Cardiff School of Computer Science and Informatics
Coursework Assessment Pro-forma

Module Code: CMT219
Module Title: Algorithms, Data Structures and Programming
Lecturer: Nervo Verdezoto Dias, Yuhua Li
Assessment Title: Algorithms Data Structures and Design Patterns
Assessment Number: 2
Date Set: 30 Apr 2021
Submission Date & Time: 26 May 2021 at 9:30am British Summer Time
Return Date: 23 June 2021

This coursework is worth 50% of the total marks available for this module. If coursework is submitted late (and where there are no extenuating circumstances):

1. If the assessment is submitted no later than 24 hours after the deadline, the mark for the assessment will be capped at the minimum pass mark;
2. If the assessment is submitted more than 24 hours after the deadline, a mark of 0 will be given for the assessment.

Your submission must include the official Coursework Submission Cover sheet, which can be found here: https://docs.cs.cf.ac.uk/downloads/coursework/Coversheet.pdf

Submission Instructions

Your coursework should be submitted through Learning Central. Your submission should include:

	Description		Type	Name
	Cover sheet	**Compulsory**	PDF (.pdf) file	[student number].pdf
Part 1	***ONE*** ZIP archive file (and no more than one) containing all the source code files for your answer to questions of Part 1	**Compulsory**	One ZIP (.zip) archive	[student number]_Part1Code.zip e.g., C1234567_Part1Code.zip
	ONE PDF file (and no more than one) which contains a written justification for your design of the program and screen shots showing an example of the output of each application	**Compulsory**	One PDF (.pdf) file	[student number]_Part1.pdf e.g., C1234567_ Part1.pdf
Part 2	***ONE*** PDF file (and no more than one) which contains the diagrams, descriptions, explanations, discussion, program listings (extracts from the code which show clearly what you did in order to complete the Tasks).	**Compulsory**	PDF (.pdf) file	[student number]_Part2.pdf e.g., C1234567_ Part2.pdf
	ONE ZIP archive file (and no more than one) containing all the source code files for your answers to Tasks 1 to 3	**Compulsory**	ZIP (.zip) archive file	[student number]_Part2Code.zip e.g., C1234567_Part2Code.zip

Any code submitted will be run on a University machine and must be submitted as stipulated in the instructions above.

Any deviation from the submission instructions above (including the number and types of files submitted) will result in a reduction of 20% of the attained mark.

Staff reserve the right to invite students to a meeting to discuss coursework submissions

Assignment

Part 1 – Algorithms and Data Structure

[**Total: 60 marks**]

All code must be written by you, although you can use the lecture notes (and lab exercises), textbooks, and the Oracle Java website for guidance.

A) Extracting words from a text document is the first step for many text-based applications of artificial intelligence, e.g., detecting abusive tweets on Twitter. This task asks you to extract all valid words from the document "**an_article.txt**" based on a given vocabulary "**google-10000-english-no-swears.txt**[1] " (you can download both files from the Learning Central). Specifically, if a word token from "**an_article.txt**" matches a word in "**google-10000-english-no-swears.txt**" (case insensitively), you keep that word, otherwise you discard it. You should use *ArrayList* to store data within the Java program, so the return of this task is an *ArrayList* containing all valid words from "**an_article.txt**". Note that these two text files are relatively big, you should consider how to make your program efficient.

[**30 marks**]
(Functionality: 16, Design: 10, Ease of use: 2, Presentation: 2)

B) Implement the merge sort (the pseudocode for merge sort is available in the lecture slides) in order to sort the words obtained from above in alphabetical order, i.e., the output of your program will be the sorted words in alphabetical order. For the merge sort algorithm write a method e.g. *mergeSort(…)*, measure
- time that is needed to sort 100 of the words, 200 of the words, 300 of the words, etc. up to all words by the algorithm.
- count the moves/comparisons that occur while sorting elements
  - (Before attempting this exercise you should work through the Algorithms lab exercises, available on Learning Central. The techniques used there will help you to work out how to approach this part of the coursework, in particular there are examples of how to time algorithms and count the moves and swaps.)

