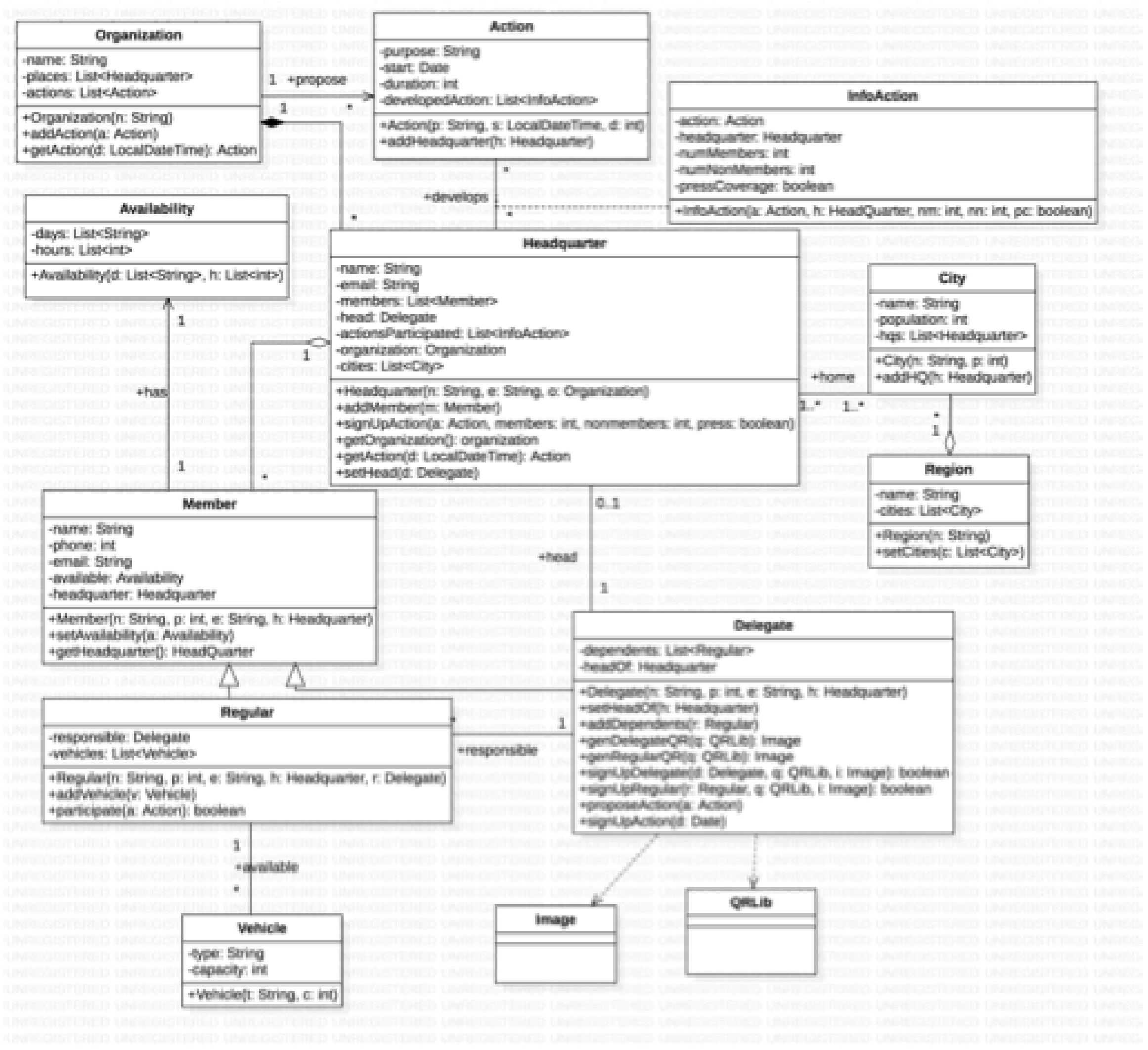
Laia Tomás
Daniela Quimis

Lab Session 3: Implementing an application using libraries and inheritance

Introduction

In this lab we implement the structure of the organization and the territory, seen in Seminar 3, also the functionalities that allow it to sign up new members. This lab session has similarities to Lab 2, but using different kinds of relationships and with different functions, following the given class diagram, we start designing our Lab 3.

