Software Engineering and Programming Basics - WS2023/24
Instructions for the Mandatory Assignment

TECHNISCHE UNIVERSITÄT CHEMNITZ

Professorship of Software Engineering

10| 2023

General Instructions

The assignment is a mandatory task that you need to complete over the course of the semester. Please read all the instructions carefully. Submissions that do not adhere to the instructions will be graded with 5.0.

- You need to create a Java-Project which contains three packages with the following names:
  - interfaces
  - classes
  - tests
- Download the interfaces.zip and tests.zip folders from OPAL. Add the downloaded interfaces to the package *interfaces*. You are not allowed to make any changes to the interfaces.
- Add any classes and enums which are required to run your program to the package *classes*. When creating classes, you must adhere to the class names specified in this document.

  Here is an overview of the required classes you need to implement: Animal, Assistant, Case, Doctor, Person, List, Node, Veterinary, AnimalKind, Treatment
- You are generally allowed to make additions, such as additional classes and methods which are not specified within this document. Just note that additional methods should not be added to the interfaces.
- You are allowed to use classes and methods from the Java standard library that are introduced in the Bonus Lecture Video "Important Java Classes". This includes the classes java.lang.Object, java.lang.String, java.lang.StringBuilder, java.lang.Math, and java.util.Arrays. **You are not allowed to import any other classes, unless explicitly stated otherwise.** There are the following two exceptions:
  - You are allowed to import the class *java.util.ArrayList* within the class *Veterinary*.
  - You are allowed to import additional classes for testing purposes within the package *tests*.
- Note that you are not allowed to use any pre-made data structure (such as java.util.ArrayList) to create functionality for your own *List* class in any way, shape or form.
- You are *not* allowed to use any third party libraries!
- Your program needs to pass the given unit tests from the tests.zip folder provided via OPAL without fail. Add the unit tests to your *tests* package. A few more notes on these tests:
  - These tests also function as examples for the required functionality of methods, thus you should not make any changes to them! You can, however, add more tests of your own, including modified copies of the provided unit tests.
  - We recommend that you add the tests to your project *after* you have created all the required classes, to avoid your IDE showing a lot of errors.
  - You need to add JUnit5 to your project in order for the tests to work. Please follow the instructions of your IDE.
  - If you have any troubles with getting the tests to run, please do not hesitate to contact the instructors for help. We will also cover how unit tests work at a later point in the exercises.
- Since this is an examination, you need to solve the task on your own. You can talk to other people about it, however it is not allowed that someone else writes the program for you or that you copy code from each other. Each submission will be checked for plagiarism! If plagiarism is found, *all* of the concerned submissions will be graded with 5.0, regardless of which one was the original!

- You are allowed to use code snippets that you find online. However, to avoid being flagged for plagiarism (in case someone else finds and uses the same code snippet), you need to clearly indicate which parts of your code have been copied by stating the source in a comment in the code.
- You are also allowed to use AI-based tools such as ChatGPT. Like with code from other sources, you need to clearly indicate any part of your code which was created with the assistance of such tools, by stating it in a comment in the code.
- You need to submit your code as .java files. No other type of file will be accepted.
- You need to submit your complete *classes* package. You can also submit your *tests* package, but it is not mandatory.
- In addition to your source code, you will need to submit a **short documentation**. Details follow below.
- Submit your Source Code and your Documentation in OPAL in the "Assignment Submission" course node.
- Deadline for submissions is the **02.02.2024**.

If you have any questions, please post them in the corresponding thread in the OPAL-Forum! Since this is an examination, it is important that all students have access to all information. If you have a question, it is very likely that someone else has the same question as well, thus it benefits everyone if you post your question directly in the forum! Additionally, please also check the forum first if maybe your question has already been answered there.

Grading

Minimal requirements to pass

- Your submission needs to contain all required classes *and* the required documentation.
- The program needs to compile and run.
- The program needs to pass the basic Unit Tests that are provided in the tests.zip folder in OPAL.
- The program needs to fully implement the given interfaces and there must not be any changes to the interfaces.
- The program needs to pass the plagiarism check.

If any one of these points is not fulfilled, the submission will be graded with 5.0!

Further grading criteria

- Functionality (60%)

  Including edge cases that are not covered by the provided unit tests.
- Clean Code (30%)
  - Use appropriately describing identifier names for variables, methods and classes.
  - Keep your methods and classes a sensible size.
  - Avoid code duplication.
  - Make sensible use of comments.
  - In general: Write code that is easily readable.
- Documentation (10%)

Task Description

Your task is to program a simulation of an animal clinic or ”veterinary”.

