Class Relationships and Separation of Concerns

Due Date: (See Slate\ Assignments)
Date: November 2022
Type: **Individual Assignment**
Weight: **10%**

Summary

Using the Object-Oriented UML model provided, create an Object-Oriented program that creates a simple banking application which allows the user to open an account, select an account to withdraw and deposit money, check balance. Analyze the partial class diagram to determine the structure of the application. Practice control statements to branch and repeat as needed. Practice the principles of separation of concerns to divide the business logic appropriately between classes and to create additional methods as needed.

Submission checklist (Please submit one zip file containing the Report Document, the Visual Paradigm Project, the Python Project, and the Git repo)

1. An assignment report created as a Word Document that has
    a. A title page with your name, date and assignment title, page numbers, and sections to identify the answer of each of the assignment parts.
    b. Requirements for the report are provided in the assignment.
    c. The assignment report shall be written in 3rd person and shall be understandable on its own without the need for the reader to know or reference the assignment description.
2. UML Class Diagram created using Visual Paradigm
3. The project folder including all source files and GIT repository.
4. Link to the private BitBucket repository containing the project with read access to shalinisingh.jaspal@sheridancollege.ca

    IMPORTANT NOTE: *Submission is done in electronic format **in SLATE. DO NOT email your submission**.*

Application Description

Create a simple banking application that allows banking operations like opening an account and performing monetary transactions. The structure of the program is shown in **Appendix 1**. Complete program structure and define all classes and their relationships as shown in **Appendix 1**.

Detailed Requirements:

Initialization: For each class, define all required field variables and methods. Define and implement required accessor and mutator methods. Initially, provide a skeleton definition of any other methods (using pass and #TODO comments). Ensure you detect and correct syntax errors early and execute your code to detect and fix any runtime errors. Commit changes often to show stepwise refinement process you have followed. Refer to **Part V Program Development Process** for a guideline.

Part I User Interaction (10 Points):

- **Class Application**: implements all the interaction between the user and the Bank. **No other class is allowed to interact (input or print) with the end user.**
    - Define a method **showMainMenu** that loops to display the following options until the user chooses to exit the application:
        1. *Open Account*: allows the user to open a new account ***To be implemented for Bonus**
        2. *Select Account*: this allows the user to enter the account number of the account they want to work with. Upon searching the account successfully, the application will call the method *showAccountMenu* to display the *Account Menu* as described next.
        3. *Exit*: allows the user to exit the application

    - Define a method ***showAccountMenu*** that loops to display the following options until the user chooses to exit the Account Menu:
        1. *Check Balance:* Display the balance of the selected account
        2. *Deposit:* Prompt the user for an amount to deposit and perform the deposit using the methods in account class.
        3. *Withdraw:* Prompt the user for an amount to withdraw and perform the withdrawal using the methods in the account class.
        4. *Exit Account: go back to Banking Main Menu*

    - Define and call the method run() to show the main menu to the end user.

Consider creating methods to implement functionality for each menu option other than *exit*.

Part II Business Logic (50 Points): The classes described in this part are not allowed to interact (input or print) with the end user.

- **Account (12 Points)**: *Represents a bank account*. It is the base class for classes SavingsAccount and ChequingAccount. Define the class as per the details provided in the class diagram.
- **SavingsAccount (12 Points)**: *Extends the class Account to represent the savings accounts*. This account requires the account holder(s) to maintain a minimum balance in the account. Override the method *withdraw* of the base class to reject the transactions that would bring the current balance of

the account below the minimum balance. E.g. if the minimum balance is 5000 CAD and the current balance in the account is 7000 CAD, the maximum withdrawal that can be allowed is 2000 CAD

- **ChequingAccount (12 Points)**: *Extends the class Account to represent the chequing accounts.* This account allows overdrafts, i.e., the account holder(s) can withdraw an amount that is more than their current balance. Generally, there is a fee associated with overdrafts, but this application will not keep a track of the fee for the sake of simplicity.

  Override the method withdraw of the base class to reject transactions that cannot be completed even after using the overdraft limit. This means if an account has an overdraft limit of 5000 CAD, the account holder is allowed to withdraw up to 5000 CAD more than the money they have in the account.

