ICT167 Principles of Computer Science

Semester 2, 2023

Assignment 1 (worth 20% of unit assessment)

Due Date: Midnight, Friday – 22 September 2023

All Students: Submit the Assignment via LMS by the due date.

Late penalty: 10% per day penalty for delayed submissions unless prior extension of deadline is obtained from the unit coordinator.

You should keep a copy of your work.

You may be asked by your tutor to explain your submission. Make sure you understand everything you are submitting.

We are aware of websites providing code/solution for assignments, if you use those services, you will be reported for Academic Misconduct.

If Test Tables are not submitted, the assignment will not be marked and receive a 0 mark.

References and Pre-requisites:

- You will need to familiarise yourself with materials covered in Topics 1 to 6.
- Lab Practices 1 to 5 have been attempted, even if not submitted.
- Textbook Chapters 1 to 7

If you are unsure of anything about the assignment question, you need to get clarification early. Read this document very carefully.

OBJECTIVES:
- Apply the Object-Oriented design paradigm to construct solutions in a modular way.
- Implement important concepts of information hiding and encapsulation.
- Implement the design using the JAVA programming language.
- Demonstrate working knowledge of the array data structure.
- Demonstrate searching in the array of class objects.

Worth:
This assignment is worth 20% of the total assessment for the unit.

This is an individual assignment and must be completed by you alone. Any unauthorised collaboration/collusion and plagiarism may be subject to investigation and result in penalties if found to be in breach of University policy.

Question

This question reinforces concepts from Lab Practices 1 - 5. Best practice of Class and Method design should be demonstrated. This will require a good understanding of class design concepts and method design concepts with some fundamental design principles: ***modularisation, code re-use, high cohesion and low coupling****.*

*Before attempting this question***,** *complete the pre-requisites listed on the first page of this document.*

Assignment 1 uses O-O design implemented in Java. You will need to use a user-defined class, as an array of such class objects will be required. It is quite similar to the Assignment that you have done before in the prior unit however, this is implemented using O-O design. You should explain how you implement this assignment by addressing the following O-O concepts in your external documentation: (Note: not just by providing definition only, but **highlight exactly where in the code you are implementing these concepts**). Copy and paste code from your program to show how you implemented the following in your program using your external documentation. **You should use your code to explain the concepts below.** Some of the concepts could be related and you will need to explain how they are related.

- Open/Closed Principle
- Information hiding
- Encapsulation
- Pre-conditions and Post-conditions
- Private variables
- Constructors
- Assessors and Mutators
- Helper

User-defined Class:

You will design and implement your own data class. As the data stored relates to monetary change, the class should be named *ChangeC*. The class requires at least 2 instance variables for the name of a person and the coin change amount to be given to that person. In UK, the coin denominations are 1p, 2p, 5p, 10p, 20p, 50p, £1 and £2. You can use 8 instance variables to represent amounts for each of the 8-coin denominations. There should be no need for more than these instance variables. However, if you use a different way to implement this, you **must** provide legitimate justification for their usage in the internal and external documentation. All these instance variables should be declared as **private.** Think of a way such that the client program can use the same class name and public methods for other countries with different coin denominations without changing their client program. The key point here is to work on a design such that there is no need for the client class to make any change when the *ChangeC* class needs to upgrade to accommodate the different coin denominations for different countries. Explain how your class can satisfy this in your external documentation and what concepts are used to implement this design.

Your class will need to have at least a *default constructor*, and a *constructor with two parameters*: one parameter being a name and the other a coin amount. Your class should also provide appropriate *get* and *set* methods for client usage. Other methods (including helper) may be provided as needed. However, make sure they are necessary for good class design; you **must** provide legitimate justification for their usage in the internal and external documentation. In particular, your class should **NOT** include Input and Output methods. The only way to get data out of a data class object to the client program is to use an appropriate *get* method. **The data class methods must not write data out.** Data should be entered into a data class object via a *constructor* or an appropriate *set* method.

When designing your ChangeC class, use an UML class diagram to help understand what the class design needs.

Client Class:

The client program should read the input data from the user and use the *ChangeC* class to store the data entered. You will need a data structure to store the *ChangeC* class objects according to the number of persons entered. Thus, you are to utilize an array of *ChangeC* objects. **Do not use ArrayList for this assignment.**

Getting Input (put this input method in the client class):

Input for the client program will come from keyboard (entered by the user). The input should consist of: the name of a person, and a coin value (as an integer). Names are one-word strings. You should ask the user to enter the required information using a loop with a question after each loop iteration to check if the user wants to end the input of data. It is recommended for the user to input at least 10 such data – this can be conveyed to the user using a message before entering the loop.

