OOP – Laboratory 3 Report

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

Introduction:

In this Lab we are asked to implement an organization management application where an Organization is composed of Headquarters that can be placed in different territories (namely Regions and Cities). Each Headquarter is composed of two types of members: Delegate and Regular members (for each Headquarter there is one Delegate that is the head). Each member of the organization has its own attributes and Availability. We are asked to populate the organization structure by using QR code verification. Moreover, an organization can perform a set of actions that each headquarter must sign up for. Finally, we are testing the organization structure to be sure that everything works as expected.

We'll start by explaining the more meaningful aspects of our project.

Organization class:

The Organization that we want to design has a name and two lists: one for headquarters and the other one for actions, which can be modified by adding new headquarters or proposing new actions. There are some aspects of our code that we may need to clarify:

Method *getHeads*: from all the headquarters of the organization, returns a list of all its corresponding heads.

Methods *addRegular*/*addDelegate*: very similar methods, they check if a new member can be added to the organization. Essentially, these methods are checking if an existing Delegate can sign up a new member (delegation of the task to the Delegate class, further explanation in the Delegate class section).

Methods *printMembers*/*printActions*: very similar methods, *printMembers* prints all members of each headquarter in a readable way and *printActions* prints all actions, including the headquarters that will perform those actions as well as the number of assisting members and their names. These two methods are used later, in the testing stage.

Headquarter class:

A Headquarter is composed of its name, email, a head delegate, the organization where it belongs, and lists of members that belong to it, actions that will be performed by the headquarter and cities where it is placed. There are some methods that need to be explained more deeply:

Methods *getDelegates*/*getRegulars*: from all members of a Headquarter, they check the type (Regular or Delegate) by checking the instance they belong to by using the *instanceof* keyword and return them as a list.

Method *signUpAction*: creates an InfoAction class instance from a specific action, while also adding a headquarter, the number of assisting and non-assisting members and press coverage. Then it links that information to the corresponding action and headquarter.

Methods *countAssistingMembers*/*getAssistingMembers*: in a very similar way, counts the number of members available by "asking" each of its delegates to compute its assisting developing members (calling *Delegate.countAssistingRegulars* method). The count method returns an integer, used later to propose actions. The get assistants method returns a list of all assisting members, which will be used to print a list of them in the testing stage.

Member class:

We have implemented the class Member, which is a superclass of Regular and Delegate. This way, we can create Regular and Delegate members that inherit attributes and methods from member. Every member has a name, a phone, an email, an availability and the headquarter it belongs to.

For this implementation we have used private attributes and protected methods. Using this visibility, the subclasses can access those attributes by using the methods provided. This type of implementation lets us to have cleaner code, achieve polymorphism and more encapsulation.

Method *checkAvailability*: Checks if the member is available for a specific action. That is done by trying to match the availability of the member with the starting date and hour of the action given as parameter.

Method *toString*: it can also be found in some other classes; we are overriding the *toString* method to achieve cleaner code and to be able to print linked lists of the Member class in an easy way.

Regular class:

The Regular class extends the Member class, inheriting attributes and methods from Member and adding to it a responsible delegate, alongside with a list of vehicles that the Regular can use as transport.

The most noticeable thing in this class is that when we call the constructor of Regular, we are calling the constructor of its superclass at the very beginning of it by using the *super* keyword.

Delegate class:

The Delegate class extends the Member class, inheriting attributes and methods from Member and adding to it a list of Regular dependents. It also contains a headquarter attribute, that indicates if the Delegate is the head of a headquarter or not (in that case it will be set to *null*).

As seen in the Regular class, at the beginning of the Delegate constructor there is a call to its super constructor. Also, if a Delegate is created in a headquarter that has no other members, it is set to be its head by default. There are some more methods to explain:

Methods *genDelegateQR*/*genRegularQR*: These methods generate and return an Image instance (namely QR codes) of the given Delegate/Regular. Depending on the type of the member, they assign different texts to the Image by using the QRLib class. When that is done, the BitMatrix and the bitmap of the image are set, and the image is returned.

Methods *signUpDelegate*/*signUpRegular*: These methods check if a given Delegate/Regular can be added using a specific Image. Essentially, tries to match the texts of the ideal QR code of the

given member and the given QR. If they are the same means that it can be created that way, so if that happens it returns *true.* If that is not the case, it returns *false*.

Method *proposeAction*: if the Delegate is the head of a headquarter it can propose an action to the organization and sign up its own headquarter to it. If the action is new to the organization, it is added to the list of actions that the organization has. This method will count how many of the members of the whole headquarter will be able to assist and will sign up the headquarter to the action.

