INB370 Software Development Assignment 2

Test-Driven Development and Graphical User Interface Programming
Part 2: GUI Development
Semester 1, 2014

Due date: Tuesday, 27th of May, 2014 (Week 13)
Weighting: 30% in total (20% for Part 1, 10% for Part 2)
Assessment type: Group assignment, working in pairs. INB370 and INN370 students will undertake separate assignments, so no mixing is permitted across classes.

Introduction:

This second part of the specification is here to provide some guidance on the development of the GUI application. In part 1, you were required to implement key classes for a system to simulate the operation of a car park. Please note that this task concerns *simulation of car park operation* – not management of the operation.

The GUI task here is thus simpler than it might be, and is concerned far more with reporting than it is with any sort of physical display. A physical display would be fun, but the first half of this assignment is reasonably involved, and so the second half has been adjusted down from 15% to 10%, and so the GUI work will be simpler. I will provide you with a code drop of the example code that I have given to the INN370 class, one producing a simple frame, and the other providing you with examples of the use of the JFreeChart library.

We begin with a quick summary of the tasks to complete this assignment.

Tasks to Complete the Assignment:

Most of the development tasks were well specified in the first part of the specification. In summary, and in a sensible order, the tasks are as follows:

1. Develop classes `Car.java` and `MotorCycle.java` in the `Vehicle` hierarchy (asgn2Vehicles), and the class `CarPark.java` in the `asgn2CarParks` package.
2. Develop a junit test suite for each of these classes (`asgn2Tests`). It is expected that these will be comprehensive but not ridiculous – cover the basics, with normal and valid boundary cases, and the exceptions thrown. Note that you have been given skeletons for these – with autogenerated stubs – in the code drop.
3. Modify the main program in the `SimulationRunner` class to accept command line arguments. I will make some more comments on this below.

In this document, we add some additional tasks focused on the construction of the GUI:

4. Develop a Swing application using a text area, drawing on the facilities from the first part of the assignment. This will be built around the class `GUISimulator.java`, (which probably could have been called something else, but we will live). The goal here will be to organize the text area to receive the output from the logging facilities that have been provided for you. The idea is that this class should provide the frame that should then be invoked from the main program that exists in the `SimulationRunner` class. Obviously, this will have to be adapted significantly (see below)
5. Once you have this basic structure, develop your test inputs and confirm that the logging behaves as expected – before moving on to the charting.
6. Next, consider the charting needed for the graphics display (see below), and embed it in a class called `ChartPanel.java`. This class should also go in the `asgn2Simulators` package.
7. Finally, you should finish the testing of the overall application.

GUI Development:

For GUIs of this limited sophistication, the structure is pretty clear cut. Work mainly through the Class constructor and the `actionPerformed` methods as the main public API for the classes you develop. Inevitably you will use some of the methods inherited from JPanel and JFrame and there will be an abundance of private helper methods.

The structure here also requires that you collect parameters from the GUI and pass those back to wherever you are creating the `CarPark` and `Simulator` objects. This suggests that a fair bit of the work which presently sits in the main program of the `SimulationRunner` class is going to have to sit somewhere in the `GUISimulator`. Play with the design until it makes sense. If it feels clumsy, try again until you have something reasonable. You have some flexibility here.

Remember, somehow, you are going to have to get the data from the model objects – and so you need to consider this in the design of your panels and frame. You must subclass the frame and panels as otherwise you have no control over the parameter lists. The actual design is up to you, provided that you satisfy the requirements laid out in part 1 – accepting input, running the simulation, and viewing the results.

The essential steps in the use of the GUI are as follows:

1. User enters - actually adjusts (see below) - parameters through text fields or other mechanisms.
2. [Invalid parameters should be trapped and the user advised of the error – either directly via the text area, or through some colour or other signal.]
3. The user should then run the simulation. My view is that the user should be offered the choice of the text or charting view, but if you can work out a way to have both without making it look silly, I am happy for you to consider it. If there is a mix of text area and graphics, then the user should have the option of switching between them. More details on these requirements are given below.

