Inf2C Software Engineering 2017-18

Coursework 3

Creating an abstract implementation of a tour guide app

1 Introduction

The aim of this coursework is to implement and test an abstract version of software for a tour guide app. This coursework builds on Coursework 1 on requirements capture and Coursework 2 on design. As needed, refer back to the Coursework 1 and Coursework 2 instructions.

2 Further design details

The auto-marking of your submitted code will assume you have precisely implemented further design details as described below.

2.1 Annotation class

For simplicity Annotation objects have a single field of type String. We can imagine refinements of the requested implementation defining an Annotation class that supports richer kinds of annotations.

The name Annotation.DEFAULT refers to a default annotation object that should used in cases when an explicit annotation is not desired.

2.2 Controller interface and class

Your controller implementation class ControllerImp should implement the interface specified in Java interface Controller shown in Fig 1. This interface defines the input messages your implementation must accept, specifying both arguments these messages carry and the types of any return values.

In addition ControllerImp should define a constructor

```
public ControllerImp(double waypointRadius, double waypointSeparation)
```

```
public interface Controller {

    /*
     * Create tour mode
     */
    Status startNewTour(String id, String title, Annotation annotation);

    Status addWaypoint(Annotation annotation);

    Status addLeg(Annotation annotation);

    Status endNewTour();

    /*
     * Browse tours mode
     */
    Status showTourDetails(String tourID);

    Status showToursOverview();

    /*
     * Follow tour mode
     */
    Status followTour(String id);

    Status endSelectedTour();

    /*
     *  All modes
     */
    void setLocation(double easting, double northing);

    List<Chunk> getOutput();
}
```

Figure 1: Controller Interface

Stage 0	Leg 0
Stage 1	Waypoint 0 Leg 1
Stage 2	Waypoint 1 Leg 2
⋮	⋮
Stage $n-1$	Waypoint $n-2$ Leg $n-1$
Stage n	Waypoint $n-1$

Figure 2: Tour stages

that initialises system constants for determining whether an app user is at a waypoint and setting a minimum distance between adjacent waypoints on a tour.

2.3 Error behaviour

Most input messages are valid if the app is in some states and invalid if it is in others. For example, createNewTour is only valid if the app is currently in the browse tour mode. All these messages return an object in the abstract class Status. This object is either from the subclass Status.OK or the subclass Status.Error. The subclasses have these 'dotted' names because they are defined as *static member* classes of the Status class. Static member classes are just like normal classes, except that their names are part of the namespace of the enclosing class rather than the enclosing package.

The Status.OK class defines no fields, so there is essentially only one distinct object of this class. For convenience, this object can be referred to with the Status.OK name, as the Status class defines a static field with this name that refers to a Status.OK object.

The Status.Error class defines a String field to hold an error message. Objects in this class can be created with invocations of form:

```
new Status.Error("message")
```

You should design your code so that input messages always return a Status.Error object if they are invalid. Sending in an input message should never result in your code throwing an exception. In addition, if an input message is invalid, the state of the app should always be unchanged. Your input message processing code should always check validity first before going on to code that updates the state. If an input message is valid, the Status.OK object should be returned. These requirements makes it straightforward for test code to check whether your implementation is correctly identifying invalid messages.

2.4 Follow tour mode

When a tourist is following a tour, the app should track the current *stage* of the tour. For a tour with n waypoints, we associate $n+1$ stages with waypoints and legs as shown in Fig 2.

The Coursework 1 instuctions did not mention a leg before the first waypoint, Leg 0 in the figure. This leg is included in order to hold an annotation describing how the tourist can reach the first waypoint.

At a minimum a tour is expected to have two stages: Stage 0 with Leg 0 and Stage 1 with Waypoint 0.

A tourist starts following a tour at Stage 0 and as the tour proceeds the stage increments. The stage can never decrement even if the tourist wanders back to previous waypoints.

Geometrically we imagine waypoints to be circles of a fixed radius. An app user is considered to be *at* a waypoint if their current location is on or within the waypoint's circle. At any stage except the last, there is a *next waypoint* in the following stage. The stage is advanced whenever the tourist reaches the next waypoint. If waypoints are too close together, there is the possibility that the stage could advance multiple steps all at once. To avoid this we require that adjacent waypoints on a tour are non-overlapping, that they are always separated by some minimum distance greater than double the waypoint radius.

Output information should always include a header which lists the tour title, the current stage and the number of waypoints.

Then, when a tourist is at any stage except the last, the app should output the leg annotation for the current stage and the bearing and distance information for reaching the next waypoint, in that order. Always output leg annotations, even if they are default.

