CMPT 270 – 2017-2018 Term 01

Assignment 4:
Three Layer Architecture

Submission Deadline: 3 November 2017 at 11:55pm – Submit via Moodle!

Description

The objective of this assignment is to restructure assignment #3, and add another I/O option to it. Note that changing code to improve it (but not change its functionality) is called refactoring. For projects that last for several years, it is necessary to refactor of at least parts of them from time to time.

In more detail, the longest class from assignment #3 was the one to run the system. Also, it was doing several tasks. This assignment will divide up this class using the structure of the 3 layer architecture. In doing this task, the project should be structured to use packages. In addition, all the input and output for assignment #3 was using the console. Input and output using windows can be more pleasing. You are to change the program to give the user the option of console I/O or dialog box I/O. The classes from assignment #3, other than the system class, should not be changed except to add them to the entities package. You are to use the classes posted as a solution to assignment #3 rather than your solution so that everyone starts from the same place. The `HospitalSystem` class will have quite major changes, although some of it will remain the same. Several parts of the existing `HospitalSystem` will be moved to new classes.

Deliverables

***** Upload .java files, .class files and .txt/.pdf files for documentation within one .zip file *****

Also upload an executable JAR file for your system. Check that your JAR file successfully executes before submitting it, as it is easy to have problems with the manifest file. Include the source files (as well as executable files) in your JAR file. The addition of the source files is not the default, so you need to select it.

We may run your JAR file to verify that they run as you state. You should also include .txt files for:

(i) the external documentation for the assignment

(ii) a listing of all the classes,

(iii) a listing of the output from running your classes

The markers will annotate the .java files to provide written feedback on your work.

Part I: Refactor `HospitalSystem` into a Three-Layer Architecture

The present version of class `HospitalSystem` has a method to handle each of the tasks. The method reads the data needed for the task, checks the data for validity, invokes the methods of the other classes to do the task, and handles the result of the task (error or not). Any errors are handled by throwing an exception where the error is detected, and catching it in the `HospitalSystem` class to issue the error message. For this assignment, each task is to have a class defined for it that does the actual task. The `HospitalSystem` class will still obtain the data and handle the result, but the class for the task will do the rest of the task (check the data for validity, and invoke methods in other classes to do the task). Ideally, all 8 tasks (except quit) would have classes defined for them, but for this assignment you only need to do (i) add a new patient, (ii) list the empty beds of the ward, (iii) assign a patient a bed in the ward, and (iv) release a patient from the ward. The remaining tasks can be left where they are (with a few changes) or can be converted to classes, but all 8 tasks should work correctly.

Each class for a task should be a descendant of the class `CommandStatus` of the bankSystem project. Note that you can copy this class into your project, rather than putting the bankSystem on your class path. Any errors that occur in the carrying out of a task should set the successful field to false, and record an appropriate error message in the errorMessage field. The successful completion of a task should set the successful field to true. Also, if any information is to be returned to the invoking method, the usual return of a function is not to be used. Instead, the information should be placed in appropriate fields of the object for the task, and accessors provided for the client to obtain the information. This is done so that it is possible to delay the invocation of a task and/or delay the accessing of the results of the invocation. It also allows the task to be executed remotely, by sending the task to a remote location and later receiving it back to retrieve the results. We will not use either of these properties, but they are useful properties of tasks. The bankSystem has Command classes that are similar to what you are to have in your system.

When the tasks are put into separate classes, most of these classes need to access the `Patient` map, the `Doctor` map, and the `Ward`. In order to provide this access but prevent more than one instance of any of these entities, you are to use the Singleton Pattern to implement them. The classes that need these entities can now use the static method of the container access class to access the entity. The dictionaries will start empty. The class to access the ward is a little different because the ward needs input values to create the instance. Probably the best approach is to add another static method. This method will have parameters with the information needed to create a ward, and the method will use the information to create the one instance of the ward. This method must be invoked prior to any invocation to access the instance of the ward. Because the ward access class is a little different, it is supplied to you.

The above changes restructure (refactor) the project, but should not change its functionality. The part below restructures part of the code and also adds a new feature. The two parts are quite independent of each other. As such, they can be done in either order, but the final product should include them both. As there is 2 weeks to do the assignment, it is strongly suggested that you do one part the first week so that you only have the other part to do the second week.

