Curtin University — Discipline of Computing

Object Oriented Software Engineering (COMP2003)

Semester 1, 2018

Assignment

Due: Thursday 17 May, 23:59.

Weight: 25% of the unit mark.

This assignment assesses your ability to take complex requirements and come up with a *good* design.

1 Problem Overview

Design and implement an app for use by trekkers and climbers to view possible routes, and track their progress through them.

For this assignment, this can just be either a console application (or a GUI app if you wish), ready to be converted into a mobile app by someone else once you're finished.

2 Routes

When out in the wilderness (or, in fact, anywhere), trekkers and climbers follow predefined routes. These have a name, a description, a starting location, a finish location (which may or may not be the same as the start), and a series of zero or more waypoints in between that define the route.

These points are *three-dimensional*, being made up of latitude and longitude (measured in degrees) and altitude (measured in metres above sealevel). Each of these is a real number. Latitude values must fall in the range $-90°$ (the south pole) to $90°$ (the north pole). Longitude values must fall in the range $-180.0°$ to $180.0°$. For our purposes, there are no hard constraints on altitude, which can be positive or negative[1].

For simplicity in the following discussion, we will consider the start and end locations to *be* waypoints themselves. So, if the route contains N waypoints, W_1 is the start, W_2 is the second waypoint after the start, etc., and then W_{N-1} is the second last waypoint, and W_N is the end.

[1] There are locations on land below sea level. While there are still lots of fun ways we could be extremely pedantic about the theoretical constraints on altitude, the application should not impose any such constraints.

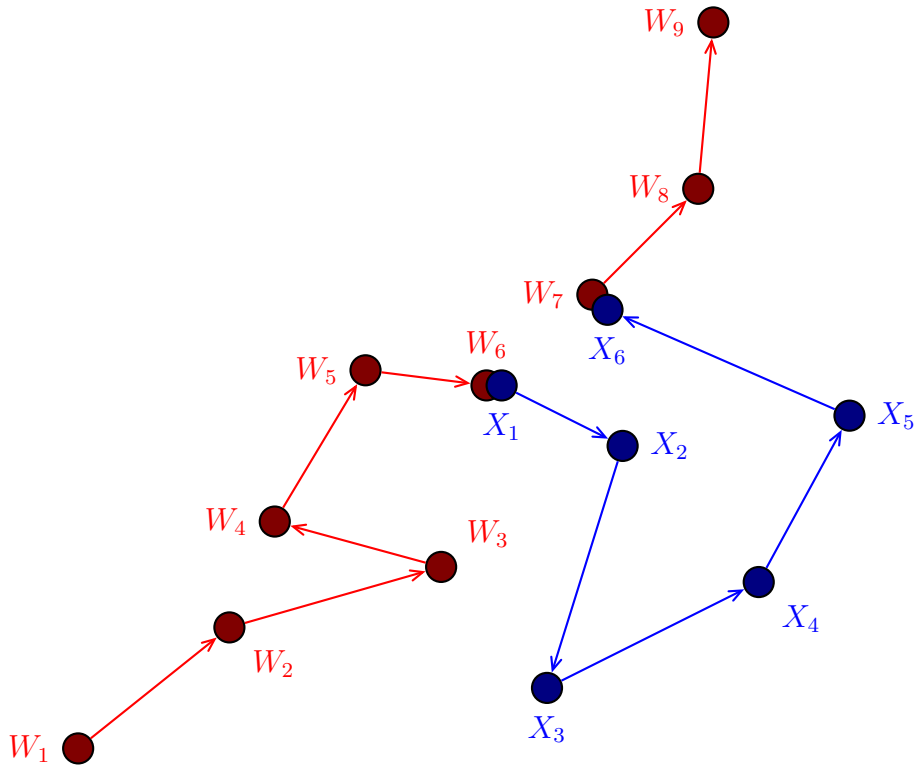

Figure 1. Route example. Here there are two routes W and X, where X is used as a subroute within W. (This only shows a top-down view, but remember that each point also has an altitude.)

