FIT3077 Assignment 2 Design Rationale (all stages) **Apoorv Kansal (27821455), Quang Ly (28688856)**

Architecture

Having a backend service poll instead of all clients minimizes the number of requests made to the FHIR service. For example, if 1 million clients wanted to observe patient A's cholesterol level, instead of making 1 million requests to get the same data per hour, there is a backend service that will make just 1 request every hour to FHIR to get the necessary health measurement (eg. cholesterol level) and then send events back to all clients via websockets. Furthermore, the backend service will only poll for patients that are being monitored on specific measurements (eg. cholesterol). This also minimizes the requests to FHIR. Furthermore, it is much easier to optimize and minimize requests further in future with server updates rather than client app updates.

Software Design

Observer Pattern

The frontend design leveraged the observer pattern to define an extensible communication workflow all the way from the received websocket events to the frontend GUI app. Given that the practitioner can start/stop monitoring a patient, the observer pattern is useful because we can easily attach and detach the necessary observers [1] (subclasses of AbstractHealthMeasurementView) to AbstractHealthMeasurementModel (subclasses of it like CholesterolModel) whenever the end-user decides to start/stop monitoring. The observer pattern allows for high extensibility [2] which has helped us to easily implement an additional monitor by simply attaching a new view (OralTemperatureView) as an observer to a new health measurement model (OralTemperatureModel) that has Subject capabilities (through inheritance). The abstract coupling that comes with this pattern [1] has also allowed us to encapsulate the Observer and Subject logic into parent classes which can be subclassed (eg. HealthMeasurementListener subclasses Subject and PatientMonitorModel subclasses Observer) and not have tight coupling between concrete business related observers and subjects.

We also used the observer pattern in the backend design to send polled data down the websockets periodically. The backend PatientMonitorModel and MonitorController class utilizes the observer pattern. PatientMonitorModel inherits from PatientMonitorSubject while MonitorController implements the MonitorControllerObserver interface. Thus, every time there is new health measurement data available to the patient (every time the PatientMonitorModel's observationLoader polls and retrieves new health measurement data), the PatientMonitorModel will notify the MonitorController. This pattern is used so that the PatientMonitorModel does not directly depend on the MonitorController class as well as giving it a mean to inform and communicate changes even with no direct dependency. We used the observer pattern to broadcast websocket data to the relevant patient monitors selected by the end-user in the frontend. This was useful because if we ever added a new type of patient monitor, we can easily observe the websocket listener and receive data just like other patient monitors.

Adaptor pattern

The Adaptor Pattern was very useful to abstract implementation details from FHIR and not tightly couple the application with their specific interfaces [3]. We used a HapiPatientModel in the frontend that stored FHIR's Patient object and the business code would only interact with HapiPatientModel to get the necessary id and name. This way, the application code did not know anything about FHIR's patient implementation. As a result, this form of decoupling between the application code and the FHIR allows us to reuse the adapter with different web systems (maybe an alternative version of FHIR) [4]. Similarly, the backend used a HapiObservationModel class to wrap FHIR's Observation object which was useful as the backend application code did not concern itself with FHIR's implementation details.

MVC pattern

The Model View Controller pattern came in handy when designing the application. It achieved a great amount of separation of concerns [1] in the system and allowed us developers to focus on each part

of the system independently, rationally and quickly [5]. The separation of concerns achieved helped organise our packages as well where we had model, controller and view packages in the frontend and model, controller packages in the backend. Although there was increased complexity [5] as shown in the frontend and backend class diagrams, making modifications to the system did not break the whole system which was great for maintainability and also extensibility of the system (eg. adding new monitors). Since the UI design is not very appealing, we can also replace the views with a new monitor UI without changing the internal business logic.

Factory pattern

The factory pattern was useful to generate objects based on preference made. In the frontend, we used HealthMeasurementViewCreator factory class to generate health measurement monitor views based on the practitioner's preference. Similarly, the backend use PatientMonitorCreator to generate a monitor based on the request data payload which specified the patient, client id and health measurement. Using the factory pattern worked well given that different workflows needed to be enabled based on user input. To add new monitors, we can simply enhance the factory's implementation to create new views for the new monitors. As such, this pattern has helped us achieve reusability, easy extensibility and also abstracted the implementation details of how views are instantiated [6]. Although the Abstract Factory pattern achieves greater decoupling [1], it also introduces more complexity [7] and is more suited to instantiating a family line of related/dependable "products" [1] whereas we just needed to instantiate one object. Hence, we chose the Factory pattern.

