INB370 / INN370 Software Development

Assignment 2: Test-Driven Development and Graphical User Interface Programming

Part 1: Test-Driven Development

Semester 1, 2013

Due date: Tuesday, 28th of May, 2013 (Week 13)
Weighting: 30% in total (15% for Part 1, 15% for Part 2)
Assessment type: Group assignment, working in pairs – generally INB with INB and INN with INN, but this is not mandatory (see below).

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 and some integrated build related work (Part 2 of the assignment).

Some Ground Rules:
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 had a number of students who work full time contact me to express their concerns about working with other people, and some are 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 been encouraging people to form teams for some weeks now. Please do this now. Here are some guidelines on pair formation.

Forming Pairs for INB370 and INN370:
Generally speaking I would prefer, for administrative reasons, that students from INB370 pair with those from INB370, and those from INN370 pair with those from INN370. I will consider requests to cross these boundaries in particular cases. Those who are able to form pairs without assistance, please do so. If you have trouble finding a partner, or aren't sure where you might find one, please use the forum established on BB (by May 3) at Assessment>Assignment Two>Assignment 2 Dating Agency.]

I require that you email me by Friday May 10 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 one of: [INB370 Confirmed Pair] or [INN370 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 one of: [INB370 Need Partner] or [INN370 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 MONDAY MAY 6. I will reply quickly to those mails early next week.

I will be pretty ruthless in assigning people to pairs early in the week of May 13, so please try to sort this out as soon as possible.

The Scenario — Preparing a Long-Distance Train for Departure

Before it begins its journey, a long-distance (e.g., intercity or interstate) railway train is assembled from rolling stock, including a locomotive, passenger cars and freight cars, after which the passengers may board. There are a number of constraints inherent in this process. For instance, the train driver is responsible for ensuring that the train has been configured correctly, so that it can operate effectively and efficiently, and the conductor is responsible for ensuring the comfort and safety of the passengers. In Part 1 of this assignment you will play the role of a small programming team tasked with the job of developing a set of Java classes which will act as part of an automated system designed to track the preparation of a long-distance train for departure. In particular, the system will help the train driver and conductor monitor the train's configuration and boarding of passengers. In Part 2 of the assignment you will demonstrate your classes' capabilities via a Graphical User Interface for interacting with them.

An outline of the specific tasks required to complete Part 1 of the assignment 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.

Note that the assignment will be released in two stages exactly one week apart. This is deliberate, so as to promote a focus on the first part of the code construction without consideration of the GUI.

Working as 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 for rolling stock, one team member must take the role of the *tester*, and the other must take the role of the *coder*. The tester will develop new unit test cases, 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 (see next week's lecture) should show that the tests are committed prior to the code…

3. Ideally, the roles should be reversed for developing the class for the departing train, so that both team members get to play both roles.

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.

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*. (In computer programming '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. 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.

Specific Tasks for Completing Part 1 of the Assignment

To complete this part of the assignment your team must finish the following programming tasks. Full technical specifications for all the program code that must be developed are given in the Javadoc API description accompanying these instructions. Details of the process to be followed and the unit tests to be produced appear in the sections below.

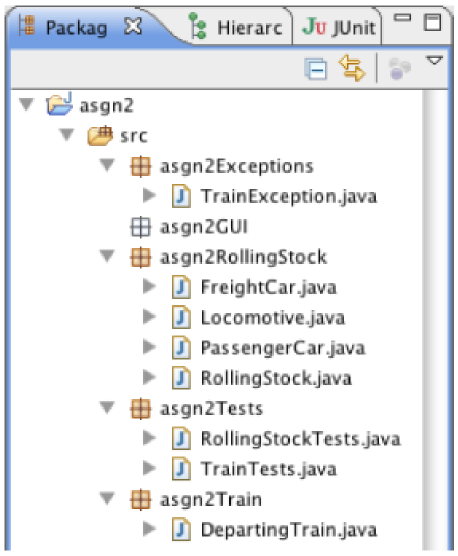
Please note that unit testing is now something we expect you to do well. For assignment 1, it was something new and we cut you some slack. Now it is something that we regard as a given and you should be able simply to deliver.

