School of ITEE
CSSE2002/7023 — Semester 1, 2020
Assignment 2 (20%)
Due: 22 May 2020 17:00
Revision: 1.0.1

Abstract

The goal of this assignment is to implement a set of classes and interfaces[1] to be used to create a simulation of a traffic management system. You will implement precisely the public and protected items described in the supplied documentation (no extra public/protected members or classes). Private members may be added at your own discretion.

Language requirements: Java version 13, JUnit 4

Preamble

All work on this assignment is to be your own individual work. As detailed in Lecture 1, code supplied by course staff (from this semester) is acceptable, but there are no other exceptions. You are expected to be familiar with "What not to do" from Lecture 1 and `https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`. If you have questions about what is acceptable, please ask course staff.

Introduction

In this assignment you will finish building a simple simulation of a traffic management system (TMS). A traffic management system monitors traffic flow in a region and adjusts traffic signals to optimise traffic flow. A TMS uses different types of sensors and signals to monitor and manage traffic flow. In the first assignment you implemented the core model for the TMS. In the second assignment you will implement some of the more advanced logic to provide a very simple simulation for the TMS.

In addition to the pressure pads and speed cameras from assignment one, you willl add a vehicle count sensor. It counts vehicles passing a location and reports the traffic flow as the number of vehicles in a time period. You need to integrate this new type of sensor into the system. This is an example of a common situation when building a large system. New features need to be added to the system. A well designed system that uses interfaces to define an API means it should be simple to add the new feature.

In assignment one, you implemented traffic lights and electronic speed signs and attached them to a route. In assignment two you will provide logic to coordinate traffic lights at intersections.

The TMS monitors sensors along routes and manages signals on routes, and at intersections, to optimise traffic flow. In assignment one, the network of routes was implicitly defined by your test code and `SimpleDisplay`. In assignment two you will implement the logic for the TMS to maintain a network of routes. This includes the ability to load a network from a data file and save a modified network to a file.

Monitoring and managing congestion requires sophisticated logic in a real TMS. In assignment one congestion was simply reported by each sensor. In assignment two you will implement logic for congestion calculators. These take the congestion data from a set of sensors and determine overall congestion for the route(s) covered by the sensors. The approach taken is to define a

[1] From now on, classes and interfaces will be shortened to simply "classes"

`CongestionCalculator` interface that provides an API. Different classes can implement this interface to provide different options for the logic of determining congestion. This is another example of a common approach to designing flexibility into the system's structure.

When implementing the assignment you need to remember that it is implementing a simulation of the TMS and not the real TMS. Interfaces are provided for the sensors to allow easy replacement of sensor implementations in the program. You will not be collecting data from real sensors but will be implementing classes that demonstrate the behaviour of sensors. They store a set of data values that are used to simulate the sensors returning different values over time. Signals are simple simulations of real signals, in that they only store the current state of the signal and allow the route to update the signal.

To manage simulation of time, there is a `TimedItem` interface and a `TimedItemManager` class, which you implemented in assignment one. Sensors implement the `TimedItem` interface, as they are items which need to react to timed events. `TimedItemManager` stores all the `TimedItem` objects in the application. The simulation's GUI tracks time passing in `MainView.run()` and it invokes `MainViewModel.tick()` once per second. The `tick` method calls the `TimedItemManager`'s `oneSecond` method, which sends the `oneSecond` message to all `TimedItem`s. This approach of tracking the passage of time and invoking an action on all relevant objects once per second was the reason that `TimedItemManager` is implemented as a singleton[2].

A simple GUI has been provided to you as part of the provided code. It is in the `tms.display` package. It will not work until you have implemented the other parts of the assignment that it uses. The GUI has been implemented using JavaFX and consists of three classes and an enum. `MainView` creates the main window for the TMS GUI. `StructureView` displays the structure of the traffic network. `MainViewModel` represents the TMS model that is to be displayed. The TMS application is initialised and started by the `Launcher` class in the `tms` package. It loads the traffic network data and creates the GUI. Most of the GUI code has been provided to you. In `MainViewModel` you need to implement some of the logic that is executed by events in the simulation and to handle keyboard input for the main application's window.

