# INB370 Software Development Assignment 2

# Test-Driven Development and Graphical User Interface Programming

## Part 1: Test-Driven Development of a Complex OO System

## 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.
***[Version of May 2, 2014**: Fixed due date to show 2014 rather than 2013, added clarification of motorcycle and small car parking].*

## Learning Objectives for the Assignment:

- To experience team-based program development using an approach similar to test-driven development (Part 1 of the assignment).
- To practice Graphical User Interface programming (Part 2 of the assignment).

lostoncampus.com.au/img/poi/medium/h-block-car-park-3306.jpg

## Pair Formation and Guidelines:

Over the years we have run this assignment, there have always been people who have either requested that they be allowed to do the assignment alone (the right way of going about it) or simply handed in an assignment that is solely their own work and hoped that we'd be ok with it (the wrong way of going about it). Please ask if there is a good reason why you can't make it work, but generally the answer will be no unless there are specific and very significant obstacles in the way.

I have in the past had a number of students who work full time contact me to express their concerns about working with other people, with some worried about having opportunities to pair up with other students. I have successful collaborations which involve code with people on the other side of the world, and this is increasingly common. The key of course is to establish the pair. Once you have had a couple of face to face meetings, it will usually work. Mostly working full time is not a sufficient barrier – I have had people working for mining companies in remote locations with lousy internet connectivity, and I have allowed them to work alone. Those in the city should normally be able to connect sufficiently, with occasional face to face meetings.

I don't like randomly allocating people to teams, even if this might happen in real workplaces. So we usually operate an Assignment 2 'dating' page on the BB site to allow people to advertise and then meet up with suitable candidates, and I have mentioned the need to work in pairs earlier in the semester. Here are some guidelines on pair formation.

**Forming Pairs across the UG/PG boundaries:**
In previous years we have permitted students from INB370 and INN370 to work in pairs across the classes. This is no longer feasible as the assignments are fundamentally different in specification, and are not even slightly related. So, you must find a partner in the same class.

**How to Find a Partner:**
Most people seem to find a partner through contact from previous classes or tutorials in the current unit. But if you cannot find one quickly, please use the forum established on BB at Assessment>Assignment Two>Assignment 2 Dating Agency. The link is the same in both units, though of course the forum is different.

I require that you email me by ***Tuesday, May 6*** with the outcome of your searches. To allow me to filter this, please use the subject lines indicated below.

**If you have found a partner**: Please send me one email from one of you, cc'ed to the other, confirming that you will be a pair for the assignment. The subject line should be [INB370 Confirmed Pair] and the mail body should contain your names and student numbers.

**If you have NOT found a partner**: Please advertise on the forum if you have not already done so, and then send me an email to say you are still looking. The subject line should be: [INB370 Need Partner] and the mail body should contain your names and student number, and the message you sent to the forum so I can help match you up if needed.

If you believe your circumstances justify working alone on this assignment, please email me with a detailed request to this effect by ***TUESDAY MAY 6***. I will be pretty ruthless in assigning people to pairs by the end of that week, so please try to sort this out as soon as possible.

## Software Development Guidelines – Working in a Team

Professional large-scale software development inevitably involves working in a team. For the purposes of this assignment you are required to work in pairs and follow the 'agile' programming practices discussed in the lectures so far, and to provide some documentation that you have followed these approaches. This supporting doc should include the use of the @author tag and source control records as discussed below.

1. When developing the classes in part 1, one team member should take the role of the *tester*, and the other should take the role of the *coder*. The tester will develop new unit test cases based on the specifications described, while the coder must respond with program code that passes the tests. [we will relax this slightly, since people never follow it properly anyway ☺ and allow you both to generate the test cases and to then respond with the code. This is the more important development lesson. But if you can do this independently with some success, please do so.]
2. The source control logs should – at least some of the time - show that the tests are committed prior to the code, and the commit log should show some evidence of regular revision and refactoring.
3. Ideally, there should be some regular reversal of these roles so that both team members get some experience of the problem.
4. Both team members should contribute to the design and implementation of the user interface (details in part 2).

**NB:** You are required to used Javadoc '@author' comments before each of the program code and unit test classes to identify who played which role. Include both your name and student number. These same conventions should be employed for source control. Note that we do not specify the source control method to be used, but most people will use Git as previously discussed.

***Producing Professional Quality Code***

The provided Javadoc API specification clearly describes the necessary packages, classes, methods, parameters and return types needed to complete Part 1 of the assignment. As a professional programmer, you must follow these specifications *precisely*. Near enough is *not* good enough.

In addition, you must submit your assignment in the specified zip archive format. More detail on the submission requirements will be released close to the deadline. Since we have already provided Javadoc-style documentation for this part of the assignment you are not required to duplicate these comments in your code, and we will supply you with the raw javadoc source so that you may use it.  However, you must properly comment any code of your own, especially private utility methods that are not part of the specified API.

