**University of Wolverhampton**

**Faculty of Science and Engineering**

**Department of Mathematics and Computer Science**

**Module Assessment**

|  |  |
| --- | --- |
| **Module** | 4CS021 – Introduction to Object-Oriented Programming |
| **Module Leader** | Hiran Patel |
| **Semester** | 1 |
| **Year** | 2023/24 |
|  |  |
| **Assessment** | Portfolio |
| **% of module mark** | 100% |
| **Due Date** | See canvas portfolio submission gateway |
| **Hand-in – what?** | **Portfolio as specified in this document** |
| **Hand-in- where?** | Canvas |
|  |  |
| **Pass mark** | 40% |
| **Method of retrieval** | Submit the resit assessment (will be distributed at the end of the module) by end of resit week (July) |
| **Feedback** | Individual feedback via Canvas, in addition verbal feedback is available in class. |
| **Collection of marked work** | N/A |

(Before you read the assessment brief, watch the assessment video which explains how the end portfolio will look).

This assessment will be split into 3 sections. For each section, you will be provided with a Unified Modelling Language (UML) Class diagram to show what functions you require in each class along with whether the class needs to inherit another class. Refer to the lecture/lecture notes on how to read a UML class diagram.

The first part (65%) will test your knowledge of object-oriented programming using Java. For this part, you will create the classes and functions required for a simple banking system. For this section, you will be required to create the following classes (brief description):

1. Customer class – will hold basic customer information such as the first name and last name.
2. Account class – will hold information such as the account number, account balance and will extend the customer class to allow customer details to be used within the constructor. This class will also hold functions such as deposit and withdraw.
3. Transaction class – will hold a method called transfer to allow accounts to transfer money between them.
4. Main – where you will create accounts and test your functions from the account and transfer class.

Part 2 of the assessment (20%) is the addition of file input. Rather than creating the accounts in the main method manually, you will read from a csv file provided on Canvas. This file has simply 4 rows, each containing a first name, last name, account number and balance. This will involve an additional class:

1. ReadAccounts class – will read the data and contain 4 functions to return a list of first names, last names, bank accounts and balances.

Part 3 of the assessment (15%) is the addition of creating a graphical user interface (GUI). The GUI will hold all the features required to activate some of the functions used to text your banking system. For example, you will have text fields to allow users to type in the amount they wish to transfer/withdraw from which account (account number will also need to be an input). This will also involve an additional class:

1. GUI class – hold the different attributes such as buttons, text fields and labels. You will set the layout in this class of how you want the interface to look along with setting up the functionality. For example, setting the action for what happens when each button is clicked or changing the text of the labels.

Below shows the 3 different parts of the portfolio in the form of UML diagrams. Each one builds from the previous.

**A picture containing table

Description automatically generatedPart 1 (65%) – Basic banking system UML class diagram**

**Customer class**

You should create a class called Customer. In here, create two string variables named firstName and lastname. These should be set to private variables (-). This class should contain 4 functions:

* getFirstName – should return the firstName variable
* getLastName – should return the lastName variable
* setFirstName – should set the global variable to FName
* setLastname – should set the global variable to LName

These functions will allow us to set and get the names of a specific account when created in the account class.

**Account class (extends Customer class)**

The next class you should create is the account class. This should inherit the variables and functions from the customer class. Once you have extended this class to use the customer class, you should create 2 new variables; balance and accountNumber. Both should be an integer. You then need to create the following methods:

* Account – this is a constructor which will be called when an object has been created using this class. When an object of this type has been created in the main method or elsewhere, it will require 4 parameters. The first two are linked to the customer class. You should take the FName and LName and use the set methods from the customer class which can be used within the account class without an object to be created to set firstName and lastName. The other 2 parameters should be used to set the global variables balance and accountNumber. You don’t need a function to do this, simply set them equal to whatever is passed into the parameters.
* getBalance – this should return the global variable balance
* getAccountNum – this should return the global variable accountNumber
* deposit – this function has a parameter called “amount”. Whatever is passed into this “amount” variable should add onto the previous balance. For example, if the balance is set to 1000 when an account has been created, and the function deposit(100) is called, this should add 100 to 1000.
* withdraw – this function has a parameter called “amount”. Whatever is passed into this “amount” variable should subtract from the previous balance. For example, if the previous balance is 1000 and the function withdraw(55) is called, 55 should be subtracted from the 1000.

