CSSE2002 / CSSE7023 **Semester 1, 2014**

Assignment 2

Goal: The goal of this assignment is to gain practical experience with procedural abstraction – how complex functionality can be broken up between different methods and different classes.

Due date: The assignment is due at **9:00am on Monday 12 May**. 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) to the lecturer (email is acceptable) or the ITEE Enquiries desk on Level 4 of GPSouth Building) prior to the assignment deadline.

Problem Overview: In this assignment, you will continue to implement a journey planner program for finding journeys in a public transport network.

Task Overview:

In brief, you will write a method for reading in a timetable – a description of routes and their services – from a file, and you will write a method for finding a journey from one station to another using an algorithm that is specified below in this handout. If you are a CSSE7023 student you will also be required to write a JUnit4 test suite for testing the journey-finder method.

More specifically, you must code method `read` from the class `TimetableReader` and method `findJourney` from the class `JourneyFinder` that are available in the zip file that accompanies this handout, according to their specifications in those files.

If you are a CSSE7023 student, you will also need to complete a systematic and understandable JUnit4 test suite for the `JourneyFinder.findJourney` method in the skeleton of the `JourneyFinderTest` class from the `planner.test` package. You may write your unit tests assuming that the classes that `JourneyFinder.findJourney` depends on (e.g. the `Route`, `Service`, `Leg` and `Journey` classes and any of the Java 7 SE API libraries) are implemented and functioning correctly. That is, you don't need to create test stubs for these classes. You may also assume the existence of a correctly implemented `TimetableReader` class – since that might make it easier to define timetables to test with.

As in Assignment 1, you must complete these methods and classes as if other programmers were, at the same time, implementing classes that use it. 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 7 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.
- Any new methods or fields that you add to `TimetableReader` or `JourneyFinder` must be private (i.e. don't change the spec of these classes.)
- Your source file should be written using ASCII characters only.

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.)
- Any additional methods that you write, and fields that you introduce should be private to hide implementation details and protect invariants.
- Private methods that you write must be commented using preconditions and postconditions (require and ensures clauses). Informal description is OK.
- 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.
- 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.

I recommend that you attempt to write loop invariants for all non-trivial while-loops in your code, but this is not compulsory.

The Zip file for the assignment also includes some other code that you will need to compile your classes as well as some junit4 test classes to help you get started with testing your code.

Do not modify any of the files in package `planner` other than `TimetableReader` and *`JourneyFinder`*, since we will test your code using our original versions of these other files. Do not add any new files that your code for these classes depends upon, since you won't submit them and we won't be testing your code using them.

The junit4 test classes as provided in the `package planner.test` are *not intended to be an exhaustive test for your code*. Part of your task will be to expand on these tests to ensure that your code behaves as required by the javadoc comments. (Only if you are a CSSE7023 student will you be required to submit your test file `JourneyFinderTest.java`.) We will test your code using our own extensive suite of junit test cases. (Once again, this is intended to mirror what happens in real life. You write your code according to the "spec", and test it, and then hand it over to other people … who test and / or use it in ways that you may not have thought of.)

If you think there are things that are unclear about the problem, ask on the newsgroup, ask a tutor, or email the course coordinator to clarify the requirements. Real software projects have requirements that aren't entirely clear, too!

If necessary, there may be some small changes to the files that are provided, up to 1 week before the deadline, in order to make the requirements clearer, or to tweak test cases. These updates will be clearly announced on the Announcements page of Blackboard, and during the lectures.

The algorithm for the `JourneyFinder.findJourney` method:

Given a timetable `timetable` satisfying the preconditions of the `findJourney` method as given in the `JourneyFinder` class, distinct start and end stations `startStation` and `endStation`, and a start time `time`, the following algorithm can be used to find a journey – if there is one – from the start to the end station that departs the start station no earlier than the start time and arrives at `endStation` no later than any other journey with those constraints.

In this algorithm we will classify stations to be either `finalised` or `unfinalised`. For each station we will also keep track of a `fastestKnownJourney` from `startStation`, departing `startStation` no earlier than `time`, to that station. The `fastestKnownJourney` to a station may be `defined` – if there is such a journey – or `undefined` if there is not.

For each station that is `finalised`, the `fastestKnownJourney` to that station is in fact a journey from `startStation`, departing `startStation` no sooner than `time`, to that station that arrives no later than any other such journey.

