FIT3077 Design Patterns and Principles- safeyeet

For this project, we have tried to abide by multiple different design principles in our system. We also incorporated what we consider to be the necessary design patterns for a successful system of this size.

Design principles

The design principles that we used for this project were, but not limited to the:

- Open closed principle
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

We also tried to limit ourselves to repeating unnecessary code, following DRY (don't repeat yourself) by introducing abstract classes. The design patterns that we incorporated in our system were:

VMC (Views, Models, Controller)

This design pattern separates the application into three components, where we could separate the user interface with the backend interface. We implemented this design, as it was a requirement for us to have a graphical interface in this project, as well as communication with a backend which would use API calls, we decided that this design model satisfied our needs. The design was assisted by our other design models, namely the repository and queries.

Model
The model namespace contains the bulk of our code, as it contains our implementation of Patients, Practitioners and observations. It contains the interface for our patients and practitioners, as well as containing our builder and factory classes.
Views
Handles the UI elements, this is what is used to represent our graphic interface. Although we have a components folder, which contains most of our front end functionality, Vue components needs the views folder to route information to the components.
Controller
We have 3 controller classes, which launch different parts of the system. They allow user input, and passes this information to other parts of our system.

Repository Design Pattern

Repository is a design pattern, where we use collections to act as a mediator between our applications. It is a way to allow us to construct the API url that we needed to call.
Using this design model increases abstraction, as it limits information that is able to be accessed by other classes. If a local database or cloud database is used instead of querying from the FHIR server manually using web API, this layer would remain the same although

the implementation might change a bit. Repository would also serve as an abstraction layer for getting the data so the Service layer which will be calling the Repository layer do not need to know whether the Repository is getting the data by SQL or JSON as the return type is consistent.

Factory Design Pattern

The Factory model allows us to create object classes without declaring them directly by specifying their concrete classes.
In our system, the factory design pattern is only used by Observation, as it only has few fields and does not require the customizability of Builder design pattern. So our factory model is able to create object classes for Cholesterol, Tobacco, Systolic and Diastolic Blood pressure.

Builder Design Pattern

The builder design pattern is used to slowly create our objects step by step. It is similar to the factory design pattern as it is used to create objects without declaring them directly. We decided to use builder for objects that we would probably need to extend more often (patient and practitioner), as this would help us extend our code later on. Similar to factory, this design pattern helps us encapsulate our data. It also follows the Open-Closed principle.

Conclusion

The main justification of using all of these design patterns was to make our system more maintainable and extendable.
By using the VMC pattern, we are able to increase encapsulation. Although there are dependencies between the patient and practitioner, we have tried to separate the models so that we obey the interface segregation principle.

By using the Repository design pattern, we are able to produce an abstraction layer, to separate our data and logic. This assists in our design, as even though we are using API calls to obtain data, there may be a need for us to extend data functionality later, and adding an abstraction layer now would mean less work later.

By using the Factory and Builder design patterns, we are able to create objects without calling them directly. This allows us to encapsulate our code, and give us more control over our objects. Our Observation factory method also followed the Liskov substitution principle, where with our parent classes, we could substitute it with any of our child classes (Cholesterol, BloodPressure, Tobacco, DiastolicBloodPressure, SystolicBloodPressure)

In conclusion, We have used a variety of different design patterns to follow the design principles that we deemed the most suitable for this project. Outside of the design patterns, we have tried to encapsulate our data and reduce dependencies, as well as provided means for being able to add extensions for further functionality in the future.

Sources used:

Coding Horror, Understanding Model, view, controller Jeff Atwood, 2008
https://blog.codinghorror.com/understanding-model-view-controller/

Repository Design Pattern in C#,
https://dotnettutorials.net/lesson/repository-design-pattern-csharp/

[MS2008] Microsoft Corporation, Model-View-Controller, Microsoft Patterns & Practices Developer Center, 2008 http://msdn.microsoft.com/en-us/library/ms978748.aspx

Robert C. Martin, Design Principles and Design Patterns, 2000