**Transaction class**

The third class you need create is the transaction class. In this class, you only need to create one function called transfer. Transfer takes in 3 parameters, the first two are accounts and the last is the amount you wish to transfer. The first parameter is the source account, and the second parameter is the destination account. Money from one account will transfer to the other and the amount is specified by the third parameter. As the first 2 parameters are objects, you will need to create 2 accounts initially (this will be done in the main) and you should pass those into the first and second parameter. The transfer method should grab the current balance of the first account and minus the amount, the next part is to grab the balance of the second account and add the amount and this will complete the transfer.

**Main class**

The main class is where you will do all your testing to see whether your methods have been implemented correctly. Complete the following steps to make sure your banking system is working correctly:

1. Create an object from Account called account1. When you create this object, it should force you to add 4 values in the parameters as you create the object. You can set the values to whatever you want but make sure you use the correct variable types. It should throw an error in eclipse if you are using the wrong types. For account 1, use “Jeffrey” as the first name, “Ting” as the last name, 1 as the account number and 2000 as the balance.
2. Create an object from Account called account2. Use “Hiran” as the first name, “Patel” as the last name, 2 as the account number and 1000 as the balance.

(NOTE – in this module, you are learning the concepts of OOP and using a banking system as an application. In a traditional banking system, an account number cannot be duplicated but with our current system, you can set the same account number. For testing purposes, DO NOT set the account number for two accounts the same as this will cause confusion for those who want to complete part 2 and 3).

1. Print out the balance of account1
2. Print out the balance of account2
3. Call the deposit function on the account1 object and deposit 250
4. Print out the balance of account1 – does it print out 2250?
5. Call the withdraw function on the account2 object and withdraw 500
6. Print out the balance of account2 – does it print out 500?

(The balance for account1 now should be 2250 and the balance for account2 should be 500)

1. Create an object called “t” from the Transaction class. This only has a default constructor which contains no parameters.
2. Called the transfer method using “t” and add account1 as parameter 1, account2 as parameter 2 and for the amount, set it to 250. This should now transfer 250 from account1 and add it to account2.
3. Print out balance of account1
4. Print out balance of account2

The final balance of account1 should be 2000

The final balance of account2 should be 750

***You can change the values above to whatever you want for testing, the above is just an example of how you can test your methods. You should try calling different functions on different lines and see what happens. Make sure you thoroughly test your methods!!***

**Part 2 (20%) – Banking System using a CSV file**

Graphical user interface

Description automatically generated with medium confidenceThis is an addition of Part 1 so please make sure you have the “simple banking system” working. For this part, you do not need to write to a CSV to update any bank accounts however, rather than creating manual accounts by creating objects from the Account class, you will generate a list of accounts by reading them from a CSV file. On Canvas, you will be given a CSV file called Accounts.csv. This file will contain 4 rows, each containing the same data used in part 1 for each account. These values are the first name (column 1), last name (column 2), bank account (column 3) and the bank balance (column 4). Make sure you DO NOT change the format of these rows, keep them the same! You may wish to add one or multiple more accounts but DO NOT change the format (order of data).

The above UML class diagram above shows slight modifications to the Main and the addition of the ReadAccounts class.

The ReadAccounts class holds 5 functions. The constructor “method” takes in a string called URL which is simply the path to the file. When creating this object in the main, set this parameter to the path (as a string) of the file you wish to use, in this case, it will be set to Accounts.csv if using the file from canvas. The other 4 functions should simply return a list of first names, last names, accounts and balances as a LinkedList. 2 of the 4 functions will return a string and the other two will be integers. In the main method, you will call these functions individually and set them equal to a linkedlist which will then be used to create new accounts.

**Main class for part 2**

In part 1, your main method included 2 manually created accounts where you set the information using the constructor method. You will do something similar but this time, create a LinkedList of accounts and then use the 4 returned LinkedList to create the 4 accounts.

Analyse the UML diagram above for the main method and implement the following:

1. Create a LinkedList of accounts
2. Create an object called readAccounts from the ReadAccounts class. Pass in the “file” variable as a parameter to the constructor
3. Create a LinkedList of First Names and call it firstnames. You should use the getFirstNames() function to return all the first names into this list.
4. Create a LinkedList of Last names and call it lastNames. You should use the getLastNames() function to return all the last names into this list.
5. Create a LinkedList of accounts and call it accountList. You should use the getAccounts() function to return all the accounts into this list.
6. Create a LinkedList of balances can call it balanceList. You should use the getBalanced() function to return all the balances in this list.

(At this stage, you should have 4 LinkedList full of individual information that you will now use to create some accounts).