Initially each station is classified as being `unfinalised`, and the `fastestKnownJourney` to each station other than `startStation` is set to be `undefined`. The `fastestKnownJourney` to `startStation` is marked as being `defined` as the special empty journey that conceptually starts and ends at `time`.

while (there is at least one `unfinalised` station such that the `fastestKnownJourney` to that station is `defined`)

- let `current` be an `unfinalised` station that can be reached by its (`defined`) `fastestKnownJourney` sooner than any of the other `unfinalised` stations can be reached by theirs; then let `currentJourney` be the `fastestKnownJourney` to `current` and, `currentTime` be the time that journey ends at the `current` station
- mark `current` as being `finalised`
- **if** (`current` equals `endStation`) then **return** `currentJourney` – found it!
- **for each** `route` that stops at `current` such that `current` is not the last stop of the route, and there exists a service for `route` that will stop at `current` at a some time >= `currentTime`
    - let station `adjacent` be the next stop on that route after `current`, and let `service` be the earliest service for `route` that departs `current` no earlier than `currentTime`
    - (If `adjacent` has not been `finalised` and) if `currentJourney` extended by taking `service` to `adjacent` would arrive before the `fastestKnownJourney` to `adjacent`, or if `fastestKnownJourney` to `adjacent` is `undefined`, then update the `fastestKnownJourney` to `adjacent` to be `defined` as that one.

return `null` since we have found no journey from `startStation` to `endStation` that departs no sooner than `time`

Note: to make life a bit easier I have added a `getNextStop` method to the Route class, and a new constructor in the `Journey` class that creates a new `Journey` from an old one.

Hints:

1. It may be easier to implement the `TimetableReader.read` method first first since you can use it to read in timetables to test the `JourneyFinder.findJourney` method.
2. Read the specification comments carefully. They have details that affect how you need to implement and test your solution.

Submission: Submit your files **`TimetableReader.java`**, **`JourneyFinder.java`** (and **`JourneyFinderTest.java`** and any of your timetable files that are used for testing in `JourneyFinderTest.java` if you are a CSSE7023 student) 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: If you are a CSSE2002 student, your assignment will be given a mark out of 15, and if you are a CSSE7023 student, your assignment will be given a mark out of 17, according to the following marking criteria. (Overall the assignment is worth 15% for students from both courses.)

Testing TimetableReader.read and JourneyFinder.findJourney (8 marks)

• All of our tests pass	8 marks
• At least 85% of our tests pass	7 marks
• At least 75% of our tests pass	6 marks
• At least 65% of our tests pass	5 mark
• At least 50% of our tests pass	4 marks
• At least 35% of our tests pass	3 mark
• At least 25% of our tests pass	2 mark
• At least 15% of our tests pass	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 7 compiler will be used to test code. Each of your classes will be tested in isolation with our own valid implementations of the others.

Code quality (7 marks)

• Code that is clearly written and commented, and satisfies the specifications and requirements	7 marks
• Minor problems, e.g., lack of commenting or private methods	4-6 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);
- exposing implementation details by introducing methods or fields that are not private
- not commenting any private methods that you introduce using contracts (pre and postconditions specified using @require and @ensure clauses).
- not having appropriate comments for fields and local variables (except for-loop variables), or tricky sections of code;
- monolithic methods: if methods get long, you must find a way to break them into smaller, more understandable methods using procedural abstraction. (HINT: very important!!)
- incomplete, incorrect or overly complex code, or code that is hard to understand.

JUnit4 test – CSSE7023 ONLY (2 marks)

We will try to use your test suite `JourneyFinderTest` to test an implementation of `JourneyFinder.findJourney` that contains some errors in an environment in which the other classes `JourneyFinder.java` depends on exist and are correctly implemented.

Marks for the JUnit4 test suite in `JourneyFinderTest.java` will be allocated as follows:

- Clear and systematic tests that can easily be used to detect most of the (valid) errors in a sample implementation and does not erroneously find (invalid) errors in that implementation. 2 marks
- Some problems, e.g., Can only be used easily to detect some of the (valid) errors in a sample implementation, or falsely detects some (invalid) errors in that implementation, or is somewhat hard to read and understand. 1 marks
- Work with little or no academic merit or major problems, e.g., cannot be used easily to detect (valid) errors in a sample implementation, or falsely detects many (invalid) errors in that implementation, or is too difficult to read or understand. 0 marks

Note: code submitted with compilation errors will result in zero marks in this section. A Java 7 compiler will be used to test code.

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**.