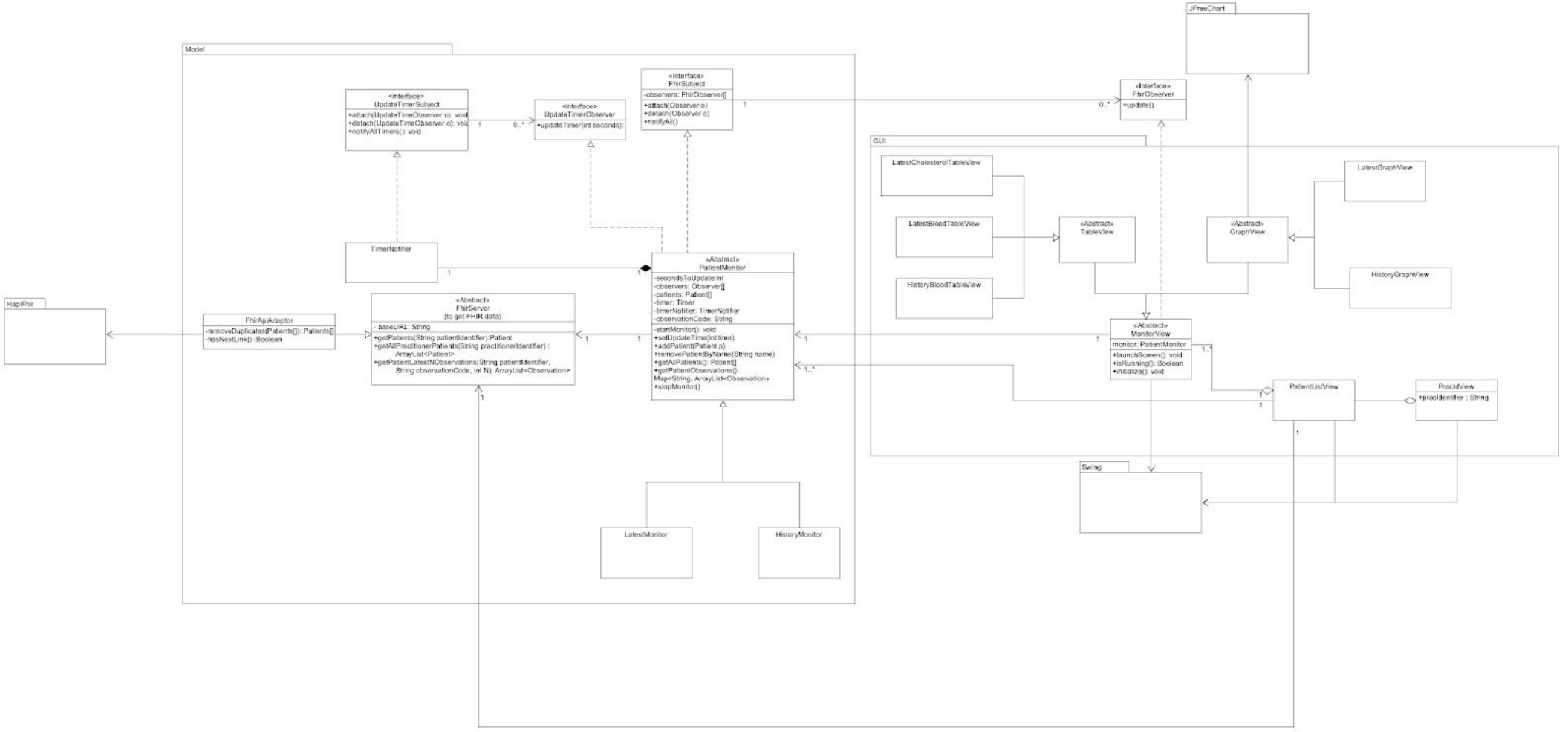
FIT3077 Assignment 3 : UML Class Diagram

Team Indomie: Richard Pranjatno and Jason Agustino



FIT 3077 A3 Design Rationale

Previously, we built the design of our system around the observer design pattern, where the cholesterol table receives an update from the subject, and changes its own state with the received information. In this current design, we implemented the Model View Controller design pattern, particularly the active Model variant. This variant of MVC still utilizes the observer design pattern to update the View whenever the Model changes. It should be noted that since we are using Java Swing for our GUI, and the package itself combines View and Controller (i.e. both are tightly coupled), our class diagram does not look like the “traditional” MVC representation. Instead, we have represented both the View and Controller components into our GUI package and its corresponding classes.

Using the ideas of MVC, PatientListView acts as a “controller” for the MonitorView (View) and PatientMonitor (Model), in which it passes single instances of the PatientMonitor to its corresponding MonitorView. This class is the only class that will send requests to the PatientMonitor, such as adding/removing patients from the monitor, and toggling a monitor on/off, which in return changes the display in our views. The advantage of this is that the MonitorView classes have single responsibility only, which is to view and display the model in its own interface. The PatientMonitor is independent of the MonitorView, and a modification in the MonitorView will not alter the Model. Implementing the MVC framework also allows us to create multiple views for a particular Model. For example, in our previous assignment stage we had a PatientCholesterolMonitor that displays a single PatientCholesterolTable (View) class. With this new design, we can now create multiple views that display the single instance of that Model, and in our case we created a CholesterolGraphView to view the model in a graphical representation.

We have also separated the TableViews for BloodPressureMonitor and CholesterolMonitor instead of displaying it in a single view. Our reason for this is that we wanted to keep a one to many relationship between the models and views, where a single view can only display a single monitor and a single monitor can have multiple views attached to it. An advantage of this is that it allows for easier extensions and clear separation for each view. This means that when a new Model (such as LatestBloodMonitor and HistoryBloodMonitor) is created, existing views such as CholesterolTable and CholesterolGraph do not have to be modified, and the only thing required is to create the required views for this model, while keeping the one model to many views design. It may create complications for extensions if we added everything into one view, which creates more dependencies within the views and is harder to manage for additional implementations. However, a disadvantage of this design is that it requires for the practitioner to open multiple windows of view, when monitoring a patient’s observation. Since the table and graph representations are similar with different model displays, we decided to create abstract classes TableView and GraphView for it, to avoid repetitions and easier modification for all the child views.

We had also refactored some of the class names and class methods as the requirements have changed and thus requires more appropriate namings. Some of the duplicated methods which are common between classes due to the change in requirements has also been refactored, by creating suitable abstract classes or interfaces (pull up method), which also increases the extensibility of the design. The classes were also organized into packages, GUI package for classes dealing with views, and model package for classes dealing with the server and subject models. We used packages to introduce further encapsulation of related concepts, and to show a clearer connection between classes to write better maintainable code. Package-level design considerations are also

introduced, such as the Common Closure Principle (GUI package classes are likely to change together, and same for the model package classes), and the Acyclic Dependencies Principle (no cycles between packages, so no cyclic dependencies and continuous code change).