Curtin University — Discipline of Computing

Object Oriented Software Engineering (COMP2003)

Semester 1, 2019

Assignment

Due: Thursday 23 May, 23:59 (pending class vote).

Weight: 25% of the unit mark.

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

You have a choice between *two* problem descriptions, detailed in the appendicies. Pick one of them. But first take a look at the marking guide.

1 Marking Criteria

This will be the same regardless of which problem description you pick.

Subtractive marks – Missing or incorrect functionality, to be assessed by demonstration.

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.

Demonstrations will be conducted in the practical sessions following the assignment due date. Please make yourself available then, or contact the lecturer to arrange an alternative time.

Also note: the problem descriptions will come with certain notionally pre-existing classes, or rather declarations. You *will lose marks* if you change these declarations, as it's part of the assignment problem to work with them as they are. (On the other hand, you can and will need to *implement* them in order to test your code.)

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.

A number of design patterns taught in the unit are based on polymorphism, and it's broadly the intention for you to use some of them. However, the marking guide does not allocate marks for any specific ones here.

Instead, we'll simply be looking for *two* separate instances of polymorphism, which broadly means two inheritance hierarchies.

However, there are some important caveats here:

- Your polymorphism must be *useful* in some way. In fact, it is probably going to form the very heart of your design. You can't just tack it on afterwards.
- Simply having inheritance is not enough. Remember that polymorphism occurs when you make a method call without knowing which subclass's method you're actually calling.

  A good question to ask yourself is: "Hypothetically, could I add in a completely new subclass, with new functionality, and make it work without touching the calling code?" If so, then you have polymorphism. (You are, of course, still allowed to modify the code that *instantiates* those subclasses.)

  If the calling code would require additional if/switch statements to cope with a new subclass, you *probably don't* have polymorphism. If you have the calling code and the instantiation code in the same file, then you *definitely don't* have polymorphism!

2 marks – Testability.

Ensure you understand what testability is. Don't confuse it with simply "doing testing".

To help make your code testable, get dependency injection happening. This may be assisted by careful use of factories, and may be *hindered* by use of singletons (unless you're very careful).

2 marks – Error handling.

We expect your application to handle errors gracefully and comprehensively. We expect you to anticipate all the different things that might go wrong at runtime, and deal with them properly.

Define your own exception classes as appropriate. Don't misuse exceptions. Don't catch high-level exception types. When handling exceptions, be careful to respect the division of responsibilities between classes and packages.

2 marks – UML consistency and relative completeness.

Ensure you understand the difference between association, aggregation, inheritance and usage dependencies. Remember that (in general) only non-static class fields are going to result in association or aggregation, as these are *long-term, object-to-object* relationships.

Ensure your diagram is consistent with your code! You can make simplifications to promote readability, but it has to be clear that this is intentional. Don't miss out important classes, or association/aggregation/inheritance relationships.

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

This will be part of a 2–3 page written document accompanying your design and code. Don't underestimate the effort required here. Even once you have your design, *explaining* it can take significant time and effort in its own right, even if the end result is not that long.

Imagine yourself trying to communicate your design to a colleague who doesn't know anything about it. Ask yourself what your colleague would want to know, first and foremost.

3 marks – Explanation of two plausible design alternatives.

This is the other part of the written document, and the purpose is for us to see that you're thinking deeply about the design.

There are always a virtually unlimited number of design options. A software designer mustn't just randomly pick one design, but be aware of several and weigh up their situation-specific advantages and disadvantages.

To this end, in addition to your actual design, you must describe *two more* possible designs, and again there are some caveats:

- Your alternative designs must be *plausible*. We're not interested in things that are ridiculous and/or wouldn't work. We're interested in relatively sensible alternatives that actually *would* work, but which you simply didn't choose, for more subtle reasons.
- You must give enough detail for us to see how your alternative proposals address the problem(s) at hand. Simply saying "I could have used pattern X instead of pattern Y" doesn't give enough information. Perhaps you could, but *how* would the rest of the system be redesigned to accommodate it? There could well be several different ways to use the same pattern!
- Your two alternatives must be reasonably different from one another, and from the actual chosen design. If the differences wouldn't show up at the level of a UML class or object diagram, they're probably not different enough.

Total: 20 marks

2 Your Task

Design and implement *either* the system described in Appendix A, or the system described in Appendix B (not both, unless you really have a lot of free time!)

(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 the classes described (e.g. `GeoUtils`, `SMS`, etc.) 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 advantages and disadvantages.

3 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.

Avoid the .rar, .zipx, .7z formats, or any other non-standard archive/compression format. Do not assume the marker has extraction tools for anything but ordinary .zip and .tar.gz files. If the marker cannot unpack your work, then it won't be marked!

You are responsible for ensuring that your submission is correct and not corrupted. Once you have submitted, you are *very strongly advised* to download your own submission and thoroughly check that it is intact.

You may make multiple submissions, but ordinarily only your last one will be marked. Your last submission time will be the time used to determine whether a late penalty applies, and what that penalty is. See the Unit Outline for further details on the late submission policy.

