Assignment 4

Introduction

In this exercise you will create a complex system for doctors, medicines, prescriptions and patients using the classes you have written in previous assignment 2 and 3.

To do this effectively you're going to need some new classes. In addition, the task will require you to do some extensions in the classes you have already created. Depending on how you've solved assignments 2 and 3 you may also need to make other changes in your classes so that they act as task requests.

Especially in section D there will be many elements that will be connected. Here you should spend time looking at the design of the main program and asking yourself questions about the next steps, for example: Can this part of the program be simplified by placing it in a method (for example, to find out if a doctor exists)? How can one easily check if the user's input is valid?

If you do assumptions about the assignment text, it is expected that you explain these carefully.

Part A (there is no part A, starts from part B)
Part B: The class Patient

B1: Write class Patient.
A patient is a typical user of prescriptions. The patient has a name and a dateOfBirth-text string. When a new patient is registered, this patient must also receive a unique ID. Patients also have a list of prescriptions they've got conscripted. Since the patient often will use a prescription shortly after it is printed, we use a *Stabel <Prescription>* to save the patient's prescriptions. It should both be possible to add new prescriptions and extract the whole prescriptionlist.

B2: Change classes that take an *int pasientid* to take a *Patient patient.*

Part C: Itererbare lists

To easily run through our lists, we will make sure they are itererbare. This should be done "from the top" by modifying the interface from Liste <T> so that it extends the Java interface *Iterable <T>* . It is important to distinguish between the interfaces *iterable* and *iterator.*

iterable is an interface in Java used by our lists to make them itererbare. This implementation allows us to write a for each-loop that runs through our list. The interface looks like this:

```
interface Iterable < T > {
    Iterator < T > iterator ();
}
```

iterator is another interface which describes the *iterator object* we use to go through the list. Since all lists are not the same, it needs a separate Iterator class that implements *Iterator <T>* , specially-adapted to our lists. This interface looks like this:

```
interface Iterator < T > {
   boolean hasNext ();
   T next ();
   void remove ();
}
```

One advantage of the implementation of this is that we get access to use for-each-notation:

```
for   ( E e : itemlist ) {
      // do something with e ...
}
```

... which is actually the short form of "as long as the iterator finds a next element in the list (hasNext () returns true), extract it (next ())."

C1: Make sure the Liste <T> interface extends Iterable <T>.

C2: Write class *LenkelisteIterator (Hint: This does not own type parameter)* that implements *Iterator <T>* and thus methods *boolean hasNext* and *T next* . Method *void Remove ()* is voluntary and does not need to be implemented.

Hint: If Node class is an internal class in Lenkeliste<T> should iterator class also be it!

C3: Expand the class Lenkeliste <T> with the method *Iterator iterator ,* which returns a new LenkelisteIterator-object.

Part D: Class Doctor

Later in the task we want to be able to sort doctors.

D1: Expand class Doctor so that it implements interface *Comparable <Doctor>* and hence the method *compareTo.* Doctors should be able to be sorted alphabetically by name, so that a doctor by the name "Dr. Paus "comes before (that is less than)" Dr. Ueland ".

D2: class Doctor should also be able to keep track of what prescriptions it has written. Extend the class with an instance Lenkeliste<*Prescription*> *printedPrescriptions* and functionality to bring out this list of prescriptions.

D3: Doctor should have methods to create instances of the four Prescription-classes you can make instances of (White prescription, p-prescription, military prescription and blue prescription). When a prescription object is created, it must be added into the list of the doctor printed-prescriptions, before they are returned.

Method-signatures should look like this:

```
public WhitePrescription  writeWhitePrescription ( Medicine
medicine, Patient patient, int reit) throws illegalPrinting;

public  MilitaryPrescription  writeMilitaryPrescription ( Medicine
medicine, Patient  pasient, int reit) throws  illegalPrinting ;


public P-Prescription writeP-Prescription ( Medicine medicine,
Patient patient) throws illegalPrinting ;
```

```
public  bluePrescription writeBluePrescription ( Medicine medicine, Patient patient, int reit)
throws illegalPrinting ;
```

If an ordinary doctor trying to prescribe a narcotic medicine, it throws exception *illegalPrinting* :

```
Public class illegalPrinting extends Exception{
    illegalPrinting(Doctor 1, Medicine 1m){
        super("Doctor" + 1.hentNavn() + "is not allowed to print" + 1m.hentNavn());
  }
}
```

(this class must also be added to your answer)

Specialists can always print Narcotic medicine.

Hint: You can check whether a medicine is Narcotic by using *instanceof* operator.

Part E: Doctorsystem

You should now program the actual doctorsystem. The program will keep track on multiple lists with information on medicines, prescriptions, doctors and patients. This means that you have to think through what happens when new objects that depend on other objects are added.

