CSSE2002 / CSSE7023 **Semester 1, 2017**

Assignment 3

<u>Goal:</u> The goal of this assignment is to gain practical experience with implementing GUIs using the Model-View-Controller design pattern.

<u>Due date:</u> The assignment is due at **noon on Friday the 26th May**. Late assignments will lose 20% of the maximum mark immediately, and a further 20% of the maximum mark for each day late.

Only extensions on Medical Grounds or Exceptional Circumstances will be considered, and in those cases students need to submit an application for extension of progressive assessment form (http://www.uq.edu.au/myadvisor/forms/exams/progressive-assessment-extension.pdf) no later than 48 hours prior to the submission deadline. The application and supporting documentation (e.g. medical certificate) must be submitted to the ITEE Coursework Studies office (level 4 of GPSouth) or by email to studentenquiries@itee.uq.edu.au. If submitted electronically, you must retain the original documentation for a minimum period of six months to provide as verification should you be requested to do so. Assignment extensions longer than 5 calendar days will not be possible due to the incremental nature of the assignments.

<u>School Policy on Student Misconduct:</u> You are required to read and understand the School Statement on Misconduct, available on the School's website at:

http://ppl.app.uq.edu.au/content/3.60.04-student-integrity-and-misconduct

<u>This is an individual assignment.</u> If you are found guilty of misconduct (plagiarism or collusion) then penalties will be applied.

If you are under pressure to meet the assignment deadline, contact the course coordinator as soon as possible. <u>You should not discuss or post solutions to the assignment on public forums, like piazza.</u>

<u>Problem description:</u> In this assignment you will develop a GUI for allocating events to the venues in a municipality using the Model-View-Controller design pattern. This extends the work you have done in Assignments 1 and 2. (Note that you won't need to use the `Allocator` class in this assignment, but is included in the files that accompany this handout for reference.)

Exactly what the GUI looks like is up to you, but in order to meet testing requirements, it must be capable of performing the following tasks:

- When the event allocator program is executed (by running the main method in `EventAllocator.java`), the program should
  - o load the venues defined in the file named `venues.txt` using the `read` method from `VenueReader`, setting the *venues in the municipality* to those which have been read from the file.
  - o set its *current allocation of events to venues*, so that no events are allocated to any venues.

  An appropriate error message should be clearly displayed (in the graphical user interface) if there is an error reading from the file or there is an error with the input format (i.e. if `VenueReader.read` throws an `IOException` or a `FormatException`).

  The error message displayed to the user should clearly identify the reason for the error. That is, it should identify that the file `venues.txt` could not be loaded, and why it could not be loaded. The reason should include whether or not the load error was due to an input/output error reading from the file, or because of an error with the input format of the file, and it should include the detail message of the exception thrown to help the user track down the exact reason for the problem.

If the venues could not be loaded then the program, after informing the user of the problem, is not obliged to perform the remainder of the tasks specified here. It should either exit gracefully or allow the user to close the program without being able to perform any other functions. If the venues could be loaded, then the user should be able to see that there are currently no events allocated to venues, no traffic caused by that allocation, and the program should be ready for use.

Note that a sample file named `venues.txt` has been included in the assignment zip file, but that your code should not be hard-coded to only handle the content of that input file – the content of the file should be able to be updated in any way.

Assuming that the venues for the event allocator could be loaded (so that the *venues in the municipality* could be appropriately set), the user should be able to use the program to perform the following tasks.

- At any point the user should be able to see the *current allocation of events to venues*. For this, the user should clearly be able to see, in a readable format, every event that is currently allocated to a venue and the venue that it has been allocated to. In particular, for each event which is currently allocated to a venue, the user should be able to see, in a readable format:
    - the name and size of the event
    - the name and capacity of the venue that it has been allocated to (the capacity traffic of the venue does not need to be shown)

  In the view of the *current allocation of events to venues*, the events should be listed in alphabetical order of the name of the event. Within this ordering, events with equal names should be ordered in ascending order of their size. (Events that are not currently allocated to a venue should not be displayed.)