4 Demonstration

You will be required to demonstrate your application to a marker in-person, *only* in order to address the first item on the marking guide. This will involve rebuilding and running your application to show that it performs as expected.

You must either be available in your practical sessions in the weeks following the assignment due date, or otherwise schedule an alternate time (in agreement with the marker/lecturer) *before* the start of the exam weeks.

5 Academic Integrity

Please see the *Coding and Academic Integrity Guidelines* (available alongside this specification on Blackboard).

In summary, this is an assessable task. If you use someone else's work or assistance to help complete part of the assignment, where it's intended that you complete it yourself, you will have compromised the assessment. You will not receive marks for any parts of your submission that are not your own original work. Further, if you do not *reference* any external sources that you use, you are committing plagiarism and/or collusion, and penalties for academic misconduct may apply.

Curtin also provides general advice on academic integrity at
http://academicintegrity.curtin.edu.au/.

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.

A Problem Description #1: Route Tracker

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.

A.1 Routes

When out in the wilderness (or, in fact, anywhere), trekkers and climbers follow pre-defined 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.

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.

[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.

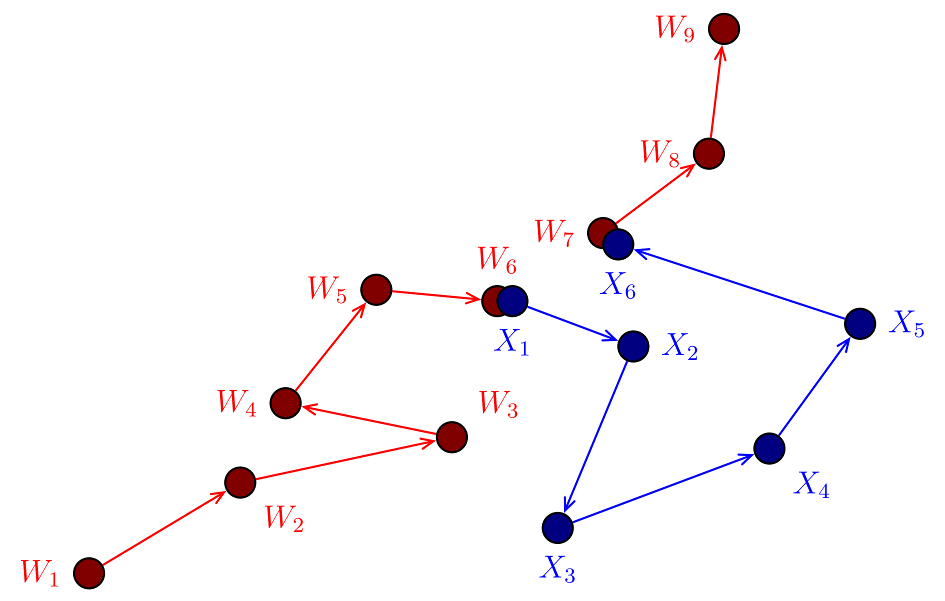

Figure 1: Route example. Here there are two routes W and X, where X is used as a subroute within W. (This only shows a top-down view, but remember that each point also has an altitude.)

A.2 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]:

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

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.

A.3 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 Amazing views!
    -31.94,115.75,47.1,Easy start
    -31.94,115.75,55.3,Tricky, watch for drop bears.
    -31.94,115.75,71.0,I*feel,like.over-punctuating!@#$%^&*()[]{}<>.?_+
    -31.93,115.75,108.0,Getting there
    -31.93,115.75,131.9

mainRoute Since I was young
    -31.96,115.80,63.0,I knew
    -31.95,115.78,45.3,I'd find you
    -31.95,115.77,44.8,*theStroll
    -31.94,115.75,47.1,But our love
    -31.93,115.72,40.1,Was a song
    -31.94,115.75,47.1,*theClimb
    -31.93,115.75,131.9,Sung by a dying swan
    -31.92,115.74,128.1

theStroll Breathe in the light
    -31.95,115.77,44.8,I'll stay here
    -31.93,115.76,43.0,In the shadow
    -31.94,115.75,47.1
```

As you can see, descriptions can consist of any arbitrary sequence of characters, except for a newline, and except that *segment* descriptions can't *start* with an asterisk.

A.4 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:

```java
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 A.3). 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 A.3.
- 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).

B Problem Description #2: Election Campaign Manager

Design and implement a system for helping to coordinate election campaigns using social media.

Your client, and prospective sole user, is the chief "political strategist" for a political party; let's call it the Australian Hedonism Party. During an election campaign, they must manage a large team of volunteers, and devise "talking points" for those in the Australian Hedonism Party to follow.

However, election campaigns can also be very fast-moving and quickly-changing, and social media has become a powerful force. Your client needs an automated system to monitor Twitter and Facebook for a series of keywords, so they can organise rapid responses to trending issues. In addition, the system must be able to notify a range of people via Twitter, Facebook or SMS.

