org.hl7.fhir.r4.model
Practitioner
Patient
1
1
Model
ca.uhn.fhir
0..*
0..*
PractitionerAdapter
- practitioner: Practitioner
PatientAdapter
- patient: Patient
- healthData: Map<Condition, HealthData>
<<interface>>
PractitionerInterface
+ getId(): String
+ getLastName(): String
+ getFullName(): String
+ getFirstName(): String
<<interface>>
PatientInterface
+ getHealthData(Condition c): HealthData
+ setHealthData(Condition c, HealthData data): void
+ getFullName(): String
+ getDateOfBirth(): String
+ getPatientAddress(): String
+ getPatientGender(): String
0..*
0..n
HealthData
- value: String
- unit: String
- timeUpdated: String
+ getValue(): String
+ getUnit(): String
+ getTimeUpdated(): String
<<interface>>
FhirClientInterface
+ validateId(String id): boolean
+ retrievePractitioner(String id): PractitionerInterface
+ retrieveAllPatients(String id): Map<String, PatientInterface>
+ retrievePatientDataForCondition(List<String> ids, Condition c): Map<String, HealthData>
JSONReader
- map: Map<String, Object>
- jsonObject: JSONObject
+ getJsonObject(): JSONObject
+ getMap(): Map<String, Object>
FhirClient
- ctx: FhirContext
- client: IGenericClient
- monashBaseUrl: String
- flaskBaseUrl: String
- searchPractitioner(String id): Bundle
gets data from ▸
<<interface>>
FlaskHealthAppBackendInterface
+ getAssociatedPatients():
+ getPatientDetails():
+ verifyAndReturnPractitioner():
1
0..*
1
gets data from ▴
1
manages ▴
manages ▴
1
<<interface>>
ReadOnlyFhirEntityManager
+ getPractitioner(): PractitionerInterface
+ getAllPatients(): Map<String, PatientInterface>
+ getPatientById(String id): void
+ getSelectPatientsWithCondition(List<String> ids, Condition c): Map<String, PatientInterface>
<<interface>>
FhirEntityManager
+ setPractitioner(String id): boolean
+ setPatients(): boolean
+ setUpMonitor(List<String> ids, Condition c): boolean
+ refreshMonitor(): boolean
MonitorModel
- client: FhirClientInterface
- practitioner: PractitionerInterface
- patients: Map<String, PatientInterface>
- monitoredPatients: List<String>
- monitoredCondition: Condition
FlaskHealthAppBackend
<<enumeration>>
Condition
CHOLESTEROL
1
1
<<creates>>
reads ▴
controls ▴
Driver
<<creates>>
<<creates>>
<<creates>>
GetPatientDataWorker
- queue: Queue
- patientIDArray: List
+ run():
GetPatientCholesterolDataWorker
- queue: Queue
- patientsCholesterolData: List
+ run():
View
MonitorPane
- patientTable: JTable
- scrollPane: JScrollPane
- addRemovePatientsButton: JButton
- startRefreshButton: JButton
- cancelRefreshButton: JButton
- refreshTextField: JTextField
- condition: Condition
- initComponents(): void
- highlightPatients(): void
- getSum(int column):int
- getAverage(int column): float
+ showPatients(Map<String, PatientInterface> patientDetails)
PatientsPane
- cholesterolButton: JRadioButton
- openMonitorButton: JButton
- conditionsPanel: JPanel
- scrollPane: JScrollPane
- conditionGroup: ButtonGroup
- patientTable: JTable
- initComponents(): void
+ showPatients(Map<String, PatientInterface> patientDetails): void
+ getSelectedCondition(): Condition
+ getSelectedPatients(): List<String>
MenuPane
- welcomeLabel: JLabel
- viewPatientsButton: JButton
- initComponents(): void
LoginPane
- layout: GroupLayout
- idPrompt: JLabel
- idInput: JPasswordField
- enterButton: JButton
- initComponents(): void
1
<<interface>>
View
+ showLoading(): void
+ stopLoading(): void
+ disableLoginInput(): void
+ enableViewPatients(): void
+ enableStartRefresh(boolean enabled): void
+ loginComplete(boolean success): void
+ patientRetrievalComplete(boolean success): void
+ healthDataRetrievalComplete(boolean success): void
+ addRemovePatients(): void
+ addActionListeners(Controller c): void
+ setModel(ReadOnlyFhirEntityManager m): void
+ display(): void
Controller
1
<<interface>>
Controller
+ requestLogin(String id): void
+ requestViewPatients(): void
+ requestMonitor(List<String> patientIdList, Conditition c): void
+ refreshMonitor(int refreshTime): void
+ cancelRefreshMonitor(): void
+ requestAddRemovePatientsFromMonitor(): void
informs ▸
1
1
MonitorController
- view: View
- manager: FhirEntityManager
- worker: SwingWorker
- patientRetrievalSuccess: boolean
- dataRetrievalTime: int
- dataRetrievalActivated: boolean = false
- requestPatients(): void
threading
Thread
1
1
1
1
1
ColoumnColorRenderer
- backgroundColor: Color
- foregroundColor: Color
- chosenRows: List<Integer>
+ getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column): Component
PatientDetailsWindow
- dobLabel: JLabel
- genderLabel: JLabel
- addressLabel: JLabel
- dobField: JTextField
- genderField: JTextField
- addressField: JTextArea
- detailsPanel: JPanel
- scrollPane: JScrollPane
- initComponents(): void
<<creates>>
MainWindow
- layout: CardLayout
- panels: JPanel
- loginPane: LoginPane
- menuPane: MenuPane
- patientsPane: PatientsPane
- monitorPane: MonitorPane
- model: ReadOnlyFhirEntityManager
- createPanels(): void
- createWindow(): void
- getUserId(): void
- getSelectedPatients(): List<String>
- getCondition(): Condition
- getRefreshTime(): int
+ showPatientDetails(MouseEvent evt): void
javax.swing