Additionally, when a tourist is at a waypoint, the waypoint annotation should be displayed. As a consequence, if the tourist leaves the waypoint of the current stage, the waypoint's annotation goes away, but if they then immediately revisit that waypoint, the annotation should be displayed again. Waypoint annotation output should be displayed after the header and before the leg and direction output information.

A tourist can only start following a tour when the app is in the browse mode. A tour can be ended at any stage, there is no need for the tourist to reach the final stage. When a tour is ended, the app switches back to the browse mode and shows the overview of the available tours.

2.5 Browse Tours mode

The browse tours mode has two sub-modes, one in which an overview is presented of all tours, listed in order of their tour ids, one in which details of a particular tour are displayed. When the app is first started (a Controller object is created), the app should be in the overview sub-mode of the browse mode.

2.6 Create Tour mode

A tour is created by the tour author walking the route of a tour, alternately adding legs and waypoints. The app allows annotations on legs to be optional. The absence of an annotation on a leg should be realised by attaching a *default* annotation to the leg rather than say using a null reference. See Section 2.1.

The tour author indicates that a leg has no annotation by skipping sending in an addLeg message and just proceeding to sending in a addWaypoint message for the waypoint at the leg end.

Created tours must have at least one waypoint. It is invalid to finish creating a tour when the last message sent in is a addLeg message. This ensures that a tour always ends in a final waypoint rather than a leg that has no destination.

Tour creation should enforce separation between adjacent waypoints. Consider as invalid an addWaypoint message that is sent in when the current location is within the minimum separation distance of the last added waypoint.

Tour creation can only be started when the app is in browse mode. When a new tour is finished, the app should return to the overview sub-mode of the browse mode.

When in this mode, a chunk should be output giving the tour title, the number of legs added so far and the number of waypoints added so far.

2.7 Location messages

The sending in of location messages is valid at any time. A location is specified as a pair of Java double values, an *easting* and a *northing*, that indicate the location's position in metres east and north respectively of some reference position.

In operation it should be assumed that location messages are sent in by some location service at regular intervals, perhaps every few seconds. Both the creating and following tours activities take advantage of this location information. While following a tour, the sending in of a location message can trigger a state change, as described in Section 2.4.

2.8 Output

Output can be requested at any time by sending in a getOutput() message. Requesting output should never change the state of the app.

Output should be in the form of chunks as described in the Coursework 2 instructions. An abstract class Chunk is provided with subclasses for each of the kinds of chunks you need to generate. Constructors and methods for constructing chunks are shown in Fig 3. As with the Status class subclasses (see Section 2.3) these subclasses are all defined as static member classes of the Chunk class. Each subclass defines an equals() method that can be used in tests to compare the actual chunks output with the expected chunks.

See the Chunk class for further details.

3 Your tasks

There are several concurrent tasks you need to engage in. These are described in the following subsections.

3.1 Construct code

There is no requirement that you stick to the design you produced for Coursework 2.

Follow good coding practices as described in lecture. Consult the coding guidelines from Google at

```
https://google.github.io/styleguide/javaguide.html
```

```
// Create overview with no items
Chunk.BrowseOverview()

// Chunk.BrowseOverview method: add item to end of overview
void addIdAndTitle(String id, String title)

Chunk.BrowseDetails(String id, String title, Annotation details)

Chunk.FollowHeader(String title, int currentStage, int numberWaypoints)

Chunk.FollowWaypoint(Annotation annotation)

Chunk.FollowLeg(Annotation annotation)

Chunk.FollowBearing(double bearing, double distance)

Chunk.CreateHeader(String title, int numberLegs, int numberPositions)
```

Figure 3: Chunk construction

or Sun at

`http://www.oracle.com/technetwork/java/codeconventions-150003.pdf`.

A common recommendation is that you never use tab characters for indentation and you restrict line lengths to 80 or 100 characters. You are strongly recommended to adopt both these recommendations because it is good practice and, particularly, in order to ensure maximum legibility of your code to the markers. You should use an indent step of 4 spaces as in the Sun guidelines rather than the 2 in the Google guidelines, for consistency with the supplied code.

Unfortunately the default in Eclipse is to use tab characters for indents. See the Coursework 3 web page for advice on how to set Eclipse to use spaces instead and also how to get Eclipse to display a right margin indicator line.

To show you know the basics of Javadoc, add class-level, method-level and field-level Javadoc comments to the provided Displacement class.

