relationships**
   * 1. **“Is-a” Relationship (Inheritance)**

BankAccount serves as the base class, and specific account types inherit from it. This defines an “is-a” relationship because SavingsAccount, CheckingAccount, and CreditAccount are all a BankAccount.

* + 1. “**Has-a” Relationship (Composition)**

Customer has a “has-a” relationship with BankAccount because each Customer can hold one or more BankAccount objects. This is composition because a Customer “has-a” BankAccount. A Transaction “has-a” date, amount, and is associated with one or more BankAccount objects (the accounts involved in the transaction).

The BankAccount Container also has a composition relationship with BankAccount objects, as it manages a collection of accounts within the system.

1. “Jot down ideas what each class looks like. Then draw a UML diagram for all classes using correct symbols and class relationship.”

A diagram of a bank account

Description automatically generated