[**20 marks**]
(Functionality: 12, Design: 4, Ease of use: 2, Presentation: 2)

C) You should create two methods for a data structure of *LinkedList*. Your data structure should have the class name *MyLinkedList*. The two methods that should be implemented are:
- Adding element at specific location in the list:
  `public void addAtPosition(int position, String item){…}`
- Deleting an element from the list at specific position where the method should return which Node is deleted:
  `public Node deleteAtPosition(int position){…}`

[1] This vocabulary contains a list of the 10,000 most common English words in order of frequency of the Google's Trillion Word Corpus.

In both methods handle errors properly (e.g. what happens if position does not exist).

There will be a skeleton code of MyLinkedList available on Learning central that contains: Node class, MyLinkedList class with calculateSize(), addFirst(), addLast(), traverse(), findByPosition() methods and ***signatures for two methods*** *addAtPosition(int position, String item), deleteAtPosition(int position)* ***that you should implement***.

You can reuse any implemented methods in the provided skeleton code. You ARE NOT allowed to change any parts of the implemented methods of the provided code.

[**10 marks**]
(Functionality: 10, Design: 0, Ease of use: 0, Presentation: 0)

Criteria for Assessment of Part 1

Credit will be awarded against the following criteria.

Functionality
- To what extent does the program perform the task(s) required by the question and how efficient it is.

Design
- How well designed is the code, particularly with respect to the ease with which it may be maintained or extended. In particular, consideration will be given to:
  - Use of appropriate types, program control structures and classes from the core API.
  - Definition of appropriate classes and methods.
  - Creativity/consideration of achieving efficient algorithms.
  - An 'excellent' design of code will carefully consider using most suitable data structures and aiming for efficient and elegant algorithms. A 'bad' design of code pays little/no attention to data structure and algorithm efficiency.

Ease of use
- Formatting of input/output.
- Interaction with the user.
- How does the code deal with invalid user input? Will the applications crash for certain data?

Documentation and presentation
- Appropriate use of comments.
- Readability of code.
- Clear and appropriate screenshots.

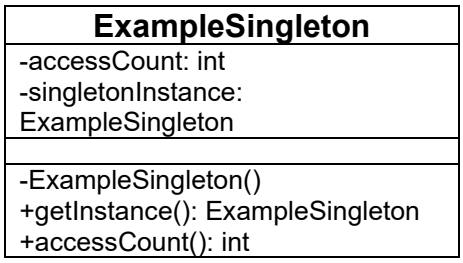
Part 2 - Design and Implementation of Design Patterns

[**Total: 40 marks**]

For each of the following tasks, a list of files is included in the present coursework specification at the end for your convenience.

Task 1

Download the zip archive file from Learning Central that contains the source code files for this task: **ExampleTest.java**. Consider the following UML class diagram, describing a particular kind of *singleton*, where **accessCount()** returns the number of times the singleton has been accessed via the **getInstance()** method:



(i) State the purpose of the *Singleton* design pattern.

(ii) Write an implementation of **ExampleSingleton.java** such that:

a. The **ExampleSingleton ()** constructor would print out the message "I, the ExampleSingleton, am being created".

b. The **getInstance ()** method should print out the message "The sole instance of ExampleSingleton is being retrieved".

c. When the **main()** method in **ExampleTest.java** is executed, it will generate the following output on the standard output stream:

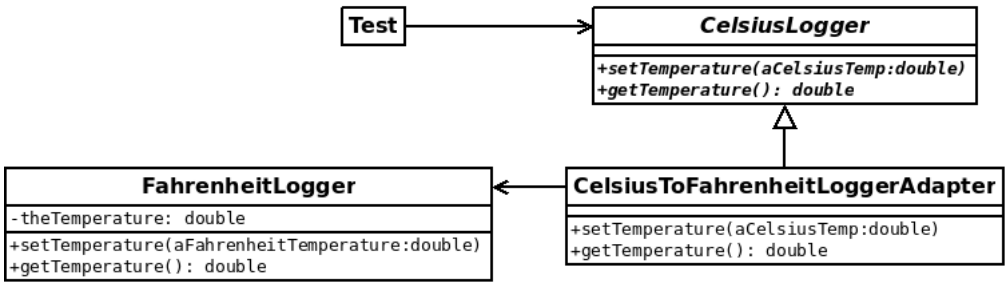
```
I, the ExampleSingleton, am being created
The sole instance of ExampleSingleton is being retrieved
The ExampleSingleton has been accessed via the getInstance() method 1 time(s)
The sole instance of ExampleSingleton is being retrieved
The ExampleSingleton has been accessed via the getInstance() method 2 time(s)
```

(iii) Explain how the **ExampleSingleton** class works, drawing attention to the features which are intended to control creation of, and access to, singletons.