Notes:

1. The logging methods are provided to you and these should be used as a model for some, though not all, of the GUI based display. You should, however, continue to maintain the file based log.
2. The Log as set up is not suitable for direct use in the text area – it does the writing directly, and does not return a String. So, you will need to create a method in the GUISimulator class which does some of the same job, taking the result from the `CarPark.getStatus(time)` method – a method which will also supply you with data for the charts.
3. It is not necessary in the GUI environment to log the parameters as these are clearly on display in the data entry fields.
4. The final dump is excessive in the GUI environment, and best left to the log, so you should conclude with the last entry from the status.
5. Now, the `getStatus` method on `CarPark` provides a string with all the information you could ever want to plot. Ideally, it would be nice if `CarPark` had a couple of other getters, but it doesn't, and I'm disinclined to change it now. So, when you have to find the current status of the number of cars, the number parked, the number dissatisfied and so on, you will have to call the `getStatus` method, and parse the `String` that comes back. A simple enough task – the data is there, the format is well known. Just do it. Some details are given below.
6. The GUI should guide the user by disabling, as appropriate, controls that are not available at particular stages of the process.

Parameters:

The constants class provides three distinct parameter groups. We consider these in turn as command line arguments and GUI data entry fields. For groups 2 and 3, you should make them available as command line parameters, which replace these default values.

Group 1: Don't even think about changing these or making them command line arguments. They are there for all time, for all simulations. We are done here.

```
//Basic simulation time parameters - unchangeable
public static final int MINIMUM_STAY = 20;
public static final int MAXIMUM_QUEUE_TIME = 25;
public static final int CLOSING_TIME = 18*60;
public static final int LAST_ENTRY = CLOSING_TIME-60;
```

Group 2: The ratios will stay roughly the same, but we may vary the numbers quite frequently

```
//Default Size parameters
public static final int DEFAULT_MAX_CAR_SPACES = 100;
public static final int DEFAULT_MAX_SMALL_CAR_SPACES = 30;
public static final int DEFAULT_MAX_MOTORCYCLE_SPACES = 20;
public static final int DEFAULT_MAX_QUEUE_SIZE = 10;
```

Group 3: The probabilities – likely to change often. I would advise that you keep the SD roughly at that fraction of the mean for the durations.

```
//RNG and Probs
public static final int DEFAULT_SEED = 100;
public static final double DEFAULT_CAR_PROB = 1.0;
public static final double DEFAULT_SMALL_CAR_PROB = 0.20;
public static final double DEFAULT_MOTORCYCLE_PROB = 0.05;
public static final double DEFAULT_INTENDED_STAY_MEAN = 120.0;
public static final double DEFAULT_INTENDED_STAY_SD =
0.33*Constants.DEFAULT_INTENDED_STAY_MEAN;
```

Group 1 is not considered further. For the others, the process is as follows:

Before you write the GUI:

- Adapt my main program to take command line arguments. These should be on an all or nothing basis.
- So if there are no command line arguments given, we just use the defaults. Otherwise, there must be ten of them, enough for the `CarPark` and `Simulator` constructors (see below)
- At this point, just make sure that you adapt the program to work – don't worry about error checking yet. You will need to deal with the command line arguments, to terminate with an error message if there are one or more arguments but not the full list and to run with the defaults if no arguments are specified. This means working with the multi-parameter constructors.
- You may find the java tutorial helpful again:
  http://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html

Then, once you start working with the GUI, groups 2 and 3 are naturally going to fall together in the UI, although there may be some subdivision. But our ideas now change:

- If there are no command line arguments, populate the text fields (or other widgets) for the simulation parameters with the defaults.
- If there *are* ten command line arguments, populate the text fields (or other widgets) for the simulation parameters with these values.
- In either case, we give the user the chance to make adjustments. The user *then* must start the simulation running (probably with a start button).

At this point, you should do the proper checks on the `CarPark`:

- maxCarSpaces, maxMotorCycleSpaces, maxQueueSize >= 0
- 0 <= maxSmallCarSpaces <= maxCarSpaces

There are similar checks to be performed on the `Simulator`, but if convenient you may rely on the exception thrown from the constructor (not true for `CarPark`). The constraints are as laid out in the javadoc for the constructor. Either way, you must make sure that the simulation cannot proceed while the data is invalid.

Optionally, you may deal with invalid data by an error message via the text area or modal dialog, or by:

1. Handling appropriate text change events in the text entry box and prevent the user launching the simulation until the input is valid
2. Disabling and/or greying out as appropriate when input or the display option is invalid.