B.1 Basic Data

The system must provide functionality to view, add and remove the following:

People – volunteers, candidates, and strategists, each identified by a unique, auto-generated ID number. Each also has a name (obviously), a type (volunteer, candidate or strategist) and one or more of the following contact details: a mobile number, a Twitter username and/or a Facebook ID.

Policy areas – used to categorise issues. Traditional policy areas include "Education", "Defence", "Workplace relations", etc., but new ones may need to be created depending on the situation. These have a name, and also a set of related keywords and talking points.

Talking points – these are very short, highly simplistic messages that campaigners and candidates can use repeatedly, to try to sway public opinion or respond to criticism. They are carefully worded (by strategists) to appeal to voters' emotions, for maximum effect. Each talking point is related to a specific policy area.

Keywords – these will be searched for in Twitter and Facebook (by pre-existing code – see below). Each keyword is a single piece of text, but it is also related to a specific policy area.

When adding talking points or keywords, the related policy area must be specified, and hence must aleady exist.

When removing a policy area, the system must first check whether there are any related talking points and/or keywords. If so, it should display these to the user before asking for confirmation. If the user *really does* want to remove the policy area, any related talking points and keywords must also be removed.

In principle, we also need to save and load all this data to/from file. However, for the purposes of this assignment, you can put stubs in place of the actual file reading/writing algorithms.

B.2 Notifications

Notifications happen under these circumstances:

When a new keyword is added. Only volunteers are notified of this.
When a keyword is trending (i.e. gaining frequent use on social media). Only volunteers and strategists are notified (not candidates).
When a new talking point is added. All types of people are notified.

The system must allow configuration of notifications on a *per-person* and *per-policy-area* basis. That is, the user may wish to notify *a particular person* when something happens related to *a particular policy area*. In order to notify a person, all available contact details are used: Twitter and Facebook (privately) and SMS. However, recall that only a subset of these will actually be available for any given person.

Notification settings can be added and removed like the other kinds of data. When adding a notification setting, the user will specify the person and policy area to which it applies.

When removing a person or a policy area, any related notification settings must also be removed. You *can* warn the user about this, but it is not essential.

B.3 Keywords and Social Media

There is existing code (see below) to help monitor social media. Your code needs to work with it and come up with a collection of keywords to look for.

The existing code will run in its own threads, and conduct periodic searches (every few minutes) for the keywords specified. It counts the number of tweets and Facebook posts containing each keyword, and reports this information back *asynchronously*.

(Note: in reality, unless you're careful, such multithreading can lead to "race conditions". This is where different threads access the same data at the same time, with surprising and unpredictable results. However, for the purposes of this assignment, you do not need to worry about race conditions.)

A keyword is said to be "trending" if it occurs in 50 or more posts (on Twitter, Facebook, or a combination of the two) over any one-hour period. When this happens, notifications must be sent to the relevant people, based on the policy areas related to trending keywords. (Although each keyword is related to only one policy area, it is possible that multiple keywords, will trend at the same time.)

However, if a keyword *continues* to trend, the software must hold off any further notifications (for that keyword only) for 24 hours after the original set of notifications. If the keyword is still trending after 24 hours, a second set of notifications should be issued, followed by a second 24-hour hold-off period, and so on. This should not affect notifications for other keywords or other events.

B.4 Existing Classes

You will need to make use of the following notionally-pre-existing classes. Note that `TwitterMessenger` and `FacebookMessenger` are partial implementations of the Template Method pattern. However, they do not extend/implement a common superclass/interface.

IMPORTANT: Do not change any declarations in these classes (except as needed to translate them into Python/C++ if desired).

```java
package edu.curtin.messaging;
public class SMS
{
    public SMS() {...}

    /** Sends an SMS to a given phone number. */
    public void sendSMS(long mobileNumber, String message) {...}
}
```

```java
package edu.curtin.messaging;
public abstract class TwitterMessenger
{
    public TwitterMessenger() {...}

    /** Sends a private Twitter message to a given user. */
    public void sendPrivateMessage(String id, String message) {...}

    /**
    * Replaces the existing set of keywords to be monitored with a new set.
    */
    public void setKeywords(Set<String> keywords) {...}

    /**
    * Called by the monitoring thread after every search. For each keyword
    * being monitored, the map parameter contains the number of times that
    * keyword has been mentioned on Twitter since the previous search was
    * performed. The timestamp parameter is the time of the search,
    * measured in seconds since the "epoch" (01/01/1970).
    */
    protected abstract void keywordsDetected(Map<String,Integer> keywords,
                                             long timestamp);
}
```

```java
package edu.curtin.messaging;
public abstract class FacebookMessenger
{
    public FacebookMessenger() {...}
    public void sendPrivateMessage(String id, String message) {...}
    public void setKeywords(Set<String> keywords) {...}
    protected abstract void keywordsDetected(Map<String,Integer> keywords,
                                             long timestamp);
}
// Note: except for the word "Facebook", the declarations and functionality
// here are identical to the TwitterMessenger class.
```

End of Assignment