Design Rationale

Our application was solely built around the idea that our application needs to be **extensible, simple to use, and easy to maintain**. We have applied in all our classes the **Single Responsibility Principle, Open/Closed Principle and the Liskov Substitution Principle.**

We have divided our classes into 5 different packages: model, view, controller, observerpattern and driver. From the **package and class perspective**, our model package depends only on the observerpattern package to use the ObservableHashMap. Our controller package depends on the observerpattern package, the model package and the view package. The view package depends only on the model package. There are no cyclic dependencies as we obey the **Acyclic Dependencies Principle** (ADP). Additionally, we've also complied with the **Common Reuse Principle** (CRP) and the **Common Closure Principle** (CCP). For example, our observerpattern package is highly cohesive, reusable and maintainable as we found a balance between **package stability** and its **abstractness** (**SDP** and **SAP**). Furthermore, The Observer class serves as a hinge in the design, which allows for **extensibility.** The Observer Pattern also uses the **Open close Principle** and the **Liskov Substitution Principle** which means that the code can be easily extended without any major change to any previous code. [1]
After complying with all of the **package cohesion principles** and **package coupling principles**, we found out that we have implemented the **active MVC model**.

The reasons the MVC architecture suits our application are as follows:
· By splitting the system into 3 components to store, update and display data, it is easy to test these components separately. This also means that simultaneous changes could be made to these layers because they are separated.
· The components in the architecture have **a low dependency**; meaning that they are easy to maintain and extend without affecting other system components.
· The model is independent of the View, which means that adding new Views to the system would not impact the Model.
· This also provides for **reusability of code**, where different views can use a Model.[2]

Additionally, we used the Observer pattern because:
· It is very easy to add new observers for an observable/ subject
· It supports **loose coupling** between the components that interact with one another
. We have a time-based trigger to update our data.
· No modification to the code is necessary to accommodate new observers [3]

Zooming in to our implementation details, we have put **a lot of consideration** into making sure that we do not overload the servers with GET Requests. We looped through each Encounter of the practitioner to get all the patients of the practitioner and we realized that there are many duplicate patient resource ids. We also found that different resource ids could

refer to the same person. With these insights, we used **3 HashMap/ ObservableHashMap(s).** First, we used the resource ID of a Practitioner to get the practitioner's identifier and then used the identifiers to find all the encounters of the practitioner, get the resource ids of the patients and placed them in a HashMap to get the **unique resource ids**. From there, we used **GET Request** to get the identifiers of the patients, placed them into another HashMap to get **unique identifiers** of patients. Only after that did we use **another** GET Request to obtain the latest total cholesterol data of the Patient. Doing this, we reduced the time it takes to complete the process **significantly**. We reduced the time it takes to get all the patients of a practitioner and the data of the patients required from **2 minutes** to **less than 20 secs** for some of the practitioners that we have tested**.** ObservableHashMap is an **Observable** that **delegates** its tasks to a normal HashMap. The difference between an ObservableHashMap and a HashMap is that after any tasks that change the data in the ObservableHashMap such as put, remove, or change; we call **notifyDatasetChanged()** to notify all **subscribed Observers** about the change that was made. This, however, led to the problem of **having too many observers.update() being called** as each **put()** caused a **notifyDatasetChanged().** The solution we came up with was to add 2 new methods to the **ObservableHashMap class.** The two new methods are **putAllThenNotify(HashMap)** and **putAllThenNotify(ObservableHashMap)**. They take in another HashMap/ObservableHashMap and only **call** notifyDatasetChanged() **after all the entries are added** to the current HashMap so the View does not get flooded with update() messages and lag.

We have also made sure our app stays responsive for users by fetching subsequent data from the server **asynchronously.** We **retrieve data on a new thread** to ensure that the user can continue working with the GUI while the App is fetching data.

References:

[1] *Design Patterns 1*. Monash University, 2020.
[2]"6 Advantages of using MVC model for effective web application development", *Brainvire*, 2020. [Online]. Available: https://www.brainvire.com/six-benefits-of-using-mvc-model-for-effective-web-application-development/. [Accessed: 17- May- 2020].
[3]"Learning Python Design Patterns", *Subscription.packtpub.com*, 2020. [Online]. Available: https://subscription.packtpub.com/book/application_development/9781785888038/6/ch06lvl1sec50/the-observer-pattern-advantages-and-disadvantages. [Accessed: 17- May- 2020].

FIT3077 Assignment 2
Done by Zhi Hao Tan and Sriram Viswanathan