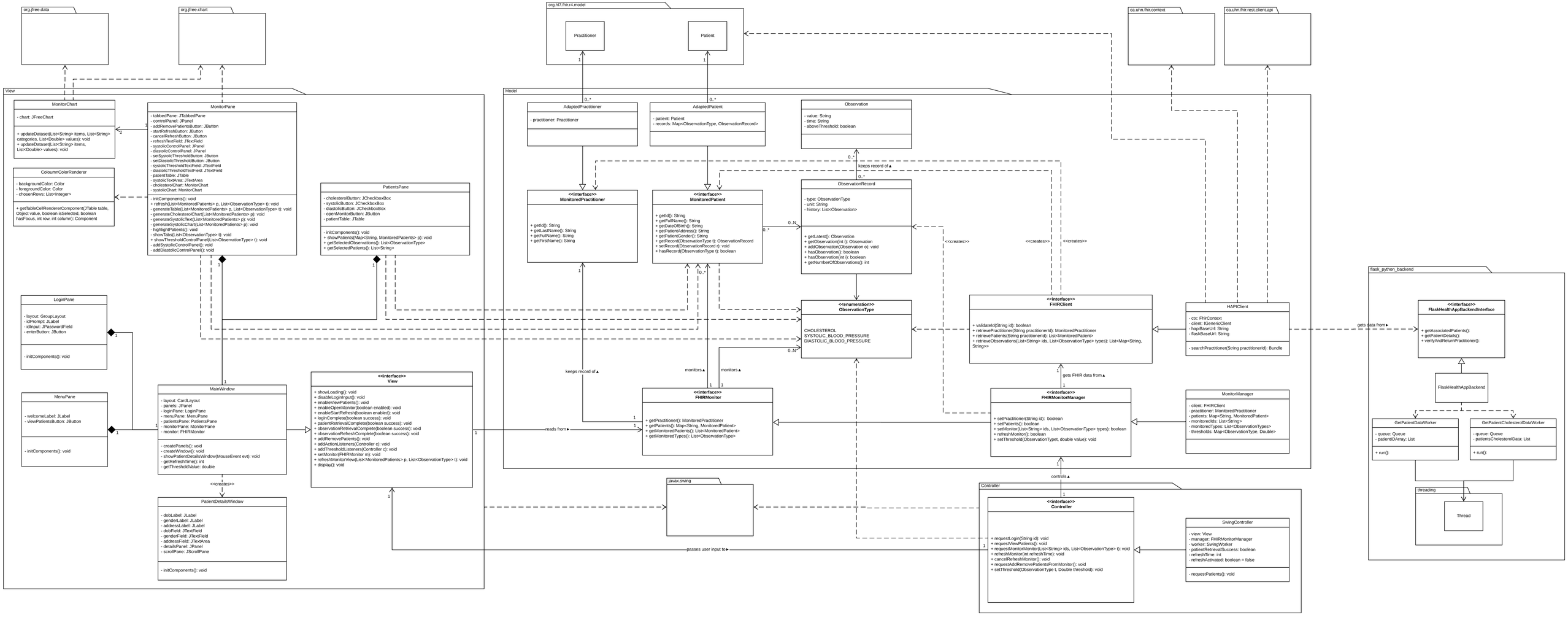
org.jfree.data
org.jfree.chart
org.hl7.fhir.r4.model
Practitioner
Patient
ca.uhn.fhir.context
ca.uhn.fhir.rest.client.api
View
MonitorChart
- chart: JFreeChart
+ updateDataset(List<String> items, List<String> categories, List<Double> values): void
+ updateDataset(List<String> items, List<Double> values): void
MonitorPane
- tabbedPane: JTabbedPane
- controlPanel: JPanel
- addRemovePatientsButton: JButton
- startRefreshButton: JButton
- cancelRefreshButton: JButton
- refreshTextField: JTextField
- systolicControlPanel: JPanel
- diastolicControlPanel: JPanel
- setSystolicThresholdButton: JButton
- setDiastolicThresholdButton: JButton
- systolicThresholdTextField: JTextField
- diastolicThresholdTextField: JTextField
- patientTable: JTable
- systolicTextArea: JTextArea
- cholesterolChart: MonitorChart
- systolicChart: MonitorChart
- initComponents(): void
+ refresh(List<MonitoredPatients> p, List<ObservationType> t): void
+ generateTable(List<MonitoredPatients> p, List<ObservationType> t): void
+ generateCholesterolChart(List<MonitoredPatients> p): void
+ generateSystolicText(List<MonitoredPatients> p): void
+ generateSystolicChart(List<MonitoredPatients> p): void
- highlightPatients(): void
- showTabs(List<ObservationType> t): void
+ showThresholdControlPanel(List<ObservationType> t): void
- addSystolicControlPanel(): void
- addDiastolicControlPanel(): void
ColoumnColorRenderer
- backgroundColor: Color
- foregroundColor: Color
- chosenRows: List<Integer>
+ getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column): Component
PatientsPane
- cholesterolButton: JCheckboxBox
- systolicButton: JCheckboxBox
- diastolicButton: JCheckboxBox
- openMonitorButton: JButton
- patientTable: JTable
- initComponents(): void
+ showPatients(Map<String, MonitoredPatients> p): void
+ getSelectedObservations(): List<ObservationType>
+ getSelectedPatients(): List<String>
LoginPane
- layout: GroupLayout
- idPrompt: JLabel
- idInput: JPasswordField
- enterButton: JButton
- initComponents(): void
MenuPane
- welcomeLabel: JLabel
- viewPatientsButton: JButton
- initComponents(): void
MainWindow
- layout: CardLayout
- panels: JPanel
- loginPane: LoginPane
- menuPane: MenuPane
- patientsPane: PatientsPane
- monitorPane: MonitorPane
- monitor: FHIRMonitor
- createPanels(): void
- createWindow(): void
- showPatientDetailsWindow(MouseEvent evt): void
- getRefreshTime(): int
- getThresholdValue: double
<<creates>>
PatientDetailsWindow
- dobLabel: JLabel
- genderLabel: JLabel
- addressLabel: JLabel
- dobField: JTextField
- genderField: JTextField
- addressField: JTextArea
- detailsPanel: JPanel
- scrollPane: JScrollPane
- initComponents(): void
<<interface>>
View
+ showLoading(): void
+ disableLoginInput(): void
+ enableViewPatients(): void
+ enableOpenMonitor(boolean enabled): void
+ enableStartRefresh(boolean enabled): void
+ loginComplete(boolean success): void
+ patientRetrievalComplete(boolean success): void
+ observationRetrievalComplete(boolean success): void
+ observationRefreshComplete(boolean success): void
+ addRemovePatients(): void
+ addActionListeners(Controller c): void
+ addThresholdListeners(Controller c): void
+ setMonitor(FHIRMonitor m): void
