**CSSE2002 / CSSE7023** **Semester 1, 2016**

# Assignment 3

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

**Due date:** The assignment is due at **4pm on Wednesday 1st June** Late assignments will lose 20% of the total mark immediately, and a further 20% of the total 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 (78-425) or by email to enquiries@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.

**School Policy on Student Misconduct:** 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

This is an individual assignment. 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**. Do not discuss or post solutions to the assignment on public forums, like piazza.

**Problem description:** In this assignment you will develop a GUI for simulating the behaviour of a train management system using the Model-View-Controller design pattern. This extends the work you have done in Assignments 1 and 2.

The train management system in question is quite simplistic. It manages the allocation of trains to sub-routes[1] for a particular track in a way that is guaranteed to prevent train collisions. The idea is that a train is only allowed to move along the sub-route it has been allocated to, but it can move freely along that sub-route without fear of colliding with another train.

There may be zero or more trains on the track. Each train on the track has a unique identifier (an integer), a route that it is following, and a sub-route of that route that it has currently been allocated to. The route a train is following must be on the track, and the sub-route that the train is allocated to must be a sub-route of the route that the train is following. The sub-routes that trains have been allocated to may not intersect.

[1] A sub-route is just a route that is part of a larger route. Trains are allocated to sub-routes of a larger route that they are following through the track.

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 train management program is executed (by running the main method in `RailwayManager.java`), the program should
  - load the track defined in the file `track.txt` using the `read` method from `TrackReader`.
  - initialise itself so that it has no trains to begin with, and so that its track is set to be the track that was read in from the file.

  An appropriate error message should be clearly displayed (in the graphical user interface) if there is an error reading from the input file or there is an error with the input format (i.e. if `TrackReader.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 `track.txt` could not be loaded, and why. 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 track 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 track could be loaded, then the track of the program should be set to be the track that was read, and it should be evident to the user that there are currently no trains on the track.

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

Assuming that the track for the train management program can be loaded, the user should be able to use the program to perform the following tasks.

- At any point, the user should be able to request that a new train is added to the track. In particular the user should be able to input:
  - the name of a file (e.g. `route.txt`) from which the route of that train can be read
  - a start and end-offset (`startOffset` and `endOffset`) on that route that specifies the sub-route of the train's route that the train initially wants to be allocated to.

  In response to such a request the program should:
  - load the route defined in the file specified using the `read` method from `RouteReader`.
  - check that the route read from the file is on the train management system's track (i.e. use the `onTrack` method of the `Route` class),
  - check that the offsets define a valid sub-route of the route, `route`, that was read, i.e. `0 <= startOffset < endOffset <= route.getLength()`.
  - check that the sub-route `route.getSubroute(startOffset, endOffset)` does not intersect with any of the sub-routes currently allocated to other trains. (Note: you can check whether or not a route intersects with another one using the `intersects` method of the `Route` class.)

  If either (i) the route could not be loaded, or (ii) the route could be loaded, but it is not on the train management system's track, or (iii) the route could be loaded and is on the track, but the offsets do not define a valid sub-route of the route that was read, or (iv) the route could be loaded, it is on the track, and the sub-route is valid w.r.t. the train's route, but the sub-route

`route.getSubroute(startOffset, endOffset)` intersects with at least one of the sub-routes currently allocated to another train, then an appropriate error message should be clearly displayed (in the graphical user interface) to the user, and the train should not be loaded, and the existing trains and their allocations should not be modified in any way. The error message displayed to the user should clearly identify the reason for the error.

Otherwise the train, with its route, is added to the system and allocated to the sub-route that it requested. It should be given a unique integer identifier that corresponds to the order in which the train was loaded (i.e. the first train loaded has identifier 0, the second train loaded has identifier 1, the third train loaded has identifier 2, etc.)

- At any point the user should be able to request to view the allocation of a particular train. The trains should be able to be selected based on their unique identifier. When a train is selected for viewing the user should be able to see, in a readable format:
    - the identifier of the train
    - the start and end offset of the sub-route that is currently allocated to the train
    - the whole route that the train is following

- At any point the user should be able to request that the allocation of a train is updated. The user should be able to:
    - select a train to update based on the train's unique identifier
    - specify a new start and end-offset (`startOffset` and `endOffset`) of the train's route that it would like to be allocated to instead of it's current allocation

  In response to the user's request to update the allocation from its existing allocation to `route.getSubroute(startOffset, endOffset)` the program should
    - check that the offsets define a valid sub-route of the train's route, `route`, i.e. `0 <= startOffset < endOffset <= route.getLength()`.
    - check that the sub-route `route.getSubroute(startOffset, endOffset)` does not intersect with any of the sub-routes currently allocated to <u>other</u> trains (i.e. trains other than this one). (Note: you can check whether or not a route intersects with another one using the `intersects` method of the `Route` class.)

  If either (i) the offsets do not define a valid sub-route of the route that was read, or (ii) the sub-route `route.getSubroute(startOffset, endOffset)` intersects with any of the sub-routes currently allocated to other trains, then an appropriate error message be clearly displayed (in the graphical user interface) to the user, and the existing trains and allocations should not be modified in any way. The error message displayed to the user should clearly identify the reason for the error. Otherwise, the allocation of the train should be updated to the requested allocation.

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.

**<u>Task:</u>** Using the MVC architecture, you must implement `RailwayManager.java` in the `railway.gui` package by completing the skeletons of the three classes: `RailwayModel.java`, `RailwayView.java` and `RailwayController.java` that are available in the zip files that accompanies this assignment. (Don't change any classes other than `RailwayModel.java` , `RailwayView.java` and `RailwayController.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 `RailwayModel.java`, `RailwayView.java` and `RailwayController.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 `RailwayModel.java`, `RailwayView.java` and `RailwayController.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 Swing library. You should consult the documentation for these classes in order to find out how they work, as well as asking questions on piazza and asking tutors and the course coordinator.

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 (https://docs.oracle.com/javase/8/docs/technotes/guides/swing/index.html) to work out what widgets (https://docs.oracle.com/javase/tutorial/uiswing/components/index.html) that you'd like to use, and

what layout options (e.g. https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html) are available.

You can read about the MVC design pattern in the lecture material from week 8 – the Calculator code example will be a useful reference.

**Submission:** Submit your files **`RailwayModel.java`**, **`RailwayView.java`** and **`RailwayController.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 `RailwayManager.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 we can't check how useable it is).
  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 `RailwayManager.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.