Design Principles and Patterns Rationale (15/80)

Throughout the process of designing the SafeHeart system, our team has applied a few of the design principles that we've learned in the unit.

First and foremost, the **OCP (Open-Closed Principle)**. The application of this principle is evident as we apply abstractions in our system. Introducing abstractions to our system provide ***hinge points*** in a design where the design can be modified via OCP as the abstract objects change less frequently. For example, the OCP is evident due to the **Observer interface.** The Observer interface is made in a way that it is open for extension, but closed for extension (Observer interface acting as a hinge point). By meaning that, it allows multiple concrete observers, e.g: **Cholesterol Observer, Blood Pressure Observer, Tobacco Observer** to extend from Observer by having the concrete observer classes overriding the interface's methods (open for extension) and add new methods as well, while modification cannot be made in the Observer interface itself (closed for modification) It also enables new concrete observers to be added. Another place where the OCP can be seen is within the **ServerDatas Package**. Within the package there is a abstract class **ServerData** which contains a common method used by all the other subclasses such as **clinicianData**, **patientData** e.t.c. This ***improves the extensibility*** as another class can be added to the package when an extension to the system needs to be made.

The other principle that we have applied is **Common Closure Principle** as improving the ***maintainability*** of the system is essential to software design. I've grouped the classes based on the likelihood that they'll change together in packages because they all have similar functionality and are likely to change together if a change is made to the system (maybe due to tight coupling). To support my point of view, I've created Users package to group **User**, **Patient** and **Clinician**, another package called ServerDatas to group **cholesterolData, bloodPressureData**, and so on. Bearing in mind that 100% closure is practically impossible, CCP states that we should group like classes that cannot be completely closed with respect to each other together in a package. In that way, we can minimise the changes made to the system and reducing the risk and likelihood that each change made between classes will propagate to the other classes, which may introduce maintainability issues such as breaking of systems. This makes extending the system by adding new functionality easier as well because have an idea of the classes which will need to be changed when extending. **Acyclic Dependencies Principle** has been applied as well where there's no cycles between the packages by viewing the diagram in package level. This ensures one update in package does not lead to successive updates of other packages, making each of the package independent.

Initially in stage 1, we had a class called **FHIRServerData** which is responsible in fetching the datas from the server itself. The data being fetched include patient, clinician, blood pressure, tobacco status and cholesterol level. However, this introduces more responsibilities to the **FHIRServerData .** The more responsibilities the class has, the more likely that we need to change it very frequently if there is a need to. Therefore, we apply the **Single Responsibility Principle** in all occasions as each class is only responsible for a **single functionality**. This can be evidently seen in the **ServerDatas** Package where each class is responsible for only one thing. E.g the **clinicianData** class is only responsible to retrieving all the Clinicians from the FHIR server, the **bloodPressureData** class is only responsible to retrieving all the bloodPressureData of a patient and so each class has only one responsibility.

We also followed the **DRY principle** through the use of **abstract classes and interfaces** which help us take out repeating code and put them in a way it can be implemented by all classes which use it. We also **decompose** the code into **functions** so that when a part of code is needed we can just call the appropriate functions.

Observer pattern has also been implemented for this task. We need to use Observer Pattern for this case because the data from the server should be updated in the program every one hour and we do this with the help of the **TimerUpdater class** which changes the state of the **concrete subject**, **PatientVitalsData** . When the state of the **concrete subject** changes all the **observers** of the **subject** are **notified** which causes the observers to run their update() function which in this case calls the server again and updates data for **Blood Pressure Levels,**

Cholesterol Levels and Tobacco Status. again The concrete observer then notifies the monitor and have monitor update the display of the data. The reason we use Observer pattern is that we noticed that we have several different types of monitor, namely cholesterol graphical monitor, blood pressure graphical monitor etc to regularly update themselves in every 1 hour. Therefore, we want to **improve the extensibility and maintainability** of the system by having specialised observers (cholesterol, tobacco, blood pressure) notify the monitor independently. This enables **ease of extensibility** when we want to introduce a new type of monitor. When we need to extend the system, e.g Monitor Blood Glucose Levels, we just need to create a new concrete observer for this which implements the **Observer Interface.** In case, Observer Patterns were not applied then a lot of code would have to change in case the case of extending the program and a lot of code would be repeated, not to mention that tight coupling issues will cause difficulty in maintaining the system. In short, observer pattern also enables loose coupling between classes. However, The Observer design pattern can cause memory leaks in the basic implementation and requires both explicit attachment and explicit detachment. This is because the subject holds strong references to the observers, keeping them alive.

Throughout the designing of the system, we have faced architectural decisions and tradeoffs had to be made. One such **tradeoff** is that we were planning on using an **interface** to implement **ServerData Class** so that we can apply the Interface Segregation Principle, since we notice fhirServerData in stage 1 has too many responsibilities (it is a fat class), by separating each components into different classes. However **interfaces** can have only **abstract** methods, **abstract** class can have abstract and **non-abstract** methods. Since we wanted a non-abstract method to be implemented we decided to use an **abstract** class. Although we managed to separate the responsibilities of the fhirServerData, we still think that implementation of serverData as an interface could have been a better option for design decisions.

References

Robert C. Martin, Design Principles and Design Patterns, 2000
Robert C. Martin, The Open-Closed Principle, 1996.
Martin, Robert C., The Dependency Inversion Principle, 1996.
Martin, Robert C., Granularity, 1997.