+ refreshMonitorView(List<MonitoredPatients> p, List<ObservationType> t): void
+ display(): void
reads from ▸
keeps record of ▲
Model
AdaptedPractitioner
- practitioner: Practitioner
AdaptedPatient
- patient: Patient
- records: Map<ObservationType, ObservationRecord>
Observation
- value: String
- time: String
- aboveThreshold: boolean
keeps record of ▲
<<interface>>
MonitoredPractitioner
+ getId(): String
+ getLastName(): String
+ getFullName(): String
+ getFirshName(): String
<<interface>>
MonitoredPatient
+ getId(): String
+ getFullName(): String
+ getDateOfBirth(): String
+ getPatientAddress(): String
+ getPatientGender(): String
+ getRecord(ObservationType t): ObservationRecord
+ setRecord(ObservationRecord r): void
+ hasRecord(ObservationType t): boolean
ObservationRecord
- type: ObservationType
- unit: String
- history: List<Observation>
+ getLatest(): Observation
+ getObservation(int i): Observation
+ addObservation(Observation o): void
+ hasObservation(): boolean
+ hasObservation(int i): boolean
+ getNumberOfObservations(): int
<<enumeration>>
ObservationType
CHOLESTEROL
SYSTOLIC_BLOOD_PRESSURE
DIASTOLIC_BLOOD_PRESSURE
<<creates>>
<<creates>>
<<creates>>
<<interface>>
FHIRClient
+ validateId(String id): boolean
+ retrievePractitioner(String practitionerId): MonitoredPractitioner
+ retrievePatients(String practitionerId): List<MonitoredPatient>
+ retrieveObservations(List<String> ids, List<ObservationType> types): List<Map<String, String>>
HAPIClient
- ctx: FhirContext
- client: IGenericClient
- hapiBaseUrl: String
- flaskBaseUrl: String
- searchPractitioner(String practitionerId): Bundle
gets data from ▸
<<interface>>
FHIRMonitor
+ getPractitioner(): MonitoredPractitioner
+ getPatients(): Map<String, MonitoredPatient>
+ getMonitoredPatients(): List<MonitoredPatient>
+ getMonitoredTypes(): List<ObservationType>
monitors ▲
monitors ▲
gets FHIR data from ▲
<<interface>>
FHIRMonitorManager
+ setPractitioner(String id): boolean
+ setPatients(): boolean
+ setMonitor(List<String> ids, List<ObservationType> types): boolean
+ refreshMonitor(): boolean
+ setThreshold(ObservationTypet, double value): void
MonitorManager
- client: FHIRClient
- practitioner: MonitoredPractitioner
- patients: Map<String, MonitoredPatient>
- monitoredIds: List<String>
- monitoredTypes: List<ObservationTypes>
- thresholds: Map<ObservationType, Double>
controls ▲
javax.swing
Controller
<<interface>>
Controller
+ requestLogin(String id): void
+ requestViewPatients(): void
+ requestMonitorMonitor(List<String> ids, List<ObservationType> t): void
+ refreshMonitor(int refreshTime): void
+ cancelRefreshMonitor(): void
+ requestAddRemovePatientsFromMonitor(): void
+ setThreshold(ObservationType t, Double threshold): void
SwingController
- view: View
- manager: FHIRMonitorManager
- worker: SwingWorker
- patientRetrievalSuccess: boolean
- refreshTime: int
- refreshActivated: boolean = false
- requestPatients(): void
passes user input to ▸
flask_python_backend
<<interface>>
FlaskHealthAppBackendInterface
+ getAssociatedPatients():
+ getPatientDetails():
+ verifyAndReturnPractitioner():
FlaskHealthAppBackend
GetPatientDataWorker
- queue: Queue
- patientIDArray: List
+ run():
GetPatientCholesterolDataWorker
- queue: Queue
- patientsCholesterolData: List
+ run():
threading
Thread