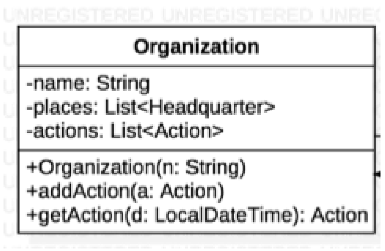

Organization
-name: String
-places: List<Headquarter>
-actions: List<Action>
+Organization(n: String)
+addAction(a: Action)
+getAction(d: LocalDateTime): Action
Action
-purpose: String
-start: Date
-duration: int
-developedAction: List<InfoAction>
+Action(p: String, s: LocalDateTime, d: int)
+addHeadquarter(h: Headquarter)
InfoAction
-action: Action
-headquarter: Headquarter
-numMembers: int
-numNonMembers: int
-pressCoverage: boolean
+InfoAction(a: Action, h: HeadQuarter, nm: int, nn: int, pc: boolean)
Availability
-days: List<String>
-hours: List<int>
+Availability(d: List<String>, h: List<int>)
Headquarter
-name: String
-email: String
-members: List<Member>
-head: Delegate
-actionsParticipated: List<InfoAction>
-organization: Organization
-cities: List<City>
+Headquarter(n: String, e: String, o: Organization)
+addMember(m: Member)
+signUpAction(a: Action, members: int, nonmembers: int, press: boolean)
+getOrganization(): organization
+getAction(d: LocalDateTime): Action
+setHead(d: Delegate)
City
-name: String
-population: int
-hqs: List<Headquarter>
+City(n: String, p: int)
+addHQ(h: Headquarter)
Region
-name: String
-cities: List<City>
+Region(n: String)
+setCities(c: List<City>)
Member
-name: String
-phone: int
-email: String
-available: Availability
-headquarter: Headquarter
+Member(n: String, p: int, e: String, h: Headquarter)
+setAvailability(a: Availability)
+getHeadquarter(): HeadQuarter
Delegate
-dependents: List<Regular>
-headOf: Headquarter
+Delegate(n: String, p: int, e: String, h: Headquarter)
+setHeadOf(h: Headquarter)
+addDependents(r: Regular)
+genDelegateQR(q: QRLib): Image
+genRegularQR(q: QRLib): Image
+signUpDelegate(d: Delegate, q: QRLib, i: Image): boolean
+signUpRegular(r: Regular, q: QRLib, i: Image): boolean
+proposeAction(a: Action)
+signUpAction(d: Date)
Regular
-responsible: Delegate
-vehicles: List<Vehicle>
+Regular(n: String, p: int, e: String, h: Headquarter, r: Delegate)
+addVehicle(v: Vehicle)
+participate(a: Action): boolean
Vehicle
-type: String
-capacity: int
+Vehicle(t: String, c: int)
Image
QRLib
+propose
+develops
+has
+home
+head
+responsible
+available
1
*
0..1
1..*

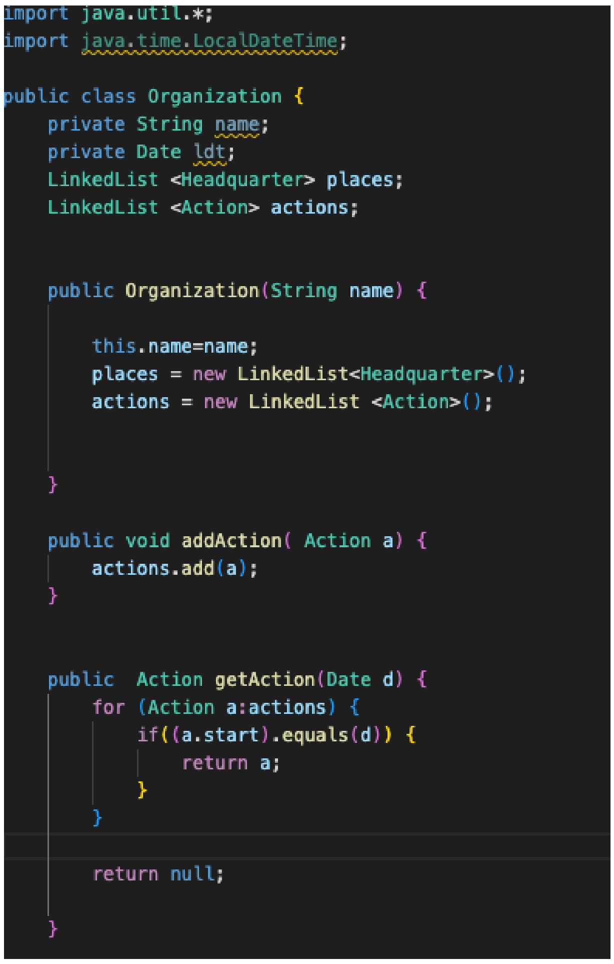
Laia Tomás
Daniela Quimis

Implementing the design

Let’s start relating our main classes, but first we need to design their class itself for each one of them:

ORGANIZATION CLASS


UNREGISTERED UNREGISTERED UNRE
Organization
-name: String
-places: List<Headquarter>
-actions: List<Action>
+Organization(n: String)
+addAction(a: Action)
+getAction(d: LocalDateTime): Action



