Banking App Design Document
CS-273-1
Kailey Cozart

Requirements Specification

I will be building a computer program that contains all of the classes and functions that would be needed in a regular bank's operations. The program should allow users to add new customers and accounts to the program. Likewise, the user will be able to make transactions. For example, deposits and withdrawals could be made to both the saving and checking accounts of a particular customer. Certain fees would be added to the transactions, along with the amount of money specified for the transaction. Likewise, interest will be added on to both the saving and checking accounts. Users should be allowed to see and change their name, address, age, telephone number, and they should be able to see their customer number and account number. Users should also be able to see their account balances. Additionally, users should be allowed to check the interest, charges, and overdraft penalties on their accounts.

Use Cases

Use Case for Adding a New Customer or Account

1. The user selects the menu option that tells the program to add a new account.
2. The system prompts the user to enter the customer's name.
3. The user enters in the customer's name.
4. The system prompts the user to choose what type of account they would like to add.
5. The user selects the type of account that they would like.
6. If the customer exists, the system adds a new account.
7. If the customer does not exist, the user is prompted to enter their address, age, telephone number, and type of customer.
8. The user inputs their address, age, telephone number, and type of customer.
9. The system gives a message to the user, telling them the account creation has been successful.
10. If the operation is unsuccessful, the process is terminated and the user is informed that there was a failure in creating the account.
11. The user is returned to the main menu.

Use Case for Viewing a the Number of Accounts a Customer Has

1. The user selects the option to view how many accounts they have.
2. The system prompts the user to enter their name.
3. The user enters their name.

4. The system gets the accounts, counts them, and displays how many accounts the user has.

Use Case for Depositing Funds

1. The user selects the option to make a deposit to a certain account number.
2. If the user enters an invalid amount or account or cancels the transaction, the process is terminated.
3. The system adds the transaction to the account's array of transactions. The transaction is given the customer number associated with the account, the amount, fees, and the type of transaction (which is a deposit in this case).
4. The system processes the transaction, adding the amount minus the fees to the account's balance.
5. The system outputs the transaction that has taken place and then returns the user to the main menu.

Use Case for Withdrawing Funds

1. The user selects the option to make a withdrawal out of a certain account number.
2. If the user enters an invalid amount or account or cancels the transaction, the process is terminated.
3. The system adds the transaction to the account's array of transactions. The transaction is given the customer number associated with the account, the amount, fees, and the type of transaction (which is a withdrawal in this case).
4. The system processes the transaction, subtracting the amount plus the fees to the account's balance.
5. The system outputs the transaction that has taken place and then returns the user to the main menu.

Use Case for Checking Account Balances

1. The user selects the option to view a certain account's information/balances.
2. The system responds by finding and displaying the balance of that particular account.
3. When the user selects the option to leave, they are returned to the main menu.

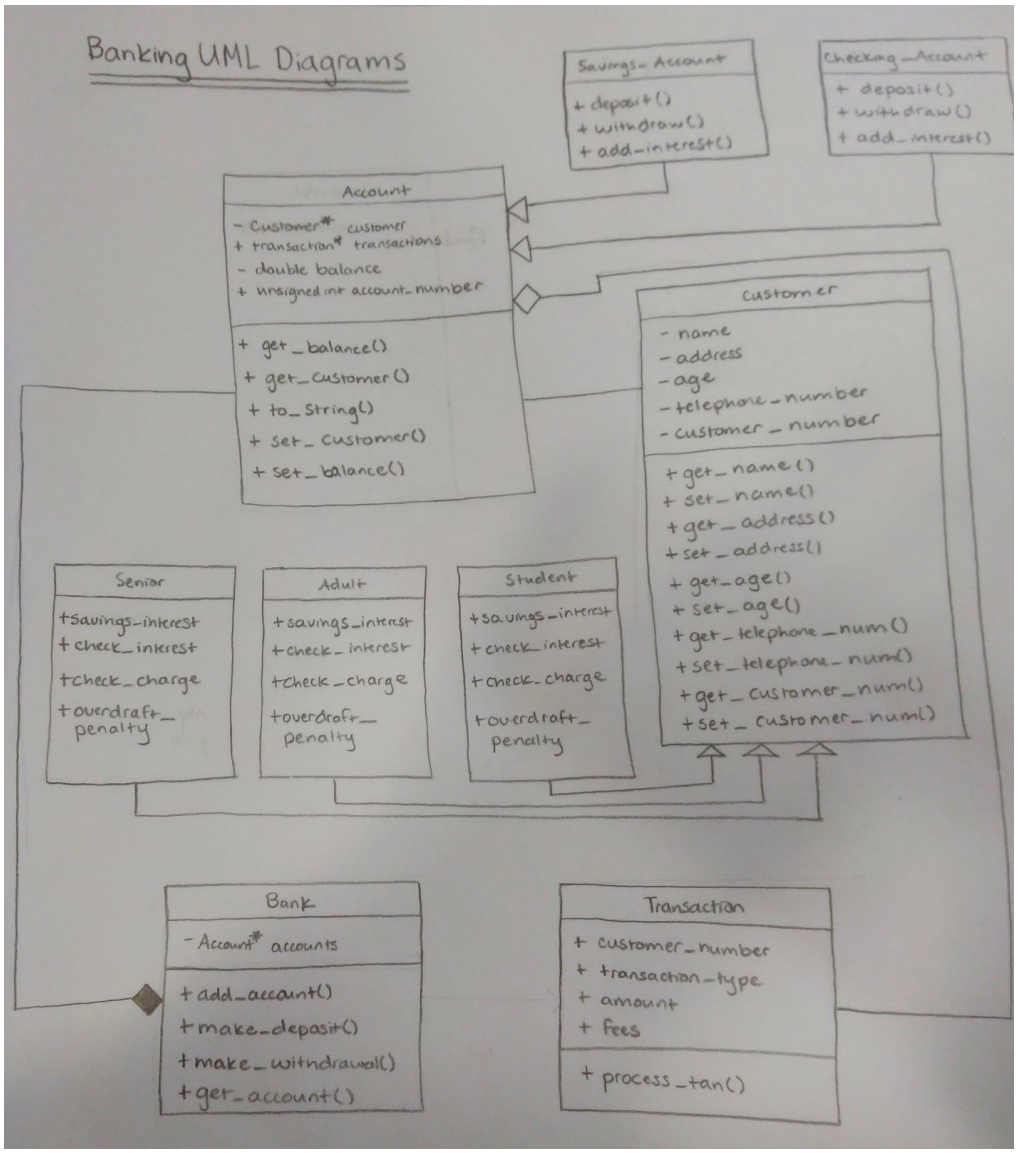
Use Case for Viewing Transaction History