Design Patterns

Model-View-Controller and Code Extensibility

The extensibility of the system was hinged heavily upon the Model-View-Controller (MVC) architecture that was already implemented. This architecture made it possible to easily break down the implementation into sections to add to either the model, view or controller, allowing the entire system to be utilised. Furthermore, refactoring could be more focused to the specific section that needed refactoring - this is evident through the vast majority of the changes only being done on the View section Monitor Pane since the majority of the features required us to mainly change the visual aspect of the system.The contractual and encapsulative nature of such architecture meant that the functionality of each feature could be split, which was the case for highlighting patients, as the model handled whether each patient was above the threshold for each observation whilst the View was tasked with just highlighting the values on the table. Another example was when the historical data of the systolic observation needed to be known, this only required that the Model section to change without the need to modify extensively either the View or Controller.

Design Principles

Our design in assignment 2 relied heavily upon the **Dependency Inversion Principle (DIP)** which decoupled classes around several ‘hinge points’ in the system. It allowed the functionality for assignment 3 to be implemented by only modifying a small number of classes which required it (e.g. MonitorPane, MonitorManager, ObservationRecord etc.) without worrying about breaking the rest of the system because classes were mostly dependent on *interfaces*.

We also made use of the **Liskov Substitution Principle (LSP)** which has also carried over from our design in assignment 2. It was used primarily to give the View read-only access to the Model’s data via getter methods without exposing its setter methods which are meant for the Controller. This was achieved by storing a reference to FHIRMonitor in the View (which contained only getters), and FHIRMonitorManager in the Controller (which extends FHIRMonitor with setters). Only a single implementation of these interfaces was required (MonitorManager) because anything that implements FHIRMonitorManager also conforms to FHIRMonitor making it a true subtype in accordance with LSP.

One of the package cohesion design principles we applied was **Common-Closure Principle (CCP)** which ensured that any changes to the system were contained within packages. This was achieved by packaging the most highly coupled classes together (e.g. UI components in View and FHIR related classes in the Model) as they are most likely to change together.

The **Acyclic Dependencies Principle (ADP)** was mentioned in the design rationale for assignment 2 and still applies after the changes made for assignment 3. The View and Controller packages depend on the Model which in turn does not depend on any internal packages. This decouples the Model and allows it to be modified independently.

Refactoring Techniques

One of the main refactoring techniques applied for assignment 3 was moving the related methods which calculated whether Observation values were above a certain threshold from the MonitorPane (in the View) to the MonitorManager (in the Model). Not only did this conceptually make more sense and strengthened the cohesion of the respective classes, it also made the code more extensible. It allowed the threshold values to be calculated differently for multiple ObservationType (e.g. cholesterol threshold by average and blood pressure by user input).

Another refactoring technique was to rename some of the classes and their associated methods to better reflect their purpose. For example, “ReadOnlyFhirEntityManager” became “FHIRMonitor” as the former is too verbose. This makes the code more maintainable.