[**10 marks**]

Task 2

Download the zip archive file from Learning Central that contains the source code files for this task: **CelsiusLogger.java**, **Test.java**. Consider the following UML class diagram, describing a particular kind of *Adapter*, involving two basic types of temperature logger: one which logs the most recently measured temperature in degrees Fahrenheit; and the other, in degrees Celsius:



(i) State the purpose of the *Adapter* design pattern

(ii) Write an implementation of **FahrenheitLogger**. The **getTemperature()** and **setTemperature()** methods should be written as *accessors* which get and set the values for **theTemperature** variable, without any side-effects.

(iii) Write an implementation of the class **CelsiusToFahrenheitLoggerAdapter**. When the adapter is created, it should create the **FahrenheitLogger** to which it refers. Its methods should do the following:

- **setTemperature**() should call the **FahrenheitLogger**'s **setTemperature**() method with parameter (aCelsiusTemperature * 9/5 + 32)
- **getTemperature**() should call the **FahrenheitLogger**'s **getTemperature**() method and return (((the FahreneitLogger's temperature) – 32) * 5/9)

(iv) Describe in your own words how the implementation of the **CelsiusToFahrenheitLoggerAdapter** class works, drawing attention to the role of each of the classes implemented to realize this design pattern.

[**10 marks**]

Task 3

Download the zip archive file from Learning Central that contains the source code files for this task: **WeatherRecorder.java, WeatherObserver.java, WarningWatcher.java, TestWeather.java**. This Java program is an incomplete implementation that uses the *Observer* design pattern to print out alerts when specific weather forecast events occur. The observers are monitoring for "Warning" events. One monitors for warning events for England, the other for Wales.

The **TestWeather** class illustrates how the WeatherRecorder/WeatherObserver code might be used. The program should produce the following output:

```
The WarningWatcher watching for Warnings for Wales
has noticed a new warning:
"Very Windy"

The WarningWatcher watching for Warnings for England
has noticed a new warning:
"Very Snowy"
```

(i) State the purpose of the *Observer* design pattern

(ii) Draw a UML class diagram representing the resulting Observer design pattern in this program and explain the role of each of the classes in realising this design pattern.

(iii) Modify the following methods, as indicated in the attached code comments, in order that the above output can be obtained:

- The **WeatherRecorder attach()** method
- the **WeatherRecorder notifyObservers()** method
- the **WarningWatcher update()** method.

(iv) Carefully explain how your minor changes in the methods contribute to the implementation of the *Observer* design pattern.

[20 marks]

Criteria for Assessment of Part 2

Credit will be awarded against the following specific criteria. Ability to:

- Understand the purpose of a Java program and apply principles of good object-oriented design in order to create a critique and/or modify the supplied programs (check the source code files provided);
- Apply and implement design patterns that pertain to a specified programming task;
- Reflect on the application of design patterns in OO Java programs.

Feedback on your performance will address each of the above criteria. The total mark is a weighted sum of marks for the different tasks.

The following will help you to interpret the marks awarded for each task:

0-49%: The submitted answer only addresses the tasks to a very limited extent. Some attempt has been made to address the tasks, and the code mostly works.

50-59%: The tasks have been completed, the code "works", but minimal insight has been shown.

60-69%: The tasks have been completed, the code "works", and some clear insight has been shown.

70-100%: An excellent implementation, showing a good degree of insight and reflection, and the code is of good quality.

