Marking Criteria

This document contains my response to the specific marking criteria for this assignment. Justification of my design choices will be discussed.

1. Clear and distinct package/class/interface/method responsibilities

Given that the MVC system was not appropriate for this application, I opted to package parts of my software based on the individual design patterns I used and the containment of specific tasks. I felt that this appropriately isolated sections of my code into specific categories based on their use and implementation. These packages are listed below:

- ApiObserverPattern
  - Contains classes required to implement the Observer Design Pattern
- CommandsStrategyPattern
  - Contains classes required to implement the Strategy Design Pattern
- CommandValidation
  - Contains classes required to handle invalid commands. Includes custom exception class.
- RoverStatePattern
  - Contains classes required to implement the State Design Pattern specifically for the current state of the Rover.
- VisibilityStatePattern
  - Contains classes required to implement the State Design Pattern specifically for the current visibility state of the Rover.

In order to decouple tasks, my application has been separated into clear and distinct classes which contain task specific methods. For example, all the code relating to the validation of incoming commands is contained within a single class: `CommandValidation`. Within this class, there are specific and isolated methods to handle the variety of validations required.

2. Appropriate error handling

General command validation

I created a custom Exception to handle the validation of incoming commands.

- `CommandException`

All incoming commands are validated and any invalid commands throw a `CommandException`. This exception will format all error messages as mentioned in the assignment specification.

- `! ...` - where "..." represents the invalid command.

These exceptions are caught from the `ApiData.updateApi()` method where the error message is then sent back through the `EarthComm.sendMessage()` method.

State based command validations

In the event that a command is of a valid format, but is unable to execute due to a currently running command (such as a Drive command while currently performing Soil Analysis), the application will send a error message through the `EarthComm.sendMessage()` method directly.

These situations are handled through the implementation of the State Design Pattern for the Rover State.

3. Implementation of dependency injection

In order to improve the maintainability and testability of my code, i made sure to implement dependency injection. I have used two injectors, as can be seen within the `MainApp` class.

All the core class objects are created within the `MainApp` class, and injected as required. Please see the screenshot below for example:

```
public static void main(String[] args)
{
    EarthComm eComm;
    Sensors sens;
    EngineSystem engSys;
    SoilAnalyser soil;
    ApiData apiData;
    RoverState roverState;
    VisibilityState visState;
    Rover rover;

    eComm = new EarthComm();
    sens = new Sensors();
    engSys = new EngineSystem();
    soil = new SoilAnalyser();
    apiData = new ApiData(eComm, sens, engSys, soil); //Dependency injector

    roverState = new Stopped();
    visState = new NormalVisibility();
    rover = new Rover(eComm, sens, engSys, soil, apiData, roverState, visState); //Dependency injector

    apiData.updateApi();
}
```

Particularly, with the injector for the creation of the rover object, this allows the object to be tested with the full variety of Rover States (Stopped, Driving, AnalysingSoil) and Visibility States (BelowFourVisibility, NormalVisibility, AboveFiveVisibility).

All the class constructors i have implemented, consists of basic data initialisation with class methods handling any extensive logic. Example of the `ApiData` constructor:

```
public ApiData(EarthComm eComm, Sensors sens, EngineSystem engSys, SoilAnalyser soil)
{
    this.eComm = eComm;
    this.sens = sens;
    this.engSys = engSys;
    this.soil = soil;

    obs = new HashSet<>();
    command = "";
    temp = 0.0;
    vis = 0.0;
    light = 0.0;
    totalDist = 0.0;
}
```

4. Use of polymorphism - Observer Pattern

I have utilised the Observer Pattern to identify when a new command and any additional required information has been sent to the Rover. The `ApiData` class acts as the subject as it runs the event loop waiting for the following incoming data:

- updated/new command - `EarthComm.pollCommand()`
- updated environment values - `Sensors.readTemperature()`, `Sensors.readVisibility()`, `Sensors.readLightLevel()`
- updated total distance travelled - `EngineSystem.getDistanceDriver()`
- updated soil analysis results - `SoilAnalyser.pollAnalysis()`

The `Rover` class is then the observer, that relies on the above mentioned data, in order to operate. Within each event loop, the `notifyObservers()` method is called to update the `Rover` class with the newly acquired information.

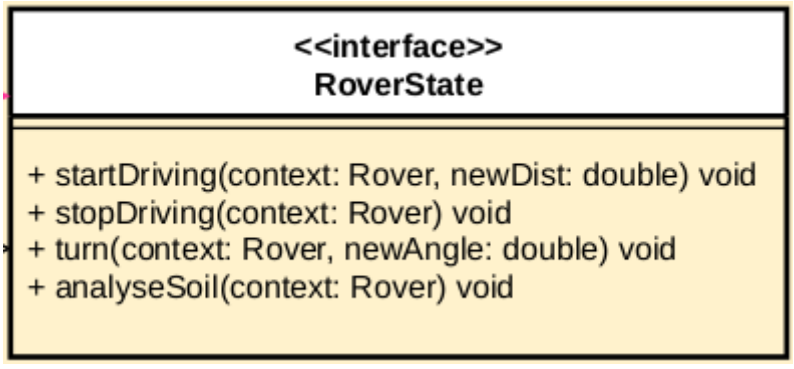
This implementation effectively decouples the `Rover` class from the `ApiData` class, as neither classes are concerned with how either of them operate. The `ApiData` class essentially acts as an intermediary for any incoming data to the `Rover` class from the external API classes.

