OOSE Report 19127639

This report will discuss in detail the design of my software for my OOSE Assignment submission. The problem this report is addressing is the GeoLocation problem.

Design Discussion

A brief introduction to my design is as follows. The software itself was designed around the state pattern and the states of the system. From the specification I gathered that the system is in 1 of 2 states, the system is either tracking or it is not tracking. This dictated a fair majority of the rest of the systems design. The first state being non-tracking state, which I called *DataMode*, would encapsulate all of the functionality to deal with the problems in A.1 through to A.3 and then the other state, tracking state, would deal with the problems in A.4, although some functionality/design was used between them for reuse purposes. For both of the states I used the composite pattern with a slight modification to represent the structure of a route. From that structure I used controllers to perform operations on the data and in turn achieve the functionality. Basically that's the software. Going more in depth however there were numerous patterns in the design of the software that were used to achieve functionality, and also to achieve things like polymorphism, good reuse ect.

The patterns used in the design:

State Pattern - As already stated the state pattern is used to represent the overall structure of the entire software. There were two states, both were system states that represented the state the entire system was in and the functionality it could perform while it was in said state. Those states were tracking (TrackingMode) and non-tracking (DataMode). The pattern is the basic version of the State Pattern, the only adaptations were simple getters and setters for the data needed by each of those states, e.g the tracking mode needs a route to track which it gets from the non-tracking state. Apart from that no other real "adaptations" were made and the generic state pattern was applied. What they accomplish however is quite a lot. The most important thing that the use of the state pattern in my design accomplishes is the clear division of responsibility and the division of functionality. Almost all of the classes are bounded/restricted by the current state of the application, which is a good thing from a separation of concerns point of view. This was also important and helped me when dealing with all the various problems as the division meant I knew what classes needed to deal with certain problems and what classes had nothing to do with other certain problems, which again is good for separation of concerns.

Composite Pattern - The composite pattern was used to represent the complex object structure of the problem, that is the routes, all the points (Waypoints and Segments) the routes contain and the fact that it can contain multiple routes. A leaf node superclass represented the waypoints and the segments, whilst the composite was the route/s. The composite pattern in my design was adapted to accommodate the fact that the route/s could contain multiple waypoints and segments. So the routes could contain any number of the leaf-nodes and also any number of other routes (subroutes). This adaptation was achieved through use of collections but also during runtime when creating the composite pattern itself. Smaller adaptations in the methods were also made in order to accomodate for the fact that the leaf-node was a superclass which could be extended by any number of subclasses. What the pattern itself accomplishes in the given situation is it allows the complex structure defined in the specification to be easily represented in the system and therefore be operated on a lot easier. There were however some major disadvantages with the design in terms of the composite pattern and how it was used in my design. The biggest issue being the ability to perform operations on the object structure whilst keeping the polymorphism/coupling. When it came to DataMode state and performing the functionality on the object structure created by the composite pattern it was very simple as the structure would perform methods

that would iterate over itself and perform operations. For example all the nodes had print methods for displaying, so you just call one of them on the route and it will iterate through and print everything without actually knowing if whatever it was printing was a route or a waypoint ect. The problem with the tracking state and the operations it needed to perform on the composite object structure was that it needed to know if a child node was a waypoint, a segment or another route. This kind of undid all of the good the composite pattern did in the first place because it required me to typecast and in turn required the tracking state and it's controllers to know more about the composite structure. This was a major design flaw and looking back on the whole process if I had picked this up sooner I probably would have attempted to make some serious changes to the design.

Template Pattern - Template pattern was used when calculating the distance, in both states. No real adaption was made to the pattern, just a standard template pattern implementation with two subclasses that implemented the hook method, one of them getting a list of all nodes (for total distance) and another for getting a list of remaining nodes (for the remaining distance). The beauty of my design here is that it didn't matter what state the application was in, you just construct a hook method for the template pattern which would get the list of required nodes and it would perform the operation. Main advantage of this was reuse, it was reused multiple times in the design of the software. Another advantage was it made things simpler, no heavy computation or anything like that. Only disadvantage was you couldn't really make any modifications if you needed to, to the template method. Although since my algorithms were very basic that wasn't really a problem.

Injection Dependency Pattern - Injection dependency was used quite a bit here. Injection Dependency was adapted to the situation at hand by not having any direct object dependencies and having minimal indirect object dependencies in the system. The injectors was main class and the state classes. Main advantage of this pattern was testability, when things went wrong it allowed to quickly isolate what object was the problem and in turn helped fixing it. It also improved the maintainability of the system a bit. It also helped in the early stages of my design when i was changing objects around a lot and whatnot, due to the lack of dependencies in the classes not much had to be changed when I removed an object or changed some aspect of the design it was just the injectors that needed to change. The pattern accomplished easy modifiability and testability, as stated above.