Your code should also be presented in a professional style. Following a recognised coding convention, such as that described in the *Code Conventions for the Java Programming Language* (see http://www.oracle.com/technetwork/java/codeconvtoc-136057.html) is recommended.

## The Scenario — Managing a Car Park

All of us are uncomfortably familiar with the experience of parking in the centre of a major city. Car parks may have substantial capacity, but inevitably this capacity is finite, and at times we are stuck with the problem of waiting in a queue to gain entry. Queues themselves have limits, as only so many cars can line up for entry before there is an obstruction to traffic. Part of the design process for a new car park involves modelling its operation in order to assess its effectiveness. It is not merely a question of capacity, but rather a question of how that capacity can be utilized when we make assumptions about the length of time each vehicle might stay, the number of spaces available, and the number of vehicles permissible in the queue at a given time.

In Part 1 of this assignment you will play the role of a small programming team tasked with developing a set of Java classes to support an automated system to simulate the operation of a car park. This will lead, naturally enough, to a GUI using these classes in the second part of the assignment. Part 1 (20%) will require that you produce a text based simulator and Part 2 (10%) will require that you use some of these ideas in a GUI environment.

An outline of the specific tasks required to complete Part 1 is given below. Details of the technical requirements for each of the classes to be produced are given in the Javadoc API specification accompanying these instructions. This will require a lot of exploration before you come to grips with what is required. Read the spec and the javadoc carefully, but you should pay particular attention to the assumptions that we have made.

**Building in Stages:**
While it is not compulsory to do so, you may find it easier to consider Part 1 development in two stages: a simpler version without the queue, and then a full version including the queue. The CRA will reflect this division, and those who do not handle the queue may still achieve a passing mark, but it will necessarily be reduced. If you follow this path, a good test of these two versions is to ensure that you get the same results with a maximum queue size of zero as you did with no queue at all.

**Basic Assumptions (the queue specific assumptions are in italics):**

- **Note – these assumptions are lengthy, and intended as something of a checklist. Their overall logic is embodied in the `SimulationRunner.run()` method which is supplied. But you need to understand how all this works.**
- Vehicle Types: The car park will support cars and motorcycles, with the maximum number of each specified by the user.
- The total available car spaces will include a limited number of spots suitable only for small cars. We will specialise the car class to encompass this possibility, and we will assume, perhaps optimistically, that small cars will fill the small car slots first.
- Each simulation period takes place over an 18 hour period from 6AM through to 12 midnight. Updates to the system state take place at one minute intervals, and we will track the time since opening in minutes – you may use an int rather than a date type.
- At the start of the simulation, we assume that the car park is completely empty, *and no vehicles of any type are waiting in the queue.*
- At each update point, **from the first minute onwards**, at most *one* of *each* of the two vehicle types may arrive at the car park. We will make these decisions randomly, based on probabilities supplied by the user (the `Simulator` class will supply the methods). So the absolute maximum number of vehicles that can appear in an hour is 120. For any sensible parameters, this is wildly improbable.
- Once a *car* 'arrives' we perform another random test to see if this is a small car – see the javadoc for car
- On 'arrival', we park the vehicle in the carpark if we are able to do so. *If not, we will place it in a queue if there is space available. If the queue is full, or no queue exists, then a vehicle that cannot park straight away must leave (See dissatisfied customers below).*
- We assume that no vehicle enters the car park in the final hour, and that all remaining vehicles leave the car park at 12 midnight.

**Essential Parameters:**
In the release version, the essential simulation parameters are *all* set up for you in the class Simulators.Constants. It is your job to modify our main program later to accept

arguments. Some arguments should be regarded as fixed, some not. Have a look at the parameters to `CarPark` and to `Simulator` for an idea of how these work.

*Overall Simulation Parameters:*

- The maximum number of car spaces in the car park [User]
- The maximum number of small car spaces in the car park [User]
- The maximum number of motorcycle spaces in the car park [User]
- *The number of vehicles permitted in the queue [User]*
- The RNG Seed [User]
- The average length of stay of cars and motorcycles – we will assume the same for each, and provide a Gaussian RNG to make this work. [User]
- The standard deviation of this stay [User]
- The time people are prepared to wait in a queue before giving up [Fixed]
- The minimum length of stay for cars and motorcycles [Fixed]
- Last entry time [Fixed]
- Closing time [Fixed]

*The cars:*

- The probability that a single car has arrived in the current minute [User]. Note that if this probability is say $\alpha_{car}$, then we are expecting $60\alpha_{car}$ cars to arrive in the next hour. So think about how many cars you might expect.
- The probability that a car arriving is a small car. [User] This is again an issue of random selection. This probability is the expected fraction of small cars.