The functionality you need to implement in `MainViewModel` is to:

- Save the state of the network to a file in response to the user selecting the save command. This is to be implemented in `MainViewModel.save()`.
- Allow the simulation's execution to be paused and unpaused. This is to be implemented in `MainViewModel.togglePaused()`.
- Process time passing in the simulation. This is to be implemented in `MainViewModel.tick()`.

Keyboard input is handled by the `accept` method in the `MainViewModel` class. It needs to process input from the user in the main window to perform actions in the simulation. Pressing the 'P' key will toggle whether the simulation is paused or not. The 'Q' key will quit the simulation. The 'S' key will save the current network to a file called "DefaultSave.txt". A shell for this method has been provided because it is already hooked into the GUI.

Persistent Data

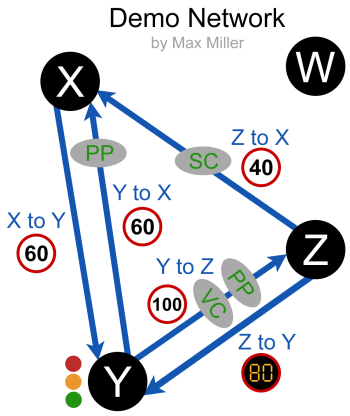
You need to implement loading a network from a data file. The JavaDoc for the `loadNetwork` method in the `NetworkInitialiser` class describes the format of a network data file. Saving a network is done by the `save` method in the `MainViewModel` class. A network data file is structured as follows:

- The first line is the number of intersections (ni) in the file.
- The second line is the number of routes in the file.
- The third line is the duration of a yellow light.

[2] `https://refactoring.guru/design-patterns/singleton` provides a reasonably detailed description of the singleton pattern.

- The following *ni* lines are the intersection details.
  - The first part of an intersection line is its id.
  - This is optionally followed by a ':', a duration, another ':', and a sequence of intersection ids which are separated by commas.
- The final set of lines are the route details, including any sensors on the routes.
  - Each route is on a separate line. The sensors for a route are on the lines immediately after the line for the route.
  - A route is described by the id of the from intersection, followed by a ':', then the id of the to intersection, followed by a ':', then the default speed for the route, followed by a ':', then the number of sensors on the route, then optionally a ':' and the speed of the electronic speed sign on the route if it has one.
  - If the route has any sensors, each sensor follows on separate lines.
  - The first part of a sensor line is its type 'PP', 'SC' or 'VC'. This is followed by a ':', then its threshold value, a ':', and then a comma separated list of the data values used to simulate the data returned by the sensor.
- Any line that starts with a semi-colon ';' is a comment and is to be ignored when reading the data from the file.
- Attempting to read an invalid network data file should throw an `InvalidNetworkException`.

An example data file, called `demo.txt`, is provided in your repository in the `networks` directory. It corresponds to the diagram below.



Supplied Material

- This task sheet.
- An example network data file.
- Code specification document (Javadoc).[3]
- A Subversion repositiory for submitting your assignment called `ass2`.[4]
- A simple graphical user interface for the simulation, which is in the `display` package.
- A sample solution for the first assignment. You are to use this as the base for your implementation of the second assignment. As the first step in the assignment you should create a new project by checking out the `ass2` repository from Subversion.

[3] Detailed in the `Javadoc` section
[4] Detailed in the `Submission` section

Javadoc

Code specifications are an important tool for developing code in collaboration with other people. Although assignments in this course are individual, they still aim to prepare you for writing code to a strict specification by providing a specification document (in Java, this is called Javadoc). You will need to implement the specification *precisely* as it is described in the specification document.

The Javadoc can be viewed in either of the two following ways:

1. Open `https://csse2002.uqcloud.net/assignment/2/` in your web browser. Note that this will only be the most recent version of the Javadoc.
2. Navigate to the relevant assignments folder under `Assessment` on Blackboard and you will be able to download the Javadoc `.zip` file containing html documentation. Unzip the bundle somewhere, and open `docs/index.html` with your web browser.

Tags in the Javadoc indicate what code has been implemented in assignment one and what code you need to implement in assignment two. Some code from assignment one will need to be modified. There are tags indicating places where you can expect to modify the assignment one code but these are not guaranteed to be all of the places where you may end up modifying code from assignment one.