Strategy Pattern - Strategy pattern was used for measuring the distance between two waypoints where the strategies was how that measurement was performed. I.e 1 strategy implementation for measuring the horizontal distance between 2 waypoints and another for measuring the vertical climbing distance between 2 points, ect. Although I didn't really implement this pattern for the polymorphism, it was more just because all of those takes were related to one another so by using it we could simplify the entire software design down a bit. Again, no real adaptations were made, it was just a typical strategy pattern implementation with 3 strategies and 1 method. Advantages was that the measurement would be performed regardless of the type/you didn't have to specifically know which type as that was up to a factory.

I've already commented on coupling, reuse and cohesion for each of my classes in the program code itself, through the use of comments, so i'm not going to go over individual classes again. Instead I'll just run through the design of the system as a whole and how I was able to achieve the levels of coupling,reuse and cohesion through the use of my specific design. Because of the strategy pattern, the template pattern and the way they were incorporated into my design I think it's fair to say that in terms of reuse the design does that quite well as it saves quite a bit of extra redundant code that would be needed to perform the distance calculations if I didn't use those patterns. I also think that in terms of cohesion it also achieves this quite well, especially since my design doesn't exactly use MVC. The design uses it on the surface to classify classes and responsibilities of those classes and their functions but there is no real MVC pattern in terms of observer pattern between the model and view ect. Cohesion itself hasn't really been achieved through a single pattern, although the use of the state pattern in my design

did mean that cohesion and responsibility was partitioned fairly well, but rather through most of the classes. All the classes has comments up the top about their purpose and the cohesion in that class and for the most part the methods in all of the classes fall within the scope I outlined. So ye in terms of cohesion I'd say the design achieves that well and that for each class it has one main and clear responsibility that it's methods achieve. In terms of coupling I'd say that overall it was average. In some classes there is no coupling whatsoever but other classes there is high degree of coupling. Coupling in this design was basically how I could achieve the task/problems with as little classes knowing about each other, so in some cases like measurement calculation it doesn't need to know about any other classes as it just takes two numbers and performs the algorithm but other classes such as the view or state classes need to know things about other classes in order to perform their job. Again, this is due to the design and it not being a completely low coupling design, the design requires some classes need to know about others. Overall coupling is alright as there is no real cases were there is too much coupling in one area.

Evaluation

Plausible Alternative 1 - The first alternative would address the whole composite pattern issue I discussed earlier on. That is the operations the tracking state needed to perform required that the system know more about the nodes, ie what type of nodes they were and access to their functions. So this alternative is using a different object structure, and not using the composite pattern. This different object structure would most likely be something like a Graph, where the vertices would be waypoints and the edges would be segments. How is that any different to the composite pattern? The composite pattern treats both leaf-nodes and composite nodes as Nodes. This presents an issue with the problems I was facing as in order to perform high level operations on the nodes you need to know their type and need to typecast them so you can use their methods, for example if a leaf-node had a method that a composite node did not and you're going through the list of node children, you need to determine if the child node is a leaf node and then call the method, which is counter intuitive to the whole reason you choose the composite pattern, to have the ability to treat those objects the same/similarly. With a graph edges and vertices aren't treated the same, edges are not vertices and vertices are not edges, both a separate entities which would mean performing operations could be a lot easier as a graph or a route in this case would contain a list of segments and a list of waypoints and for the computation you just figure out which one you're up to and then perform your calculations. So from an operations point of view, in this kind of context, a graph like object structure would have been more beneficial then the composite structure. The main disadvantage is that a graph can't adequately represent something as complex as this problem in terms of routes containing sub routes and whatnot, without the graph itself being over complex. This was why i ended up going with the composite pattern for the object structure, because it's a lot easier to represent the object structure of a route containing many sub routes than it would be for a graph.

Plausible Alternative 2 - The other plausible design alternative was using the observer pattern and basing the design of the Tracking state around observers. The journey controller would have a bunch of observers registered in it, waiting for some event in regards with the tracking to occur. It also could have been used for the GPSLocation and periodically updating the location. Send an event when that happens and go from there. The main advantage of using this alternative in my design would have been that the classes would be less coupled and there would have been a higher level of cohesion as the JourneyController would not have had to handle a lot of the logic for the tracking and the logic could have possibly been broken down into several classes. The main disadvantage, and the reason I didn't do this alternative, was because the Observer pattern is only really good for big complex programs that have lots of events. The system described in the specification doesn't really have many events and the problem itself isn't that complex. Because of that I thought that my existing design would capture the whole problem better.