In the basic case, the path between any two waypoints is assumed to be a straight line. This is called a "segment", and also comes with a textual description (e.g. stating what the user will encounter, what equipment they'll need, any risks, etc.).

However, some routes can also *contain* other routes (*sub-routes*), where two adjacent waypoints in one "big" route (say W_i and W_{i+1}) are the same as (or very close to – see below) the start and end points for another "small" route. In this case, there is no segment directly connecting W_i and W_{i+1}, but rather a whole sequence of segments defined by the small route. A route could contain several sub-routes, at various points, and the sub-routes themselves can also contain their own sub-routes, etc.

What defines "very close"? Waypoint W_i in the "big" route doesn't need to be *exactly* the same as the start of the small route, but it does have to be within 10 metres horizontally, and 2 metres vertically. The same goes for waypoint W_{i+1} in the big route, and the final waypoint of the small route. See below for details on calculating distance.

There are no rules either against a route retracing its own path. The same set of coordinates can appear more than once in different parts of the sequence of waypoints. Entire sub-routes can be repeated as well. A given route could contain a single sub-route at, say, three distinct points in the sequence of waypoints.

3 Measuring Distance

There are several cases where you'll need to measure the horizontal distance (i.e. latitude and longitude only) between two sets of coordinates. The mathematics of this are beyond the scope of the assignment, but you will have access to the following notionally-pre-existing method:

```
public class GeoUtils
{
    /**
     * Returns the horizontal distance (across the Earth's surface) in
     * metres between two points expressed in degrees of latitude and
     * longitude.
     *
     * (If any arguments are out of range, this submodule will erase your
     * hard drive.)
     */
    public double calcMetresDistance(double lat1, double long1,
                                     double lat2, double long2) {...}

    ...
}
```

If you'd like to consider a possible implementation of this method (apart from hard drive erasure), the following formula[a], is about the simplest reasonble way to do it, but note that it is actually only an approximation[b]:

$$d = 6371000 \cdot \cos^{-1}\left[\sin\left(\frac{\pi \cdot \texttt{lat1}}{180}\right) \cdot \sin\left(\frac{\pi \cdot \texttt{lat2}}{180}\right) + \cos\left(\frac{\pi \cdot \texttt{lat1}}{180}\right) \cdot \cos\left(\frac{\pi \cdot \texttt{lat2}}{180}\right) \cdot \cos\left(\frac{\pi \cdot |\texttt{long1} - \texttt{long2}|}{180}\right)\right]$$

Also note that this is only for testing purposes. You may assume that it's someone else's responsibility to devise the real implementation.

[a] Weisstein, Eric W. "Great Circle." From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/GreatCircle.html

[b] The formula here assumes the Earth is a sphere exactly 6371km in radius. It's actually closer to an "oblate spheroid", but it's also a bit bumpy and uneven.

4 Before Setting Out

When it starts up, the app will display a table of routes, giving their name, start/end coordinates, and overall distance. The distance in particular will actually be broken down into three values, all expressed in metres:

- Horizontal distance,
- Vertical climbing (distance up), and
- Vertical descent (distance down).

These are calculated by adding up the relevant values for each segment of the route (including sub-routes). If a segment ends higher up (in altitude) than it begins, the difference contributes to "vertical climbing", and otherwise to "vertical descent".

For any given route, the user can choose to see the details. This includes the route description, along with a list of all waypoints and segments, and their coordinates and descriptions, including those from sub-routes.

You do not need to provide a UI feature to create routes, as this will be done using an online system (to be developed by someone else).

Rather, in your app, there should be a user-selectable option that will cause the app to download an updated set of routes from the central server. To do this, you'll need to invoke the following pre-existing method:

```
public class GeoUtils // Note: same class as above
{
    /**
     * Attempts to contact the central server to retrieve the latest version
     * of the route data.
     *
     * @return A string containing formatted route information.
     * @throws IOException if the server cannot be contacted, or the
     * connection is interrupted.
     */
    public String retrieveRouteData() throws IOException {...}

    ...
}
```

The method returns all route data, at once, in a single specially-formatted string. The string consists of multiple lines of text in the following format:

- Spaces at the start and end of lines should be ignored, along with blank lines and lines that consist only of spaces.
- The whole string then consists of one or more routes.
- Each route occupies one initial line, plus a line for each waypoint, including the start and end (thus, three lines at a minimum).
- The initial line for a route consists of the route name, followed by a space, followed by the route description. The route name consists of letters, digits and underscores (roughly the same rules as for Java identifiers – variable/method/class names). The route description can contain any characters except for a newline.
- Each waypoint line contains latitude, longitude and altitude, separated by commas. They are expressed as real numbers, in degrees or metres as appropriate, but without the actual units.