The overall Java project comprises several packages as shown in the hierarchy below. Ensure that you follow these package and class names precisely (and do not introduce any classes other than those shown here). All packages are to be developed in Part 1 of the assignment, except for the user interface package asgn2GUI which will be developed in Part 2.

Task 1: Implementing a rolling stock hierarchy

In this task you will use Test-Driven Development to develop a class hierarchy and corresponding unit tests for the rolling stock from which a train is constructed. With respect to the class hierarchy above, this task will produce packages asgn2Exceptions and asgn2RollingStock, and unit test class RollingStockTests. API specifications can be found in the accompanying Javadoc description.

The first step is to introduce the trivial TrainException class. This merely needs to provide a single, simple constructor and has exactly the same structure as other such classes in various INB370 pracs and demos. Either team member can write this and it doesn't follow TDD as understood above.



```
asgn2
  src
    asgn2Exceptions
      TrainException.java
    asgn2GUI
    asgn2RollingStock
      FreightCar.java
      Locomotive.java
      PassengerCar.java
      RollingStock.java
    asgn2Tests
      RollingStockTests.java
      TrainTests.java
    asgn2Train
      DepartingTrain.java
```

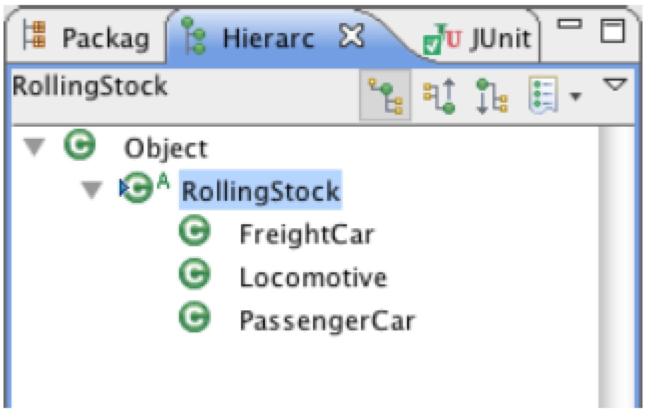
The main step is developing the class hierarchy for rolling stock in the asgn2RollingStock package and the corresponding RollingStockTests class. Begin by choosing who will play the role of 'principal tester' and who will play the role of 'principal coder'. Then:

1. The *tester* must add new unit tests to class RollingStockTests in package asgn2Tests. (look at the API spec for inspiration). Although test-driven development is usually depicted as adding tests one at a time, it may be more practical to add small groups of related tests (especially when trying to cover boundary cases). However, avoid the temptation to add several *unrelated* tests in one step, as this makes the job very difficult for the coder, and violates the TDD philosophy.

2. The *coder* must add code to package asgn2RollingStock to pass the tests, and must refactor the code appropriately to generalise the solution.

Ensure that all classes contain '@author' comments to identify who played which role. Include both your name and student number. As noted, please use these conventions for the source control logs as well.

For the purposes of this assignment we assume that there are three kinds of railway carriages of interest, locomotives, passenger cars and freight cars, each with different characteristics. In particular, all rolling stock has a fixed gross weight, whereas locomotives also have a certain 'pulling power', passenger cars have a fixed seating capacity, and freight cars are designed for different kinds of goods. Therefore, your rolling stock classes should form a class hierarchy as shown below. See the Javadoc API specification for details. Note that `RollingStock` is an abstract class, and that the others inherit from it, specialising as needed to the particular types of carriage.