Doctorsystem should make use of the lists you wrote in last assignment. You choose the structure of doctorsystem, as long as it meets the requirements of subtasks. Where the objects can be identified both with a unique ID and name (for example, when we are going to find a medicine to create a prescription) choose yourself what is most appropriate.

E1: Write a method to read the objects from the file. Follow the file format in Appendix 2. Use *writePrescription*- methods in doctorobject to create Prescription objects. If an object is invalid, it should not be entered into the system.

Hint: Remember to treat exceptions that can be thrown.

The file format is given by appendix 2 .

Note: If the file format should be correctly it is important that your unique ID counters start at 0.

E2: Make sure the user is presented with a command-loop that runs until the user chooses to exit the program.

Command-loop will present the following options:

- Printing a complete list of patients, doctors, medicines and prescriptions (subtask E3).
- Creating and adding new elements to the system (subtask E4).
- Use a given prescription from the list of a patient (subtask E5).
- Printing various types of statistics (subtask E6).
- Writing all data to file (subtask E8).

E3: Implement functionality to print a neat overview of all elements of the doctorsystem. Doctors shall printed in *ordered sequence* .

E4: Add functionality to allow user to add a doctor, patient, prescription or medicine. Prescriptions should be created through the Doctor's writePrescription *()*. Make sure you check if it is possible to create the desired object before it creates - for example, should not be allowed to make a prescription without a valid prescribing doctor. If the user provides invalid information, they must be informed of this.

Hint : To find out if the data provided is valid, we should use the iterator we created and search after them in the relevant lists!
You should also make sensible type checks along the way- for example, the program should provide an error message and return to the main menu if a user attempts to enter anything else than a number amount of medicine- but this is not a requirement. *Hint :* Catch up *NumberFormatException* if needed!

E5: Add the possibility of using a prescription. Illustration of the proposed interaction with the user (from the user has indicated that they want to use a prescription) you will find at the bottom of the task

(attachment 1) .

E6: Create the ability to view statistics about the elements in the system. This may for example be presented as a "sub-menu" of the user-menu. The user should be able to see the following statistical information:

- Total number of printed prescriptions on addictive medicine.
- Total number of printed prescriptions on narcotic medicine.
- Statistics about possible misuse of narcotica should appear as follows:
  - List the names of all the doctors (in alphabetical order) who have written at least one prescription of narcotic medicine, and the number of such prescriptions per doctor.
  - List the names of all patients who have at least a valid prescription for narcotic medicine, and for these, print number per patient.

E7: Add functionality to select the type of prescription that will be created under the command loop.

E8: Allow the user to write all elements of the current system to file. file should be formatted in the same way as the input file-example from the previous subtask. You do not need to store the items ordered by ID, but note that if you choose to do so can you *read from the same file that you write to* .

Summary

You will deliver the classes that make up the main program and all classes required for the main program to work (including both modified and unmodified classes of assignment 2 and 3).

Attachments

1) Suggested interaction for prescription use:

```
Which patient will you see prescriptions for?
0 : Anne ( dob 12121212121)
1 : Johnny ( dob 32323232323)
> 1

selected patient : Johnny ( dob 32323232323 ).
Which prescription will you use?
0 : Prozac ( 3 REIT)
1 : Ibux ( 2 REIT)
2 : Tylenol ( 0 REIT)
> 1

Used prescription of Ibux .Quantity remaining reit : 1

Main menu:
[...]

Which patient will you see prescriptions for?
0 : Anne ( fnr 12121212121)
1 : Johnny ( fnr 32323232323)
> 1

elected patient : Johnny ( fnr 32323232323 ).
Which prescription will you use?
0 : Prozac ( 3 REIT)
1 : Ibux ( 1 REIT)
2 : Tylenol ( 0 REIT)
> 2

could not use prescription Tylenol ( no remaining REIT ).

Main menu:
[... ]
```

appendix 2

2) File format for reading file:

```
# Patients (name, dob)
Jens Hans Olsen, 11111143521
Petrolina Swiq, 24120099343
Sven Svendsen, 10111224244
June Olsen, 21049563451
# Medicine (name, type, price, active substance, [strength])
Predizol, narcotic, 450,75,8
Paralgin Forte, addictive, 65,400.5
Placebo Pianissimo, common, 10.0
Ibux, common, 240.200
# Doctors (name, control ID / 0 if regular doctor)
Dr. Cox, 0
Dr. Wilson, 0
Dr. House, 12345
Dr. Hillestad Lovold, 0
# Prescriptions (medicineNumber, doctorName, patientID, type, [reit])
1, Dr. Cox, 2, white, 7
2, Dr. Hillestad Lovold, 3, blue, 1000
0, Dr. House, 1, military, 12
3, Dr. Hillestad Lovold, 3, p,
```