Design Rationale

Package Cohesion Principles and Package Coupling Principles

As seen in our package level diagram, included in the appendix, our system exemplifies the acyclic dependencies principle. The acyclic dependency principle states that "the dependencies between packages must not form cycles." [Mar2000]. This occurred as a consequence of us implementing the MVC design pattern. This minimized the effect of the "morning after syndrome," thus, we were able to split work between the two of us efficiently. For example, if one of us worked on the Database package independently and the other worked on the View or Controller package separately, we were able to do so without having problems merging our code and causing errors in our application.

The common closure principle was applied within the Database package. The CCP states that "classes that change together, belong together" [Mar2000]. This package contains classes that are highly associated and dependent on each other (through the implementation of interfaces or regular associations/dependencies). Thus, it is important to group these classes in one package to remind ourselves that we should rebuild and retest all the classes in this package if we were to make a change. Furthermore, it will minimize the number of packages to be affected by a change. For example, if we make a change to this Database package, the Controller and View packages will not be affected at all. Again, this helped us split up the work equally without problems.

Interface Segregation Principle and Dependency Inversion Principle

The Database package was implemented with interface segregation and dependency inversion principle in mind. From observing the FHIR server, it is clear that the format of the documents were different based on the type of resource (i.e., Observation, Encounter, Patient, etc.). Implementing one DAO interface for the entire fetching and storing process would be incredibly messy and classes would need to implement methods that it does not use since the format of the resources are different. Thus, implementing ISP fixed this problem. We applied the dependency inversion principle when we separated the concrete classes for fetching and storing data and the class that actually interacts with the GUI (DBModel) by interfaces. This way, DBModel does not know anything about the concrete implementations and only relies on the methods provided by the interface. This helped us since we refactored the implementations for fetching and storing data a lot as we found better methods, however the abstract class remained the same. This will help us extend our project as well as the abstractions provide hinge points. It will be easy to add new functionality by simply adding a new method to the interface.

Model-View-Controller Design Pattern

We applied the GUI with the model-view-controller design pattern. The view and the logic for each page is separated- LogInView and LogInController, PatientsView and PatientsController and the DBModel. Implementing the GUI this way helped eliminate any cyclic dependencies. We are able to separate concerns very well. The views manage the display of information, the controllers manage all the user inputs, listeners, etc. while the model queries the database and responds to actions as requested from the controllers. This made it easier for us to debug our code as if we know we had a problem with one module, it was completely separate from the other and we didn't need to retest that. Furthermore, since the model does not depend on the view at all, we can add new features to the view without caring about how it would affect the model. This helps with the extensibility of the project as we can add new pages and views without worrying about causing errors in the database queries.

References

- [Mar2000] Robert C. Martin, “Design Principles and Design Patterns”, 2000. Available on-line from Object Mentor: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

Appendix

Package Level Diagram