5. Use of polymorphism - State Pattern

Given that there are specific conditions under which certain commands are invalid, such a "Drive" command while currently performing a soil analysis, it made sense to establish particular states that the rover would be in while executing it's actions. These states are as follows (please see `RoverState` interface:

- Stopped - where the rover is essentially idle
- Driving
- Analysing soil

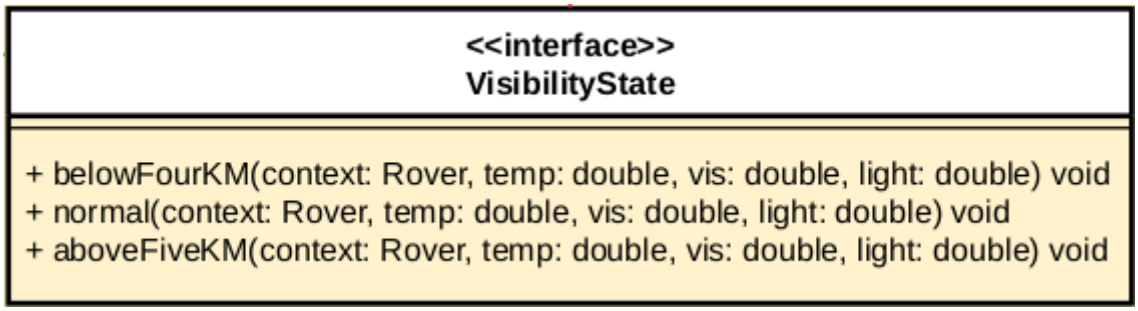
All actions that are dependent on the state of the rover are implemented as interface methods in the `RoverState` interface. These methods being:



Now each of these above mentioned actions, will be handled differently based on the state of the Rover. For example: The `analyseSoil()` method will send an error message if it is called while the Rover is currently in the `Driving` state.

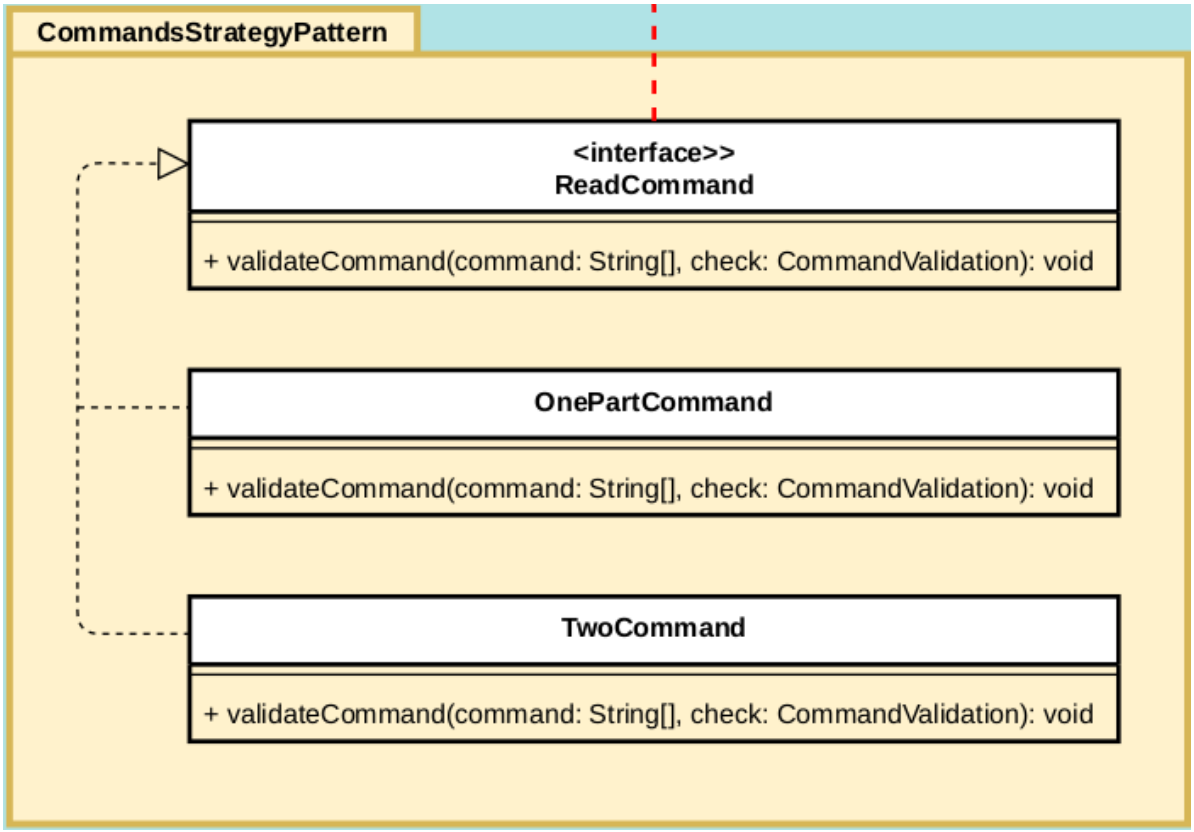
This implementation helped simplify the code within the `Rover` class and allow the `RoverState` implementing classes handle any validation logic.

I have also utilised the State Pattern to handle the automatic message alerting for when the visibility crosses the 4 km and 5 km thresholds. It is implemented in a similar manner as mentioned above.



Additional use of polymorphism - Strategy Pattern

I have also utilised the Strategy pattern to handle incoming commands. The `ReadCommand` interface is implemented by the `OnePartCommand` class and the `TwoPartCommand` class, which both override the `validateCommand()` method to execute code specific to each respective implementing class.



6. Clear and correct UML

The UML class diagram for this application clearly illustrates the separation of classes into their corresponding packages. Each of the design patterns I have implemented can be seen to be contained in their our individual packages. All inheritance hierarchies have been depicted using standard and consistent UML design conventions. The diagram has been updated to reflect the current version of the application, as at submission date.