- At any point the user should be able to see the *traffic caused by the current allocation* of events to venues. The *traffic caused by the current allocation* is the sum of the traffic generated by hosting each of the events at their respective venues in the allocation. (That is, the traffic that would be caused by hosting all the events at their respective venues at the same time in the municipality. The same definition was used in assignment 2 for the `allocate` method – refer to that handout for an example if you are not sure). For each traffic corridor with greater than zero traffic caused by the current allocation, the user should be able to see, in a readable format:
    - the `toString()` representation of the traffic corridor (which includes its start location, end location and maximum capacity)
    - the amount of traffic on that corridor caused by the current allocation

  The listing of those traffic corridors and their respective traffic, should appear in the order defined by the natural ordering of the `Corridor` class (like in the `toString()` method of the `Traffic` class). Traffic corridors with 0 traffic caused by the current allocation should not appear in the view of the traffic caused by the current allocation.

- At any point, the user should be able to request that an event is added to the *current allocation of events to venues*. In particular the user should be able to input:
    - the name of the event
    - the size of the event

  and they should be able to
    - select a *venue from the municipality* that they would like to allocate the event to

        - A choice of the *venues from the municipality* should be available (i.e. visible) to the user so that they don't have to have prior knowledge of the venues read in from the file when the program is initialised.
        - The user should only be able to select a venue from the municipality.
    - and then request that the specified event be allocated to the selected venue.

  In response to such a request the program should:

  If either (i) the name of the event is the empty string "" or the size of the event is not an integer greater than 0, or (ii) the specified event is currently allocated to a venue in the municipality, or (iii) the selected venue is currently allocated to another event, or (iv) the venue is not large enough to host the event, or (v) the venue is large enough to host the event, but allocating the given event to the selected venue would make the *traffic caused by the current allocation* unsafe (what it means for a traffic object to be safe is specified by the `isSafe()` method of the `Traffic` class), then an appropriate error message should be clearly displayed (in the graphical user interface) to the user, and no changes to the *current allocation of events to venues* (and hence the *traffic caused by the current allocation*) should occur. The error message displayed to the user should clearly identify the reason that the allocation could not be made. Otherwise, the *current allocation of events to venues* should be updated by allocating the given event to the specified venue, and the view of the *current allocation of events to venues*, and the *traffic caused by the current allocation* of events to venues should be updated accordingly.

- At any point the user should be able to request that an event is removed from the *current allocation of events to venues*. The user should be able to
    - select an event that has currently been allocated to a venue, and request for it to be removed from the allocation.

  In response to the user's request, the event should be removed from the *current allocation of events to venues*, and the view of the *current allocation of events to venues*, and the *traffic caused by the current allocation* of events to venues should be updated accordingly.

Your program should be robust in the sense that incorrect user inputs should not cause the system to fail. Appropriate error messages should be displayed to the user if they enter incorrect inputs.

Task: Using the MVC architecture, you must implement `EventAllocator.java` in the `planner.gui` package by completing the skeletons of the three classes: `EventAllocatorModel.java`, `EventAllocatorView.java` and `EventAllocatorController.java` that are available in the zip files that accompanies this assignment. (Don't change any classes other than `EventAllocatorModel.java`, `EventAllocatorView.java` and `EventAllocatorController.java` since we will test your code with the original versions of those other files.)

You should design your interface so that it is legible and intuitive to use. The purpose of the task that the interface is designed to perform should be clear and appropriate. It must be able to be used to perform the tasks as described above.

As in Assignment 1 and 2, you must implement `EventAllocatorModel.java`, `EventAllocatorView.java` and `EventAllocatorController.java` as if other programmers were, at the same time, implementing the classes that instantiate them and call their methods. Hence:

- Don't change the class names, specifications, or alter the method names, parameter types, return types, exceptions thrown or the packages to which the files belong.

- You are encouraged to use Java 8 SE classes, but no third party libraries should be used. (It is not necessary, and makes marking hard.)
- Don't write any code that is operating-system specific (e.g. by hard-coding in newline characters etc.), since we will batch test your code on a Unix machine.
- Your source file should be written using ASCII characters only.
- You may define your own private or public variables or methods in the classes `EventAllocatorModel.java`, `EventAllocatorView.java` and `EventAllocatorController.java` (these should be documented, of course).

Implement the classes as if other programmers are going to be using and maintaining them. Hence:

- Your code should follow accepted Java naming conventions, be consistently indented, readable, and use embedded whitespace consistently. Line length should not be over 80 characters. (Hint: if you are using Eclipse you might want to consider getting it to automatically format your code.)
- Your code should use private methods and private instance variables and other means to hide implementation details and protect invariants where appropriate.
- Methods, fields and local variables (except for-loop variables) should have appropriate comments. Comments should also be used to describe any particularly tricky sections of code. However, you should also strive to make your code understandable without reference to comments; e.g. by choosing sensible method and variable names, and by coding in a straightforward way.
- Allowable values of instance variables must be specified using a class invariant when appropriate.
- You should break the program up into logical components using MVC architecture.
- The methods that you have to write must be decomposed into a clear and not overly complicated solution, using private methods to prevent any individual method from doing too much.