- **Bank (14 Points)**: *Implements the business logic required for the banking. Keeps track of all the accounts. Allows the user to open a new account or to search for an existing account.* The class shall define a List of Account objects. The list is to be populated with instances of SavingsAccount / ChequingAccount
  a. Define a constructor that populates the account list with hardcoded of three ChequingAccount instances and three SavingsAccount instances.
  b. Define and implement the *searchAccount()* method that accepts an *account number* as parameter. The method should find and return the account with matching account number from the list of accounts. This method is to be used by ***Application*** to retrieve an account using the *account number* entered by the user and perform transactions on the account.
  c. Define a method openAccount (**Required only for bonus**: **described in bonus section**)

Part III (10%) Error Handling. Using standard error checking (if-else) as well as exception handling (try-except) to ensure the user input is valid both in terms of range and type of data entered. The application should not crash at any point due to data input / processing. If recovery from the error(s) is not possible, inform the user about the error and terminate the application. Examples:

- Negative values for withdrawal/deposit amounts should not be allowed.
- If conversion of the user input to a number fails, the code should handle the exception.

Part IV (10%) UML Model. Create a Visual Paradigm project with the class diagram shown in **Appendix 1** updated with all the implementation details and the correct relationship details. All class symbols should include all attributes and methods. For this assignment, please label the relationships with their name: USES, HAS-A and IS-A.

Part V (10%): **Program Development Process**. The project is to be developed iteratively in small increments. Code must be version controlled using GIT. Each milestone must have at a minimum one commit at the end of the milestone. For best evaluation ensure changes are committed often (more than once per milestone) and the commit messages are informative.

Milestone 1. *Project Creation*. Create the initial application folder and add the project to version control using GIT. Use GIT effectively throughout the development of the project.

Milestone 2. *User Interaction Setup*. The application can display the two menus asking the input from the user.

Milestone 3. *Simple Bank Setup*. The application can search for accounts based on account number.

Milestone 4. *Savings Accounts Offered*. The application allows searching for Savings Accounts and performing transactions on them.

Milestone 5. *Chequing Accounts Offered*. The application allows searching for Chequing Accounts and performing transactions on them.

Milestone 6. *Bug Fixing and Polishing*. Test the application thoroughly and fix any problems you have detected

Milestone 7. *Bonus Completed*: The application allows opening and performing transactions on new chequing and savings accounts based on the choice of end user.

Part VI (10%): **Report Requirements (10%)**: In your assignment report:

1. Add a copy of your class diagram
2. Describe the application, purpose of all classes and all the class relationships.
3. Describe your experience as you progress through the development process. **Remember to describe specific difficulties encountered in each milestone and your approach to overcome them**.
4. List the limitations of the app that you would like to address in future versions.

Bonus (5%): Define a method openAccount in the class Bank. The method should accept the data required to open a new account as parameters and create a new instance of Savings or Chequing account. This new account should be added to the list of accounts in the bank. Call this method when the user selects the main menu option to *open an account*.

Notes:

5. The **professionalism of your submission**, clarity of written **communication** is extremely important. The ability to communicate your knowledge is as important as the knowledge itself.

    a. Up to 20% of the mark for any written work can be deducted due to poor presentation / communication: *title page (5%)*, *document organization (5%)*, *layout (5%)* and *grammar and spelling* (5%).

    b. Up to 20% of the mark for a program can be deducted due to poor presentation / communication: quality of names according to our *naming and coding conventions (10%)* and *comments (10%)*.

6. **All assignment shall be submitted by the deadline.** Late submissions will be penalized with 10% per day for up to 3 calendar days after which the assignment cannot be submitted anymore. **An email must be sent** should you choose to submit a late assignment.

7. This assignment shall be **completed individually**. Remember that completing the assignment by yourself will ensure your success on the midterm and final exam. See the Academic Honesty at Sheridan.

8. Submission is done in electronic format **using SLATE Dropbox**. **DO NOT email your submission.**

Appendix 1: Simple Banking Application Model

Figure 1 presents the structure of the application with the required classes and their relationships in their most basic form. This structure and specifically the list of field variables and methods can be expanded as needed. **The parameters and return values of methods have been left out on purpose. Make sure to add the parameters and return types in the Visual Paradigm Project you submit.**