*The motorcycles:*

- The probability that a single motorcycle has arrived in the current minute [User]. If this probability is say $\alpha_{motorcycle}$, then we would expect $\alpha_{motorcycle} \ll \alpha_{car}$, i.e. a significantly lower probability for motorcycles, with a correspondingly smaller capacity.

*The Simulation* *`(see SimulationRunner.run())`*

- As each vehicle arrives, if there is capacity in the car park, the vehicle gains entry and is parked.
- If the car park is full, the vehicle is placed in the queue. If the queue is full, then the vehicle is turned away and we have a ***dissatisfied customer***.
- Each queued vehicle waits until there is a vacant parking space – at which point it enters the car park – or until the maximum tolerable waiting time is reached.
- Vehicles already in the queue have priority over recent arrivals, which are necessarily placed at the end of the queue. Drivers or riders in the queue have exactly the same tolerance level or patience, and so all of them choose to leave if they wait longer than this maximum waiting time.
- Drivers or riders who leave after waiting too long are also seen as ***dissatisfied customers.***
- Once parked, the car remains in the car park for some random time according to a Gaussian random number generator with specified mean and variance. There is no limit on the number of cars which may leave during a particular update period.
- We assume a minimum stay of 20 minutes for each car that gets to park, a limit that is hard-wired for all simulations.
- No car will arrive after 11PM, and all cars remaining at 12 Midnight will leave without delay at that time, emptying the car park in readiness for the next morning.

Our task here is not to manage the operation, but to see how well the design works. The main development tasks are considered in turn.

## *Specific Tasks for Completing Part 1 of the Assignment*

You are supplied in this assignment with extensive support through the packages:

- `asgn2Exceptions`
- `asgn2Simulators`

These basically allow you to run the assignment and to understand the simulation. You will also be given some assistance with some logging code for the `CarPark` class (see later file). Your tasks in assignment 1 are then to implement the following classes as specified in the javadoc supplied. Vehicle is an abstract class, so you may rely on `CarTests` and `MotorCycleTests` to cover the bases.

The recommended order for completing the tasks is:

- `asgn2Tests`
- `asgn2Vehicles`
- `asgn2CarParks`

More specifically, this entails:

- `asgn2Tests.CarTests`
- `asgn2Tests.MotorCycleTests`
- `asgn2Tests.CarParkTests`
- `asgn2Vehicles.Vehicle`
- `asgn2Vehicles.Car`
- `asgn2Vehicles.MotorCycle`
- `asgn2CarParks.CarPark`

***Some guidance:***

Implementing the vehicle hierarchy is relatively straightforward, and `MotorCycle` really is as trivial as it seems. The `CarPark` class, however, is not easy. You will need to understand the iteration properly if you are to implement the more complex methods. Remember that these get used in the `run()` method and so you should have a good idea of what is going on.

Remember that a vehicle is either New (N), Queued (Q), Parked (P) or Archived (A) or in transition between these states. The Vehicle object keeps track of the state that it is in. The CarPark object keeps track of where it is stored, and so these must match up. The CarPark object must therefore maintain stores of some nature for:

- The current carpark spaces
- The queue
- The archive of vehicles that have been processed – this includes those that have been parked successfully, those that have left the queue in frustration, and those that arrived when the queue was full and not shifting (the ones turned away).

When we have a call to `Carpark.enterQueue(v),` we take the vehicle and physically add it to the queue store, but we also use the method `Vehicle.enterQueuedState();` to change its internal state. In this way, we have a record for when the vehicle is archived.

If you understand the principle of the transitions, you will get the assignment. See these methods and the transitions that they cover:

**`CarPark.tryProcessNewVehicle()`**
N > P : the ideal case – we arrive and park
N > Q: car park full for this category, so we queue
N > A: car park full, but so is the queue so we are turned away

**`CarPark.processQueue()`**
Q > P: good – we are clearing the queue and parking
Q > A: bad – people are queuing too long and leaving (archive)

**`CarPark.archiveDepartingVehicles()`**
P > A: Success – we have parked and are leaving at the end of our stay.

These methods are supported by a range of public and private helpers. But work these out and you will be well on the way to completing the CarPark class, and then all is well.