Be sure you are familiar with common interfaces in the Java collections framework such as List, Set and Map and common implementations of these such as ArrayList, LinkedList, HashSet and TreeMap. Effective use of appropriate collection classes will help keep your implementation compact, straightforward and easy to maintain. If you start wanting to implement some sorting algorithm to keep tours in the tour library sorted, you have not been looking closely enough at this framework.

3.2 Create unit tests

The provided code includes a class Displacement for representing the position of one location relative to another. Add tests to the JUnit 4 test class DisplacementTest to check the correct functioning of the bearing() and distance() methods. Test for a variety of different bearings to ensure an angle θ in range $0 \leq \theta < 360$ is always returned.

3.3 Create system-level tests

Your are expected to use JUnit 4 to create system-level tests that run scenarios of each of the use cases you implement. The provided ControllerTest class tests some but not all the features described in Section 2. You are expected to add further tests to more completely test all the features.

Few tests are able to test single use cases by themselves. In nearly all cases, several use cases are needed to set up some state, followed by one or more to observe the state.

Do not run all your tests together in one large single test. Where possible, check distinct features in distinct tests. Also, when a test involves exercising some sequence of use cases and features, try to arrange that this test is some increment on some previous test. This way, if the test fails, but the previous test passes, you know immediately there is some problem with the increment.

You are encouraged to adopt a test first approach. Make use of the provided tests to help you develop your code, and when tackling some feature not already tested for, write a test that exercises it before writing the code itself.

Include all your additional tests as JUnit test methods in the ControllerTest class. Feel free to add extra auxiliary methods that handle repeatedly needed sequences of input and output events; avoid cut and pasting the same sequences multiple times.

Your ControllerTest.java file should serve as the primary documentation for your system-level tests. To this end, take care with how you structure your test methods and add appropriate comments, for example noting particular design feature being checked.

3.4 Add logging code

Add logging code to your code using logging support from the standard java.util.logging package. The provided code for the ControllerTest, ControllerImp and Displacement classes gives examples of the use of this package. Note that each class using logging includes a field

```
private static Logger logger = Logger.getLogger("tourguide");
```

that sets up a field logger that allows for easy reference to a Logger object. Logging messages are then created by statements like

```
logger.finer("Entering");
```

The logging messages automatically include the enclosing class and method names. It therefore can be sufficient for the logging text to be very brief, say just the word `"Entering"` as above, indicating the logging message is generated at the entry-point of a method. The message might also provide information about the object or arguments the method is called on or might generate a banner to improve log readability. See the provided code for examples of such logging messages.

Each log message is generated at a *logging level* with higher levels being for more important messages. Level names in increasing order of importance include *finest*, *finer*, *fine*, *info* and *warning*. Note how test-cases start with banners at the level *info*, but Displacement class methods are at the lower level *finer*. The static variable loggingLevel in the ControllerTest class controls the minimum level that at which messages are reported. Have a brief browse

of documentation on the web concerning this package to learn more about what is going on with logging.

Not every method you write needs logging. Your aim is to add enough logging that both you and the markers can follow the flow of control around your implementation objects and methods when each of your tests is run. Experiment with generating logging messages at different levels, so the volume of log messages generated can be controlled be varying the minimum logging level. See Section 4.2 for how to control this minimum logging level when running JUnit tests from a command line.

You are required to submit a file of the logging output from running your tests. This serves as documentation of the dynamic behaviour of your system. You are not otherwise required to submit any UML sequence diagrams or text descriptions of behaviour.

3.5 Keep a project log

Each member of a group is expected to keep and submit their own independent project log. This is the only task that should be done independently. All other tasks should be completed jointly.

This project log should contain an entry for each day you spend a significant time on the coursework. It should contain the following elements.

Plans What are your plans for completing the project? What activities are you going to do when? What times do you have available? As the project progresses, you can make plans for how you are going to complete the next individual tasks you have to work on.

Achievements Describe what you have achieved through each day you put in time on the coursework. Summarise outcomes of meetings with your partner. Note how much time you are putting in.

Reflection Has everything gone to plan or not? Were you optimistic or pessimistic? Did unforeseen issues come up? Are there ways in which you could improve your own pattern of work or your pattern of work with your project partner? How is your understanding evolving of the concepts involved in the coursework? Have you resolved issues that were puzzling you? Do you have new questions you need to find answers to?

Be sure to consider including remarks on working practices you experiment with. See Section 4.1.

Do not include details of design and implementation decisions in your log. Instead, refer to your report for this information. However, note that generally it is appropriate to include such information in project logs.

Your project log is expected to be no more than 2 or 3 pages. You can edit your project log in order to improve ease of reading and keep it to length.

