OOP – Laboratory 2 Report

GitHub Repo: LucaFranceschi01/UPF-POO22-G101-07

Introduction:

This laboratory consisted of simulating a university with many classes and read the information of all these classes from .xml files. In our approach to the exercise, we created the following classes:

Test University. This class was created basically for testing the university, it contains the main method where we create a new university.

University. This classed was the global class that had all the others inside it. This class stores 4 types of Linked Lists (each one for students, teachers, classrooms and courses). We read the information from the .xml files following this method for each of the classes mentioned above:

```
LinkedList<String[]> std = Utility.readXML(type: "student");
for(String[] array : std) {
    Student st = new Student(array[0], Integer.parseInt(array[1]));
    students.add(st);
}
```

After that, we also created Linked Lists of String[] for lectures, assignments and enrollments. We read the information from the .xml files following this method for each one of this 3 classes:

```
    LinkedList<String[]> lec = Utility.readXML(type: "lecture");
    LinkedList<String[]> enr = Utility.readXML(type: "enrollment");
    LinkedList<String[]> ass = Utility.readXML(type: "assignment");

    for(String[] arr : lec) {
        Lecture l = new Lecture(arr[4], Integer.parseInt(arr[2]), Integer.parseInt(arr[3]));
        Classroom cl = Utility.getObject(arr[0], classrooms);
        Course co = Utility.getObject(arr[1], courses);
        l.addClassroom(cl);
        l.addCourse(co);
        cl.addLecture(l);
        co.addLecture(l);
    }
```

Note that in the last image we can see that apart from reading the lecture to its linked list, we also added the classroom and the course to it, as well as we added the lecture to classroom and course. By doing it this way we ensured that each course and classroom had the lecture information added as well. We did the same with assignments and enrollments, adding its information to the classes necessary.

Apart from reading, in the University class we also have 3 other types of methods. The first type are getter methods that give us the information of the Linked Lists of their special class as Linked Lists of string arrays, by using the Utility.toString() method. The second type are getter methods that simply return the Linked Lists read in University. Lastly, the third type of methods in University are the ones used for the queries (including the extra ones), in which we applied overriding/delegation,

by calling from the inside of the method to other methods that were in their specific classes to do the tasks and return the value to University and finish the University method. An example of this would be the coursesOfStudents() method:

```
public void coursesOfStudent(Student s) { // propagates Student.coursesOfStudent() and prints it in a legible way for all students
    LinkedList<Course> coursesOfStud = s.coursesOfStudent();
    System.out.printf(format: "Name: %-22s | Nia: %-10d | Courses: %s%n", s.getName(), s.getNia(), coursesOfStud.toString());
}
```

This part of the code corresponds to the University class, where we call Student.coursesOfStudent():

```
public LinkedList<Course> coursesOfStudent() { // for all enrollments, returns a list of courses
    LinkedList<Course> courses = new LinkedList<Course>();
    for(int i=0; i<enrollments.size(); i++) {
        courses.add(enrollments.get(i).getCourse());
    }
    return courses;
}
```

This is the Student's class method, in which we create a courses Linked List and store there the information of the courses of the Student and then return the Linked List. All the other queries in this laboratory have been made using the same way of thinking and procedure, considering the concrete task asked in each query.

Student. This class basically stores the information of students, more precisely with the attributes: name and nia. The constructor of Student receives a name and a nia and initializes the Linked List of enrollments as empty. It also has a Linked List of Enrollments of the student. In terms of methods, this class has all the getters for its attributes, an addEnrollment(Enrollment e) to add more enrollments to the Linked Lists, and then its function for the toString() that returns the name. The methods needed for the queries that involve the class Student are also here. For instance:

```
public LinkedList<Course> coursesOfStudent() { // for all enrollments, returns a list of courses
    LinkedList<Course> courses = new LinkedList<Course>();
    for(int i=0; i<enrollments.size(); i++) {
        courses.add(enrollments.get(i).getCourse());
    }
    return courses;
}
```

Teacher. This class basically stores the information of teachers, more precisely with the attribute name and a Linked List of Assignments of the teacher. The constructor of Teacher receives a name and initializes the Linked List of assignments as empty. In terms of methods, this class has all the getters for its attributes, an addAssignment(Assignment a) to add more assignments to the Linked Lists, and then its function for the toString() that returns the name. The methods needed for the queries that involve the class Teacher are also here. For instance:

```
public LinkedList<Lecture> lecturesOfTeacherInClassroom(Classroom c){
    LinkedList <Lecture> lectures = new LinkedList<Lecture>();
    for(Assignment a : assignments){
        for(int i=0; i<a.getCourse().getLectures().size(); i++){
            if(a.getCourse().getLectures().get(i).getClassroom().toString().equals(c.toString())){
                lectures.add(a.getCourse().getLectures().get(i));
            }
        }
    }
    return lectures;
}

public LinkedList<Lecture> lecturesOfTeacher(){
    LinkedList <Lecture> lectures = new LinkedList<Lecture>();
    for(Assignment a : assignments){
        for(int i=0; i<a.getCourse().getLectures().size(); i++){
            lectures.add(a.getCourse().getLectures().get(i));
        }
    }
    return lectures;
}
```