Hints: You should watch the piazza forum and the announcements page on the Blackboard closely – these sites have lots of useful clarifications, updates to materials, and often hints, from the course coordinator, the tutors, and other students.

The assignment requires the use of a number of components from the JavaFX library. You should consult the documentation (http://docs.oracle.com/javase/8/javase-clienttechnologies.htm) for these classes in order to find out how they work. You can also ask tutors, and ask questions on piazza.

User interface design, especially layout, is much easier if you start by drawing what you want the interface to look like. You're layout does not need to be fancy (just legible, intuitive etc. as above), and we don't expect you to use a layout tool to create it – if you do then your code quality might not be very good. You'll have to have a look at the java libraries to work out what controls (http://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui_controls.htm - JFXUI336) that you'd like to use, and what layout options are available (http://docs.oracle.com/javase/8/javafx/layout-tutorial/builtin_layouts.htm#JFXLY102).

You can read about the MVC design pattern in the lecture material from week 9 – the Calculator code example will be a useful reference.

Submission: Submit your files **`EventAllocatorModel.java`**, **`EventAllocatorView.java`** and **`EventAllocatorController.java`**, electronically using Blackboard according to the exact instructions on the Blackboard website:

https://learn.uq.edu.au/

You can submit your assignment multiple times before the assignment deadline but only the last submission will be saved by the system and marked. Only submit the files listed above.

You are responsible for ensuring that you have submitted the files that you intended to submit in the way that we have requested them. You will be marked on the files that you submitted and not on those that you intended to submit. Only files that are submitted according to the instructions on Blackboard will be marked.

Evaluation: Your assignment will be given a mark out of 15 according to the following marking criteria.

Manual testing of the GUI (6 marks)

We will manually test the expected functionality (as described in this handout) of your GUI by attempting to perform a number of scenarios.

Full marks will be given if your interface can reasonably be used to perform all of the defined scenarios correctly. Part marks will be given based on the number of scenarios that your GUI is able to perform correctly. Work with little or no academic merit will receive 0 marks.

Note: code submitted with compilation errors will result in zero marks in this section. A Java 8 compiler will be used to test code. We will execute your code by running the main method defined in `EventAllocator.java`.

Usability (user interface design) (3 marks)

- Interface is legible and intuitive to use. The purpose of the task that the interface is designed to perform in clear and appropriate. The interface can be easily used to perform the tasks as defined in this handout. No additional and unnecessary features.
  3 marks
- Minor problems, e.g., some aspect of the interface is not able to be well-discerned or the interface can't be used to perform all of the tasks defined in the handout (if some aspect isn't functional, then there are usability issues).
  2 marks
- Major problems, e.g. the purpose of the task that the interface is designed to perform is not clear and appropriate, or the interface cannot be used to perform most of the tasks as defined in this handout. 1 mark
- Work with little or no academic merit 0 marks

Note: code submitted with compilation errors will result in zero marks in this section. A Java 8 compiler will be used to test code. We will execute your code by running the main method defined in `EventAllocator.java`.

Code quality (6 marks)

• Code that is clearly written and commented, and satisfies the specifications and requirements	6 marks
• Minor problems, e.g., lack of commenting	4-5 marks
• Major problems, e.g., code that does not satisfy the specification or requirements, or is too complex, or is too difficult to read or understand.	1-3 marks
• Work with little or no academic merit	0 marks

Note: you will lose marks for code quality for:

- breaking java naming conventions or not choosing sensible names for variables;
- inconsistent indentation and / or embedded white-space or laying your code out in a way that makes it hard to read;
- having lines which are excessively long (lines over 80 characters long are not supported by some printers, and are problematic on small screens);
- for not using private methods and private instance variables and other means to hide implementation details and protect invariants where appropriate
- not having appropriate comments for classes, methods, fields and local variables (except for-loop variables), or tricky sections of code;
- inappropriate structuring: Failure to break the program up into logical components using MVC architecture
- monolithic methods: if methods get long, you must find a way to break them into smaller, more understandable methods using procedural abstraction.
- incomplete, incorrect or overly complex code, or code that is hard to understand.