Methods *countAssistingRegulars*/*getAssistingRegulars*: If a head of a headquarter proposes an action, every delegate of the headquarter needs to count how many of the members (regulars) he/she is responsible for will be able to assist to the action. That is done by calling the *checkAvailability* method for each one of the dependent members of the delegate.

Availability class:

Each member has some time slots during the week in which they are available for developing actions for the organization. Those time slots are the availability of each member. These time slots are defined as two linked lists of integers: one for the days the member is available and another one for the hours the member is available for each day in the days list.

The most meaningful aspect of this class is the constructor. The constructor has two parameters: day availability (a string with the days of the week separated by '.') and hour availability (a string of integer values separated by '.'). Using regular expressions, pattern matching and arrays, we can set up the attributes of the class, and transform the raw data read in the .xml files to lists of integers that are much easier to work with.

Action class:

An Action is an event that has a purpose, that is held in a specific time with a specific duration and that can be performed or not. The organization has a list of them, and when a head of a headquarter proposes it, an InfoAction is created and added to the list of existing InfoActions. That class will contain some information about how the action and the headquarter/s interact. The most significant methods that need explanation are:

Method isPerforming: from a given LocalDateTime, checks if the action is being performed using the starting LocalDateTime of the action and the duration of it. If the LocalDateTime given is in between the beginning and ending time of the action, it will return *true* (the action is being performed at that time). Otherwise, it will return *false.* In our design the duration of the actions is expressed by minutes, as we think it gives us more flexibility and it adapts well to the real world.

Method *getHeadquarters*: creates and returns a list of headquarters that will perform the action. This is done by going through the list of InfoActions and adding each headquarter to a list. This is done mainly to print results in the testing stage.

Method *getAssistingNumMembers*: given a headquarter, returns the number of members that will assist to the action specific to that headquarter. If there are none, it returns 0. This is done

by delegating the task of counting members to each one of the delegates of the headquarter, which will have to count all its dependent members.

InfoAction class:

InfoAction is an association class that manages all the information regarding a Headquarter and an Action. An InfoAction is created whenever a headquarter decides to perform an action and, from then on, the two instances of headquarter and action are linked by means of the InfoAction class.

An InfoAction contains all the relevant information about the performance of an action: the assisting members, the assisting non-members and if there's press coverage or not.

City class:

In our implementation a City is composed of its name, its population and a list of the headquarters that are placed into it. Whenever a Headquarter is created inside a City it is added to the list oh headquarters.

In this class we also override the *toString* method to be able to print cities in a readable way in the testing stage.

Region class:

In our implementation a Region is composed of a name and a collection of cities.

In this class we also override the *toString* method to be able to print regions in a readable way in the testing stage.

Vehicle class:

In our design a Vehicle has a certain capacity and has a type (car, motorcycle, bicycle, …).

Image class (Given):

Has a bunch of attributes and methods that allow us to create Images of a certain width/height and path, where later we will also save our texts by setting the BitMatrix to the desired values (using QRLib class). The most relevant and useful methods of this class are the constructor, the setBitmatrix and the getBitmap, which we all use in order to generate the QRs for the members.

QRLib class (Given):

Contains two methods for generating the QR code BitMatrix from a text or to decode a QR code BinaryBitmap into text. We use the generateQRCodeImage method to generate the QRs for the members, and we use the decodeQRCodeImage in order to check if the QRs contain the same information in the *signUpDelegate* and *signUpRegular* methods.

Utility class:

As in other laboratories, this class helps us when reading the XML files by giving us the methods of *readXML*, *getObject* and *toString* methods. The readXML allows us to read the information from files in an easier way, while the getObject allows us to get the specific object we want from its name.

TestDelegate class:

The TestDelegate contains the main method for our project and has attributes to keep track of the organization and the regions where each headquarter is placed. In this class there are methods to read the XML files to populate the organization with members, regions and headquarters. In the main method there is mainly testing of the functions, where after each test we print to Stdout the results.

Methods *readRegions*/*readHeadquarters*/*readHeads*: these methods, read the XML files in a very similar way to past laboratories using the given Utility class. Basically, they convert information from a text file into instances of classes to populate the structure of our organization.