Learning Outcomes Assessed

This assignment particularly addresses the following module learning outcomes:

- Critically evaluate design patterns and their applicability to different scenarios
- Select and use appropriate algorithms and data structures to provide best performance in a given scenario

Code Reuse

Your solutions may make use of any classes in the Core Java API. You may also reproduce small pieces of code from:

- The CMT219 course handouts and solutions
- java.oracle.com
- any textbooks

provided:

- The section reproduced does not form the entire solution to a single question
- The source of the code is clearly referenced in your source code
- Your code is commented to demonstrate clearly that you understand how the reproduced code works (i.e., explain why types have been selected, why other language features have been used, etc.)

You may **NOT** reproduce code written by any other student or code downloaded from any other website.

If you are in any doubt about whether you may include a piece of code that you have not written yourself, ask the lecturer before submitting.

See “Referencing in code guidance” at Learning Central → COMSC-SCHOOL → Learning Materials → Referencing in code guidance.

Feedback and Suggestion for Future Learning

Feedback on your coursework will address the above criteria. Feedback and marks will be returned via Learning Central. There will be opportunity for individual feedback during an agreed time.

Program listing for Part 2

Task 1

```
public class ExampleTest {
	public static void main(String[] args) {
		ExampleSingleton s = ExampleSingleton.getInstance();
		System.out.println("The ExampleSingleton has been "
		+ "accessed via the getInstance() method "
		+ s.accessCount()
		+ " time(s)");
		s = ExampleSingleton.getInstance();
		System.out.println("The ExampleSingleton has been "
		+ "accessed via the getInstance() method "
		+ s.accessCount()
		+ " time(s)");
	}
}
```

Task 2

```
public abstract class CelsiusLogger {
	public abstract void setTemperature(double aCelsiusTemp);
	public abstract double getTemperature();
}
public class Test {
	public static void main(String[] theArguments) {
	CelsiusLogger l=  new CelsiusToFahrenheitLoggerAdapter();
	l.setTemperature(22.0);
	System.out.println("Current logged temperature: " + l.getTemperature() + " Celsius.");
	}
}
```

Task 3

```
public class WeatherRecorder {
	private ArrayList <WeatherObserver> observers = new ArrayList <WeatherObserver> ();
	private String latestUpdateType; private String latestUpdateCountry;
	private String latestUpdateText;

	public void attach(WeatherObserver o) {
		// Complete this method so it adds the observer to the observers list
	}
	public void setLatestNews(String theCountry, String theUpdateType, String theUpdateText) {
		latestUpdateType=theUpdateType; latestUpdateCountry=theCountry;
		latestUpdateText=theUpdateText; this.notifyObservers();
	}
	public String getUpdateType() { return latestUpdateType; }
	public String getUpdateCountry() { return latestUpdateCountry; }
	public String getUpdateText() { return latestUpdateText; }
	private void notifyObservers() {
		// Complete this method to go through each observer in turn,
		// sending it a message to notify that an update has occurred
	}
}
```

```
public abstract class WeatherObserver {public abstract void update(); }

public class WarningWatcher extends WeatherObserver {
	private String countryWatched; private WeatherRecorder theRecorder;
	public WarningWatcher(WeatherRecorder aWeatherRecorder, String countryToWatch) {
		theRecorder = aWeatherRecorder; countryWatched = countryToWatch;
		theRecorder.attach(this);
	}


	public void update() {
		// Modify this so that it only prints out the update text
		// if the update is a "Warning" for the country being watched
		System.out.println("The WarningWatcher watching for Warnings for " +
				countryWatched + "\nhas noticed a new warning:\n\"" +
				theRecorder.getUpdateText() + "\"\n");
	}


public class TestWeather {
	public static void main(String[] args) {
		WeatherRecorder wr = new WeatherRecorder();
		WeatherObserver englandWatcher = new WarningWatcher(wr, "England");
		WeatherObserver walesWatcher = new WarningWatcher(wr, "Wales");

		wr.setLatestNews("England", "Current Temperature", "15");
		wr.setLatestNews("Wales", "Warning", "Very Windy");
		wr.setLatestNews("Wales", "Current Rainfall", "0");
		wr.setLatestNews("England", "Warning", "Very Snowy");
	}
}
```