Liskov Substitution Principle

Using this principle allowed us to have code that did not depend on implementation details and leverage inheritance to its full power. For instance, we subclassed MonitorEventModel with various health measurement specific event models (eg. BloodPressureMonitorEventModel) in the backend which follows this principle and can be substituted wherever MonitorEventModel is used without breaking the application. Utilizing this principle means the subclasses are true subtypes that have enhanced behaviour and achieve cohesion and encapsulation as classes are categorized correctly based on behaviours [1]. It also achieves reusability and ensures that we do not have repetitive code which reduces the maintainability issue [1]. Reusability is of utmost importantance when expanding to new patient monitors (eg. adding OralTemperatureModel which subclasses AbstractHealthMeasurementModel).

Interface Segregation Principle

This principle requires the separation of different interfaces instead of having an omnipotent interface that is going to handle everything [8]. No class should depend on methods it does not need [8]. In our design, we have removed the original ObservationModelInterface to handle all the different health measurement protocols and introduced new interfaces such as CholesterolObservationModelInterfaces, BloodPressureObservationModelInterface, etc. Additionally, classes like CholesterolPatientMonitorModel and TobaccoUsePatientMonitorModel are used differently as the client is interested in the value and unit of the cholesterol measurement whereas for tobacco use, they are interested in the smoking status. Thus, the system is more robust with the cost of a few more small and manageable interfaces. Additionally, the decision to proceed with these designs lies in the fact there are only a limited amount of health measurements that can be monitored and thus, though the number of interfaces scales proportionally with the number of different types of monitorable health measurement, the design would never be unmaintainable if we decide to add in new health measurement monitors. In fact, creating more and smaller versions of interfaces and spreading the dependencies across them makes the system more extendable and easier to add more monitors [1].

Open closed principle

This principle is demonstrated by the use of health measurement model interfaces such as CholesterolObservationModelInterface, BloodPressureObservationModelInterface and many more. These interfaces are closed for modification and available for use and extensibility by other classes. Introducing this principle allows communication on interface levels which are unlikely to change and make the overall system resilient to change [1]. Resilience to change is important for this app given that new monitors will be added in the future and hence, we need reliable classes that have defined implementations and interfaces that cannot be modified but potentially enhanced through inheritance/polymorphism.

Dependency Inversion Principle

Through the use of interfaces that act as hinge points, this principle makes the system resistant to changes to lower levels [1]. This is very beneficial when we made all views depend on relevant controller and model interfaces in the frontend. This is because we can easily replace the models or controllers in future and so long as the new models or controllers respect the relevant interfaces (hinge points), the application should work. As such, this allows for great freedom to switch between different implementation and concrete class usages making the system greatly flexible and error tolerant [1].

References

1. *Freeman, E., Robson, E., Bates, B. and Sierra, K., 2004. Head first design patterns. " O'Reilly Media, Inc.". [Accessed 19 May 2019].*
2. *javaworld. 2003. An inside view of Observer. [ONLINE] Available at: https://www.javaworld.com/article/2073299/an-inside-view-of-observer.html. [Accessed 19 May 2019].*
3. *java-design-patterns. 2019. Adapter. [ONLINE] Available at: https://java-design-patterns.com/patterns/adapter/. [Accessed 19 May 2019].*
4. *geeksforgeeks. 2016. Adapter Pattern. [ONLINE] Available at: https://www.geeksforgeeks.org/adapter-pattern/. [Accessed 19 May 2019].*
5. *interserver. 2018. What is MVC? Advantages and Disadvantages of MVC. [ONLINE] Available at: https://www.interserver.net/tips/kb/mvc-advantages-disadvantages-mvc/. [Accessed 19 May 2019].*
6. *journaldev. 2011. Factory Design Pattern in Java. [ONLINE] Available at: https://www.journaldev.com/1392/factory-design-pattern-in-java. [Accessed 19 May 2019].*
7. *codeproject. 2013. Understanding Factory Method and Abstract Factory Patterns. [ONLINE] Available at: https://www.codeproject.com/Articles/35789/Understanding-Factory-Method-and-Abstract-Factory. [Accessed 19 May 2019].*
8. *hackernoon. 2018. Interface Segregation Principle. [ONLINE] Available at: https://hackernoon.com/interface-segregation-principle-bdf3f94f1d11. [Accessed 19 May 2019].*