Part II: Perform I/O with Dialog Boxes

The second part of the assignment is to change the input and output so that the user has the choice of doing I/O via the console or by dialog boxes. At the start of the project, a dialog box should be used to query the user as to which type of I/O is to be used, and then the rest of the project should use that I/O approach. This part of the assignment should NOT be implemented by having two versions of the methods of HospitalSystem. Instead, you are given an interface, InputOutputInterface, with the operations needed for I/O. This interface will have two implementations, one using console I/O and the other using dialog boxes for I/O. By this approach, a variable can be declared of the interface type. This variable is assigned an instance of the console implementation or an instance of the dialog implementation at the start. Throughout the rest of the class, whenever I/O is to be done, the appropriate method is invoked on the interface variable. By dynamic binding, this will use the console or dialog I/O dependent upon which implementation was assigned to the interface variable.

You will need to write the complete console implementation of the interface InputOutputInterface. For the dialog version, you will be given an abstract class that implements the readChoice() method. You need to extend this class to implement the other methods.

The classes of this project should be organized into packages similar to those of the 3 layer architecture. As there is no login as part of this application, you will not have a class similar to the startup class of the bankSystem. The HospitalSystem class has functionality similar to the Controller class of the bankSystem. You should put all the classes from assignment #3, except HospitalSystem, into one package, the entities package. The new version of the HospitalSystem class should be put into the startup package. Thus, you should end up with 5 packages: entities, startup, commands, containers, and userInterfaces.

Additional Guidelines

Internal Documentation

When writing these classes, be sure to properly document each method, including `@param` and `@return` comments. Also, if a method has a precondition, specify the precondition in a `@precond` comment, and throw a runtime exception if it is not satisfied. When appropriate, exceptions should be caught and handled. In particular, if any operation of the system fails and as a result throws an exception, it is reasonable to catch the exception, print the message of the exception, and then go on to the next operation. Note that these additional comments and precondition checks have already been added to the classes in the solution for Assignment 2.

External Documentation

For external documentation, include the following:

(i) A description of how to execute your test classes and system. This should be very short.

(ii) The status of your assignment. What is working and what is not working? What is tested and what is not tested? If it is only partially working, the previous point should have described how to run that part or parts that work. For the part or parts not working, describe how close they are to be working. For example, some of the alternatives for how close to working are (i) nothing done; (ii) designed but no code; (iii) designed and part of the code; (iv) designed and all the code but anticipate many faults; or (v) designed and all the code but with a few faults.

(iii) Maintenance manual. This is information for the person or persons who must keep your system running, fix any faults, and do any upgrades. For this assignment, it is sufficient to include a UML class diagram showing all the classes, the features of each class, and the relationships (inheritance,

uses and aggregation) amongst the classes. If a class has a container of items of some type, use the aggregation relationship for a container. This may well hide the particular data structure used for the container. You should not include an icon for any of the classes that are part of the Java library.

Code Structure

- Recall that in developing a system, is it bad not to have anything working. Try to have at least part of your system working.
- Also, a characteristic of good system is that I/O is not scattered throughout the system. It should be concentrated in one place, a class or subsystem, so that if the I/O interface is to be changed, only the one place needs to be changed. For this assignment, all the I/O should be in the system class, except for the main in each class that tests that class. Tests could be developed to test the methods of the system class, but for this assignment, the main of the system class can simply create an instance of the class and run the interactive application.
- Make sure that you do not have long methods. In particular, in your system class, you will probably have a method with a switch statement to determine which operation was selected by the user. When there are many cases, this method can get long. The key to keeping it from getting too long is to have as many tasks as possible (other than the actual switch statement that is determining the choice made) abstracted into other methods of the class. Thus, if something is a self-contained task, separate the task into a separate method. In particular, include a method to read the next integer. You can base your method on the code to read an integer towards the end of the slides on exceptions.
- In keeping with the principle of information hiding, the fields of a class should be private unless there is a very good reason to make them visible. When appropriate, methods should be supplied to access and set the fields.

Marking (total 50)

(4) Readability

(9) External documentation

(5) User packages, use of singleton

(10) Input and output

(11) Commands

(6) Testing

(5) Correct output

(-10) If you do not submit both an exectuable jar file and a zip file containing all .java files, .class files, and external documentation

This is an individual assignment. You are encouraged to discuss the general concepts of classes, types, containers, etc. with you classmates, but the specific details of the Hospital Management System in this assignment should be done completely individually. Students that copy / share work will be penalized.