CST2110 Individual Programming Assignment #1 (RESIT)

Deadline for submission **16:00, Friday 30th July, 2021**

General information

You are required to submit your work via the dedicated Unihub assignment link in the 'resits and deferrals' folder by the specified deadline. This link will 'timeout' at the submission deadline. Your work will not be accepted as an email attachment if you miss this deadline. Therefore, you are strongly advised to allow plenty of time to upload your work prior to the deadline.

Submission should comprise a single 'ZIP' file. This file should contain a separate, cleaned[1], NetBeans project for each of the three tasks described below. The work will be compiled and run in a Windows environment, i.e., the same configuration as the University networked labs and it is strongly advised that you test your work using the same configuration prior to cleaning and submission.

When the ZIP file is extracted, there should be three folders representing three independent NetBeans projects as illustrated by Figure 1 below.

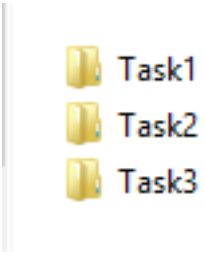

Figure 1: When the ZIP file is extracted there should be three folders named Task1, Task2 and Task3

Accordingly, when loaded into NetBeans, each task must be a separate project as illustrated by Figure 2 below.

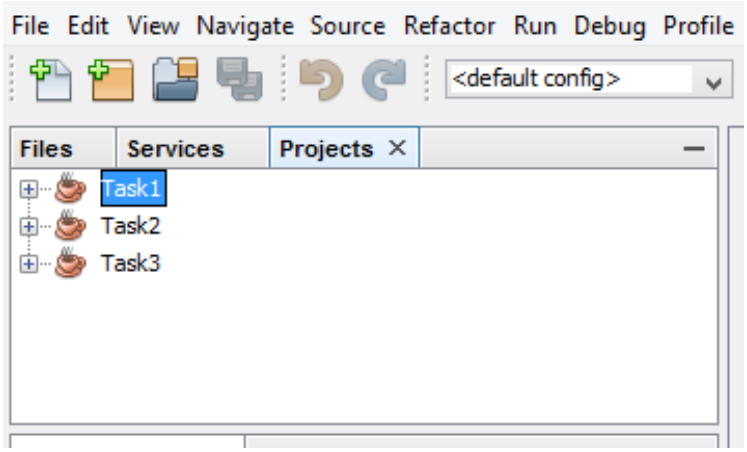

Figure 2: Each task must be a separate NetBeans project.

To make this easier, a template NetBeans project structure is provided for you to download.

[1] In the NetBeans project navigator window, right-click on the project and select 'clean' from the drop-down menu. This will remove .class files and reduce the project file size for submission.

Task 1 (20 %)

In NetBeans, open the project called 'Task1' and inspect the data file provided called *film-script.txt*. This file contains the full text of a film screenplay. You are not required to, and should not, alter this text file i.e., it is intended to be *read-only*.

Your task is to write a Java 8 program (as a NetBeans 8 project) that opens the data file, parses the text of the film script, and prints the following information to the console:

a) The total number of uppercase letters in the file.
b) The total number of lowercase letters in the file.
c) A listing of the frequency of each letter in the file (uppercase or lowercase), ordered by the highest frequency to the lowest frequency. The listing should display each letter, followed by the letter's frequency (integer count). There should be a new line between each letter and frequency value.

The frequency counts should be performed on letters only (i.e., not on non-letter characters).

On compiling and running, your program should have output that is structured as follows:

```
Total number of uppercase letters = count of all uppercase letters in the file
Total number of lowercase letters = count of all lowercase letters in the file

e --> frequency value
t --> frequency value
E --> frequency value
o --> frequency value
etc. etc.
. . .
Q --> frequency value
Z --> frequency value
```

The actual ordering of the frequency values will, of course, depend on how your program performs and the above output is to illustrate the structure of the output only. In addition to the console output, your program should also save all the output to a text file in the NetBeans project root folder. Name the file *results.txt*.

Having performed these actions, your program should close the opened file(s) properly.

Your program should handle exceptions correctly, and robustly.

Locating and opening the data file

You must not, under any circumstances, include a hard-coded file path to the test data file location i.e., a path that is only applicable to your own personal computer configuration. The data file must be accessed *relative* to the NetBeans project folder. When viewing the project folder in Windows Explorer, the data file should be located as illustrated by Figure 3 below.

