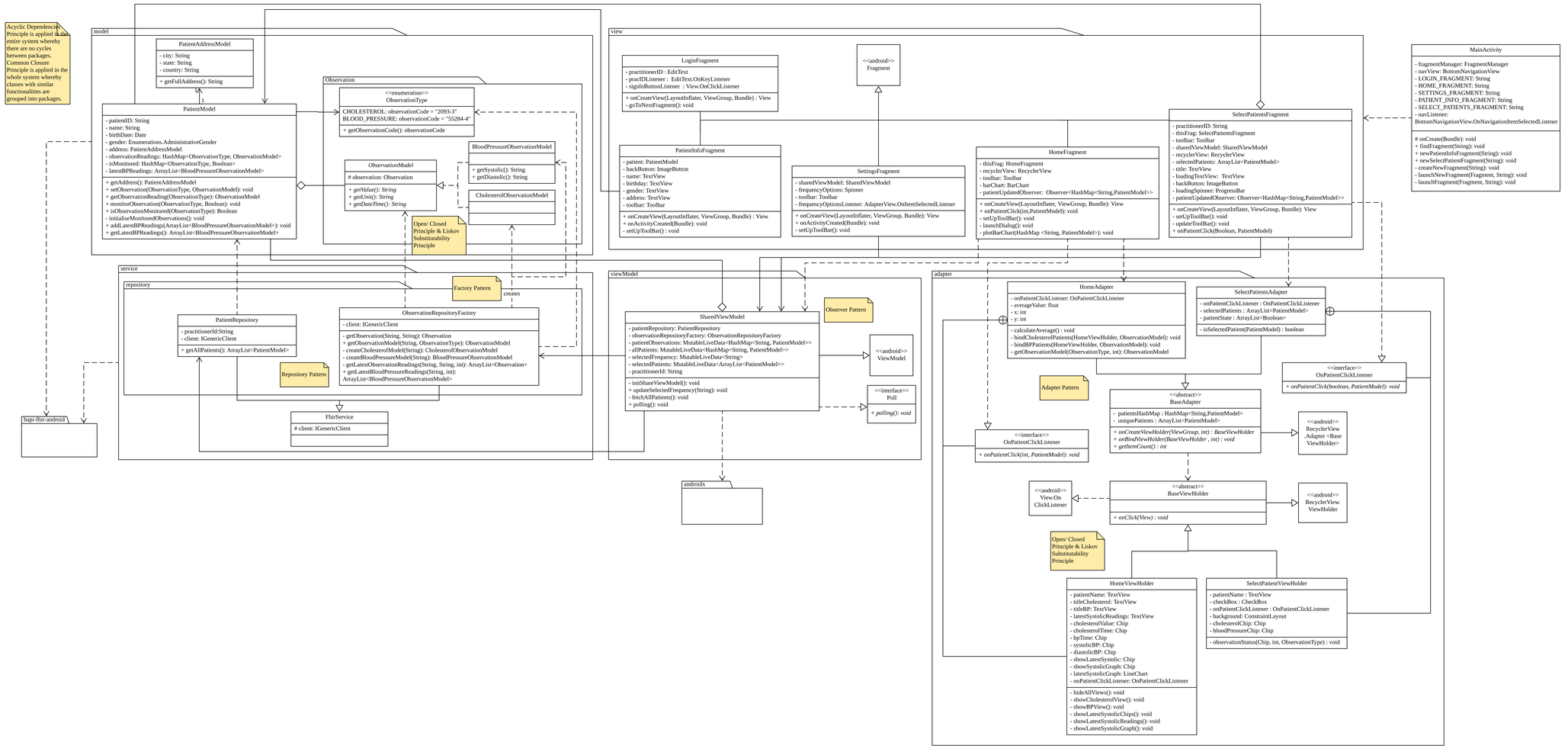
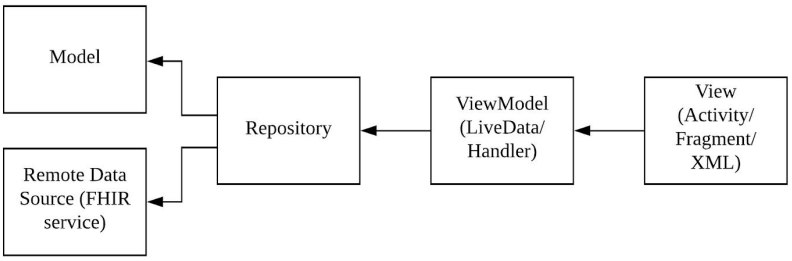
KMHC Class Diagram
Team Kimchi

by Megan Ooi Jie Yi (30101670) & Hew Ye Zea (29035546)



FIT3077 Assignment 3
Design rationale
Written by: Megan Ooi Jie Yi (30101670) & Hew Ye Zea (29035546)



Based on the **MVVM (Model-View-ViewModel) architecture** that we have implemented in our system, our system follows the **Acyclic Dependencies Principle** as the dependencies between packages do not form cycles[1]. Our system also applies the **Extracted UI approach** whereby the View/ UI and Model are placed in separate packages so that our system obeys the **Single Responsibility Principle** and also ensures that the View knows about the Model but not vice versa. By following these principles, we can work on our own assigned tasks in the frontend and backend without impacting each other's work as dependencies are easier to understand and maintain and the frontend and backend are clearly separated.

Common Closure Principle
Classes with similar functionalities are grouped together in packages[1]. For example, all the views and model classes are grouped together in their respective packages. This is beneficial to our system because when a change occurs, only the components in the same package are affected but not components in other packages. This increases maintainability because we can reduce the number of packages affected by changes and reduce the number of times we rebuild and retest our system.

How our initial design supported new requirements
Factory pattern
The factory pattern is used in the ObservationRepositoryFactory class whereby the class is responsible for getting data from the server and packaging the data to their own respective class based on the observation type. Since we are required to get blood pressure readings from the server, we are only required to create a new method which creates the BloodPressureObservationModel object. This makes maintenance easier as the subclasses of ObservationModel are all instantiated in the same class.

Open/ Closed Principle
Since we are required to represent the blood pressure data in our system, we had to create a subclass, BloodPressureObservationModel which inherits from the ObservationModel class and add its suitable methods. This follows the Open/Closed Principle as the superclass is not modified but its subclass is created and open for extension. This is beneficial as it provides a layer of abstraction and ensures **loose coupling**.

Observer Pattern
The SharedViewModel class already contains a method which polls the server to get observation readings which uses the **Observer** pattern. The HomeFragment acts as an observer and subscribes to the LiveData object so that it is notified of change. When new readings are obtained from the server, the HomeFragment simply has to refresh its view to display the latest data to the user. We just had to refactor the method to loop through all different observation types in our system and poll the server to get the required data. This makes sure that our view is always showing the latest data whenever new data is obtained from the server. The advantage of this approach is that classes that are dependent on the 'Subject' are updated automatically and are allowed to provide their own implementation when they are notified of change[2].

Refactoring
Rename Method - Some methods have been renamed to add more clarity to its functionality.
Pull Up Field - ObservationModel has been refactored to contain the field observation which is shared by its subclasses.

FIT3077 Assignment 3
Design rationale
Written by: Megan Ooi Jie Yi (30101670) & Hew Ye Zea (29035546)

[1] Robert C. M. (2000). Design Principles and Design Patterns. Retrieved from Design Principles and Design Patterns
[2] Android Open Source Project. (n.d.). LiveData Overview. Retrieved from LiveData Overview