User Stories

We have formulated some user stories to help you get an idea of what the processes in the veterinary look like that your code needs to represent:

- When a new patient arrives, they need to be registered and then get into the queue in the anteroom.
- Once it is their turn, the patient will be examined by a doctor who has the necessary qualification for the kind of animal the patient is. With the help of some assistants, the patient will be diagnosed and treated, which will take a certain amount of time.
- For a smooth process, there always should be one assistant present in the anteroom. If there are less assistants present in the veterinary than doctors, all treatments will take double the time.
- The clinic will also take emergency cases, which take priority over normal cases.

Required Classes

- Animal
- Assistant
- Case
- Doctor
- Person
- List
- Node
- Veterinary
- AnimalKind
  which can take the following values:
  - dog, cat, bird, reptile, rodent
- Treatment
  which can take the following values:
  - vaccination, injury, diagnostics, emergency

In the following, we explain in detail the requirements for each of the required classes and methods:

Person

The class **Person** implements the interface **I__Person**. It contains the following methods:

```java
public Person(String name, int age, String address)
```

This constructor creates an Object of the class Person.

```java
public String getName()
```

Returns the person's name.

```java
public int getAge()
```

Returns the person's age.

```java
public String getAddress()
```

Returns the person's address.

Doctor

The class **Doctor** extends the class **Person** and implements the interface **I__Doctor**. It contains the following methods:

```java
public Doctor(String name, int age, String address, AnimalKind[] authorizedFor)
```

This constructor creates an Object of the class Doctor.

```java
public AnimalKind[] getAuthorizedFor()
```

Returns an array containing the kinds of animals the doctor is allowed to handle.

```java
public boolean addAuthorizedAnimal(String AnimalKind)
```

This method receives a kind of Animal represented as a String. If the String is a valid kind of animal, it is added to the animals the doctor is allowed to handle. There should be no duplicate animals. If the animal is added successfully, return *true*, else return *false*.

Assistant

The class **Assistant** extends the class **Person** and implements the interface **I__Assistant**. It contains the following methods:

```java
public Assistant(String name, int age, String address)
```

This constructor creates an Object of the class Assistant.

Animal

The class **Animal** implements the interface **I_Animal**. It contains the following methods:

```
public Animal(String name, int age, double weight, String gender)
```

This constructor creates an Object of the class Animal.

```
public Animal(String name, int age, double weight, String gender,
              String [] preexistingConditions)
```

This constructor creates an Object of the class Animal where there are already preexisting conditions to consider.

```
public void setWeight(double weight)
```

Sets the animal's weight to the given value.

```
public boolean setAnimalKind(String kind)
```

This method receives a kind of Animal represented as a String. If the String is a valid kind of animal, it is set as the animal's kind and the method returns *true*. If the String is not a valid kind of animal, return *false* and do net set it as the animal's kind.

```
public void addPreexistingCondition(String[] conditions)
```

This method receives an array of Strings which represent potential preexisting conditions. It adds the content of the given Array to the animal's preexisting conditions. Note: Should any of the given conditions already be listed within the animal's preexisting conditions, do not add it a second time.

```
public String getName()
```

Returns the animal's name.

```
public int getAge()
```

Returns the animal's age.

```
public double getWeight()
```

Returns the animal's weight.

```
public AnimalKind getAnimalKind()
```

Returns what kind of animal the animal is.

```
public String getGender()
```

Returns the animal's gender.

```
public String[] getPreexistingConditions()
```

Returns the animal's preexisting conditions.

Case

The class **Case** implements the interface **I_Case**. It contains the following methods:

```java
public Case(Person owner, Animal animal, Treatment treatment)
```

This constructor creates an Object of the class Case.

```java
public Person getOwner()
```

Returns the owner of the animal associated with the case.

```java
public Animal getAnimal()
```

Returns the animal associated with the case.

```java
public Treatment getTreatment()
```

Returns the treatment that has been prescribed in this case.

Node

The class **Node** implements the interface **I_Node**. It contains the following methods:

```java
public Node(Case currentCase)
```

This constructor creates an Object of the class Node.

```java
public Node getPrev()
```

Returns the predecessor of the node.

```java
public Node getNext()
```

Returns the successor of the node.

```java
public void setPrev(Node prev)
```

Sets the given node as this node's predecessor.

```java
public void setNext(Node next)
```

Sets the given node as this node's successor.

```java
public Case getCase()
```

Returns the case contained in the node.