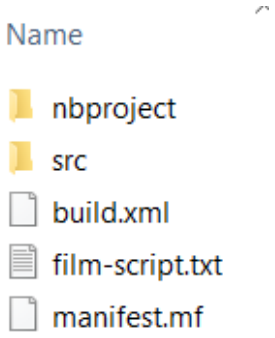

Figure 3: Location of the test data file when viewed in Windows Explorer.

Accordingly, if you select the 'Files' tab for the given NetBeans project, you will see the location of the test data file relative to the project files, as illustrated by Figure 4 below.

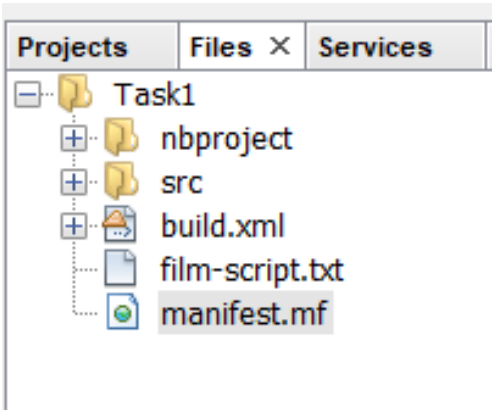

Figure 4: Location of the test data file when viewed in NetBeans (select 'Files').

With the data file located in the correct place (i.e., the 'root' of the specific NetBeans project), then the following lines of code will correctly return a relative path to the data file.

```
String fileLocation = System.getProperty("user.dir");
String dataPath = fileLocation + File.separator + "film-script.txt";
```

In order that no errors are made with respect to this structure, templates have been provided (so there really is no excuse!). If these instructions are not adhered to, and your solutions include hard-coded paths, marks will be deducted.

Assessment of Task 1

Your solution to Task 1 will be assessed according to functionality and the correctness of program output.

When checking your solution, the tutors will use a different test file, but with the same structure to that provided. In other words, the program should execute in the same way, but the program outputs will be different. The markers will examine these different outputs to verify the correctness of your program.

Task 2 (40%)

Remember, use a different (separate) NetBeans project for this task, named *Task2*. A template has been provided for you with the data file located relative to the NetBeans project root folder as before. For this task, the data file contains a list of words to be used in a simple 'word guessing' game. The data file is called *wordlist.txt*, and again, is provided as a *read-only* file that should not be altered. The data file comprises words of five letters only and is compiled from an open-source repository that is popular with word games developers. It is quite extensive and includes over eight thousand five-letter words.

Your task is to write a Java 8 program (as a NetBeans 8 project) that randomly selects one of the words provided in *wordlist.txt* file to be a secret 'codeword' and presents a console interaction for the User to guess the codeword. The User guessed word must be a valid word that is contained in the *wordlist.txt* file. If the User enters an invalid word, the program displays an appropriate message and prompts the User to try again (not counting the invalid word as a guess). For each valid guess, the program will announce how many letters of the guessed word appear in the codeword, regardless of position. The output will display the guessed word along with the number of corresponding 'hits' it produces. As the User continues guessing, the program generates a simple table of guessed words and hits until the User either correctly guesses the word or reaches a maximum number of 10 guesses without managing to guess the word. For example, suppose that the program randomly selects the five-letter word "oboes", and the following interaction takes place:

```
Word Guessing Game

Play (1) or Exit (0) > 1

Guess a 5-letter word (lower case) or enter * to give up

> trawl

There are 0 matching letters in your guess (trawl)

-------------
| trawl | 0 |
-------------

Number of guesses so far: 1

Guess a 5-letter word (lower case) or enter * to give up

> chomp

There is 1 matching letter in your guess (chomp)

-------------
| chomp | 1 |
| trawl | 0 |
-------------

Number of guesses so far: 2

Guess a 5-letter word (lower case) or enter * to give up

> botch

There are 2 matching letters in your guess (botch)

-------------
| botch | 2 |
| chomp | 1 |
| trawl | 0 |
-------------
```

```
Number of guesses so far: 3

Guess a 5-letter word (lower case) or enter * to give up

> boost

There are 4 matching letters in your guess (boost)

-------------
| boost | 4 |
| botch | 2 |
| chomp | 1 |
| trawl | 0 |
-------------

Number of guesses so far: 4

Guess a 5-letter word (lower case) or enter * to give up

Etc.
```

From the example above you will note that the position of the letters does not matter with regards to the number of hits announced. Also, hits should be counted on a one-to-one basis. For example, although there are two o's in *oboes*, the guessed word *botch* contains only one o and so scores only one hit in respect of that letter, whereas *boost* counts two for containing both. If the User either correctly guesses the secret word, or makes 10 incorrect guesses, then the secret codeword should be revealed with the option to play again.

