Java

Object Oriented Development

Bank design project

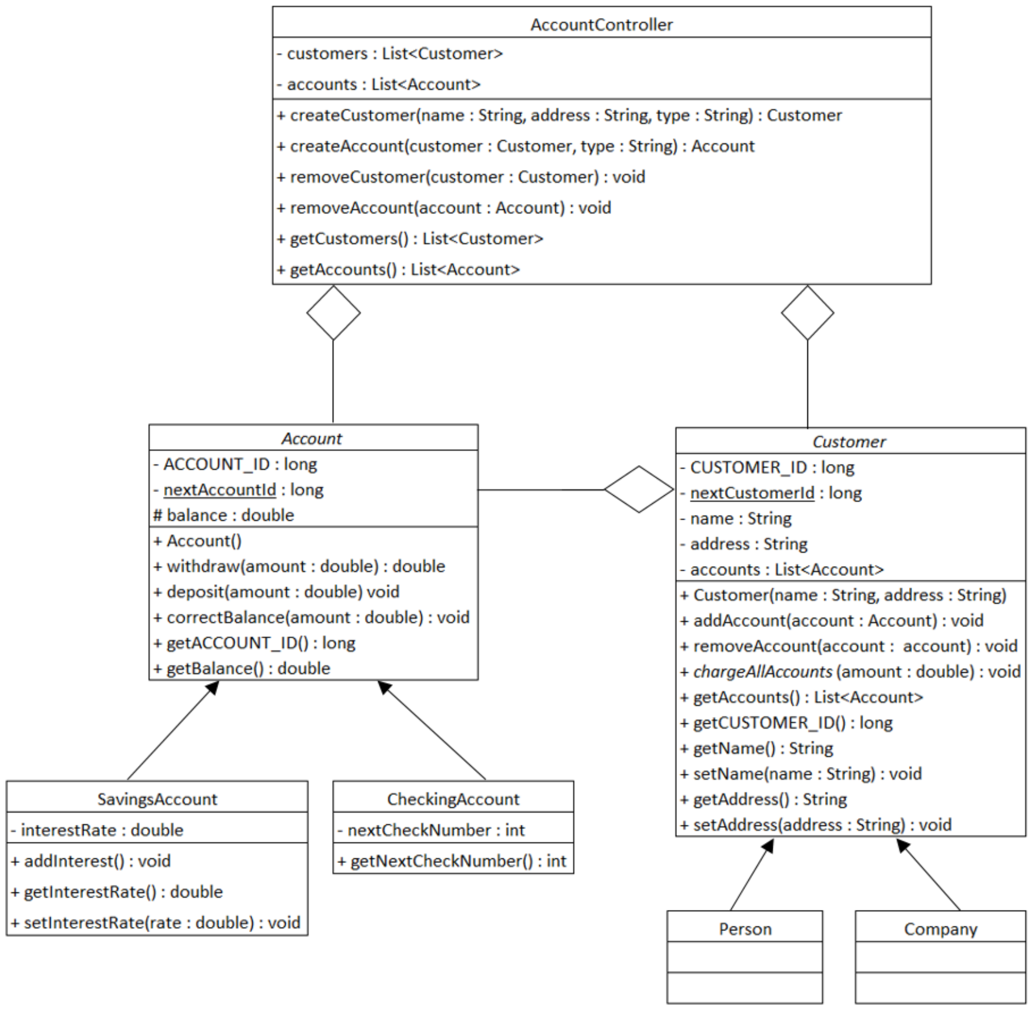
About the project

You will be writing code for an application which will be used to generate and organize data for a bank. We want to create, manipulate, and track different types of bank accounts, different types of customers, and the connections between accounts and users.

You will be working from a UML diagram showing all of the required classes, attributes and methods. Tests for the code have already been written. Your job is to create all of the classes shown in the UML and then write code which passes all of the tests.

UML

Note that the classes Account & Customer and the method chargeAllAccounts are all abstract.


AccountController
- customers : List<Customer>
- accounts : List<Account>
+ createCustomer(name : String, address : String, type : String) : Customer
+ createAccount(customer : Customer, type : String) : Account
+ removeCustomer(customer : Customer) : void
+ removeAccount(account : Account) : void
+ getCustomers() : List<Customer>
+ getAccounts() : List<Account>
Account
- ACCOUNT_ID : long
- nextAccountId : long
# balance : double
+ Account()
+ withdraw(amount : double) : double
+ deposit(amount : double) void
+ correctBalance(amount : double) : void
+ getACCOUNT_ID() : long
+ getBalance() : double
Customer
- CUSTOMER_ID : long
- nextCustomerId : long
- name : String
- address : String
- accounts : List<Account>
+ Customer(name : String, address : String)
+ addAccount(account : Account) : void
+ removeAccount(account : account) : void
+ chargeAllAccounts (amount : double) : void
+ getAccounts() : List<Account>
+ getCUSTOMER_ID() : long
+ getName() : String
+ setName(name : String) : void
+ getAddress() : String
+ setAddress(address : String) : void
SavingsAccount
- interestRate : double
+ addInterest() : void
+ getInterestRate() : double
+ setInterestRate(rate : double) : void
CheckingAccount
- nextCheckNumber : int
+ getNextCheckNumber() : int
Person
Company

Setting up

Download the project ‘bankDesignProject’ from Git Lab.

