Liam Pilling. 15520564

Part C:

1. The program is started by initialising the concrete classes of the Comm, Driver, SoilAnalyser and Camera, which are RoverComm, RoverDriver, RoverSoil and RoverCamera. It then creates the Controller class and gives it the initialised concrete classes. The Comm, Driver, SoilAnalysis and Camera classes have extremely basic implementations for testing purposes. The Controller class is passed to the RoverComm, RoverDriver, RoverSoil and RoverCamera classes as an observer.

2. The RoverComm class (concrete class of Comm) then receives a message sent by the test class. The message has the following format:
    - identifier,taskOne,taskTwo etc. where identified is an identifying integer and tasks are of the following format:
        i. Md, where d is a double between -100 and 100
        ii. Td, where d is a double between -180 and 180
        iii. P
        iv. S
        v. Li, where i is an integer that represents a previous task list identifier

    All the concrete classes have observers and these observers let the controller class know when certain methods are called. The controller class method newMessage() finally receives the message and parses it.

3. The jobs are then put into a queue and the queue is put into a map with its unique identifier. This list is then passed into the private method performTasks() by giving it the unique identifier.

4. The queue is extracted from the map using the unique identifier and the queue is iterated. As long as there is no mechanical error we keep iterating through the queue. If the program is busy (which means a task is executing) then we keep iterating until the program is no longer busy. If the program receives a Li command, then we call the performTasks() method again with the i value as the parameter. The program throws exceptions if the task is invalid, such as incorrect characters, the ranges are incorrect or the list to iterate over does not exist. If there is a mechanical error, then the method returns 0 and the list is abandoned. Other lists might still be executed after this as the specification says abandon all the tasks in the rest of this list, not other lists.

Part D:
The only design pattern that I have used is the observer pattern. Each of the concrete classes for the given classes have an observer, which is the controller. I used it because the controller class needs uses event driven programming, as certain events can only happen once a particular event has occurred. An example is the mechError() method which causes the program to quit its current list. Controller is observing constantly to see if there will be an error.

Part E:

- The program could use states rather than booleans. At the moment the program has boolean flags which represent if the controller is busy or has an error. This could be implemented as a state pattern where there could be three states that the program could be in: waiting, busy and error. I chose not to use this for simplicity and to make the system less complex in general.
- The driver could have had a template pattern implementation, where you would have two different drivers inheriting it, one for turning and one for moving. This would be useful for when you needed to be more specific on mechanical errors, which could say what caused the mechanical error, rather than saying that there is a mechanical error.

Part F:
The design achieves separation of concern by separating the functionalities of the program into the appropriate classes. The controller class is the only class in the program which performs the programs decision making, while all other classes exists only to interact with the controller. The system occasionally outputs what is happening but this is only for testing purposes. The functions are appropriately named and serve a purpose. For example, early in development the newMessage() function in the controller did all the parsing and performing of tasks, but the function was separated to become the performTasks() method, which is private and can only be called by newMessage().

Part G:

- The performTasks() method has a recursive call for completing tasks from other lists. When the program gets the Li task, it calls performTasks() and sends it the i value to get the specific task list.
- The Controller class implements four different interfaces. If you had one class for each interface implementation and then had those as fields inside the controller then you would get a lot of unnecessary code and four extra classes. Using multiple inheritance to implement four different interfaces allows the one Controller class to be notified of any events in the concrete classes.

Part H:
The controller class has dependency injection for it’s constructor. Instead of the controller class creating new variables for RoverComm, RoverDriver, RoverSoil and RoverCamera, it will take those variables in as parameters. This allows you to pass in RoverDriver which might cause a mechanical error and you can test your code for handling these errors.