Testing your program

To make it easier for the tutor to assess your program, you are required to program a small 'cheat' feature into your program which enables the assessor to view the secret code word when a new game commences. To facilitate this, you should provide a simple Boolean flag (variable) in the main method. The default value is false (i.e., the User cannot see the secret codeword), but when set to true (by the tutor) the codeword is displayed. As illustrated below:

```
public class ApplicationRunner {

    public static void main(String[] args) {

        String dataFile = System.getProperty("user.dir") +
                                          File.separator + "wordlist.txt";

        boolean displayCodeWord = false;

        . . .
    }

}
```

Failure to include the Boolean variable and the subsequent display of the secret codeword when it is set to true will result in a significant mark penalty.

As with Task 1, your solution to Task 2 will be assessed according to robustness, functionality, and the correctness of program output. In addition, the assessment of this task will consider program modularity, formatting, and code style. In other words, credit will be gained for good use of methods, neat console-based text formatting, appropriate use of Java naming conventions, and sensible use of commenting.

Task 3 (40%)

This programming task has a focus on object-oriented concepts that are introduced in Chapter 9 of the module *kortext*.

Consider the finite state machine (FSM) that is illustrated in Figures 5a, 5b and 5c below.

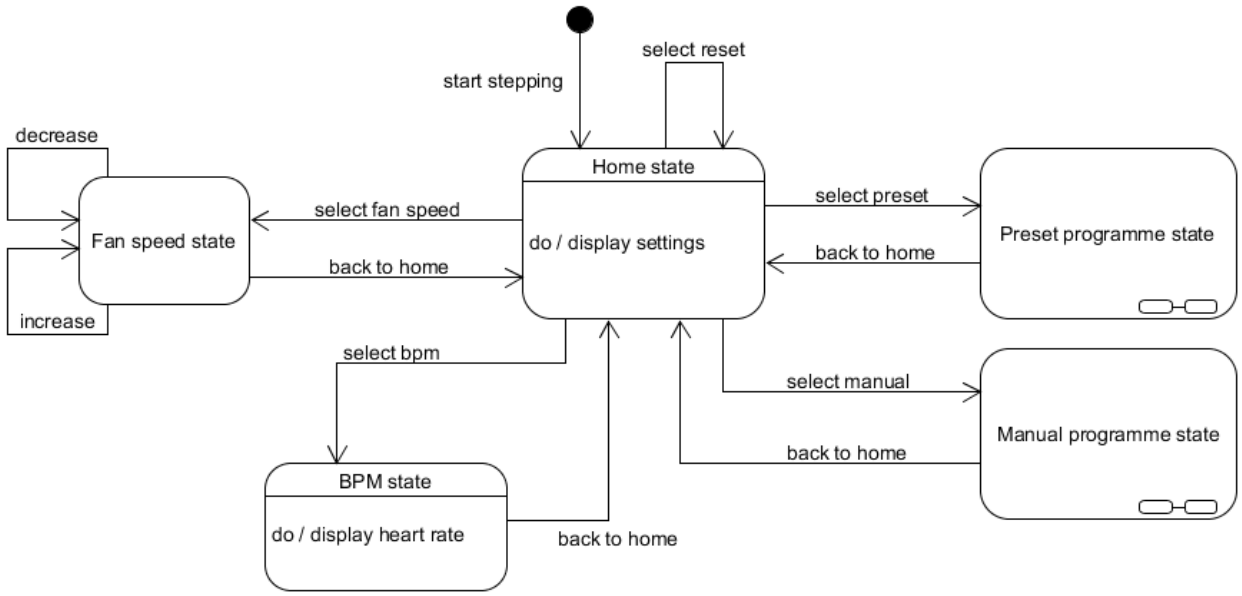

Figure 5a: Finite State Machine representation of a cross-trainer

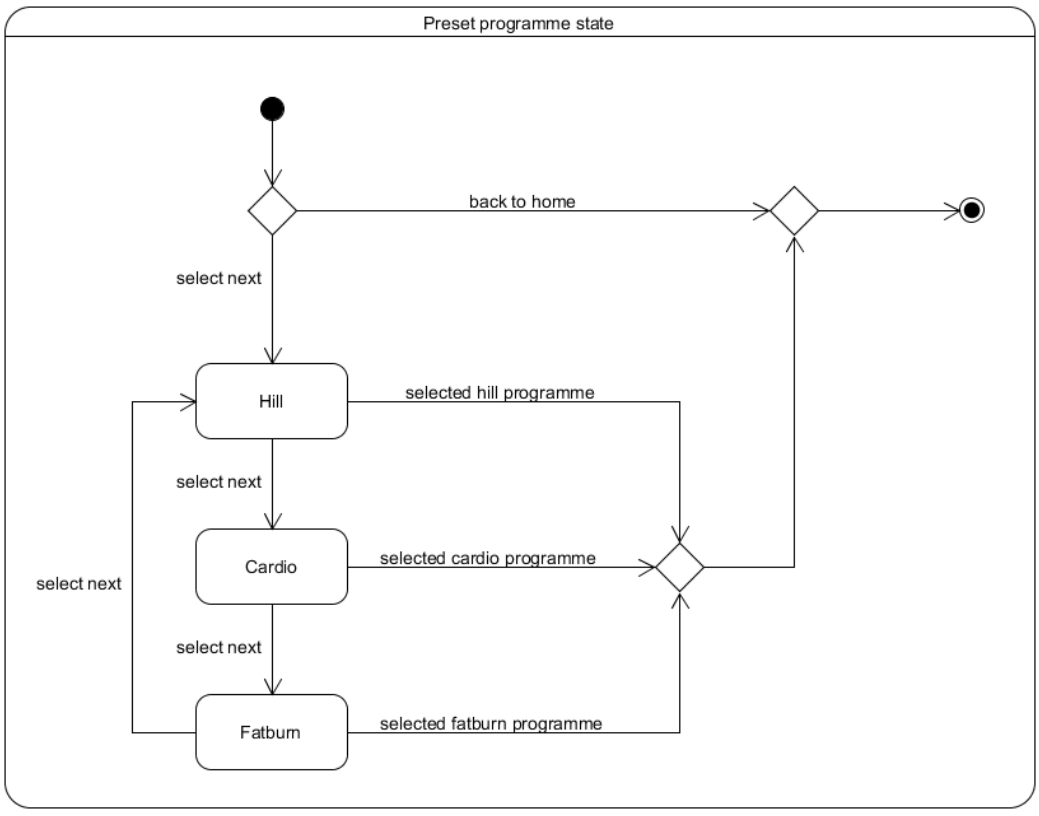

Figure 5b: FSM sub-states of preset exercise programmes.

