OOSE Assignment report

Rares Popa - 19159700

1. Discuss what design patterns have you used, how you have adapted them to the situation at hand, and what they accomplished in this situation
   - For my assignment I used 4 different patterns and 1 method to achieve the required outcome.
   - Composite
     - The composite pattern is used where we need to tread a group of objects in comparable way as a single object. This pattern is represented using a tree structure, as it can either have a leaf node class or composite node class. The composite node class aggregates its own superclass.
     - In my design I had the Path as the superclass (interface too) and the route class was the composite node while the segment class was the leaf node.
     - What occurs in my code is that if the path is a segment object then it means that it is a leaf node and aggregation stop but if it is a route object it can be aggregated because a route class *contains* segments, like a tree we continue to aggregate until there are no more composite nodes (route objects) and only the leaf nodes (segment objects) remain.
   - MVC
     - MVC stands for Model-View-Controller, this pattern is used to separate concerns
     - The model represents an object or it can have simple accessors and mutators to change the data of the model. The view represents the visualization of the data that model contains, it deals with Input/output and is usually called the UI. The controller oversees the decision-making/business-logic part, it controls the data flow into the model object and it updates the view whenever data changes.
     - In my design I did just this, I split my code into 3 files depending on what it does/contains. I made sure to keep the separation of concerns, for example I made sure that only the view can output information to the sure and can only get input from the view too. The model classes created objects while controller classes updated objects, model/controller classes also threw their exceptions to view classes so it can be handled.
     - This made my code much cleaner and easier to read, it also made sure that one class wasn't a superclass with high coupling.
   - Observer
     - The observer pattern is used when there is a one to many relationship between objects, if one is notified then the others are all identified. Observers have a common interface and the subject's observers are added and removed by another class.
     - In my design I used the observer pattern 2 ways, firstly I used it with the GivenLocation class, when the observer detects a change (when a new Gps location is given) then it updates all the other observers and then the UI which outputs the new updated statistics. Secondly I used it lightly in to observer and detect when the data is getting updated and then update the lists/maps associated with it..

        - The LocationObserver observes the model and when the observers detects changes it updates the view which allowed me to automatically detect when there is a change to the GPS without manually checking ever x seconds.
    - Factory
        - A factory is a method which creates and returns an object, a factory allows us to decouple the code, it removes the need for hardcoding. We create an object without exposing the creation logic to the user.
        - In my design I have a factory method which build the route, depending on what type we needed it would either create a segment object or a route object, this class was called RouteFactory.
        - This pattern allowed me to decouple my code and not expose the user to creation logic and just make the code nicer to read and understand in general.
    - Template method
        - Beside all the patterns I also used a method, the Template method. It demonstrates inheritance. Template methods are used when you need several algorithms that are all identical, except for one more steps.
        - I used this method with the GpsLocator class where GivenLocation was a implementation of it
2. Discuss coupling, cohesion, and reuse in your design
    - Coupling is the degree of independence a submodule has. The lower the coupling the better because high coupling makes it difficult to maintain your code.
        - In my design I tried my best to make sure that the coupling was low between submodules, one major way I achieved this is using the MVC pattern. Each class has a different task to do and doesn't need to worry about other class.
    - Cohesion refers to what a class can do, low cohesion means that the class does lots of different actions (broad) while high cohesion means the class is only focused on one main goal
        - In my design I made sure that cohesion was high, modules only did what they required todo, an example is the menu submodule. Instead of having the menu focus on updating the data, getting user input, displaying route info I placed all those into a separate class so that they can focus on that while the menu submodule only needs to focus on displaying the menu and receiving an option.
    - Reuse of code is when the coder copies an existing method/function/etc from one submodule to another without much or no change. This leads to messy code which makes it hard to maintain. If one function breaks all do and you need to change each one.
        - In my design I made sure reuse of code was kept at a minimum, a good example is user input. When asking for user input instead of reusing the code I placed it into a submodule and called it each time I needed to get user input. Making sure that if that code ever breaks I only need to go to one place to fix it = clean code and easily maintained code

3. Discuss two plausible alternative design choices, explaining their pros and cons
    - Singleton pattern
        - Another design choice which I could have done but didn't was the use of a singleton class, these classes are used usually for initialization as it restricts itself to only having one instance. In my design I could have used this pattern to configure classes, through a config class.

    - Pros include
        - Instance control – Singleton prevents other objects from instantiating their own copies of the singleton object ensuring only one object is created
        - Flexible – The class has the flexibility to change the instantiation process
    - Cons include
        - It is sometimes called the 'antipattern' as it reduces testability
- Dependency Injection pattern
    - This pattern is basically the providing of objects to an object (is dependencies) instead of having to have it constructed itself (Prevents hardcoding). It makes sure that we create low coupled classes and is intended to improve maintainability and testability. In my design I have kind of used this at times, mostly in main. I passed the objects to the object instead of initialising them inside of the constructor.
    - Pros include
        - Increases testability
        - Improves maintainability
    - Cons include
        - Makes the code much less readable
        - Makes it harder to manage all the objects in the class