List

The class **List** implements the interface **I_List**. It should realise a doubly linked list data structure to store and manage cases within the veterinary. Note: A lot of the methods for this class require you to work with indexes. You need to treat indexes like you would for arrays, meaning that the first Node of the List has the index 0. The class contains the following methods:

```
public List()
```

This constructor creates an Object of the class List.

```
public boolean append(Node node)
```

This method adds the given Node *node* at the end of the list. If the operation is successful, it returns *true*. Else it returns *false*.

```
public Node getHead()
```

Returns the Node at the head of the list.

```
public Node getTail()
```

Returns the Node at the tail of the list.

```
public boolean insert(int index, Node node)
```

This method inserts the given Node *node* at the given *index*. If the operation is successful, it returns *true*. Else it returns *false*.

```
public boolean remove(int index)
```

This method removes a node at the given *index* from the list. If the operation is successful, it returns *true*. Else it returns *false*.

```
public boolean swap(int first_index, int second_index)
```

This method swaps the node at the position *index_one* with the node at the position *index_two*. **Important note:** You need to swap the nodes themselves, not just their content! If the operation is successful, the method returns *true*. Else it returns *false*.

```
public void sort()
```

This method sorts the list so that emergency cases are moved to the front of the list. **Important note:** The order of the non-emergency cases, as well as the internal order of the emergency cases, should not be changed!
As an example: You have a list that consists of the emergency cases "E" and normal non-emergency cases "N" that looks as follows: N1 → E1 → N2 → N3 → E2 → N4
After using the sort-method, it should look like this: E1 → E2 → N1 → N2 → N3 → N4

```
public int findIndex(String ownerName)
```

This method is given the name of a person as a String and returns the index of the case in the list, where the person with said name is the owner of the animal. If the owner can not be found in the list, return -1. Note: You can assume that all owner names are unique.

Veterinary

The class **Veterinary** implements the interface **I__Veterinary**. It also imports the class java.util.ArrayList. The Veterinary class contains the following methods:

```java
Veterinary(Doctor[] doctors, Assistant[] assistants)
```

This constructor creates an Object of the class Veterinary.

```java
public Doctor[] getDoctors()
```

Returns an array of the doctors that work in the veterinary.

```java
public Assistant[] getAssistants()
```

Returns an array of the assistants that work in the veterinary.

```java
public boolean addCase(Case newCase)
```

This method adds a new case to the list of cases in the veterinary. If there is no doctor present in the veterinary that can handle the kind of animal the case is about, return *false* and do not add the case. If the case is added successfully, return *true*.

```java
public List getCaseList()
```

This method returns a list of all cases that are currently in the veterinary.

```java
public boolean removeCase(String ownerName)
```

This method receives the name of an animal owner as a String. If there is a case in the veterinary that is associated with this person, remove the case from the veterinary. If a case is successfully removed, return *true*, else return *false*.

```java
public void printMostUrgentCases(int k)
```

This method prints the most urgent cases in the veterinary up to (and including) the given index *k*. It should print the cases in the following format: "ownerName, animalName, treatmentKind, standardTreatmentTime". Notes: If *k* is an invalid index, print an appropriate error message instead. For the standard treatment times, see the description of the method below.

```java
public ArrayList<String> execute()
```

This method processes all the currently registered cases in the veterinary. Note: Emergency cases take priority over other cases. Otherwise cases are handled in their order of arrival.
Each case needs an available doctor. If all doctors are currently occupied, the next case in the list needs to wait until a doctor has finished their previous case. If there are assistants available, doctors will also require the help of one assistant. Note: Since one assistant of the veterinary should always stay in the anteroom, treatment processes will only be smooth if more assistants than doctors are present in the veterinary. Thus, if there are less or only an equal amount of assistants present compared to doctors, all treatment times will be doubled.

The standard times for treatments are:

vaccination: 15 minutes
injury: 30 minutes
diagnostics: 45 minutes
emergency: 60 minutes

The method should return a protocol of the executed cases. To do this, it returns an ArrayList of Strings. Each String protocols one case, by detailing the name of the treated animal's owner, the name of the treated animal itself, the name of the doctor working on the case, the kind of treatment that was received, as well as the starting time of the treatment and the finishing time of the treatment. The resulting String should be formatted as follows:

"$ownerName with $animalName was treated by $doctorName, $treatment, started at: $startingTime ended at: $finishingTime"

Notes: Replace the variables starting with a $ with the concrete values of each case. There should be no linebreak within the String (see the corresponding Unit Test for clarification).

The starting time of the execution is 0. Once a doctor has finished a case, they will go on to the next case in line and start it in the following minute. For Example: If there is 1 doctor present (as well as a sufficient amount of assistants) and the first case is a vaccination, the treatment will start at 0 and finish at 15. The second case will start at 16.

Once a case is finished, it should be removed from the waiting list. Thus, when the execution method is finished, there should be nor more cases remaining in the veterinary.

Documentation

Finally, you need to submit a documentation on your code. It should be in a suitable format (.pdf) and at maximum **5 pages** (or 2500 Words). Its contents should be the following:

- A rough description of each class and how the classes interact with each other (maximum 1 page or 500 words). Include a UML class diagram to support your explanation.
- Explain for each of the required classes which attributes you used and why.
- A short explanation for each additional method you created that was not given by the interface. Explain in 1 or 2 sentences what the method does/is needed for.
- A short explanation of how your implementations of the following methods work:
  - Animal.addPreexistingCondition
  - List.swap
  - List.sort
  - Veterinary.printMostUrgentCases
  - Veterinary.execute