Testing:

This testing is done on the main method of the TestDelegate class as already said above. More precisely, in this testing we test several features of the code:

Adding members: in this part of the testing, we will basically try all combinations possible to see if the members are being added or not being added correctly depending on the head/delegate that is trying to sign them up and the headquarter they are trying to be signed up to. As it is explained on the code, there is a first part where we print the original members of the organization (the heads of each headquarter given by the heads.xml). Then we have a section where we define 4 general availabilities that we will use for the members, and then we have also defined variables for all 11 heads and headquarters to make the code more compact and readable. Then we have the part where we add the members, we have specified a part where the members are added, and to which headquarter are added, and a part where the members aren't added and the reason for it. After all this, we print the final members of the organization.

Adding actions: in this part of the testing, we will try adding several newly created actions to the organization and proposing the actions to several headquarters. We have several parts: first we created 4 sample dates that we will use for the actions, and then we have 5 created actions with their purpose and their information. Lastly, we have the part in which we add the different actions to several headquarters, and then we print the result by using the method *printActions*, that prints the purpose of the action, the headquarters participating, alongside with the number of members participating and their names.

Alternative Solutions:

Our final solution is a program that allows you to create an organization with several features that allow you to increase the organization members through a QR checking process, and that allows the head of each Headquarter to propose new actions to the organization. It also allows to check in advance the availability of the members and the assistance, as well as if an action is being performed at a specified time. It also keeps track of all developed actions and their information. Our reason for choosing this implementation is because we thought it was the one with more logic behind, we tried to make it as close as possible to the implementation of this system in the real world. There were some methods that could have been understood or interpreted in other ways, but we chose this version. Having talked about the practical features of the code, it is also necessary to explain the

theoretical background that made this program possible. In order to create it, we made use of several coding tools as explained below:

Abstraction and encapsulation: By applying abstraction and encapsulation, we are trying to focus only on the essential and hide the irrelevant things for the general public in order to reduce the complexity of the code for someone that is trying to understand it from the outside. In our case, we tried to apply abstraction by grouping some methods together in a bigger method that is more understandable. For example, the method *printActions* does several things inside and calls some other methods but in the end, it just prints things, so the user will only see a print function that will help him/her to understand it better without having to worry about the concrete implementation.

Delegation: Delegation refers to calling other methods inside a method to do part of the job, is delegating the responsibility of doing a task to another object. In our case, we used delegation many times, for example in *proposeActions* where we called several methods to do some tasks: *Headquarter.countAssistingMembers* to get the number of participants and *Headquarter.signUpAction* to sign up the action (create it and add it to the proper lists).

Use of keywords: In Java there are some special words called *keywords* that fulfill some specific needs and are very useful to do some tasks. In this lab we used for example the words: *super* (used to access the superclass for either the constructor, the attributes of the methods of it), *instanceof* (that has value true or false depending if the object given is an instance of a specified class or not), *extends* (that is used for inheritance at the declaration of a class to denote that is a superclass of another class), *this* (that is used to refer to the current object in a method or constructor), etc.

Inheritance: Inheritance refers basically to the relation between two classes, in which one of them is the superclass (more general object) and the other one the subclass (more specific object). In this type of relation, the subclass inherits all the attributes and methods of the superclass and can use them as their own. In our case, we have implemented inheritance with the classes Member (superclass) and Delegate and Regular (subclasses). We can see this in the actual code by the *extends* keyword in the definition of the subclasses and by the *protected* visibility of the methods in Member so that only Delegate and Regular are able to access them.

Overriding: Basically, redefining an existing method in another class. In our case, we applied overriding on the *toString* methods on all classes to have their own specific one and not take the general one from the Utility class.

Casting: Casting is storing the same instance in a different variable of different declared type. In our case, we sometimes used Downcasting, which is making the instance to be a more specific instance (instance of the subclass). We can see this with examples of our code, where we use *(Delegate) m* or *(Regular) m* in the class Headquarters to get the Delegates or Regulars of that Headquarter.

Conclusions:

Our solution worked out in practice well. The tests in the TestDelegate part were all successful, doing exactly what they were supposed to. An example of this would be in the testing part related to adding new members to the organization. We see two cases in which the members we created can't be added to the organization due to a specific reason. They can't be added because they are trying to be added to a specific headquarter by the head of another headquarter, and thus that is not possible. This makes sense as in real life, you can not sign up to a headquarter by a member of another headquarter.

Another example would be the management of the availability for members and all the methods related to it, that checked which members were able to assist to different actions. We see in the testing that the participating members varies a lot, adjusting to each member's availability, as it should. Lastly, talking about the difficulties we encountered while designing our program, we found that the explanation of the objective of some methods was a bit ambiguous, so it took us some time to really understand the objective of those methods.