Classroom. This class basically stores the information of classrooms, more precisely with the attribute code and a Linked List of Lectures of the classroom. The constructor of Classroom receives a code and initializes the Linked List of lectures as empty. In terms of methods, this class has all the getters for its attributes, an addLecture(Lecture l) to add more lectures to the Linked Lists, and then its function for the toString() that returns the code. The methods needed for the queries that involve the class Classroom are also here. For instance:

```
public LinkedList<Course> coursesOfClassroom() { // from all classrooms, returns a list of courses
    Set<Course> courSet = new HashSet<Course>(); // for no duplicate courses
    for(int i=0; i<lectures.size(); i++) {
        //check if course is already held in classroom independently of group/slot/type
        courSet.add(lectures.get(i).getCourse());
    }
    return new LinkedList<>(courSet);
}
```

Course. This class basically stores the information of courses, more precisely with the attribute name and 3 Linked Lists: one for the enrollments, another for the assignments and another for the lectures of the course. The constructor of Classroom receives a name and initializes the Linked List of lectures, assignments and enrollments as empty. In terms of methods, this class has all the getters for its attributes, has an add method for each 3 of the Linked List, and then its function for the toString() that returns the name. The methods needed for the queries that involve the class Course are also here. For instance:

```
public LinkedList<Teacher> teachersOfCourse() { // from all assignments, returns a list of teachers
    LinkedList<Teacher> teachers = new LinkedList<Teacher>();
    for(int i=0; i<assignments.size(); i++) {
        teachers.add(assignments.get(i).getTeacher());
    }
    return teachers;
}
```

Enrollment. This class basically stores the information of enrollments, more precisely with the attributes: seminarGroup, course and student. The constructor of Enrollment receives a seminarGroup. In terms of methods, this class has all the getters for its attributes, an addStudent(Student s) to add a student, an addCourse(Course c) to add a course, and then its function for the toString() that returns the seminarGroup.

Assignment. This class basically stores the information of assignments, more precisely with the attributes: groups (as a String[], so as an array), course and teacher. The constructor of Assignment receives the array groups. In terms of methods, this class has all the getters for its attributes, an addTeacher(Teacher t) to add a teacher, an addCourse(Course c) to add a course, and then its function for the toString() that returns the groups using **Arrays.toString(getGroups())**.

Lecture. This class basically stores the information of enrollments, more precisely with the attributes: group, timeSlot, type, course and classroom. The constructor of Lecture receives a group, a timeslot and a type. In terms of methods, this class has all the getters for its attributes, an addClassroom(Classroom c) to add a student, an addCourse(Course c) to add a course, and then its function for the toString() that returns the timeSlot using **String.valueOf(timeSlot)**. The methods needed for the queries that involve the class Lecture are also here. For instance:

```
public LinkedList<Student> getStudentsOfGroup(){
    // from any lecture group, return the students of that group. Examples:
        // Group 1: if seminar is 111, 112, 121, 122.
        // Group 11: if seminar is 111, 112.
        // Group 111: if seminar is 111.
    LinkedList <Student> students = new LinkedList<Student>();
    for(Enrollment e : course.getEnrollments()){
        if(isSeminar() && e.getSeminarGroup().equals(group)) {
            students.add(e.getStudent());
        }
        if(isLab() && (e.getSeminarGroup().equals(group.concat(str: "1")) || e.getSeminarGroup().equals(group.concat(str: "2")))){
            students.add(e.getStudent());
        }
        if(isTheory() && (e.getSeminarGroup().equals(group.concat(str: "11")) || e.getSeminarGroup().equals(group.concat(str: "12")) ||
            e.getSeminarGroup().equals(group.concat(str: "21")) || e.getSeminarGroup().equals(group.concat(str: "22")))){

            students.add(e.getStudent());
        }
    }
    return students;
}
```

Alternative Solutions:

There are some things that we changed from the original class scheme, mainly because it was easier to implement, or we thought it would lead us to a more clean and readable code. For instance: setting the type of Classroom code to String instead of Integer, since we didn't have to do any operations with it, only display the value in some results; or using type Array instead of LinkedList of strings for the assigned groups of each teacher, since the implementation in the constructor of university was easier to implement (and it wouldn't affect performance).

We also used polymorphism, creating a toString() method in all our classes, since it was necessary to override the method to get the Utility.getObject() or Utility.toString() methods working.

For the queries, we created the necessary methods in the proper classes and then propagated them through University, where we created void methods to display the results in a readable way given the object needed (instead of a String representation as it was suggested). That way, we thought our code would be "more generic", and we could implement loops in TestUniversity to print tables of all students/teachers/etc. Doing it that way, we didn't need the methods given in Utility that much.

We are sure that there are many other ways in which queries could've been done, maybe some of them could've been clearer and more efficient, but those were the ones that we got figuring it out by ourselves.

Conclusions:

We can say that the solution worked as expected. We managed to initialize our classes and fill them up with the elements from the XML files provided, printed out the most meaningful information and created some queries to get even more information about the objects in a readable way.

We had some trouble implementing the queries because we couldn't see quite well how the encapsulation of those methods should work, where to place the methods, etc. and that can be seen in earlier commits of the repository. However, we could finally find a solution to most of them.

The only aspect in which we are concerned is that when we get the results of the ClassroomOfTeacher() query, we see that different lectures are held in the same timeslot and classroom and that one teacher gives different lectures in different classrooms at the same timeslot. Obviously, we think that is not possible in real life, but when we compare the results with the given XML files (assignments and lectures), we see that it is coherent with the data given, so it is supposedly implemented correctly.