Andrew O'Kins

Almat Yeraly

CS273

GP1

Requirements Specifications

The program created will contain multiple classes that help track several bank accounts and customers of varying types within a bank object along with the transactions that occur. The way program will allow the user to setup customer accounts (with options being a Senior, Student, or Adult variation), bank accounts (with options of Savings or Checkings), and to process transactions within these accounts. The user in this program should be able to make accounts that contain either new customers or existing ones, make transactions with these accounts, and to be able to display the account information.

At this time, it will be assumed that the User is a verified worker at the bank, and can be trusted to manage the information for all Customers, Accounts, and the transactions that occur within the Bank, though they will need to supply both the account number and the customer id to find the account. It will also be temporarily assumed that this program will not be closed when managing the data, and so there will be no method of saving the data onto a file when the User commands the program to close.

A change from what is stated in the Programming Project assignment is instead of using arrays the program will utilize vectors because of their ability to have a dynamic size, since there is no limit to how many transactions an account may have. The Account object will also contain virtual methods for deposit and withdraw which Savings and Checking will define, as this will make implementation easier for calling these methods and it is logical that Accounts would contain this behavior in general and not declared in the derived only. Get functions for the account and customer numbers were also added so that they can be accessed in the get\_account() method in Bank as well as so that it can be displayed to the user. We also added a new property in account (the account type bool) to make determining if an account was checking or savings easier in the to\_string()

Detailed Use Cases

Table 1. Use Case for User Starting the Program

|  |  |  |
| --- | --- | --- |
| Step | User’s Action | System’s Response |
| 1. | User starts the program file |  |
| 2. |  | The program starts and creates an initial Bank object that contains an empty array of Accounts |
| 3. |  | Prompts the User for a command to be entered |

Table 2. Use Case for User Adding a New Customer

|  |  |  |
| --- | --- | --- |
| Step | User’s Action | System’s Response |
| 1. | User inputs a command to indicate they want to add a new customer when making an account |  |
| 2. |  | System prompts for name |
| 3. | User enters name | If User cancels entry, the process is terminated |
| 4. |  | System prompts for age |
| 5. | User enters age | If User cancels entry, the process is terminated |
| 6. |  | System prompt for telephone number |
| 7. | User enters telephone number | If User cancels entry, the process is terminated |
| 9. |  | System creates a child Customer object dependent on the age that the User inputted that contains a unique customer number. The System returns this Customer object. The process is then terminated |

Table 3. Use Case for User Adding a New Account (calling add\_account())

|  |  |  |
| --- | --- | --- |
| Step | User’s Action | System’s Response |
| 1. | User inputs a command to indicate they want to add a new account |  |
| 2. |  | System prompts for account type |
| 3. | User enters account type as either Savings or Checkings | If User cancels entry, the process is terminated. If User enters an invalid type of account (not Savings or Checkings) then the system prompts again. |
| 4. |  | System prompts if the account will have a preexisting customer |
| 5. | User inputs “yes” or “no” |  |
| 6. |  | If “yes”, then System prompts for a customer\_number and then uses inputted to find a match, if no match is found the System prompts again. If “no”, then System creates a new customer based on User inputs (see Table 2) |

Table 4. Use Case for User Processing a Transaction

|  |  |  |
| --- | --- | --- |
| Step | User’s Action | System’s Response |
| 1. | User inputs a command to indicate they want to process a transaction (also specifies the type of transaction). |  |
| 2. |  | System calls appropriate type of transaction function which then prompts for customer number and account number. |
| 3. | User inputs customer number and account number. |  |
| 4. |  | System searches an account with the account number, then checks to see if the associated customer is the same within the Bank accounts array. If a match is found, the process is continued. |
| 5. |  | System prompts for what type of transaction and amount. |
| 6. | User inputs type of transaction and amount. |  |
| 7. |  | System calls the account’s appropriate make transaction function, which adjusts the balance accordingly and display’s the change in balance. |

Table 5. Use Case for User Searching for an Account

|  |  |  |
| --- | --- | --- |
| Step | User’s Action | System’s Response |
| 1. | User indicates they want to find an account |  |
| 2. |  | System calls get\_account() in the Bank object to search for the account, prompts for customer number and account number. |
| 3. | User inputs customer number and account number |  |
| 4. |  | Searches through the Bank array of accounts to find a match. If a match is found the account is returned. The account is then displayed using its to\_string() method |

