Design Rationale

In order to extend our application to fulfill the additional requirements, various refactoring techniques were used to ensure that our code was extensible, scalable, and robust. We also adopted a Model-View-Controller (MVC) pattern in the structure of our code which helped to reduce cyclic dependencies.

During the design and development of this project, we ensured that the code adhered to the MVC pattern. This eliminated any cyclic dependencies which may have appeared, adhering to the Acyclic Dependency Principle. The application is run by the Driver class which instantiates EMR to handle all server calls. All interactions between the application and the server are handled in EMR. The HealthMonitor is the main page in the application from which all other pages are accessed. This means that the HealthMonitor requires functionality to interact with the user. Because of this, it was placed in the controller package despite being a JFrame form that displays data. This design decision improved the flow of the application, making it more readable, robust and extensible. To facilitate this decision, a Panels interface was included between the HealthMonitor and all other JFrame pages. This interface allowed the Dependency Inversion Principle (DIP) to be maintained as the lower-level JFrame classes depend on the interface rather than HealthMonitor.

The controller package contains a Server package with EMR and ServerJSON. If the server changes, the only classes which would be affected by this are EMR and ServerJSON. The Common Closure Principle states that classes that are likely to change together, should be grouped together, so to adhere to this principle, EMR and ServerJSON were grouped in a Server package.

The View package contains all of the JFrame form classes and the Graph package. These classes all display the data which the View package receives from the Controller package. This data is stored in and processed by classes in the Model package.

The Model package contains an Address package and a Vitals package. The Vitals package has many incoming dependencies yet no outgoing dependencies. This means the package is very stable and thus the decision to make biologicalParameter an abstract class adheres to the Stable Abstractions Principle while making biologicalParameter easier to extend. The Model package is a passive model as it only receives information from the server via EMR in the Controller package.

Converting our code to adhere to the MVC pattern required very little modification. In assignment 2 we had already isolated the classes which stored and handled data into the Model package. Additionally, only the EMR and ServerJSON classes accessed the server which was not changed. This demonstrates that our code from assignment 2 was well designed, robust and extensible.

The bridge design pattern was used in the implementation of the Graph package for requirements 1 and 5. The design pattern was used to separate the abstraction of the data being graphed and the implementation of which graph is shown. GraphDataExchange is the abstraction which the client (HealthMonitor) provides the data to and uses to request the appropriate graph. ChartConnector is the interface that defines the graphical implementation. BarChart and LineChart are the two concrete implementations of the ChartConnector interface. The Graph Enum was also added to restrict the graph choice to only graphs which had an implementation, making the code more robust. The use of this design pattern ensures that the code is extensible and scalable. This design pattern utilises the DIP which states that abstractions should not depend on details. By including an interface between the abstraction and implementation, neither depend on each other but instead on the interface. BarChart was implemented first along with ChartConnector, Graph, and GraphDataExchange. This meant that the work required to extend the design to include LineChart was minimal.

The Single Responsibility Principle was also adhered to in the design of the application. The Render classes and RefreshDataChart were created from methods that could have been within HealthMonitor and GraphDataExchange. By employing the Extract Class refactoring technique, we were able to simplify the HealthMonitor and GraphDataExchange classes which made the overall design more readable and robust.

In order to implement requirement 6, a Statistics class and JFrame form were created. As the HealthMonitor class and JFrame form already existed, the implementation of the Statistics class was a fairly straightforward extension due to the highly flexible design from assignment 2.

One of the refactoring techniques utilised in this project was the change signature technique. The getMeasurement method in SmokingStatus originally returned an Object containing a boolean value. This was refactored to return a Boolean. This modification allowed the output to be tested without being first casted to a Boolean which resulted in the code being harder to break and therefore more robust.

In the Controller package, the HealthMonitor was updating data and controlling the UI features. Using the refactoring technique Move Method, this functionality was moved to a different class. This allowed the Patient class to be updated by classes in the View package without involving the HealthMonitor.

The refactoring technique Splitting The Function was also used in the creation of the TimeMonitor class. This class was originally a method within HealthMonitor.

Rename Method was another refactoring technique used within the development of this application.

The refactoring techniques we employed in the development of the code were necessary to ensure that the final product was robust and extensible. The minimal use of these techniques demonstrates that the existing code base from assignment 2 was well designed and implemented.