Then, except for the last point, there is a further comma and then a string value that can be one of two things:

(a) A description of the route segment beginning at the coordinates just given. The description may be empty, but it cannot contain newline characters, and it cannot start with an asterisk ("*").

(b) An asterisk ("*") followed by the name of a sub-route beginning at (at least roughly) the coordinates just given. The sub-route itself will need to be defined as a whole route somewhere else in the file (either *before* or *after* the current route).

The asterisk is not part of the sub-route's name – it is just there to distinguish between the two cases.

- The very last point consists *only* of a latitude, longitude and altitude. This is how you know that the route has ended, and that you should then look for the next route in the string.

While the server will do its best to supply valid data, your application *must not* rely on the server for validation, and must perform its own validation and error handling where applicable.

Here is an example of some (valid) formatted route data:

```
theClimb [description]
    -31.94,115.75,47.1,[description]
    -31.94,115.75,55.3,[description]
    -31.94,115.75,71.0,[description]
    -31.93,115.75,108.0,[description]
    -31.93,115.75,131.9

mainRoute [description]
    -31.96,115.80,63.0,[description]
    -31.95,115.78,45.3,[description]
    -31.95,115.77,44.8,*theStroll
    -31.94,115.75,47.1,[description]
    -31.93,115.72,40.1,[description]
    -31.94,115.75,47.1,*theClimb
    -31.93,115.75,131.9,[description]
    -31.92,115.74,128.1

theStroll [description]
    -31.95,115.77,44.8,[description]
    -31.93,115.76,43.0,[description]
    -31.94,115.75,47.1
```

Note that "`[description]`" could simply be any text – I haven't gone to the trouble of providing individual examples of it.

5 Tracking Progress

When the user is ready to go, they will select a particular route and then select "go" (or equivalent). At this point, the app must enter a completely different mode of operation – "tracking" mode.

As the user progresses along their chosen route, the app must track their progress, using the mobile device's GPS reader. The GPS location will only be periodically updated (as continuous use would drain the battery rapidly). To receive the GPS location, you must make use of the following notionally-pre-existing class:

```
public abstract class GpsLocator
{
    ...
    /**
     * When GpsLocator has received a new set of coordinates, it calls
     * this hook method.
     */
    protected abstract void locationReceived(double latitude,
                                             double longitude,
                                             double altitude);
}
```

This class is a partial implementation of the Template Method pattern, where the actual template method hasn't been shown.

You may assume that constructing it is sufficient for it to set up a new thread, connect to the GPS reader hardware, and call the hook method whenever it receives new coordinates (and that this can happen *at the same time* as you might be doing something else, like reading user input). We won't worry about race conditions, though, for simplicity.

When in "tracking" mode, the app must do several things:

- Show the GPS location on the screen, whenever it is updated.
- Show the remaining distance, broken down into horizontal distance, vertical climbing, and vertical descent (as described in Section 4). However, note the difference between *remaining* distance and the *total* distance described in the previous section. Here we only count up:
  (a) The distance from the last known GPS location to the next waypoint (the remaining part of the current segment); and
  (b) The distance of each segment *after* the current one (including sub-routes).
- Determine which waypoint the user is up to. Do this by comparing the current location to the next expected waypoint, and check whether the distance is less than 10m horizontally and 2m vertically. When this happens, notify the user.

  Take sub-routes' waypoints into account too. For instance, based on the example in Figure 1, the sequence of waypoints includes: $\ldots W_6, X_2, X_3, X_4, X_5, W_7, \ldots$

Once a waypoint has been reached, the app must then start looking for the next waypoint. Once the *end* is reached, the app must revert back to the original list of routes described in Section 4.

