Way-to-Go Design Report

by Callum France

Architectural Pattern

Way-to-Go used the model-view-controller architectural pattern.

The model includes all information the program uses, and all calculations performed on said information. This is further separated into two sub-sections - DirectoryModel and TrackerModel. The reason two sub-models were used is because some parts of the model are instantiated when the program begins (DirectoryModel), whilst others are created and destroyed during the course of the program (TrackerModel). The program will always need to have one Directory object representing all known data in the program, however, when it enters tracking mode, needs to create a sub-set of the known information and perform specialized calculations on them.

Way-to-Go's view is simply the user interface. In this case, the view is console-based, simply outputting lines to the console, and reading inputs from the user.

The programs controller links the model to the view, and will call methods within itself when the programs changes state. The controller is the only class accessible from main.

Composite Pattern

Way-to-Go uses the composite pattern to store route information. Because it is possible for a route to contain another route, and so on, it was necessary to use a composite pattern to allow this kind of 'object recursion' to take place. Along a routes pathway, each point could either represent another route, or a description with coordinates. Thus, the interface Segment was created in the UML, of which both Description and Route inherited from. Description was the leaf node (and also inherited from the base coordinate class Waypoint). Route was the branch class, both inheriting and aggregating from Segment. In this instance, the root level class in the composite pattern was always going to be a Route, and as such Directory simply inherits from Route instead of Segment. It should also be worth noting that since Way-to-Go was written in Python, duck-typing has been used instead of writing a Segment class.

Dependency Injector Pattern

The Directory class uses a factory to create its map of routes in `Directory.update_directory()`. If this method was to create its own factory class directly, it would be incredibly hard to test the Directory class in isolation of SegmentFactory. Thus, dependency injection was included. The method has a SegmentFactory parameter, and Directory's parent class (Controller) creates the SegmentFactory object that is passes into it. This increases testability of the code because you can now mock SegmentFactory when testing this method - otherwise the test code will create a fully functional SegmentFactory.

Factory Pattern

The Directory class needs to populate itself with information from GeoUtils, however this is a rather long and complicated process. The SegmentFactory class was created to do this, decreasing the size and complexity of Directory. This factory has the sole purpose of reading in data from GeoUtils and outputting a map (in Python, a dictionary) of populated routes, accessed via their route name. This means that in the future, if SegmentFactory or GeoUtils were changed, the Directory class is unscathed.

Iterator Pattern

Often, the program needs to access all elements stored inside a list or set and perform an operation on each of them. When the Tracker class needs to update its observer, the iterator pattern for-loop is used to go through all elements in the 'tracker observer' set -

```
def notify_tracker_change_obs(self):
    for o in self.tracker_change_obs:
        o.tracker_update(self)
```

Observer Pattern

The observer pattern allows the program to update itself in response to a certain event or action. In Way-to-Go, two different observers are implemented - after the Directory has finished being updated; and after an 'accessible' class field in Tracker has changed.

Using observer to do this allows the view to display updated context-specific information without resorting to some kind of polling in Controller. Instead, the Directory/Tracker notifies all their observers, which is directly inherited in the appropriate nested concrete observer class inside UI, allowing the display to instantly update.

The concrete observers are nested classes inside appropriate View methods. This is because the observer implementations are only used in very specific and small sections of the UI. The concrete observers are created and added to their appropriate Model observer sets when the UI method begins, and removed when the method finishes. This is to prevent future Directory updates from erroneously overriding the current console display because it notified the UI directory observer.

Template Pattern

Way-to-Go uses the abstract hook method `GpsLocator.locationReceived()` to update the current location in Tracker. This is achieved by allowing Tracker to inherit from GpsLocator and writing the concrete hook method - which updates Tracker's current location.

Alternative Designs

by Callum France

Alternative 1

Instead of using a composite pattern to store routes (that may contain other routes and so on), an alternative may be to simply enforce that a Route object can only contain Description objects. If a Route does contain a sub-route, then that sub-route is expanded into a list of Description objects and this list is inserted in its place.

The biggest advantage of this alternative is that it streamlines the use of an iterator pattern in Routes. Because every segment in Routes is a Description, you will never have to recurse down into a sub-route to find the next Description object.

One downside that is normally expected from this implementation is that if the data inside one Route (that was also a sub-route in another Route) were to change, because this Route is not referenced in the other Route, the data will not be changed in the other Route, despite no longer being correct. However, because GeoUtils will only ever update all data at once, this scenario will never occur.

One real consequence of this alternative is memory. If a Route is a sub-route, then the Descriptions inside this sub-route are stored multiple times despite being identical.

Alternative 2

An alternative to Way-to-Go's current UI class would be to refactor it using the State pattern. There are currently three separate calls to the View UI class within Controller - the UI methods stemming from these calls could all become their own separate View classes in different python files. Each of these View classes could aggregate a different 'State' interface, and the inheriting classes of these states would each represent a single state of the current UI class.

The UI class currently contains complicated control flow statements to write specific messages to the console. Using states and separate classes will separate the control flow clutter out of the UI class and into concrete state classes - i.e. code hiding. This will make the resultant UI classes highly readable.

However the biggest drawback to this alternative is that it requires a lot more classes to be written and tested, taking more time to implement in actual code.

Furthermore, it may increase inter-class coupling through the Directory and Tracker observers. The concrete observers would be added and removed from the respective concrete UI class. As such, the notify observers method will need to travel from their origin in the Model, through the observer interface, into the View UI class, then through the state interface and finally into the state class.