Mention in your project log the overall time you have spent on this coursework and give some idea of how this broke down into different activities.

3.6 Write a report

This report is expected to be significantly shorter than that you produced for Coursework 2. A title page plus 2 or 3 further pages is fine. The report should have the following sections:

UML Class diagram Include the Controller class, the Displacement class, and any other classes you create. Assume the reader already has been given information about the provided classes such as Annotation, Chunk and its subclasses and Status.

Do not try to exhaustively list all the methods and fields of the classes. Instead, only list a few of the most important ones. As is commonly the convention, do not show fields that implement associations drawn in the diagram. Do show the navigability of associations and do include multiplicities on their ends.

High-level design description This could include some of what you produced for Coursework 2. However, only cover the parts you have actually implemented. Be sure to include discussion of your design decisions. Include remarks about how your design might have changed since Coursework 2.

Implementation decisions Discuss the main implementation decisions you have made. For example, state the kind of Java collection class you have used to implement each of the one-to-many associations in your design, and explain why.

4 Practical details

4.1 Working practices

This coursework is a small-scale opportunity to try out ideas that have been mentioned in lectures. For example, you could try writing tests before code and using the tests to drive your design work. You could also try pair programming.

You are strongly encouraged to take an incremental approach, adding and testing features one by one as much as possible, always maintaining a system that passes all current tests.

Sometimes it is seen as important to have development teams and testing teams distinct. This way there are two independent sets of eyes interpreting the requirements, and problems found during testing can highlight ambiguities in the requirements that need to be resolved. As you work through adding features to your design, you could alternate who writes the tests and who does the coding.

4.2 Getting started

Download the skeleton code zip file tourguide-skeleton.zip from the Coursework 3 web page:

`http://www.inf.ed.ac.uk/teaching/courses/inf2c-se/Coursework/2017/cw3.html`

This can be placed anywhere, a temporary directory for example.

Unzip the file. This creates a top-level directory `TourGuideSkeleton` containing subdirectories `src`, `bin` and `lib`.

The code can be compiled and run from the command line. First `cd` to the `TourGuideSkeleton` directory. Then run:

```
export CLASSPATH=bin:lib/junit-4.12.jar:lib/hamcrest-core-1.3.jar
javac -sourcepath src -d bin src/tourguide/*.java
java tourguide.AllTests
```

If all goes well you will see a bunch of logging messages followed by the text

```
TEST RESULTS
Number of tests run: 7
SOME TESTS FAILED
Number of failed tests: 6
```

and some details about each of the failed tests. To run with reduced or no logging messages, try

```
java tourguide.AllTests info
```

or

```
java tourguide.AllTests off
```

Other options for the logging level argument include `fine` and `finer`.

Instructions on how to import this code into Eclipse are available from the Coursework 3 web page.

4.3 Provided code

The provided code includes the following.

- The Controller interface and associated classes such Annotation, Chunk and Status that are used in the interface methods.
- A ControllerImp class providing a skeleton implementation of the controller.
- A partially complete JUnit test class ControllerTest for your whole system.
- A class Displacement for representing the position of one location relative to another.
- A skeleton JUnit test class DisplacementTest for the Displacement class.
- A class AllTests with a main method which enables all JUnit tests in the ControllerTest and DisplacementTest classes to be run from the command line.

The provided code should compile fine. However, all the provided tests except an example test in DisplacementTest will fail.

4.4 Modes of testing

In Eclipse you can run individual JUnit tests in either the ControllerTest class or DisplacementTest class, all tests in one of these classes, or all tests in both classes using the JUnit test suite defined in the AllTests class.

It is also possible to run the AllTests class as a Java application, as it defines a static main method. See Section 4.2.

4.5 Singleton classes

Your ControllerImp class and perhaps others too will be singleton classes; you will only intend ever to create one instance of each such class. Do *not* do anything special to realise the Singleton design pattern. For example, do *not* use static fields to hold a singleton's data, a static field to refer to the single instance of a singleton class or a static method to retrieve the single instance. Just create one instance of the class when setting up your system.

This ensures that each of your tests starts with a completely fresh state and you do not encounter subtle testing bugs when state from one test persists into the next.

4.6 What can and cannot be altered

- Do **not** alter the provided Controller interface, the Chunk and Status classes and their subclasses, and the Annotation class. Auto-marking compilation and test will rely on these being as supplied.
- In order to complete the coursework you will need to modify the code in the ControllerImp class, ensuring the class still implements the Controller interface. However, do not change the types and number of the provided constructor's arguments; testing assumes that the ControllerImp class defines a constructor of form:

```
public ControllerImp(double waypointRadius, double waypointSeparation) {
    ...
}
```

- The coursework requires you to add Javadoc comments to the Displacement class. However, do not change the fields and methods of this class.
- You are free to modify the provided ControllerTest, DisplacementTest and AllTests test classes.