- Allow the user to manually indicate that they have reached a waypoint.

  This needed because automatic detection is not going to be perfect. The user could deviate past the next waypoint and never technically be within range of it. We could also fail to get a GPS signal for part of the route (for various reasons).

6 Your Task

Design and implement this system, providing the following:

(a) Your design, expressed in UML, containing all significant classes, class relationships, and significant methods and fields.

(b) Your complete, well-documented code, in Java, C++ or Python (your choice). Do not use any third-party code without approval, in writing, from the lecturer.

Although `GeoUtils` and `GpsLocator` don't really exist, *your code is expected to actually work* should those classes be written. Thus, you will need to develop stubs for them for your own testing and debugging purposes.

Then, in 2–3 pages total:

(c) Discuss what design patterns have you used, how you have adapted them to the situation at hand, and what they accomplish in this situation.

(d) Discuss coupling, cohesion, and reuse in your design.

(e) Discuss two plausible alternative design choices, explaining their their pros and cons.

7 Submission

Submit your entire assignment electronically, via Blackboard, before the deadline.

Submit one .zip or .tar.gz file containing:

A declaration of originality – whether scanned or photographed or filled-in electronically, but in any case *complete*.
Your source code – your .java, .py or .cpp/.h files (and build.xml if you have one).
Your UML – one .pdf, .png or .jpg file.
Everything else – exactly one .pdf file.

Note: please avoid the .rar format.

You are responsible for ensuring that your submission is correct and not corrupted. You may make multiple submissions, but only your newest submission will be marked. The late submission policy (see the Unit Outline) will be strictly enforced.

8 Marking Criteria

The marking criteria for this assignment are broadly as follows.

(Note: The advice given for each criteria below is only intended to address commonly-occurring issues observed in past assignments. It is not a complete guide to the assignment, and you should not mistake it for a promise to grant marks.)

Subtractive marks – Missing or incorrect functionality.

While this assignment is fundamentally about software design, one cannot have a good design that does not actually solve the problem at hand. Subtractive marks will apply if you omit required functionality. If you omit *a lot* of functionality (or somehow manage to have the wrong functionality), it will also be difficult to award any marks for other criteria below.

Subtractive marks – Notably poor quality code or commenting.

2 marks – Clear division of responsibilities.

Avoid having a "god class". Be very clear about what responsibilities each class, interface and package (or namespace) has. Have meaningful packages/namespaces.

2 marks – Minimal redundancy.

Avoid repetition, and otherwise unnecessarily long code.

4 marks – Extensibility and decoupling through polymorphism.

Ensure you understand what polymorphism actually is! You won't get these marks simply for using the `extends` or `implements` keywords in your class declarations.

2 marks – Testability.

Ensure you understand what testability is. (It isn't the same thing as simply conducting testing!)

2 marks – Error handling.

2 marks – UML consistency and relative completeness.

3 marks – Explanation of key aspects of the chosen design.

3 marks – Explanaton of two plausible design alternatives.

These must be significantly different from each other, and from the actual chosen design. You must give enough detail that I can see how your proposals address the problem(s) at hand. Vague answers get zero marks.

Total: 20 marks

9 Academic Misconduct – Plagiarism and Collusion

This is an assessable task, and as such there are strict rules. You must not ask for or accept help from *anyone* else on completing the tasks. You must not show your work to another student enrolled in this unit who might gain unfair advantage from it.

These things are considered **plagiarism** or **collusion**.

Staff can provide assistance in helping you understand the unit material in general, but nobody is allowed to help you solve the specific problems posed in this document. The purpose of the assignment is for *you* to solve them *on your own*.

Please see Curtin's Academic Integrity website for information on academic misconduct (which includes plagiarism and collusion).

The unit coordinator may require you to provide an oral justification of, or to answer questions about, any piece of written work submitted in this unit. Your response(s) may be referred to as evidence in an Academic Misconduct inquiry. In addition, your assignment submission may be analysed by Turnitin and/or other systems to detect plagiarism and/or collusion.

End of Assignment