Model-View-Controller (MVC) Architecture

We utilised a form of Model-View-Controller (MVC) architecture in order to establish a separation of concerns between the UI, application and business logic. This would allow us to work on the view, controller and model components independently, with the only concern being that each component complies with its public interface. We designed a passive model because it is conceptually simple; the view can pull data from a read-only reference to the model (an interface with only getters) when the controller tells it that the model has changed. Using this design, the view is able to update itself without manipulating or constantly polling the model. This does introduce some coupling between the view and controller, however following MVC means making strong separation between presentation (view & controller) and domain (model) [1], so some coupling is to be expected.

Cohesive Encapsulations

Our classes have high cohesion because they are highly focused and serve singular purposes due to separation of concerns (from using MVC). Encapsulation boundaries between classes are enforced using data hiding, interfaces and by putting related classes in a package (e.g. all UI classes go in the View package). Most dependencies exist between classes within the package and any cross-package dependencies are acyclic in order to satisfy the Acyclic Dependencies Principle (ADP). In our design, the Controller package depends on both the View and Model Packages, the View depends only on the Model and the Model itself is independent, thus maintaining ADP.

Patient and Practitioner Adapter

Since the Practitioner and the Patient were classes that belonged to the FHIR package, the design needed to adapt such class so that they were able to be reused in the context of the Health App. The form of Adaptation was of a Object Adapter wherein the "PatientAdapter" and "PractitionerAdapter" had adaptees of "Patient" and "Practitioner" respectively. The consequence of such design meant that the interface for these classes could be tightly controlled, and meant that any changes that were made to the Patient or Practitioner would not directly interfere with the app, as it relied instead on the Interface and Adapters.

Code Extensibility

The extensibility of the code is showcased by its many interfaces which act as "hinge points" throughout the design. An example of such a hingepoint is the "View" interface, which allows the application to be open to the extension of many different types of views.
Furthermore the use of interfaces follows the design principle of **dependency inversion** the use of which is through the Controller (which is the connecting layer between the Model and View in MVC architecture). The 'MonitorController' is never not dependent on a concrete class of either the Model nor the View, allowing the concrete implementations of the Model and View to change as required.
Finally, even as the majority of the system is dependent on the "Condition" Enum, this is highly justified as Enumeration is a stable entity when compared to classes. Indeed the remarkable extensibility of the System is showcased by realising that an entirely new Condition can be monitored **by simply adding the condition to the Enum and implementing a search for the condition in the Flask backend.**

References

[1] Fowler, M., 2006. *GUI Architectures*. [online] martinfowler.com. Available at: <https://martinfowler.com/eaaDev/uiArchs.html> [Accessed 20 May 2020].