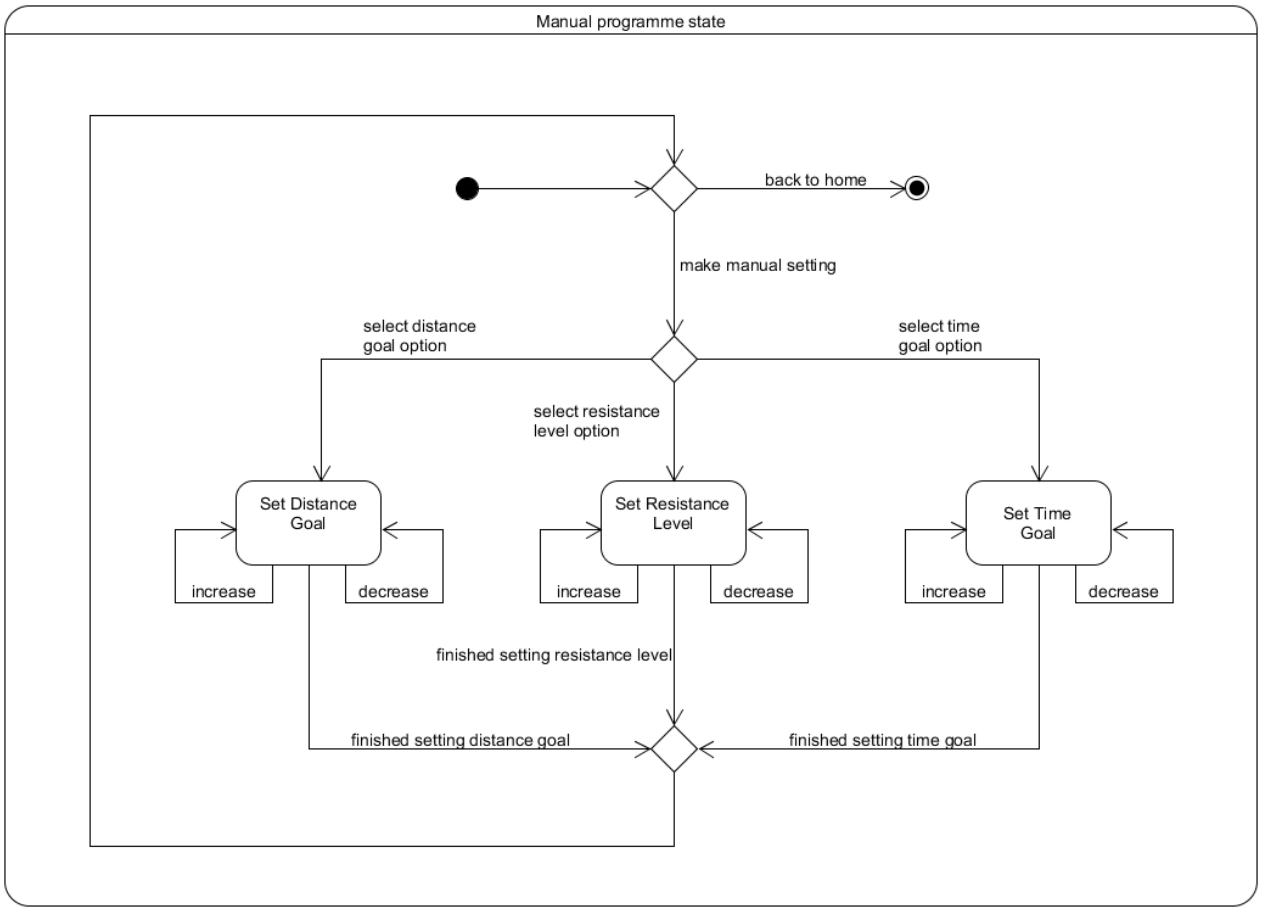

Figure 5c: FSM sub-states of the manual exercise programme.

The FSM illustrated above represents a 'cross-trainer' exercise machine typically found in a gym. The FSM essentially represents the set of states that the cross-trainer can be in, with transitions resulting from events produced by interaction with a touchscreen.

Your task is to write a Java 8 program (as a NetBeans 8 project) that is a simulation of the cross-trainer. In other words, the program must (on execution) demonstrate the FSM illustrated in Figures 5a, 5b and 5c. As part of this program, you are required to design and implement (as a minimum) a dedicated Java class that represents the cross-trainer. As with all other tasks in this assignment, User interaction must be text-based at the NetBeans console (i.e., not a GUI), and, in this case, it should offer a simple menu-selection interaction using mutually exclusive integer values (i.e., each menu option maps to a different integer value to be read in at the prompt). The menu interaction must support all state transition events depicted in the FSM and allow the User to exit the program. The console interaction should operate as follows:

- On entry to the home state, the console should display default programme setting values for target distance, a target time duration, a resistance level, and the default fan speed setting should be 'off'. The User should be presented with options (via a menu on the NetBeans console) to transition from the home state to one of four modes (states) in which the exercise programme settings can be changed: fan speed, BPM, preset programmes, and manual programme.

- When the User enters the fan speed state, they should be allowed to increase or decrease between four mutually exclusive fan speed settings: off, slow, medium, and fast. On completing the selection

of the fan speed setting, the User should have an option to return to the home state whereby the updated fan setting is reflected in the home state display (i.e., printed to the NetBeans console).