UML Diagram

|  |
| --- |
| Bank |
| - vector<Account\*> accounts |
| + void add\_account()  + void make\_deposit()  + void make\_withdrawal()  + Account\* get\_account() |

|  |
| --- |
| Account |
| # Customer\* customer  # float balance  # int account\_number  # bool a\_type  # vector<transaction\*> transactions |
| + float get\_balance()  + Customer\* get\_customer()  + string to\_string()  + void set\_customer(Customer updated)  + void set\_balance(float new\_balance)  + void virtual deposit(float amount)  + void virtual withdraw(float amount)  + int get\_account\_number()  + bool get\_type() |

|  |
| --- |
| Savings\_Account |
| + void add\_interest()  + void add\_charge() |

|  |
| --- |
| Transactions |
| - int customer number  - bool transaction\_type  - float amount  - string fees |
| + void process\_tran(int c\_num, bool type, float amount, string list\_of\_fees)  + string get\_type()  + int get\_amount()  + string get\_fees()  +int get\_customer() |

|  |
| --- |
| Student |
| -const float SAVINGS\_INTEREST  -const float CHECK\_INTEREST  -const float CHECK\_CHARGE  -const float OVERDRAFT\_PENALTY |
| + float get\_savings()  + float get\_check()  + float get\_charge()  + float get\_penalty() |

|  |
| --- |
| Adult |
| -const float SAVINGS\_INTEREST  -const float CHECK\_INTEREST  -const float CHECK\_CHARGE  -const float OVERDRAFT\_PENALTY |
| + float get\_savings()  + float get\_check()  + float get\_charge()  + float get\_penalty() |

|  |
| --- |
| Senior |
| -const float SAVINGS\_INTEREST  -const float CHECK\_INTEREST  -const float CHECK\_CHARGE  -const float OVERDRAFT\_PENALTY |
| + float get\_savings()  + float get\_check()  + float get\_charge()  + float get\_penalty() |

|  |
| --- |
| Checking\_Account |
| + void add\_interest()  + void add\_charge() |

|  |
| --- |
| Customer |
| # string name  # string address  # int age  # long int telephone\_number  # int customer\_number |
| + void set\_name(string new\_name)  + void set\_address(string new\_address)  + void set\_age(int new\_age)  + void set\_telephone(long int new\_phone)  + string get\_name()  + string get\_address()  + int get\_age()  + long int get\_telephone()  + int get\_customer\_number() |

PseudoCode for Bank Project

Algorithm for bank Function in main.cpp

* Initialize a bank object containing no accounts
* while command is not “exit”
  + Prompt user for a command
  + Read user’s command
    - If command is “new account”, Call Bank’s add\_account function to execute the command
    - If command is “make withdrawal”, Call Bank’s make\_withdrawal function to execute the command function
    - If command is “make deposit”, Call Bank’s make\_deposit function to execute the command
    - If command is “display account”, Call Bank’s get\_account function to execute the command
    - If not any of these, report to the user that it is not known

Algorithm for get\_account function in Bank

* while true
  + prompt user to enter an account number
  + prompt user to enter a customer id
  + read user’s inputted account number
    - search in Bank’s account array for an account that has a matching account number and customer number
      * if none are found
        + report to user that no matching account was found
        + return null pointer
      * if one was found,
        + report to user that a matching account was found
        + return account pointer

Algorithm for add\_account function in Bank

* prompt the user for type of account (either “savings” or “checking”)
* declare bool variable unique that is False
* while not unique,
  + declare variable new\_a\_id that is a random value between 9999 and 1000
  + set unique to True
  + if new\_a\_id is equal to any account numbers in the accounts array of bank
    - Set unique to False
  + else, continue
* while true,
  + prompt the user if the new account will have an already existing user
  + if no,
    - prompt the user to input a name
    - prompt the user to input an address
    - declare input\_age to be -1
    - while input age is less than 0 or greater than 125,
      * prompt the user to input age into input\_age
    - declare input\_telephone to be -1
    - while input\_telephone is less than 100 or greater than 999999999999999,
      * Prompt the user to input phone number into input\_telephone
    - if input\_age is less than 18,
      * if the type of account inputted is “savings”,
        + create a new account with a new student customer that has new\_a\_id for account number
      * if the type of account inputted is “checking”,
        + create a new account with a new student customer that has new\_a\_id for account number
    - if input\_age is greater than or equal to 18 and less than 65,
      * if the type of account inputted is “savings”,
        + create a new account with a new adult customer that has new\_a\_id for account number
      * if the type of account inputted is “checking”,
        + create a new account with a new adult customer that has new\_a\_id for account number
    - if input\_age is greater than 65,
      * if the type of account inputted is “savings”,
        + create a new account with a new senior customer that has new\_a\_id for account number
      * if the type of account inputted is “checking”,
        + create a new account with a new senior customer that has new\_a\_id for account number
    - display the customer and account numbers
  + If yes,
    - Prompt for customer\_id to identify the customer
    - If an account has a customer with customer number equal to the inputted customer\_id,
      * If the type of account inputted is “savings”,
        + Make a new savings account with customer pointer found
        + Break loop
      * If the type of account inputted is “checking”,
        + Make a new checking account with customer pointer found
        + Break loop
      * display the customer and account numbers
    - If none were found, report to the user none were found

