This report is to justify the design for the covid app in regards with design principles, software code patterns and software architecture while mentioning all the changes from the previous assignment in terms of extending the design and refactoring.

Firstly, the design patterns used in this software is the strategy design pattern. This can be seen in the implementation for commands. In the strategy design pattern, additional classes are made to extend the functionality. This results in high extensibility as to extend a feature all we must do is just add the strategy which is extending the command class in the system. This can be seen in figure 1 below. The image below shows our applied strategy design pattern in command. We chose this strategy as this seems like the easiest and cleanest way to extend the design as we represent each functionality with different classes. Since this is the case, the strategy code pattern was chosen to solve this problem. However, there are also a few disadvantages to this design being applied in our software. This results in the high number of classes in our code, and this might make refactoring harder if there are hundreds of classes. However, since our code is modular, it makes isolating the logic in functionality easy as each of those classes have different logic and this makes refactoring easy as the bugs are isolated in each class. To show our design is extensible, I have shown the new functionalities by circling them in blue in figure 1. Like the strategy design pattern, all we had to do to extend is to add another class and add the logic inside it. This makes our design very extensible and easy to use, refactor and extend. This also results in our software obeying the Single Responsibility principle as each class is responsible for one feature and eliminates God classes in our software. Open closed principle is also followed as all we must do to extend is adding new code. Our code also obeys Liskov substitution principle as the command is replaceable with any of the subclasses. Finally, our code also follows the dependency inversion principle as all our code is dependent on abstract classes like the interface class, api caller class and the command class.

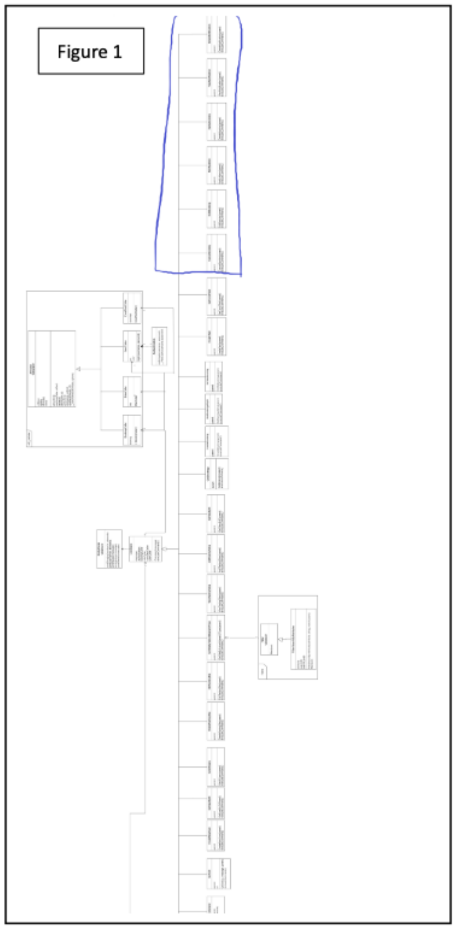
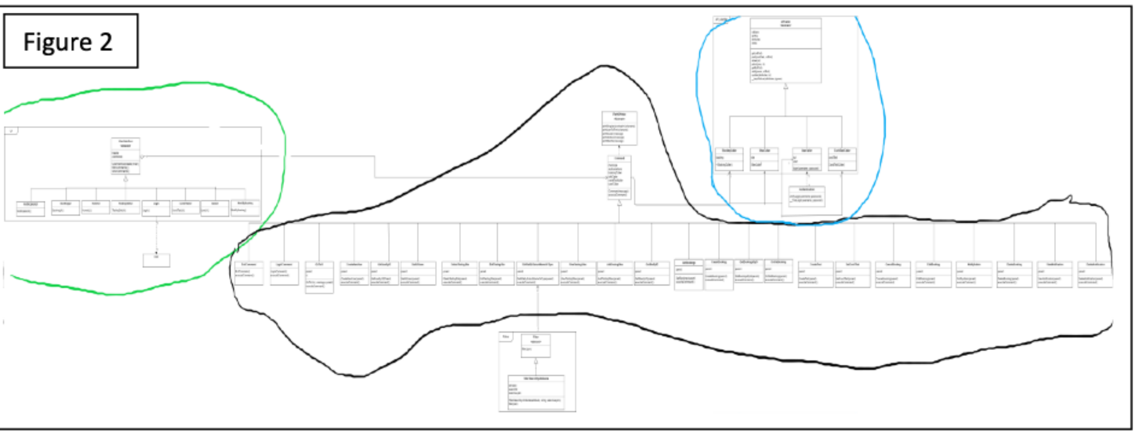


Figure 1

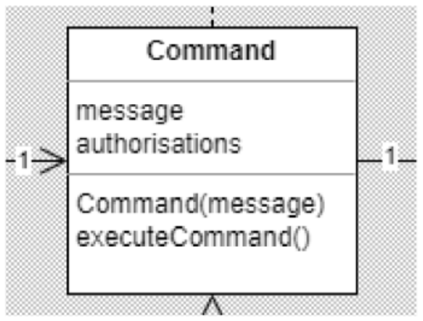
Secondly, we used the layered architecture as we decided on segregating the different components of the software. In our software, we have 3 layers. The display layer, the logic layer, and the database layer. This can be shown in figure 2 below with the display layer being circled in green, the logic layer in black and the database layer in blue.

Figure 2

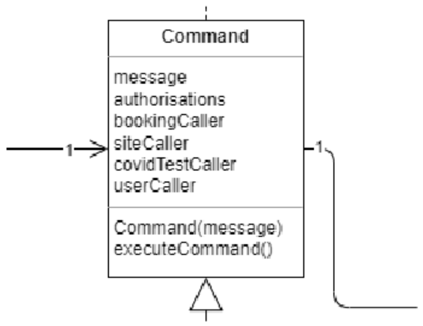


The green layer is the display layer as it is responsible for all the UI and display elements and the black layer is the logic layer as that is where most of the functionalities are and the blue layer is the database layer as it retrieves data from the database. The reason why we decided to use this software architecture is because it isolates each component of the software, and it makes testing easy. It also provides extensibility as we can add layers easily in the future if there is additional functionality that needs to be made. We also decided to choose this software architecture as we can see how the data flows easily and this makes it easy to understand and refactor or debug. No changes were made regarding software architecture from the previous assignment.

Finally, we refactored a few things as some of the system functionality was full of bugs and therefore, we had to refactor it. However, the major refactoring that we did do was using the API caller class as objects inside the command class. By doing this, we only need to instantiate the objects once and then it can be used by all the subclass for commands. This can be seen in the before and after pictures below. This majorly removes dependencies from the subclasses.



Before



After