- When the user selects the BPM option (to simulate grasping the heart rate sensors on the exercise machine) then the console should display a heart rate (just a fixed value of 70 bpm for this simulation). When finished, the User should have an option to return to the home state and the BPM is no longer displayed.

- When the FSM transitions to the manual programme state the User should be allowed the option to set values for a target distance (up to 10Km in increments of 500 metres), a time goal (up to 1 hour in increments of 1 minute) and a resistance level (a value between 1 and 30 in increments of 1). These are not mutually exclusive options, and the User may elect to set any combination of these options. The User should be able to both increment and decrement the relevant setting values rather than set an absolute value. When finished, the User should have an option to return to the home state whereby the settings are reflected in the home state display (i.e., printed to the NetBeans console).

- When the FSM transitions to the preset programme state the User simply needs to cycle through the three possible preset modes (hill, cardio, and fat-burn) and accept default values for distance, resistance level and time (i.e., you can fix appropriate values in your program). When finished, the User should have an option to return to the home state whereby the settings are reflected in the home state display along with an indication of the chosen preset programme.

- When in the home state, the main menu of the console user interaction should also offer an option to reset the programme settings back to the default values.

Task 3 will be assessed according to correct logic (i.e., correct implementation of the FSM) within the simulation program, the production of an appropriate menu-driven user interface at the NetBeans console, and your use of object-oriented concepts.

Please note that all task weightings are provisional and subject to adjustment during moderation of the overall module assessment. A provisional assessment rubric is provided in the module handbook.