CMPT 270 – 2016-2017 Term 01

Assignment 4:
Three-Layer Architecture

Submission Deadline: (extended) November 16 2016 at 11:55pm – Submit via Moodle!

Description

The objective of this assignment is to restructure Assignment 3. 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 (refactor the project). In doing this task, the project should be structured to use packages. 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 classes from Assignment 3, other than the system class, should not be changed except to add them to a package. The `FlightReservationSystem` class will likely have quite major changes, although some of it will remain the same.

Deliverables

***** Upload .java files and .txt 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 will run your JAR file to verify that they run as you state. You should also include .txt files for:

(i) your external documentation

(ii) a listing of all the classes,

(iii) the output from the console of your system

Refactor `FlightReservationSystem` into a Three-Layer Architecture

The present version of class `FlightReservationSystem` has a method to handle each of the main operations (for example, `addPassenger` from `FlightReservationSystem`). The method reads the data needed for the operation, checks the data for validity, invokes the methods of the other classes to do the operation, and handles the result of the operation (error or not). Any errors are handled by throwing an exception when the error occurs, and catching it in the run method of the `FlightReservationSystem` class to issue the error message.

For this assignment, each of these operations is to have a **class** defined for it that does most of the work. The `FlightReservationSystem` class will still obtain the data and handle the result, but the class for the operation will do the rest of the task. Ideally, all 8 operations would have classes defined for them, but for this assignment you only need to do (i) add a new passenger, (ii) display empty seats on a flight, and (iii) book a regular passenger on a flight. The remaining operations can be left as they are or can be converted to classes. Each class for an operation should be a descendant of the class CommandStatus of the bankSystem project (available on Moodle; 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 an operation should set the successful field and record an appropriate error message in the errorMessage field. In particular, should an exception occur while doing the operation, the exception should be caught, the successful field set to false, and the errorMessage set to the message from the exception. To handle an operation, the system class needs to read the data for it, create the command object, execute the method of the command object to do the operation, and then invoke the methods of the command object to determine the results.

When the operations are put into separate classes, most of these classes need to access the `Passenger` dictionary and/or the `Flight` dictionary. In order to provide this access but prevent more than one instance of either of these dictionaries, you are to use the Singleton Pattern to implement them. As an example, see the class CustomerSetAccess from the bankSystem, although you can start with an empty dictionary. The classes that need these dictionaries can now use the static method of the container access class to access the dictionary. As these containers will no longer be in the system class, the methods for operations not converted to commands will also need to access the dictionaries via the static 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 `FlightReservationSystem` class has functionality similar to the Controller class of the bankSystem. You can put all the classes from Assignment 3 (except the system class) into one package, the 'entities' package.

Thus, you should end up with 5 packages: entities, startup (containing `FlightReservationSystem`), commands (discussed in two paragraphs above), containers (containing the containers), and interfaces.

Additional Guidelines

Internal Documentation

When writing new 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.

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.

Code Structure

- Recall that in developing a system, one of the worst things is to reach the deadline for completion but not have anything working. Be sure 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.