Tasks

1. Implement the classes and methods described in the Javadoc as being requried for assignment two.
2. Implement the indicated features of the user interface.
3. Write JUnit 4 tests for all the methods in the following classes:
   - `AveragingCongestionCalculator` (in a class called `AveragingCongestionCalculatorTest`)
   - `IntersectionLights` (in a class called `IntersectionLightsTest`)
   - `NetworkInitialiser` (in a class called `NetworkInitialiserTest`)

Marking

The 100 marks available for the assignment will be divided as follows:

Symbol	*Marks*	*Marked*	*Description*
F	45	Electronically	Functionality according to the specification
R	35	Course staff	Code review (Style and Design)
J	20	Electronically	Whether JUnit tests identify and distinguish between correct and incorrect implementations

The overall assignment mark will be $A_2 = F + R + J$ with the following adjustments:

1. If $F < 5$, then $R = 0$ and code style will not be marked.
2. If $R > F$, then $R = F$.

For example: $F = 22, R = 25, J = 17 \Rightarrow A_2 = 22 + 22 + 17$.
The reasoning here is to place emphasis on functional code and to not to give marks to well styled code and well implemented JUnit tests when the code is not functional.

Functionality Marking

The number of functionality marks given will be

$$F = \frac{\text{Functionality tests passed}}{\text{Total number of functionality tests}} \cdot 45$$

Each of your classes will be tested independently of the rest of your submission. Other required classes for the tests will be copied from a working version of the assignment.

Code Review

Your assignment will be reviewed and style marked with respect to the course style guide, located under `Learning Resources > Guides`. The marks are broadly divided as follows:

Naming	5
Commenting	7
Structure and Layout	7
Code Design	8
Object-Oriented Practices	8

For Code Design we are looking for well thought out code that is easily followed and is maintainable. For Object-Oriented Practices we are looking for the course content to be applied to solve certain challenges present in the assignment.

Note that style marking does involve some aesthetic judgement (and the marker's aesthetic judgement is final).

JUnit Test Marking

The JUnit tests that you provide in `AveragingCongestionCalculatorTest`, `IntersectionLightsTest` and `NetworkInitialiserTest` will be used to test both correct *and* incorrect implementations of their respective classes. Marks will be awarded for test sets which distinguish between correct and incorrect implementations[5]. A test class which passes every implementation (or fails every implementation) will likely get a low mark. This will be assessed by running your JUnit test classes on a number of correct and incorrect assignment implementations. Marks will be rewarded for tests which pass or fail correctly.

There will be some limitations on your tests:

1. if your tests take more than 20 seconds to run, or
2. if your tests consume more memory than is reasonable or are otherwise malicious,

then your tests will be stopped and a mark of zero given. These limits are very generous (your tests should not take *anywhere* near 20 seconds to run and in all probability could take well less than 1 second).

Electronic Marking

The electronic aspects of the marking will be carried out in a linux environment. The environment will not be running Windows, and neither IntelliJ nor Eclipse (or any other IDE) will be involved. OpenJDK 13 and JUnit 4.12 will be used to compile and execute your code and tests.

It is critical that your code compiles. If one of your classes does not compile, **you will recieve zero** for any electronically derived marks for that class.

[5] And get them the right way around

Submission

Submission is via your Subversion repository. You must ensure that you have committed your code to your repository *before* the submission deadline. Code that is submitted after the deadline will **not** be marked. Failure to submit your code through your repository will result in it **not** being marked. Details for how to submit your assignment are available in the `Version Control Guide`. Your repository url is:

`https://source.eait.uq.edu.au/svn/csse2002-s???????/trunk/ass2` — **CSSE2002** students
or
`https://source.eait.uq.edu.au/svn/csse7023-s???????/trunk/ass2` — **CSSE7023** students

Your submission should have the following internal structure:

`src/`	folders (packages) and `.java` files for classes described in the Javadoc
`test/`	folders (packages) and `.java` files for the JUnit test classes