Charting:

Those seeking full marks for the assignment must produce a chart based report using JFreeChart. You should have access to any java.awt colour. These are suggestions.

Progress:
X-axis:
time – as an integer

Plotted Series (based on getStatus):
Black – Number of vehicles to date
Blue – Currently parked vehicles (total)
Cyan – Total cars in car park
Gray – Number of small cars in car park

DarkGray – Number of motorcycles in car park
Yellow – Number of vehicles in Queue
Green – Number archived
Red – Number Dissatisfied

Chart 2: Summary (at end of simulation):
A final, simple bar chart showing the total number of vehicles with a bar beside it showing the number of dissatisfied customers.

There are many other reports possible, including end of simulation reports, but these would require parsing of the final report and this is already enough for the marks.

GUI Testing:

Both of you should be responsible for the look and feel of the interface, but testing requires that you work through some scenarios, and it is best if you can try to test the logic the other person coded.

Testing of the GUI will be based on a series of usage scenarios and the data associated with them. These will include a few standard examples that I will supply later, but for now, explore:

- High car prob
- Small car parks
- High small car prob
- No queue

In the final submission, you will document the scenario, the data used, and a series of screenshots to show me that it worked. Something in the vicinity of five such scenarios will be sufficient. The easiest way of presenting these may be to use a PPTX or pdf slide deck.

Here is a formal version of one of these from an earlier assignment, which had a locomotive system [PAG means Perform a Gesture, which is a neutral way of saying this without making it a button or some other specific widget]. The plain font is what you do. The italicised text is what the system does.

Script 1: Build a Train

1. PAG to add a Locomotive of class 4D of weight 180 tonnes
2. *The System display shows the locomotive in the train configuration, indicates that the train is not overloaded, and shows a capacity of zero passengers and that the train is full.*
3. PAG to add a passenger carriage of weight 80 tonnes and capacity 50
4. *The System display shows the locomotive and a single passenger carriage in the train configuration, indicates that the train is not overloaded, shows a remaining capacity of 50 passengers, occupancy of 0/50 in carriage 1, and indicates that the train is not full.*
5. PAG to add a passenger carriage of weight 80 tonnes and capacity 50
6. *The System display shows the locomotive and two passenger carriages in the train configuration, indicates that the train is not overloaded, shows a remaining capacity of 100 passengers, occupancy of 0/50 in carriage 1 and 2, and indicates that the train is not full.*

In our case, we follow this style, even if the test cases are simpler.

Assessment

GUI Design:

We have previously noted that GUI design need not be world class, but that it should not be an abomination either. Some basic principles follow, and these also are clear in the CRA. Your GUI should:

- Be well laid out with related information and functional widgets grouped together;
- Have a good alignment between the display widgets chosen and the type of information to be displayed;
- Automatically update information displays affected by other operations;
- Not appear cluttered – Google's is, or at least used to be, an example of a clean interface.

Students are permitted to use a GUI Builder such as the Visual Editor for Eclipse provided that this is explicitly declared in the submission, and provided they understand that a higher standard of GUI design will be expected for the same mark level. As always, students are permitted to work in environments other than Eclipse but the final product *must* be submitted as a working Eclipse archive. We will not under any circumstances work through a cumbersome import process for an assignment.

If you wish to use other libraries in addition to `swing` for the GUI display please contact us directly. We will look at this on a case by case basis, but if we say yes the expectations will follow the same path as for the GUI builder, with a higher standard expected for the same mark level.

Code Quality:

Whilst GUI code can at times be very ugly, there is no excuse for sloppy coding style and documentation. The standard of elegance for GUI code is lower than for other contexts, but there are still some violations that are plainly not acceptable. To make your code more readable, try:

- Some terse single line comments above a block of code - highligting material that is less than obvious;
- Lots of private methods to make sense of lots of widget creation and organization. The refactoring functions on the right click menu are your friends. Get to know them.
- As always, we recommend following a recognised coding convention, such as that described in the *Code Conventions for the Java Programming Language*[1].

Source Control:

As noted elsewhere, we expect that you will use a recognized source control program to manage the codebase for this assignment, and that your logs will be submitted as part of the assignment. Please do not fake your logs to make them nicer – unless you do your first commit some time the night before the assignment is due, you will almost certainly get a decent mark for this. For most of you, this is the first time you have used source control in any decent way, and so we are really just looking to see that you have given it a proper go. We will indicate our submission requirements for these in the submission doc.

