Design Rationale

Introduction

In assignment 2, the model-view-controller design pattern was implemented. This helped us to separate concerns and eliminate cyclic dependencies. Package cohesion and coupling principles were also implemented in efforts to separate classes into different packages based on their functionalities. Furthermore, interface segregation principle and dependency inversion principle were used to implement the backend. The requirements of assignment 3 called for a variety of refactoring techniques. We implemented this whilst focusing on the following quality attributes: modifiability, extensibility and reusability.

Design Patterns

Adding the functionality to retrieve blood pressure measurements from the server only required adding a few more methods to ObservationRepository and DBModel classes. The ease of adding this functionality demonstrates the extensibility of our system. This success can be attributed to our use of interface segregation principle and dependency inversion principle to organize the classes that deal with the collection of data from the server and database as we have implemented in assignment 2.

On the other hand, adding the ability to add various displays of the measurements required some refactoring. One refactoring technique we implemented was "extract class." Previously, we combined the landing page with all patient names and cholesterol monitor table into one view and controller class. From this, we extracted the logic for the cholesterol monitor and created new view and controller classes (CholesterolTableView and TCTableController). This helped minimize the dependency between these two views which increased the modifiability of the system. For example, if we decide that we no longer need the cholesterol monitor table but still need the view for the patients list, we can simply exclude the cholesterol monitor view and controller classes from the GUI. This modification can be done easily at runtime. Furthermore, changes to each view can be done independently of each other, which facilitates testing. With this, for each different type of display, we simply created new view classes- i.e., CholesterolChartView, BloodPressureTableView, SBPChartView- as well as their respective controller classes. This also improves upon the reusability of the system as each new class that is added can access the functionality of each separate view by simply linking to it without having to write duplicate code to implement similar functionality.

In assignment 2, there was no need to implement the Observer pattern since we only had one main view and controller class. However, for this assignment, the Observer design pattern is necessary in order to update each separate view (acting as observers) automatically for a fixed time interval. Thus, every time observations are fetched from the server and a patient is updated with new measurements, all views are notified and refreshed accordingly. One advantage of this includes the fact that the abstract observer and subject class are now hinge points in the design where extension is possible. Furthermore, we do not need to implement separate timers for each view as all views will be updated automatically according to one timer that is set in PatientsView. This minimizes redundant code and helps with the overall extensibility of our design. For example, if we have a new display that needs to be updated in the same manner, we can simply make its controller class implement the Observer abstract class and register it as an observer to the concrete subject class, PatientUpdater.

Along the way, refactoring techniques such as renaming methods, moving methods, inlining and self-encapsulated fields were implemented to improve upon the overall design of the software.

Package Principles

The organisation of our packages in assignment 2 already exemplified the acyclic dependencies principle and the common closure principle. This proved to be successful as it facilitated testing and individual changes to classes in one package can be made without worrying about breaking the functionality of classes in other packages. In this assignment, one additional package is introduced, the observer package, which contains the

abstract Observer and Subject class for the observer design pattern. This exhibits the stable abstractions principle which states that “stable packages should be abstract packages” [Mar2000]. The controller package, which is not stable as it depends on many classes in the database and view packages, depends on this abstract observer package. Thus, this is a direct implementation of the stable dependencies principle which states that a package should depend on the more stable package. The advantage of this is that the abstract package is completely independent and the classes that depend on it (i.e., the controller classes in our system) do not have to worry about its code breaking as the abstract package is extremely robust.

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

Design Rationale

In the same document as the class diagram, provide a written explanation of the design principles and patterns that you have applied in the design of your system. You must explain the reasoning behind your class model in terms of the principles and patterns described in lectures, or in other design texts. All sources must be properly cited. This document (including class diagram and explanation) should be no longer than two A4 pages. (5 Marks)

- Students must explain WHY their design is as it is, not simply WHAT the design is (the UML documents do that). This explanation must be in terms of the principles and patterns described in lectures, or in other design texts. Patterns and principles must be referred to by name. All sources must be properly cited.
- The advantages and/or disadvantages of the patterns and principles applied must be described explicitly. It is not enough to say something like "we designed it like this because it is good", or even "we used the Observer pattern because it is well-known and good". There should be focus on extensibility.
- Students should discuss how their initial design (developed in Assignment 2) supported the new extensions. If you decided to start from scratch, you should discuss the reason for this, e.g.: “In our initial design we had implemented the Observer design pattern. The new requirements made it clear that we needed a design pattern that can support multiple views, hence we decided to completely change our design and implement MVC.”
- Refactoring techniques applied during the extension of the system must be discussed.

