CMPT 270 – 2017-2018 Term 01

Assignment 3:

Inheritance and Data Structures

Submission Deadline: 20 October 2017 at 11:55pm – Submit via Moodle!

Description

The objective of this assignment is to learn about inheritance in Java and get familiar with the basic data structures of Java. In the process, four classes, that extend the hospital application of Assignment 2, will be built and tested.

The four classes that you write should extend and use the sample solutions provided for Assignment 2 (so that everyone is starting from the same place). These will be posted on Moodle on Oct 10 (after everyone has had the chance to submit Assignment 2).

In particular, you must create the three new classes, a `Patient` class, a `Doctor` class, and a `Surgeon` class. You will also modify the `Ward` class. Each class should have a constructor, and the fields and methods specified below. For each of the classes, all the fields should be declared as private. Be sure to follow the specification given below, since the classes will be used in subsequent assignments in build a simple hospital management system. In addition, each class is to have a static main method that should be used to test the class following the principles of regression testing as discussed in class.

Deliverables

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

We may run your classes 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.

Hospital Application Extended

Each of the classes is now discussed in more detail:

`Patient` class:

The `Patient` class extends the `Person` class. The `Patient` class should have a field to store the integer label of the bed for the patient, with -1 used if the patient is not assigned a bed. Also, the patient should have a linked list to store all the doctors of the `Patient`. When a patient is created, she/he is not in a bed, and has no doctors. This class will need methods to handle the two fields, and the toString() method should be overridden to output the bed label (if any) and the name of each doctor of the `Patient`. Note that in the toString() method, if you include all the information for a doctor of a patient, you might end up with an infinite loop.

`Doctor` class:

The `Doctor` class is a descendant of the `BasicDoctor` class. A `Doctor` is to have a linked list of `Patient`s and methods to handle this list. Initially a doctor has no patients.

`Surgeon` class:

The `Surgeon` class is a special type of `Doctor`. This class will be kept very simple by only having a constructor, and overriding the toString method to include “Surgeon” on the line prior to the line with the name of the doctor.

Modifications to `Ward` class:

The `Ward` class from Assignment 2 should be modified so that the array has type `Patient`, rather than `Person`. When this change is made, a number of the methods will need to be changed in order to be consistent with type `Patient` being stored in the array. In addition, two methods need to be added to the `Ward` class. The first is to obtain a list of the empty beds in the ward. The second one is to remove a `Patient` from a specific bed. These changes are to be done to the existing `Ward` class, i.e., you are not define a descendant of `Ward`.

`HospitalSystem` class:

The last class to write is one to run a simple hospital system. To keep it simple, the system will have only one ward. The class should also have two containers: one for all the patients known to the system, and the other for all the doctors known to the system. Both containers should be keyed dictionaries, with the key for a patient being their health number, and the key for a doctor being her/his name. When the system starts, there are no patients and no doctors. Also at the start, the name of the ward, and the integer labels for its first and last beds should be read into the system in order to initialize the ward. During the running of the system, the system should display a message to the user for the user to select a task. When a task is selected, it should be carried out, and then another task selected. It is easiest to handle task selection is by numbering the tasks and having the user enter an integer.

Note that when a value is read using Scanner, other than by nextLine(), none the characters after the value are read. Thus, a subsequent nextLine() read will read those following characters up to and including the next end-of-line character(s). Such a read often just reads end-of-line character(s) that mark the end of the line after the previous read, so that the nextLine method simply returns the empty string. It does not read the characters on the next line (which might be what was wanted), since it finds end-of-line characters first. So be careful, and use the debugger!!

In the full system, the following tasks would be supported:

1. quit
2. add a new patient to the system
3. add a new doctor to the system
4. assign a doctor to a patient
5. display the empty beds of the ward
6. assign a patient a bed
7. release a patient
8. drop doctor-patient association
9. display current system state

For this assignment, you only need to complete all tasks except for `5` and `7` (these will be done in later assignments). In each of the tasks, patients are identified by their health number, doctors by their name, and beds by their (external) integer label. When the user quits, the system should print out the system state at that time. Note that when the task is to add a new doctor, the user should be asked whether the new doctor is a `Surgeon` or not. If so, a `Surgeon` should be created. Note that as will be discussed in class soon, an object of a certain type can be assigned to a variable of an ancestor type. Thus, a `Surgeon` can be assigned to a `Doctor` variable, and a `Surgeon` can be placed in the dictionary of `Doctor`s.

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. Be sure to include a meaningful error message in the String parameter for the exception constructor. Note that these additional comments and precondition checks have already been added to the Person and `BasicDoctor` classes of Assignment 2, but they should be added into the `Ward` class, as well as the new classes. When appropriate, exceptions should be caught and handled. In particular, if a task of the system fails and as a result throws an exception, it is reasonable to print the message of the exception, and then try the next task. The system should not crash when a user enters invalid data. To achieve that, a try-catch will be needed for the invocation of any method to handle a task that might throw an exception. These try-catches will probably be in the system class, as it is the one that has the invocations of the methods in the entity classes. Note that you might be able to handle all situations with only a couple try blocks.

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. Be sure to include tests in the main method for each of the new entity classes (`Patient`, `Doctor`, and `Surgeon`), and additional tests in `Ward` for its new methods. 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. However, in your system class, you will probably have a method with a switch statement to determine which task was selected by the user. When there are many cases, this method can get fairly long. However, as much of the code as possible (other than the actual switch statement that is determining the choice made) should be abstracted into other methods that are invoked from the switch statement.

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. This has already been done for the classes of Assignment 2. When appropriate, methods should be supplied to access and set the fields. It is anticipated that there might be many types of specialists (descendants of `Doctor`). Thus, the list of patients for a doctor might be set `protected`, so that descendant classes can directly access this list.

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 100)

(4) Readability

(8) UML class diagram

(6) Good use of object-oriented style

(57) Correct design

- (3) `Surgeon`
- (7) `Doctor`
- (11) `Patient`
- (10) `Ward` improvements
- (26) `HospitalSystem`

(15) Correct output

(10) Submit 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.