It is expected you will introduce several new classes in addition to the provided classes. While it is possible to produce a functioning implementation by adding no further classes, such an implementation would have very poor object-oriented design.

5 Good Scholarly Practice

Please remember the University requirement as regards all assessed work for credit. Details and advice about this can be found at:

`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`

and links from there. Note that, in particular, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work in some web repository, then you must ensure that access is restricted to only members of your coursework group. Be aware that not all free repository hosting services support private repositories, repositories with access restrictions. There are some more details about this on the Coursework web page.

6 What to submit

6.1 Working code, tests and an output log

Your code **must** compile and run from a DICE command line, **exactly** as described above in Section 4.2. This is essential for the auto-marker. Please be aware that the default ECJ compiler used in Eclipse is a *different* compiler from the command line `javac` compiler. ECJ will compile and run code that `javac` will not compile. If you develop your code in Eclipse or some other IDE, or if you use some other non-DICE platform, you must double check it runs OK under DICE from a DICE command line before submitting.

Auto-marking will use Java 1.8.

Do not use any libraries outside of the standard Java 1.8 distribution other than the JUnit libraries provided in the lib subdirectory.

Create a zip file src.zip of your code and tests by setting your current directory to the main project directory containing the src, bin and lib directories, and running the command

```
zip -r src.zip src
```

Again, it is important you follow these instructions **exactly** so that the auto-marking scripts can run without problems.

Capture the output of running your JUnit tests using the command

```
$ java tourguide.AllTests &> output.log
```

This output.log file is a required part of your submission.

6.2 Report

This should be a PDF file named report.pdf. Please do not submit a Word or Open Office document.

If you are using draw.io to draw diagrams, export your diagrams as PDFs, not as bit-maps (e.g. .png files). Make sure your PDF is viewable using the evince application on a DICE machine. The report should include **a title page with names and UUNs of the team members**.

6.3 Project log

Keep this as say a Word or Open Office document, but convert it to PDF for submission. Group members should submit their project logs separately and name their project log files project-log.pdf. See the submission instructions below.

6.4 A team members file

Create a text file named team.txt with only the UUNs of the team members (one UUN on each line) as shown,

```
s1234567
s7891234
```

Do **not** include any other information in this file, such as the names of the team members. At the start of marking a script has to be able to process these files.

7 Marking Scheme

1. **Code quality**: Use of coding standards, ease with which code can be read and understood. Use of Javadoc for Displacement class. (5%)

2. **Report**: This mark is based on the quality of the design and implementation information. (35%)

3. **Code output log**: Is it easy to follow the execution of your use cases by reading the log along with your class diagram? (5%)

4. **Code correctness and completeness**: This will be assessed by auto-marking, followed by a review of the auto-marking results. Auto-marking will be more thorough than the provided tests. of the phases. It will only test use-case features described in this or previous Coursework handouts. (30%)

5. **Project log**: (20%)

6. **Quality of unit tests**. Have you written unit tests in the javaDisplacementTest class to adequately test the methods of the provided Displacement class? (5%)

8 How to submit

Each member of a group has to submit some files.

Ensure you are logged onto a DICE computer and are at a shell prompt in a terminal window. Place all the requested files in the same directory and cd to this directory.

Only one group member should submit the code, output log and report. That group member should run the command

```
submit inf2c-se cw3 src.zip output.log report.pdf project-log.pdf team.txt
```

If the group has two members, the second group member should run the command

```
submit inf2c-se cw3 project-log.pdf
```

This coursework is due

16:00 on Tuesday 28th November

The coursework is worth 50% of the total coursework mark and 20% of the overall course mark.

9 Document history

v1. (8 Nov) Initial release.

v2. (10 Nov) Fixed class named in Section 7 that needs Javadoc comments; the class that Javadoc comments should be added to is the Displacement class as noted in Section 3.1. Adjusted slightly the description of singleton classes in Section 4.5. Fixed a few minor grammatical errors.

v3. (10 Nov) Added a new section, Section 4.6, clarifying what provided code can and cannot be altered. This replaces and expands on text that used to be at the end of Section 4.3.

v4. (12 Nov) The provided test code required a specific header chunk to be output when in create tour mode, but this was not described in Section 2.6. Updated Section 2.6 to also include this information.

Paul Jackson. v4. 12th November 2017.