Task 2: Implementing a class for departing trains

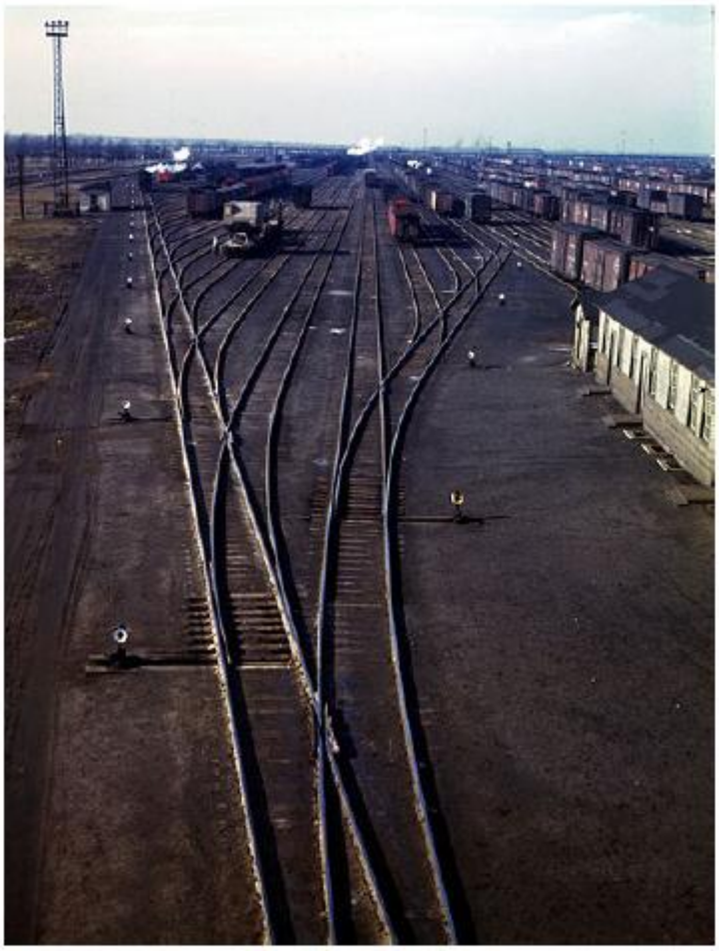
In this task you will implement a class, DepartingTrain, and its unit test class, TrainTests, which allows us to track the configuration and passenger boarding status of a whole train. We assume that trains are assembled by adding carriages to the rear, starting with a locomotive. This process occurs in a marshalling yard, such as that shown overleaf, so that carriages can be added to the middle of the train, if necessary, by removing carriages from the rear and temporarily shunting them to a spur line while the new carriage is being added. Thus the essential operations for assembling a train are the ability to add new carriages to the rear and remove existing carriages from the rear. We also require that a train must be configured so that it begins with a locomotive, followed by zero or more passenger cars, followed by zero or more freight cars. Thus a train in which a passenger car appears after a freight car is considered to be invalidly configured. Similarly, there can only be one locomotive and it must be the first carriage.

Once the train has been assembled passengers can begin to board. Essential capabilities at this stage are the ability to monitor how many passengers are on board and how many seats are available on the whole train. Also, for safety reasons, no further carriage shunting operations are allowed when any passengers are on the train. It is not necessary to include specifications for derailments or for criminal activities nearby.

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

The DepartingTrain class is the most complex one in this part of the assignment. To complete it you should swap the *tester* and *coder* roles played so far.

The *tester* must add new unit tests to class TrainTests in package `asgn2Tests`.
The *coder* must add code to class `DepartingTrain` in package `asgn2Train` to pass the tests, and must refactor the code appropriately to generalise the solution. Again, please ensure that both classes contain '*@author*' comments to identify who played which role, and that these conventions are also required in the source control logs.



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. Marks will be awarded proportionately to the percentage of unit tests passed and defective programs detected.

The precise assessment criteria for the assignment are supplied with this release, but 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 next week, and there will be fewer pictures of trains.

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 avoid incurring a late penalty. 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.** To be sure of avoiding a late penalty, submit your solution well before the deadline.

Details on the use of source control will be provided at the week 10 lecture, and you should use the coming days to make a start. It is expected that your source logs will make some sense from next week, but we will forgive you if they appear odd for the first week or so…