Principles used:

- Open-Closed Principle
- Dependency Inversion Principle
- Interface Segregation Principle

Design Patterns used:

- Model-View-Controller (Hinge point)
- Observer (Hinge point) - Refactoring because we needed to update multiple views (blood pressure table, cholesterol table, cholesterol chart, blood pressure charts).

Refactoring:

- Refactored model view controller. (Extract class)
    - Created new view classes and controller classes.
    - Separated Cholesterol Monitor table and Patients View so now they have two different classes and controller classes.
- Reduced dependency between Encounter DAO and Observation DAO
- Rename Method
    - renamed a bunch of methods to make it clearer (addToPatientList- addToPutList)
- Getters and setters for everything and do not access private variables (self encapsulate field)
- Move method
- inlining: reduce the number of unnecessary methods/variables and simplify the code
- Advantages:
    - without refactoring, the design of the software will decay

    - badly designed code usually uses more code to do the same thing as well-designed code
    - an important goal of refactoring is thus to remove duplicate code (easier to maintain, the more code there is, the more there is to understand)
    - the use of refactoring as an aid to understanding can also help find bugs
- known refactorings:
    - composing methods (extract method, replace temp with query, introduce explaining variable, replace method with method object, etc.)
    - moving features between objects (move method, move field, extract class, hide delegate, remove middle man)
    - organizing data (replace data value with object, change value to ref. duplicate observed data, encapsulate field, replace type code with subclasses)

Quality Attributes

- Modifiability (views are very modifiable)
    - elements must be designed so that changes have limited impact (how easy design of system can be changed)
    - modifiable systems are easier to change/evolve
    - modifiability should be assessed in context of how a system is likely to change
    - no need to facilitate changes that are highly unlikely to occur
    - a demonstration of how the solution can accommodate the modification without change
    - minimizing dependencies increases modifiability
        - changes isolated to single components likely to be less expensive than those that cause ripple effects across the architecture
    - techniques:
        - decrease coupling (hide information (encapsulation), reduce communication paths, break dependencies)
        - increase cohesion
        - reduce module size
        - defer binding time
        - patterns
- Extensibility
    - inter-component usage must be managed
- Reusability
    - inter-element coupling must be restricted so that it is possible to extract an element without having to bring a large part of its “home” environment with it
    - never need to look at the source code of the module (except for the documentation)
    - only need to link to the module to access its functionality
    - we receive a new version of the module whenever it is fixed or enhanced

Package cohesion principles

- Composing packages makes them reusable and easily maintainable
- Release reuse equivalency principle:
    - The granule of reuse is the granule of release. only components that are released through a tracking system can be effectively reused. this granule is the package.

    - If anything in the module is changed, the client needs to reintegrate and retest even if they were not using the changed part.
    - ***Only components that are released through a tracking system can be effectively reused.***
        - clients of the module need to be notified of the new release.
        - releases need to be tagged with version numbers or names.
        - clients need to be able to choose which version of the module they want to use.
        - older versions need to be support for a reasonably long time.
    - group reusable classes together into packages.
- common reuse principle:
    - classes that aren't reused together should not be grouped together.
    - a dependency on a package is a dependency on everything in the package.
    - classes that collaborate with other classes should belong together in a package.
    - advantage:
        - helps to minimize how often we do re-build, re-test and re-lease the client software.
- common closure principle:
    - related to open-closed principle.
    - CCP helps to minimize the number of packages to be affected.
    - minimizes the number of packages that must be re-released when a change is made.
    - classes that change together, belong together.

package coupling principles

- acyclic dependencies principle:
    - the dependencies between packages must not form cycles.
- break cycles by:
    - the addition of a new package
    - via the use of ISP or DIP
- stable dependencies principle:
    - a package should depend on the more stable package
- stable abstractions principle:
    - stable packages should be abstract packages

observer:

- a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- an abstract observer class is introduced
    - now the concrete subject (implementing the abstract subject interface) can have many different kinds of observer and new kinds can be added easily
    - the subject no longer depends on any concrete class
- the abstract observer class is now a "hinge point" in the design where extension is possible
    - open-closed principle and liskov substitution principle