A complete submission would look like:

```
src/tms/congestion/AveragingCongestionCalculator.java
src/tms/congestion/CongestionCalculator.java
src/tms/display/ButtonOptions.java
src/tms/display/MainView.java
src/tms/display/MainViewModel.java
src/tms/display/StructureView.java
src/tms/intersection/Intersection.java
src/tms/intersection/IntersectionLights.java
src/tms/network/Network.java
src/tms/network/NetworkInitialiser.java
src/tms/route/Route.java
src/tms/route/SpeedSign.java
src/tms/route/TrafficLight.java
src/tms/route/TrafficSignal.java
src/tms/sensors/DemoPressurePad.java
src/tms/sensors/DemoSensor.java
src/tms/sensors/DemoSpeedCamera.java
src/tms/sensors/DemoVehicleCount.java
src/tms/sensors/PressurePad.java
src/tms/sensors/Sensor.java
src/tms/sensors/SpeedCamera.java
src/tms/sensors/VehicleCount.java
src/tms/util/DuplicateSensorException.java
src/tms/util/IntersectionNotFoundException.java
src/tms/util/InvalidNetworkException.java
src/tms/util/InvalidOrderException.java
src/tms/util/RouteNotFoundException.java
src/tms/util/TimedItem.java
src/tms/util/TimedItemManager.java
src/tms/Launcher.java
test/tms/congestion/AveragingCongestionCalculatorTest.java
test/tms/intersection/IntersectionLightsTest.java
test/tms/network/NetworkInitialiserTest.java
test/tms/JdkTest.java
```

Ensure that your assignments correctly declare the package they are within. For example, `CongestionCalculator.java` should declare `package tms.congestion`.

Do not submit any other files (e.g. no `.class` files). Note that `AveragingCongestionCalculatorTest`, `IntersectionLightsTest` and `NetworkInitialiserTest` will be compiled without the rest of your files.

Prechecks

Prechecks will be performed on your assignment repository multiple times before the assignment is due. They will assess whether your folders and files are in the correct structure and whether your public interface aligns with the expected public interface. **Successfully passing a precheck does not guarantee any marks. No functionality or style is assessed.**

```
Precheck #1: Approximately 5pm on the 11/05
Precheck #2: Approximately 5pm on the 18/05
Precheck #3: Approximately 5pm on the 20/05
```

You should endeavour to have code written and in your repository before at least one of these prechecks, preferably the first, in order to make the most of them. No additional prechecks will be run for people who did not start the assignment in time, or who neglected to commit their code to their repository. Prechecks are valid only for currently released version of the Javadoc, if an update is made it may invalidate the precheck results.

Late Submission

Assignments submitted after the due date will recieve a mark of zero unless an extension is granted.

Do not wait until the last minute to commit the final version of your assignment. A commit that starts before 17:00 but finishes after 17:00 will **not** be marked.

Extension Requests

You may request an extension if exceptional circumstances affect your ability to complete the assignment on time. Circumstances that are within your control will not be accepted (e.g. multiple assignments due at the same time, losing work due to a computer crashing, ...). The ECP provides details of how to apply for an extension and the maximum allowed length of an extension. Your extension request must include appropriate evidence for the impact of the circumstances and the duration of that impact. Extensions will not be granted beyond the length of the duration of the impact.

You are expected to start work on the assignment well before the due date. Tutor support for the assignment is available up until the assignment due date. Some support will be provided after the due date for students who have an extension but you cannot expect the same level of support as is provided before the due date. You should not expect assignment help over a weekend. You should not expect an extra pre-check to be run on your submission after the last scheduled pre-check above. That is only two days before the assignment is due and you should have completed enough work to be checked for conformance by then, even if you have an extension on the assignment.

Remark Requests

To submit a remark of this assignment please follow the information presented here: `https://my.uq.edu.au/information-and-services/manage-my-program/exams-and-assessment/querying-result`.

Revisions

1.0.1 – The Javadoc for `Network.makeTwoWay` and `NetworkInitialiser.loadNetwork` has been updated to correct minor issues with their descriptions.

- `Network.makeTwoWay` – The descriptive text in the second paragraph of the Javadoc had 'to' and 'from' the wrong way around.
- `NetworkInitialiser.loadNetwork` – The bullet points listing conditions for invalid files has been expanded to be more descriptive and precise.