***Some Details of Parking:***

1. Parking requires that you generate an intended duration for this vehicle, which, given the arrival time enables you to create and store a departure time. This is provided by the `Simulator.setDuration()` method using the constants set.
2. You then need to change the state of the vehicle using one of the vehicle methods and then use the `CarPark parkVehicle` to add to the carPark. [There are similar processes for removing the vehicle]

***Some Notes on MotorCycles, Small Cars, Spaces and Blocking:***
The javadoc notes some possibilities in the parking of small cars and motorcycles which were not properly explained in the first version of this doc. The `@return` comments for the methods `getNumMotorCycles()` and `getNumSmallCars()` from CarPark.java are as follows:

```
@return number of MotorCycles in car park, including those occupying a small car space
@return number of small cars in car park, including those not occupying a small car space.
```

This is sort of obvious, but not, and was missed in the earlier version. Basically, we allow some overflow. This is not normally a problem with the standard settings, but it can be nasty if we are not careful. The trick is to keep track of the spaces filled.

The rules are as follows:

- If we fill *all* of the motorcycle spaces, and there are small car spaces available, then a motorcycle can fill one of these, but *NOT* a general car park space. Needless to say, this overflow – if it happens - reduces the number of small car spaces available for small cars…
- Similarly, if we fill the *all* of the small car spaces, and there are general car spaces available, then the small car can take up one of these. Once again, this will limit the availability of the general car park space for other cars.

For many settings – depending of course on the probability that the car is a small car - the problem will be that the general spaces will be filled and that only small cars will be permitted in the car park. In the example log entry from the javadoc, we have:

262::276::P:91::C:84::S:14::M:7::D:48::A:176::Q:9CCCCCCCCC|C:P>A||C:Q>P||S:N>P|

Here the maximum car park spaces is 100, but we have reserved 30 of them for small cars (probability that a car is small is here 0.2). So there are 84 cars = 70 (max general cars ☹) + 14 small cars in the car park. There are 16 small car spaces available still, and these can be taken up by small cars or, if the number goes above 10, motorcycles. But we are blocked on ordinary, general vehicles, which is why the queue looks like: Q:9CCCCCCCCC. Note that there is some movement though – the C leaves the car park and goes to the archive, meaning that the queue goes down in size from 10 (full on the previous iteration) to 9, as we can park a general car. The newly arrived small car goes straight into the car park.

***Vehicle creation:***

1. This takes place through calls to the `motorCycleTrial()` and `newCarTrial()` methods of the `Simulator` class using the specified probabilities.
2. If the car is created, you must also determine whether it is a small car or not.

**Adding arguments:**

Finally, you are required to implement command line argument processing in the simulator main supplied so that you can use lots of different parameters. See the constructors for details. More details later on values to try.

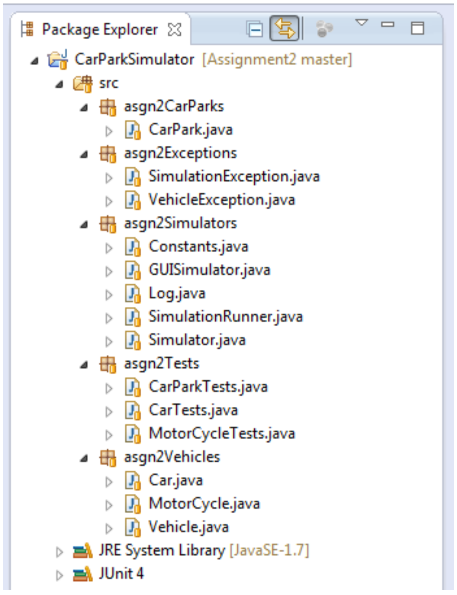
**Coding and Testing:**

The basic coding tasks are laid out above. Unit testing is now something we expect you to do well, and a tool to be used as part of the development process. For assignment 1, it was something new and we cut you some slack. Now it is something that we regard as a given and you should be able simply to deliver. As previously advised, we shall also be far sharper in our marking of code quality defects.

The overall Java project comprises the packages as shown in the hierarchy below. Ensure that you follow these package and class names precisely (and do not introduce any classes, public methods or public fields other than those specified). All packages are to be developed in Part 1 of the assignment, and the principal task for part 2 of the assignment will be to complete the GUI version of the simulator in the package shown.

**The Structure:**

The project is formally called CarParkSimulator. You will be required to implement a series of packages and classes, shown below. (The decorations are provided by egit, and the bottom of the picture has been clipped).



```
Package Explorer
CarParkSimulator [Assignment2 master]
  src
    asgn2CarParks
      CarPark.java
    asgn2Exceptions
      SimulationException.java
      VehicleException.java
    asgn2Simulators
      Constants.java
      GUISimulator.java
      Log.java
      SimulationRunner.java
      Simulator.java
    asgn2Tests
      CarParkTests.java
      CarTests.java
      MotorCycleTests.java
    asgn2Vehicles
      Car.java
      MotorCycle.java
      Vehicle.java
  JRE System Library [JavaSE-1.7]
  JUnit 4
```

Details of all of the required operations can be found in the API specification in the code drop.

***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 that document.

***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.