Object Oriented Software Engineering (COMP2003) Assignment

Software Design Report based on program written to address Problem Description #1: Route Tracker

Name: Minh Dao

Student ID:

Key Aspects of Chosen Design

Composite Pattern

The composite pattern was used in the program to address the problem of how to construct the Route/Segment data structure in a flexible manner by using recursive aggregation. This is done by first constructing an abstract class Path that contains the common methods/classfields between Routes and Segments. In our program our Segment would be considered the “Leaf”, as in end a Segment contains only 2 points and an altitude. In this program, the Route class is considered the “composite” class as Routes contains a list of Path objects. What this allows us to do is that we can treat all the Paths in the in-Route list in a similar way without caring or knowing if that Path is a Route (Sub-route) or a segment. An example of this can be seen in the program via the calculation of the total horizontal distance. The common abstract class Path defines an abstract method of calculating horizontal distance and Segment and Route override it and implement it in their own way. This way when total horizontal distance is calculated inside a Route, it can iterate through the Path list and call horizontal distance and add the values up. It doesn’t have to care whether that Path is a Segment or a Route. We are decoupling through polymorphism. Using the composite pattern in this way aids in extensibility as well as you can add more different types of “Leaves” to your program without much adjustment if it extends Path. An example would be if in the future a Route could also contain Mountains. You would create a Mountain class

State Pattern

State pattern within the view in order to set the behaviour of the UI based on the state of the program. The program has 2 distinct states, the main menu state, where the user can choose what route they want to take, and the tracking route, where the user is tracking through. This is achieved by creating 2 menu state classes, MainMenuState and TrackingMenuSate, which both implement the interface MenuState and a context class Menu. Menu’s behaviour will change depending on what the set state is. Extensibility is also achieved by implementing the state pattern, as new states will can be added easily. New states can be created, and the Menu context class would just have to set the state as that new Class if the program’s behaviour changes to that state. An example for Route Tracking is if a new 3rd state was to be added that allows the user to map out their own route.

Factory Pattern

The factory pattern was used in our program in order to create the Route and Segment objects based off the information sent by the GeoUtils server. Since we don't know what Routes and Segments we need at Runtime, the RouteManger will use the factory [PathFactory| to read in the GeoUtils gives from the retreiveRouteData method and decides which object to make a, Route or a Segment and if the that Route is a sub-route for another larger route.

Observer Pattern

The observer pattern is used within our program in order to notify our program whenever a new GPS location has been received by the GPSLocator. This is achieved by first having our TrackingProgress class, the class that contain information about the trek that the user is on implement the GPSObserver interface. A GPSWrapper was also created that wraps around the GPSLocator. This class as the event-source, containing a list of observers and the event being when a new location is received from the GPSLocator. When this happens the GPSWrapper notifies all the observers in the list of the event and how they handle it is dependent on the GPSObserver. In the current state of the program, there is one observer and when its gets notified of a new location, it moves the user to that current location and recalculates all the new distances remaining based off of that new location, and determines weather or not the user has reached a new waypoint, or finished the trek. This design allows for extensibility as there can be multiple observers observing the same event, an example being 2 TrackingProgress objects, each tracking a different individual.

Other Design Patterns

The MVC (View, Model, Controller) was used to determine the overall structure of the software such that Model, only contains classes whose main function is to store information about the program. View contains classes that concern themselves with the UI of the program and Controller contains all the classes that govern how the program should behave/ classes that complete calculations. This is done by creating 3 packages, View, Model and Controller and placing the appropriate classes in them. Dependency injection was also added in order to improve maintainability and testing. It was achieved by not hard coding any dependencies in any of the classes, rather creating any required objects first, then passing them over to the required Object.

Design Alternatives

Decorator Pattern

An alternative to using the Composite pattern to create the data structure for Routes and Segments is using the decorator pattern. In this situation, rather than having 2 Classes that both extends a common abstract class, you instead have a single base class (In our program that would be point) and a single linked list of points. An abstract decorator class would then be created, and all the decorators created would extend that class. In this situation that would be things such as Segment, Route, sub-route. This way when the program knows that a point forms part of a segment, it decorates that point with a Segment decorator. Likewise, when it detects that a point is the start or end of a route, it would attach the Route decorator. This alternative provides the added benefit of being able to add decorators during runtime (for example a future release of this program, might add the ability to create a sub route from an existing route, or a longer segment).

Strategy Pattern

An alternative to using the state pattern in order to govern how the UI, operates in the program is using the strategy pattern instead. This can be achieved by creating a common interface that the 2 menus implement. The program then chooses the correct strategy depending on what option the user decides on run time. The affect this would have is that, you would have to change up the way the UI works. Rather than present the main menu at the start, the user would be prompted to either start the trek or view/choose a route. Depending on what they choose the program would adopt the appropriate strategy the show the user the correct UI.