1. The user selects the option to view their account transaction history.
2. The system prompts the user to enter their account number.
3. The user enters their account number.
4. If the user cancels the operation, the process terminates.
5. The system responds by displaying the transactions that have taken place.
6. When the user selects the option to leave, they are returned to the main menu.

Use Case for Viewing or Editing Customer Information

1. The user selects the option to view their information.
2. The system prompts the user to enter their account number.
3. The user enters their account number.
4. If the user cancels the operation, the process terminates.
5. The system displays the customer information to the screen.
6. If the user selects a certain piece of information, the system allows the user to change that piece of information.
7. If the user selects the option to leave, they are returned to the main menu.

UML Diagrams



Pseudocode

Algorithm for Add_Account() method in Banking_Application.cpp

1. Prompt user to enter name
2. Save the name in a string
3. Prompt user to enter which account type they desire to create
4. Save the account type in a variable
5. Create an account object and attempt to add it to the bank's array of accounts using the customer's name and account type as parameters
6. If the account does not exist
7. Prompt user to enter address
8. Save the address in a string
9. Prompt the user to enter telephone number
10. Save the telephone number in an string
11. Prompt the user to enter age
12. Save the age in an integer
13. Prompt the user to enter the customer type
14. Save the customer type in a variable
15. Add the account to the bank's array of account (the above data are parameters)
16. If the account does exist
17. Tell the user what their new account ID is
18. Else
19. Tell the user that the account was not successfully created

Algorithm for make_deposit() methods in Bank.h and Banking_Application.cpp

1. Create an account number variable
2. Prompt user to enter account number
3. Save the user's entered number into the account number variable
4. Create and amount variable
5. Prompt user to enter the amount
6. Save the user's entered amount into the amount variable
7. Call the bank's make_deposit() function using the amount and account number as parameters

Algorithm for make_withdrawal() method in Banking_Application.cpp

1. Create an account number variable
2. Prompt user to enter account number
3. Save the user's entered number into the account number variable
4. Create and amount variable
5. Prompt user to enter the amount

6. Save the user's entered amount into the amount variable
7. Call the bank's make_withdrawal() function using the amount and account number as parameters

Algorithm for Overloaded add_account() methods in Bank.h

Parameters: Customer and Account Type

1. Create an account object
2. If the account type is savings
3. Create a savings account
4. If the account type is checking
5. Create a checking account
6. Return the account number

Parameters: Name and Account Type

1. Call find_customer and the customer object with the parameter name is returned
2. If the customer is not found
3. Return NULL
4. Return the add_account function passing the customer and the account type as parameters

Parameters: Name, Address, Telephone Number, Age, Customer Type, Account Type

1. A customer object is created
2. If the customer type is adult
3. Create an adult object
4. Else if the customer type is student
5. Create a student object
6. Else if the customer type is senior
7. Create a senior object
8. Add the customer to the customers vector
9. Return the add_account function passing the customer and the account type as parameters

Algorithm for get_account() method in Bank.h

1. Calls the function find_accounts_by_name using the name of the customer as a parameter

Bank Data Storage Description

The account numbers will be unique account numbers that we create in the add_account() method in Bank.h. (The add_account() method that uses the customer and account type as parameters.) In order to make the numbers unique, a certain number of possible account numbers could be pre-generated into a list, randomly selected, and then removed from the list when they have been used to create an account object. An account number that has been used to create an object will be stored in that account object's account_number variable. Moreover, the account objects are stored in the bank object's vector of accounts, and the account numbers can be accessed with the bank's get_account() method.

The customer id numbers will also be unique numbers that we create in the add_account() method in Bank.h. (The add_account() method that uses name, address, telephone number, age, customer type, and account type as parameters.) To make the customer id numbers unique, a certain number of possible customer ID numbers could be pre-generated into a list, randomly selected, and then removed from the list when they have been used to create a customer object. A customer id number that has been used to create an object will be stored in that customer object's customer_number variable. Also, the customer objects are stored in the bank object's vector of customers, and the customer numbers can be accessed when the bank accesses the customer object's get_customer_num() method.

The accounts will be linked to customers because each account has a customer object that is used when that account object is created. The transactions will be linked to the accounts because each account has a vector of transaction objects. Thus, the customers will be linked to the transactions because the customer and transaction objects are all in the account object.

GitHub Repository Link

https://github.com/KaileyCozart/Data-Structures-HW05-Inheritance-Polymorphism