Design Rationale

The new required functionality meant that we had to make significant changes to our app. With the previous UI library we used, Atlastk, there was no way to add graphs or images to the UI. As such, we transitioned to Tkinter and essentially redid our whole UI. However, this also provided us with an opportunity to transition to an MVC architecture, as Atlastk library has unavoidable strong coupling between View and ViewController, while Tkinter doesn’t have such constraint. We chose to convert the design pattern to MVC in order to separate the concerns and break down different aspects in the system (input logic, business logic and UI logic). By minimizing the coupling between system components, this provides us a change to manage the complexity of the system.

In using MVC, we also follow the Acyclic Dependencies Principle. This means that the View and the Controller can be extended without modifying the Model, which facilitates future app extensions. Additionally, we follow the Common Reuse Principle by putting all classes that deal with business logic in the Model package. These classes are interdependent and would most likely be used together if they were to be reused in another app. In this scenario, the new app could just depend on the data and logical structures in the Model without having to depend on the View and ViewController classes as well.

Our initial design used a MonitoringList class which stored Patients who in turn each stored one Encounter subclass (analogous to an Observation in FHIR) containing the measurement and its date. DIP was used here since both Patient and the Encounter subclasses depended on the abstract Encounter class, meaning MonitoringList can store any Patient with any measurement. This was useful when we extended our app to monitor blood pressure and had the benefit of keeping all the monitors independent. However, the trade off of this design is that Patient information (name, id, etc.) is stored in multiple monitors if they are being monitored for multiple measurements.

The previous MonitoringList also calculated the average of the measurements of the monitored patients, which was not needed for blood pressure. As such, we used the ‘Extract Subclass’ refactoring method to move the average calculation to a MonitoringListAverage subclass which inherits from a simplified and more generalised MonitoringList. This makes the way in which blood pressure is monitored simpler, cleaner and more efficient.

Additionally, we made the decision to keep MonitoringList as a way to store just one type measurement for each Patient rather than two. The drawback of this was that we needed two calls to the server to monitor both types of blood pressure, as well as two monitors for each type of blood pressure. However, this choice had the benefit of keeping the types of blood pressure independent, which is useful in the (admittedly unlikely) event a Patient only has Systolic but not Diastolic measurements. More importantly, it allows the user to choose to monitor only one type of blood pressure, which is the case for the historic monitor, without having to also monitor the other type at the same time.

For the historic monitor, we refactored Patient to store a list of Encounters rather than just one Encounter. Now, MonitoringLists all have a constant that defines how many historic values to monitor (a non-historic MonitoringList has this value initialised to 1). This refactor made MonitoringList more generalised. However, it did require us to make changes to the implementation and interface of WebServiceManager, e.g. Webservicemanger.fetchEncounter() takes an additional ‘num’ parameter that specifies the number of measurements to fetch. These interface changes lead to changes in any class that uses WebServiceManager. Despite this disadvantage, we decided that it was preferable to having to design new measurement storage and monitoring classes that allow for multiple types of measurement. From our analysis of the server, blood pressure is the only type of Observation that has two values. Changing our app to allow storage of multiple measurements in the same place would hence be unlikely to improve extensibility.

Reference

sourcemaking.com. (n.d.). *Design Patterns and Refactoring.* [online] Available at: https://sourcemaking.com/refactoring/extract-subclass [Accessed 18 Jun. 2020.