Import it into Eclipse. It will look like this:

You will be writing all of your classes within the com.iabdinur.bankDesignProject package.

Phase 1

In your com.iabdinur.bankDesignProject package, create all the classes and enumerations shown in the UML. Pay attention to detail and ensure that all attributes, methods, arguments, return types, relationships and modifiers are exactly as shown in the diagram.

For the moment **you should not write any code** within the methods unless they are getters or setters.

Do not add any additional attributes or methods that are not in the UML.

Make sure that there are no compilation errors in your classes.

Testing

There are no tests to run for this phase. However the tests you'll be running for later phases rely on you exactly matching the UML. You'll know when this has been done correctly as all the compile errors in the 9 test cases will disappear:

Step 1

When Account, SavingsAccount and CheckingAccount have all been created correctly, the compile errors will disappear from TestAccountMethods, TestAccountInitialisation, TestCheckingAccount and TestSavingsAccount.

Step 2

When Customer, Person and Company have all been created correctly, the compile errors will disappear from TestCustomerMethods, TestCustomerInitialisation, TestPerson and TestCompany.

Step 3

When TestAccountController has been created correctly, all tests will compile.

Phase 2

In this phase you'll be writing code within the methods of your classes to make the tests pass.

Account

- Each account should have a constant, unique id. Id numbers should start from 1000 and increment by 5. The ACCOUNT_ID attribute should be initialised when the account object is created. The id should be generated internally within the class, it should not be passed in as an argument.
- The deposit method should increase the balance by the value passed in as an argument.
- The withdraw method should reduce the balance by the value passed in as an argument. It should return the value of the argument
- The account should be able to go overdrawn
- The correctBalance method should change the balance to match the value passed in as an argument.

Testing your code for Account

- Run TestAccountInitialisation to check that the ID has been set correctly.
- Run TestAccountMethods to check that all of the methods work.

Customer

- Each customer should have a constant, unique id. Id numbers should start from 2000000 and increment by 7. The CUSTOMER_ID attribute should be initialised when the account object is created. The id should be generated internally within the class, it should not be passed in as an argument.
- The customer's name and address should be set when the customer object is created.
- The addAccount() method should take an account object as an argument and add it to the list of accounts stored within the customer.
- The removeAccount() method should take an account object as an argument and remove it from the list of accounts stored within the customer.

Testing your code for Customer

- Run TestCustomerInitialisation to check that id has been set correctly.
- Run TestCustomerMethods to check that the addAccount() and removeAccount() methods work and that the name and address have been set correctly.

SavingsAccount

- It should not be possible for the savings account to go overdrawn. In the case that an amount larger than the balance is passed into the withdraw method, nothing should be subtracted from the balance and zero should be returned.
- The addInterest() method should calculate the interest due on the account and add it to the balance. The formula to calculate the interest due is balance * interest rate / 100.

Testing your code for SavingsAccount

- Run TestSavingsAccount.

CheckingAccount

- The getNextCheckNumber() method should return the value of the nextCheckNumber variable. The first check should have number 1. Each subsequent check number should count up by one.

Testing your code for CheckingAccount

- Run TestCheckingAccount

Person

- The chargeAllAccounts method should be implemented so that the amount passed in is subtracted from from the balance of each of the person's accounts. (1 mark)

Testing your code for Person

- Run TestPerson

Company

- The chargeAllAccounts method should be implemented so that the amount passed in is subtracted from the balance of all of the company's checking accounts. The amount subtracted from the balance of all of the company's savings accounts should be **double** the amount passed in.

Testing your code for Company

- Run TestCompany

AccountController

- The createCustomer method takes a name, address and customer type as arguments.
    - If the type is "person", createCustomer should create a Person object with the correct name and address. It should add the Person object to the account controller's list of Person objects. It should return the Person object.
    - If the type is "company", createCustomer should create a Company object with the correct name and address. It should add the Company object to the account controller's list of Company objects. It should return the Company object.
- The createAccount method takes a Customer object and an account type as arguments.
    - If the type is "checking", createAccount should create a CheckingAccount object, add the object to accountController's list of accounts and to the customer's list of accounts. It should return the CheckingAccount object.
    - If the type is "savings", createAccount should create a SavingsAccount object, add the object to accountController's list of accounts and to the customer's list of accounts. It should return the SavingsAccount object.
- The removeAccount method takes an Account object as an argument.
    - It should remove the account from the account controller's list of accounts.
    - It should remove the account from the customer's list of accounts.
- The removeCustomer method takes a Customer object as an argument.
    - It should remove the customer from the account controller's list of customers.
    - It should remove all of the customer's accounts from the account controller's list of accounts.

Testing your code for AccountController

- Run TestAccountController

Marking

Phase 1

- Classes match the UML.
- Abstract keyword is used for the classes and method specified in the UML.
- Final, static and protected keywords are used where specified in the UML.

Phase 2

- Account class correctly implemented.
- Customer class correctly implemented.
- SavingsAccount class correctly implemented
- CheckingAccount class correctly implemented
- Person class correctly implemented
- Company class correctly implemented
- AccountController class correctly implemented

Clean code implementation