NOTE: for the purpose of testing the program by your tutor, you should provide a method in the **client class** that hardcodes data into at least 10 ChangeC objects and stores these objects into the array provided by your program. In this case, your tutor would **not** need to manually key in the data to test the program when assessing your work. Thus, you should provide a call to this method (commented out) in the main function; this can be uncommented by your tutor as needed.

Example of inputs as follows:

```
Recommendation: Please enter at least 10 records to test the program.

Please enter the name of the person:
Jane
Please enter the coin value for the person:
30

Do you have more person to enter (Y/N):
Y

Please enter the name of the person:
John
Please enter the coin value for the person:
50

Do you have more person to enter (Y/N):
Y
```

```
Please enter the name of the person:
Fred
Please enter the coin value for the person:
94

Do you have more person to enter (Y/N):
Y

Please enter the name of the person:
Janet
Please enter the coin value for the person:
35

Do you have more person to enter (Y/N):
Y
```

… (*assuming this is repeated at least 10 times*)

```
Do you have more person to enter (Y/N):
N
```

… (*go out of the loop*)

Your program should check that no same name should be entered. If the same name has been entered, the program should inform the user that the name has existed in the system.

Make sure you have hardcoded test cases of the above for your tutor to test for these situations.

Methods would then need to be called to calculate the required output corresponding to the coin amounts stored in the array of objects. Output change values must consist of the following denominations: 1p, 2p, 5p, 10p, 20p, 50p, £1 and £2 coins. The program should aim to give as much of the higher valued coins as possible. A poor solution for an input of 30 pence is to give six 5 pence coins. The better solution is to give a 20 pence coin and a 10 pence coin.

Once the data input has been completed, your program should then display a menu screen as illustrated below. The program will continue to show the menu and execute the menu options until "Exit" is selected by entering the value 5 at the menu prompt.

1. `Enter a name and display change to be given for each denomination`
2. `Find the name with the smallest amount and display change to be given for each denomination`
3. `Find the name with the largest amount and display change to be given for each denomination`
4. `Calculate and display the total number of coins for each denomination`
5. `Calculate and display the total amount (i.e. NOT the total number of coins) for the sum of all denominations`
6. `Exit`

When the user enters the value 1 at the menu prompt, your program will ask for a name. As an example, if the user enters the name Jane (as in the example input above), the program will output:

```
Customer:
Jane 65 pence

Change:
50 pence: 1
10 pence: 1
5 pence:  1
```

N.B. change values of 0 are not shown for screen output.

If the user enters a non-existent name (eg: Donald) at menu option 1, which would therefore not be in the array of objects, your program will print:

```
Name: Donald
Not found
```

After processing the output for menu option 1, the menu is re-displayed.

When the user enters 2 at the menu prompt, your program will search all objects in the array to find the object with the smallest coin amount. Then the program will output the name for the person, and the denomination breakdown of their change. After processing the output for menu option 2, the menu is re-displayed.

When the user enters 3 at the menu prompt, your program will search all objects in the array to find the object with the largest coin amount. Then the program will output the name for the person, and the denomination breakdown of their change. After processing the output for menu option 3, the menu is re-displayed.

When the user enters 4 at the menu prompt, your program will access all objects in the array to calculate and display the total number of coins for each denomination. Do take note that this option displays the number of coins and NOT the amount of each denomination. After processing the output for menu option 4, the menu is re-displayed.

When the user enters 5 at the menu prompt, your program will access all objects in the array to calculate the sum of the amounts from all denominations. Do take note that this option is different from option 4; this option displays the amount and NOT the number of coins. After processing the output for menu option 5, the menu is re-displayed.

When the user enters 6 at the menu prompt, your program will write an appropriate farewell message to screen and exit.

<u>The client program requires the submission of a structure chart, a high-level algorithm and low-level algorithm (i.e. suitable de-compositions of each step in the high-level algorithm).</u>

Important Points:

You need to provide a test plan to **fully test** your algorithm and program. As well as keyboard input, do not forget to provide a method in the **client class** that hardcodes data into at least 10 ChangeC objects and stores these objects into the array provided by your program. Think carefully about how to construct this test data. If done well, you should be able to do nearly all required testing with this set of hardcoded test data.

Your solution (user-defined class, client class program and algorithm) should be modular in design. For methods, use a **high cohesion** and **low coupling** design approach. These principles will also demonstrate good **code re-use** if done properly.

Note that for this problem, the principle of code re-use is particularly important and a significant number of marks are allocated to this. You should attempt to design your solution such that it consists of a relatively small number of methods that are as general in design as possible, and you should have methods that can be re-used (called repeatedly) in order to solve the majority of the problem (actual calculations). If you find that you have developed a large number of methods where each performs a similar task, OR there is a lot of code that is repeated in the methods, then attempt to analyse your design to generalise the logic so that you have just one general version of required methods.

