Design Patterns and Principles Used

Throughout the planning of the structure and design of the application, we considered extensibility of the program, in terms of the possibility of adding different measurement types, as well as observations of the patient. In order to make the design easily extensible as possible, design patterns and principles were used in order to achieve this.

MVC

The Model-View-Controller pattern was used in our Patient Monitor application and helped separate the logic behind the system and the visual representation of it. One of the drawbacks of the decision in using this design pattern is that it increases the complexity of the system. However, since the Model component depends on neither of the View nor the Controller, we can build and test it alone, which is an example of the Acyclic Dependency Principle- reducing cycles of dependencies within our application. The model component in our design involves a column of data for each measurement in the table model, by doing this, we may extend the model by adding more columns and have the view adopt such changes without modifying the view manually.

Bridge

We integrated a Bridge pattern to decouple the initial design of classes that were created for the purpose of observing cholesterol related attributes. This design allows the implementation details behind each measurement to be hidden from the client, as the system only needs to know about the abstraction of the measurement types.

Observer Pattern

The Observer Pattern was found to be appropriate for the application as the table needed to be notified when there was a change in the latest total cholesterol value, and potentially other measurements as well.

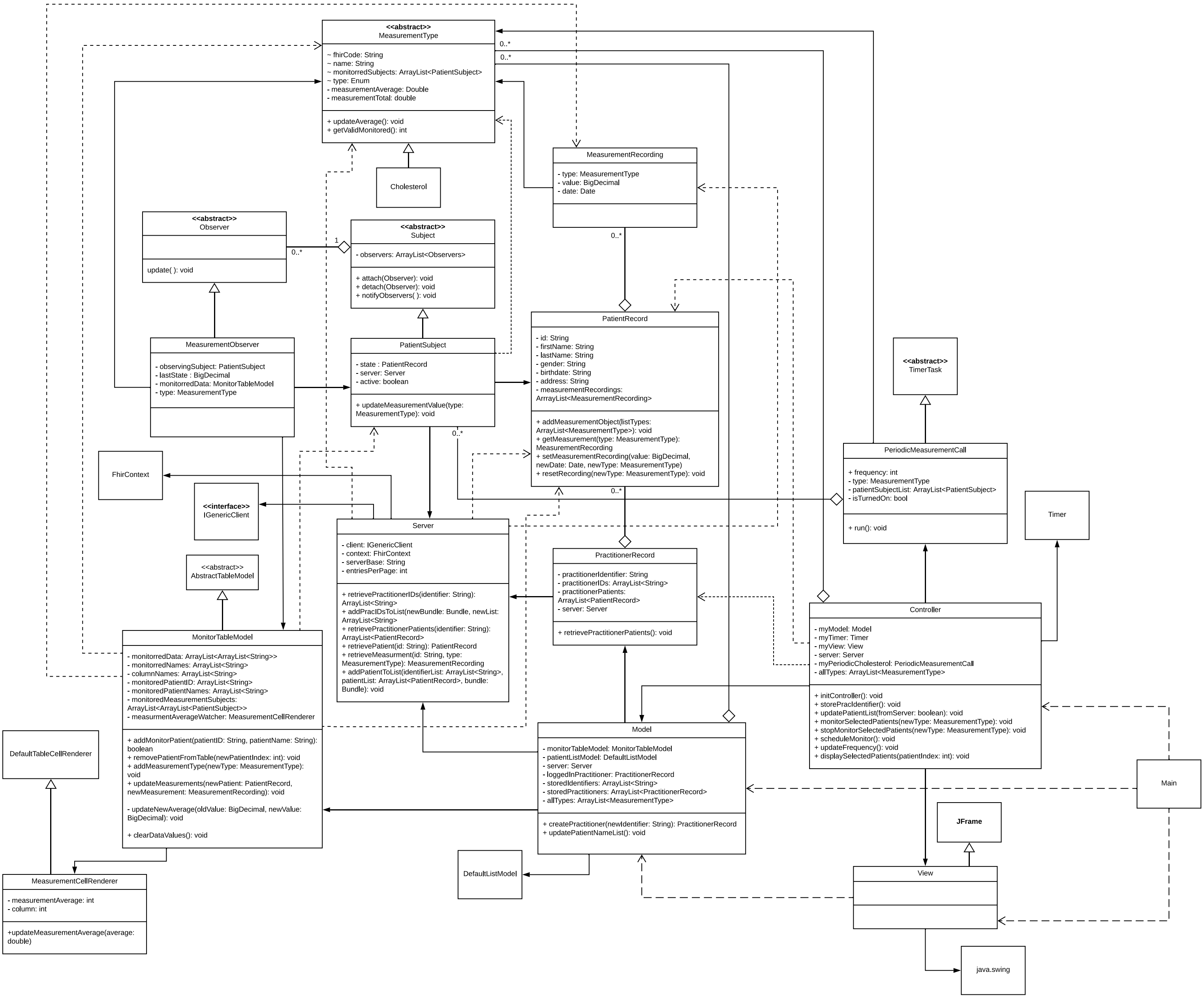
In our design, we have implemented the patient as the concrete subject in which measurements of the patient can be observed by the concrete observer which detects changes in measurements and notifies the table.

The hinge point in using this pattern is that it promotes loose coupling the extension of different observers (i.e. another observer may observe different values in a different way) to be attached to the concrete subject without it knowing of the subclass of the observer, which makes the design more extensible. However, a disadvantage is that it must be implemented with care to avoid complexity issues when notifying the observer.