---

[1] http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

[Copied across from part 1]

Academic Integrity

Please read and follow the guidelines in QUT's *Academic Integrity Kit*, which is available from the INB370 Blackboard site on the Assessment page. Programs submitted for this assignment will be analysed by the MoSS (Measure of Software Similarity) plagiarism detection system (http://theory.stanford.edu/~aiken/moss/).

Assessment

Part 1 of submitted assignments will be tested automatically, so you must adhere precisely to the specifications in these instructions and on Blackboard. Your program code classes will be unit tested against our own test suite to ensure that they have the necessary functionality. Your unit test classes will be exercised on defective programs to ensure that they adequately detect programming errors. As before, we will supply a testing program to enable you to make sure that your code is consistent with the spec. The precise assessment criteria for the assignment are supplied in a separate document.

Part 2 of submitted assignments will be assessed based on the functionality of your Graphical User Interface. Further details of assessment criteria for Part 2 will be released with the guide for the GUI (this document + the released CRA).

Submitting Your Assignment

Full details of required file formats for submissions will appear on Blackboard near the deadline. You must submit your solution before midnight (actually 11:59) on the due date to ensure that your assignment is accepted. You should take into account the fact that the network might be slow or temporarily unavailable when you try to submit. **Network problems near the deadline will not be accepted as an excuse for late assignments, and QUT's assignment submission policy is now very strict.**

Please make sure that you complete the pracs for week 7, 8 and 9 as these are crucial to your ability to undertake this assignment. The work you are doing depends particularly on the Swing libraries, and on the use of source control and build tools. As noted in the prac guide for week 8, we expect you to work with at least two distinct branches in the source control system, one called development or just dev, the other the master, where you will obviously explore new work on the dev branch and merge it back to the master branch as it becomes stable.

FAQs:

Q: What Vehicle ID should I use?

A: I have used the following. Internal to the cp object, I maintain a count of all the vehicles that have been created. When I create a Car, I use the following:

```
"C"+this.count
```

And for motorcycles:

```
"MC"+this.count
```

See the examples in the question below.

Q: When you are trying to create a car or a motorcycle in the current time step, which one do you do first?

A: Very good question. Completely arbitrary choice in theory, but can make a difference in practice. So please try to make a car and then try to make a motorcycle. Don't forget to test for a small car if the car is created. (I do this in the parameter list of the constructor).

Q: What does your `toString()` output look like?

A: Some examples:

```
Vehicle vehID: C9
Arrival Time: 9
Vehicle was not queued
Entry to Car Park: 9
Exit from Car Park: 124
Parking Time: 115
Customer was satisfied
Car cannot use small parking space

Vehicle vehID: C111
Arrival Time: 104
Vehicle was not queued
Entry to Car Park: 104
Exit from Car Park: 124
Parking Time: 20
Customer was satisfied
Car can use small car parking space

Vehicle vehID: C166
Arrival Time: 157
Vehicle was not queued
Vehicle was not parked
Customer was not satisfied
Car cannot use small parking space
```

```
Vehicle vehID: C143
Arrival Time: 136
Exit from Queue: 147
Queuing Time: 11
Entry to Car Park: 147
Exit from Car Park: 247
Parking Time: 100
Customer was satisfied
Car cannot use small parking space

Vehicle vehID: MC155
Arrival Time: 146
Vehicle was not queued
Entry to Car Park: 146
Exit from Car Park: 274
Parking Time: 128
Customer was satisfied
```

The CarPark version is something like this, though you may need to change things a bit.

```
public String toString() {
      return "CarPark [count: " + count
      + " numCars: " + numCars
      + " numSmallCars: " + numSmallCars
      + " numMotorCycles: " + numMotorCycles
      + " queue: " + (queue.size())
      + " numDissatisfied: " + numDissatisfied
      + " past: " + past.size() + "]";
}
```

```
//Create the service object using defined key

NgramServiceFactory factory =
NgramServiceFactory.newInstance(NGramStore.Key);

if (factory==null) {

   throw new NGramException("NGram Service unavailable");

}
```

We will not require anything more substantial and I *will not require this exception throw to be tested*. We *can* set up a test to break things, but we can't easily control the exceptions and errors that are coming back without spending too much time for it

to be worth it.