Algorithm for make\_withdrawal function in Bank

* call Bank’s get\_account function to access the account pointer
* if account is NULL /\*none were found\*/,
  + report to the user that it is exiting the function
  + exit function
* while true
  + prompt the user to input transaction amount greater than 0
  + if user input is less than or equal to 0
    - report to user that the amount is invalid
    - continue loop
  + If user input is greater than 0
    - display the current balance
    - call account’s withdraw function with user input as argument
    - display the current balance /\*should have changed\*/
    - break loop

Algorithm for make\_deposit function in Bank

* call Bank’s get\_account function to access the account pointer
* If account is null /\*none were found\*/,
  + Report to the user that it is exiting the function
  + Exit function
* while true
  + Prompt the user to input transaction amount greater than 0
  + If user input is less than or equal to 0
    - Report to user that the amount is invalid
    - continue loop
  + If user input is greater than 0
    - call account’s withdraw function with user input as argument
    - break loop

Algorithm for deposit function in Savings and in Checkings//

* adds the argument (a float) to the balance and any interest
* adds a new transaction in the transactions array
* calls the new transaction’s process\_tran() function with the account’s customer id, 0 for type, argument for amount, and string of fees

Algorithm for withdraw function in Savings and in Checkings//

* subtracts the argument (a float) to the balance and any fees
* adds a new transaction in the transactions array
* if balance is less than 0,
  + subtract the balance by the customer’s overdraft penalty
  + set list\_of\_fees to “overdraft penalty”
  + calls the new transaction’s process\_tran() function with the account’s customer id, 1 for type, argument for amount, and list\_of\_fees
* else,
  + set list\_of\_fees to “no unusual fees”
  + calls the new transaction’s process\_tran() function with the account’s customer id, 1 for type, argument for amount, and list\_of\_fees

Algorithm for add\_interest function in Savings

* set variable interest to be (Customer’s SAVINGS\_INTEREST)
* return interest

Algorithm for add\_interest function in Checkings

* set variable interest to be (Customer’s CHECK\_INTEREST)
* return interest

Algorithm for add\_charge in Savings and Checking

* set variable interest to be (Customer’s CHECK\_CHARGE)

Algorithm for process\_tran function in Transaction (has arguments for necessary information)

* set the transaction’s customer number to be the customer number of the account
* set the transaction type to be True for deposit, or False if withdrawal
* set amount to be the amount the balance has changed
* set the fees to string describing unusual fees made in the transaction

Algorithm for to\_string function in Account

* declare string info
* add (“Customer group: ” + [customer type] + [new line]) to info
* add (“Address: ” + [customer’s address] + [new line]) to info
* add (“Age: ” + [customer’s age] + [new line]) to info
* add (“Phone number: ” + [customer’s telephone\_number] + [new line]) to info
* add (“Customer ID: ” + [customer’s number] + [new line]) to info
* add (“Account ID: ” + [account number] + [new line]) to info
* add (“Balance: ” + [balance] + [new line]) to info
* declare i = 0
* while i is less than length of transactions,
  + add (“Transaction # ” + [i] + “: ” + [transaction’s customer id] + [transaction type] + [transaction amount] + [transaction’s list of fees]) to info
  + set i to i +1
* return info

Bank Data Storage Description

Account numbers and customer id numbers will be randomly generated with a check to make sure that it is not a duplicate by comparing it to other accounts and customer ids (a new account number will compare other accounts, same for customer id). To ensure a consistent size of numbers, account numbers and customer ids will be about 4 digits long (1000 to 9999). While this will limit how many accounts and customers the Bank can store, this will produce a large enough quantity that it will suffice for the scope of this project. To store these numbers, the accounts will have individual numbers while customer id’s will be tied to the customer object that the account contains, and when a new account is made it will point to the same customer object (assuming the new account is using an already existing customer). This will also be how the accounts and customer information will be linked. To link the transactions to their respective customers, this will be done by having a search ability where the which will find all the accounts that have the same customer and then access the account transactions array.