Assumptions:

- Each practitioner has one unique practitioner identifier
- Any patient who does not have a cholesterol recording initially on the server will not have any cholesterol values added after

Class Diagram


<<abstract>>
MeasurementType
~ fhirCode: String
~ name: String
~ monitorredSubjects: ArrayList<PatientSubject>
~ type: Enum
- measurementAverage: Double
- measurementTotal: double
+ updateAverage(): void
+ getValidMonitored(): int
0..*
0..*
Cholesterol
MeasurementRecording
- type: MeasurementType
- value: BigDecimal
- date: Date
0..*
<<abstract>>
Observer
update( ): void
0..*
1
<<abstract>>
Subject
- observers: ArrayList<Observers>
+ attach(Observer): void
+ detach(Observer): void
+ notifyObservers( ): void
PatientRecord
- id: String
- firstName: String
- lastName: String
- gender: String
- birthdate: String
- address: String
- measurementRecordings:
ArrrayList<MeasurementRecording>
+ addMeasurementObject(listTypes: ArrayList<MeasurementType>): void
+ getMeasurement(type: MeasurementType): MeasurementRecording
+ setMeasurementRecording(value: BigDecimal, newDate: Date, newType: MeasurementType)
+ resetRecording(newType: MeasurementType): void
0..*
<<abstract>>
TimerTask
MeasurementObserver
- observingSubject: PatientSubject
- lastState : BigDecimal
- monitorredData: MonitorTableModel
- type: MeasurementType
PatientSubject
- state : PatientRecord
- server: Server
- active: boolean
+ updateMeasurementValue(type: MeasurementType): void
0..*
PeriodicMeasurementCall
+ frequency: int
- type: MeasurementType
- patientSubjectList: ArrayList<PatientSubject>
- isTurnedOn: bool
+ run(): void
FhirContext
<<interface>>
IGenericClient
Timer
Server
- client: IGenericClient
- context: FhirContext
- serverBase: String
- entriesPerPage: int
+ retrievePractitionerIDs(identifier: String): ArrayList<String>
+ addPracIDsToList(newBundle: Bundle, newList: ArrayList<String>
+ retrievePractitionerPatients(identifier: String): ArrayList<PatientRecord>
+ retrievePatient(id: String): PatientRecord
+ retrieveMeasurment(id: String, type: MeasurementType): MeasurementRecording
+ addPatientToList(identifierList: ArrayList<String>, patientList: ArrayList<PatientRecord>, bundle: Bundle): void
PractitionerRecord
- practitionerIdentifier: String
- practitionerIDs: ArrayList<String>
- practitionerPatients: ArrayList<PatientRecord>
- server: Server
+ retrievePractitionerPatients(): void
<<abstract>>
AbstractTableModel
Controller
- myModel: Model
- myTimer: Timer
- myView: View
- server: Server
- myPeriodicCholesterol: PeriodicMeasurementCall
- allTypes: ArrayList<MeasurementType>
+ initController(): void
+ storePracIdentifier(): void
+ updatePatientList(fromServer: boolean): void
+ monitorSelectedPatients(newType: MeasurementType): void
+ stopMonitorSelectedPatients(newType: MeasurementType): void
+ scheduleMonitor(): void
+ updateFrequency(): void
+ displaySelectedPatients(patientIndex: int): void
MonitorTableModel
- monitorredData: ArrayList<ArrayList<String>>
- monitorredNames: ArrayList<String>
- columnNames: ArrayList<String>
- monitoredPatientID: ArrayList<String>
- monitoredPatientNames: ArrayList<String>
- monitoredMeasurementSubjects: ArrayList<ArrayList<PatientSubject>>
- measurmentAverageWatcher: MeasurementCellRenderer
+ addMonitorPatient(patientID: String, patientName: String): boolean
+ removePatientFromTable(newPatientIndex: int): void
+ addMeasurementType(newType: MeasurementType): void
+ updateMeasurements(newPatient: PatientRecord, newMeasurement: MeasurementRecording): void
- updateNewAverage(oldValue: BigDecimal, newValue: BigDecimal): void
+ clearDataValues(): void
DefaultTableCellRenderer
Model
- monitorTableModel: MonitorTableModel
- patientListModel: DefaultListModel
- server: Server
- loggedInPractitioner: PractitionerRecord
- storedIdentifiers: ArrayList<String>
- storedPractitioners: ArrayList<PractitionerRecord>
- allTypes: ArrayList<MeasurementType>
+ createPractitioner(newIdentifier: String): PractitionerRecord
+ updatePatientNameList(): void
Main
JFrame
DefaultListModel
View
MeasurementCellRenderer
- measurementAverage: int
- column: int
+updateMeasurementAverage(average: double)
java.swing

References:

GeeksforGeeks. (2018, February 08). “MVC Design Pattern.” [Online]. Retrieved from www.geeksforgeeks.org/mvc-design-pattern/.

Pham, T (2020), *Design Patterns 1*, FIT3077 Software Engineering: Architecture and Design, Lecture 5 [Powerpoint Slide 36], Semester 1 2020, Monash University. Retrieved from https://moodle.vle.monash.edu.au/

Pham, T (2020), *Design Patterns 2*, FIT3077 Software Engineering: Architecture and Design, Lecture 6 [Powerpoint Slide 4], Semester 1 2020, Monash University. Retrieved from https://moodle.vle.monash.edu.au/