1. Using the LinkedList of accounts called “accounts,” you need to populate these by using a loop (preferably a for loop). By setting the limit of the for loop to the size of one of the LinkedLists populated (which will be the amount of rows in the csv), you can then add an account by using accounts.add(new Account(…….)). The space in between the brackets of the “new Account()” is what you will need to populate by grabbing each element of the firstNames, lastNames, accountList and balanceList.
2. Use the same testing procedure as part 1 but this time, you have 4 accounts (if you used the standard Account.csv file from canvas). This means you have 4 different accounts to work with. To grab each account, you can use account.get(index) to grab a specific account. This can then be followed by a function such as deposit, withdraw or if you wish to make a transfer, then make sure you make a transaction object.

**Part 3 (15%) – Integration of a Graphical User Interface (GUI)**

The final section of the assessment requires you to build a basic graphical user interface (GUI) for your banking system. This section takes all of part 2 and adds on a GUI class where you will create your buttons, text fields, labels, layout and calling the functions from the Account class and Transaction class. Bloe shows the UML class diagram for part 3:

A picture containing text

Description automatically generated

As you can see, there is a much larger class called GUI with slight modifications to the main class to enable visibility of the GUI. There is also a sub-class being used within the GUI class to create functionality on the features within the GIU. For example, when we click a button, it will call the actionPerformed(ActionEvent e) method where we can modify the actions after a button has been pressed. The “implements ActionListener” section after declaring the sub-class name forces this class to use the methods within ActionListener which is an interface (not a class), in this case, the actionPerformed method. If you’re unsure what an interface is, refer back to the lectures around creating and using a GUI. Here are the list of variables and functions within the class with a brief description of what they must do:

1. transferObject – this is similar to the transfer object used within the main for part 1 and part 2. In this section of the assessment, you will use this object in the GUI class to perform the transfer feature of the banking system
2. sbAllData : StringBuilder – this StringBuilder object called sbAllData will be used as the variable to hold all the account data in a long string
3. globalAccounts – this is a global LinkedList which will hold all the accounts from the CSV file. The CSV will be read in the main method, passed into the constructor and then this global variable will hold the same data. You will need this LinkedList later on to be used within the actionPerformed function
4. showAllData – This is a label which will show all the account data
5. showAllButton – this button will trigger the showAllData label to show all the data
6. depositButton – this button will take whatever is typed into the accDeposit and depositInput text fields and deposit a certain value to the appropriate account
7. withdrawButton - this button will take whatever is typed into the accWithdraw and withdrawInput text fields and withdraw a certain value from the appropriate account
8. transferButton – this button will take the data from acc1Transfer, acc2Transfer and transferAmount text fields and perform the transfer function. This should take the transferAmount from acc1Transfer and transfer it to acc2Transfer
9. accDeposit – Text Field to hold account number for the deposit feature
10. accWithdraw – Text Field to hold account number for the withdraw feature
11. acc1Transfer – Text field to hold the account number for account 1 to be used to transfer money from
12. acc2Transfer – Text field to hold the account number for account 2 to be used to transfer money to
13. depositInput – Text field to hold the amount to be deposited into an account
14. withdrawInput – Text field to hold the amount to be withdrawn from an account
15. transferAmount – Text field to hold the amount of money to be withdrawn from account 1 and deposited into account 2
16. GUI (accounts : LinkedList<Account>) – Constructor method for the GUI class. In this method, you will initialise all of the GUI features such as labels, buttons and text fields and store data for the sbAllData variable. Within this method perform the following:
    1. When you create the constructor for this class, you can name the frame of the GUI using the super(“title”) method. So firstly, call super() and name the frame whatever you’d like, I would suggest something like “banking system.”
    2. The next line after you’ve set a title, call setLayout(null); This allows you to set the layout how you would like by setting the bounds on each feature such as buttons, text fields and labels later on.
    3. Set the global linkedList variable (globalAccounts) to the passed in LinkedList variable. This now populates that global variable with all of the accounts passed in from the main
    4. Using a for loop, populate sbAllData with the account data (global variable) in a suitable/readable format
    5. Initialise all of buttons by calling the constructor which allows you to label them. For example, use showAllButton = new JButton(“Show all”) to add “Show All” on top of the button. Do this for every button that you’re using.
    6. Use the .setBounds(int x, int y, int width, int height) method to position and set a size of your buttons. Place them wherever you like using the x and y parameters.
    7. Repeat step e and f for your text labels and text fields. The string you add in the constructor method will be the default text you will see in the text fields and labels
    8. Use the add(Component c) method to add all the features to the graphical user interface. If you do not add all the features, you will not be able to see them in the frame. For example, add(showAllButton); add(showAllData); add(accDeposit) etc
    9. Create an object called handler which is referencing the HandlerClass. When you create this, this will cause an error as this class does not exist yet, you will now create this to add functionality.
17. Create a private class (still within the same file) called HandlerClass and this needs to implement ActionListener. The syntax for this is “private class HandlerClass implements ActionListener.” ActionListener is an interface which forces you to use methods created inside it. In this case, we must use one called actionPerformed. Those who are using eclipse can simply click on the message, import this method and you will see an @Override above it. This indicates to us that a method has been transferred from the interface and we can modify it. You can remove the @Override bit if you want as it doesn’t matter but it’s a label to show it’s something not created by us, but a pre-defined method from an interface.
18. Within actionPerformed(ActionEvent e), use the e parameter to read what features have been triggered or used. For example, when a button is clicked, you can use if(e.getSource == depositButton). This will check if the deposit button has been pressed, if so, then perform a certain action within the condition brackets. Within this HandlerClass, you will detect which button has been pressed, read the inputs within the text fields and call the deposit, withdraw and transfer functions appropriately.