Be mindful of the cohesion exhibited by a method. If you have a method that is doing more than one task, then cohesion is low, and you will need to re-design it to have high cohesion.

Distribution of marks for assessment

For the programming question, an approximate distribution of marks for assessment is given below. The question will be marked out of 100 as follows:

- Correct solution design (class and method) and implementation: **50 marks**
- Programming style (internal documentation (comments), use of methods, parameters, readability, etc.): **25 marks**
- External Documentation (problem specification, algorithm, program limitations, program listings, software design principles used, program test strategy, and test results, etc.): **25 marks**

- **If Test Tables are not submitted, the assignment will not be marked and receive a 0 mark.**

For internal documentation (i.e. in the source code) we require:

- A beginning comment clearly stating title, author, date, file name, purpose and any assumptions or conditions on the form of input and expected output;
- Well-formatted readable code with meaningful identifier names and blank lines between components (like methods and classes).

The program should also include a method (eg: StudentInfo( )) to output your student details (name, student number, mode of enrolment, tutor name, tutorial attendance day and time) to the screen before the input data is read in.

Required External Documentation for each question:

1. **Title**: a paragraph clearly stating title, author, date, file names, and one-line statement of purpose.
2. **Requirements/Specification**: a paragraph giving a brief account of what the program is supposed to do. State any assumptions or conditions on the form of input and expected output.
3. **User Guide**: include clear instructions on how to compile and run the program, and how to use the program during execution. **State clearly which version of IDE you are using**.
4. **Structure/Design/Algorithm**: Outline the design of your program (it is here that you will need to justify any extra instance variables and class methods that you have included in *ChangeC* class). Give a written description, use diagrams (UML class diagram for the class and structure chart for the client program), and use pseudocode for algorithm.
5. **O-O design concepts or software design patterns**: Explain clearly using the code and solution provided to explain exactly how those concepts or software design patterns listed in page 2 are addressed in your submission. Do not just provide definition only as this will receive a 0 without referring to the solution provided.
6. **Limitations**: Describe program shortfalls (if any) e.g.: the features asked for but not implemented the situations it cannot handle, etc.
7. **Testing**: describe your testing strategy (the more systematic, the better) and any errors noticed. Provide a copy of your results of testing in a document saved in Word format. Note that a copy of the sample test data and results from a program run for each program is required (copy from the program output window and paste to a Word file). Your submitted test results should demonstrate a thorough testing of the program.

Use the following example tables.

Test Table

A set of test data in tabular form with expected results and desk check results from your algorithm. Each test data must be justified – reason for selecting that data. No mark will be awarded unless justification for each test data is provided.

Add rows to the following table as needed. Table can span more than one page. Each test id tests only one condition for the desk check.

Test id	**Test description/justification – what is the test for and why this particular test.**	**Actual data for this test**	**Expected output**	**Actual desk check result when desk check is carried out**	**Desk check outcome – Pass/Fail**
1					
2					
3					
4					

Results of Program Testing

Results of applying your test data to your final program (tabular form), including a sample printout of your program in operation.

Add rows to the following table as needed. Table can span more than one page.

Each test id tests only one situation for the test run of the program. Table is copy/paste of the desk check with actual output column showing results of the program output. There should be no duplicated reasons listed in the second column.

Test id	**Test description/justification – what is the test for and why this particular test.**	**Actual data for this test**	**Expected output**	**Actual program output when test is carried out**	**Test run outcome – Pass/Fail**
1					
2					
3					
4					

After the above test table, copy/paste sample printouts of your program in operation. You can screen capture and paste here. Make sure you label each printout with the correct *Test id*.

8. Source program listings: submit all your program listing and IDE project files.

All of the external documentation for the question must be included in the above order saved in a file in MS Word format. Your final external documentation submission should be prepared as a single Word document containing each of the questions in order and each section within these also in order.

It is also necessary to submit the Java source code and compiled version of your program (i.e. all classes that you have designed and implemented). You have to develop the program using the IDE. It will make it easy to collect sample output. The whole project should be submitted, and state clearly which version of IDE you are using.

The external documentation together with the source code files and compiled versions for the question must be compressed in a .zip file before submitting. Note that the whole IDE project folders is zipped. Make sure that all necessary files are submitted so that the programs can be viewed, compiled and run successfully. The final versions of the programs should compile and run under Java SE (JDK). If you program cannot be run, the submission may not be marked.

Associate Professor Kevin Wong
ICT167 Unit Coordinator