In the Main method, you will create a GUI object called gui and set the visibility to true, the exit feature (EXIT\_ON\_CLOSE) and size of the layout (I used 1000,1000). Once the GUI is open, perform the testing used in part 1 to make sure all the functions are working.

Assessment Video – In addition to the video to explain what the assessment is, there will be 4 additional videos. The first will show how the main method will look for part 1. The second is the main method for part 2. The third is the main method and GUI demo for part 3. The 4th video shows how to export your project ready for submission via eclipse.

**Submission of work**

Your completed work for assignments must be handed in on or before the due date. ***You must keep a copy or backup of any assessed work that you submit.  Failure to do so may result in your having to repeat that piece of work.***

**Penalties for late submission of coursework**

Standard Faculty of Science and Technology arrangements apply.

**ANY late submission (without valid cause) will result in 0 marks being allocated** **to the coursework**.

**Procedure for requesting extensions**

If you have a valid reason for requiring an extension you must request an extension using e:vision. **Requests for extension to assignment deadlines should normally be submitted at least one week before the submission deadline and may be granted for a maximum of seven days (one calendar week).**

**Retrieval of Failure**

A pass of 40% or above must be obtained overall for the module (but not necessarily in each assessment task).

**Where a student fails a module they have the right to attempt the failed assessment(s) once, at the next resit opportunity (normally July resit period).  If a student fails assessment for a second time they have a right to repeat (i.e. RETAKE) the module.**

**NOTE: STUDENTS WHO DO NOT TAKE THEIR RESIT AT THE NEXT AVAILABLE RESIT OPPORTUNITY WILL BE REQUIRED TO REPEAT THE MODULE.**

**Mitigating Circumstances (also called Extenuating Circumstances).**

If you are unable to meet a deadline or attend an examination, and you have a valid reason, then you will need to request via e:vision **Extenuating Circumstances.**

**Feedback of assignments**

You will be given feedback when you demonstrate your work.

You normally have **two working weeks** from the date you receive your grade and feedback to contact and discuss the matter with your lecturer. See the Student’s Union advice page <http://www.wolvesunion.org/adviceandsupport/> for more details.

**Registration**

Please ensure that you are registered on the module. You can check your module registrations via e:Vision You should see your personal tutor or the Student Support Officer if you are unsure about your programme of study. The fact that you are attending module classes does not mean that you are necessarily registered. A grade may not be given if you are not registered.

**Cheating**Cheating is any attempt to gain unfair advantage by dishonest means and includes **plagiarism** and **collusion.** Cheating is a serious offence. You are advised to check the nature of each assessment. You must work individually unless it is a group assessment.

**Cheating** is defined as any attempt by a candidate to gain unfair advantage in an assessment by dishonest means, and includes e.g. all breaches of examination room rules, impersonating another candidate, falsifying data, and obtaining an examination paper in advance of its authorised release.

**Plagiarism** is defined as incorporating a significant amount of un-attributed direct quotation from, or un-attributed substantial paraphrasing of, the work of another.

**Collusion** occurs when two or more students collaborate to produce a piece of work to be submitted (in whole or part) for assessment and the work is presented as the work of one student alone.