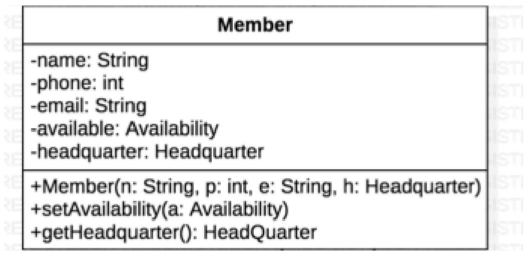
```
import java.util.*;
import java.time.LocalDateTime;

public class Organization {
    private String name;
    private Date ldt;
    LinkedList <Headquarter> places;
    LinkedList <Action> actions;


    public Organization(String name) {

        this.name=name;
        places = new LinkedList<Headquarter>();
        actions = new LinkedList <Action>();


    }

    public void addAction( Action a) {
        actions.add(a);
    }


    public  Action getAction(Date d) {
        for (Action a:actions) {
            if((a.start).equals(d)) {
                return a;
            }
        }

        return null;

    }
```

First we import the necessary libraries, that are (java.util.* and java.time.LocalDateTime) we see there’s a new type that is *Date*, that’s why we need to import a new library, so Organization main methods consist on <u>add</u> and <u>get</u> Actions. In ***getAction***, it’s required an argument of type Date (previously mentioned library), this will check if our wanted action coincides with our list of actions, if it does it’ll return it, otherwise it doesn't return anything.

Organization Class has a composition relation with Headquarter and association relation with Action.

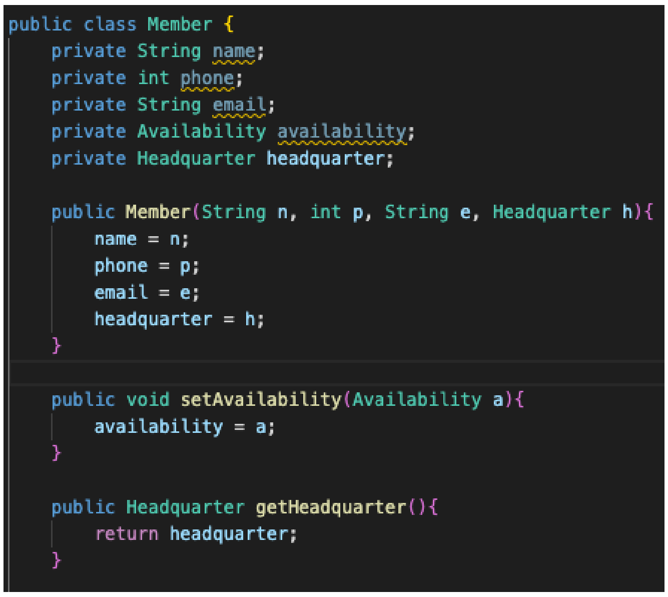
Laia Tomás
Daniela Quimis

MEMBER CLASS



Member
-name: String -phone: int -email: String -available: Availability -headquarter: Headquarter

+Member(n: String, p: int, e: String, h: Headquarter)
+setAvailability(a: Availability)
+getHeadquarter(): HeadQuarter

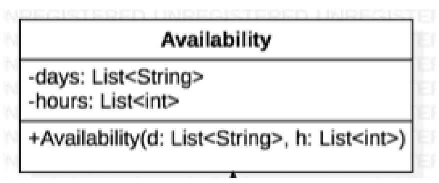


```java
public class Member {
    private String name;
    private int phone;
    private String email;
    private Availability availability;
    private Headquarter headquarter;

    public Member(String n, int p, String e, Headquarter h){
        name = n;
        phone = p;
        email = e;
        headquarter = h;
    }

    public void setAvailability(Availability a){
        availability = a;
    }

    public Headquarter getHeadquarter(){
        return headquarter;
    }
```

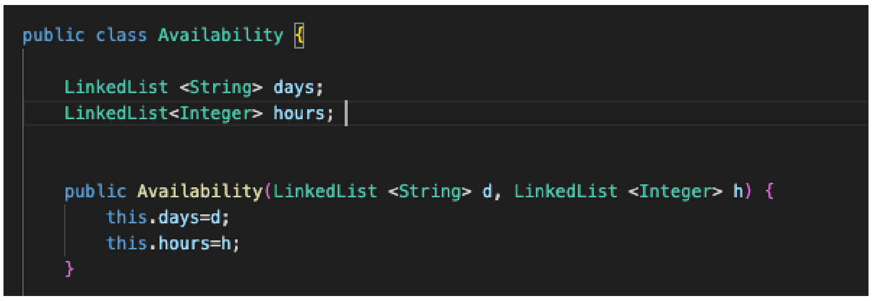
Member class contains the basic data of the members (name, phone, email….) and their main headquarters. Their methods are just setters for availability of each member and getters for which are their headquarters.

Member has an association relation with Availability , aggregation relation with Headquarter, and it’s a parent for Regular and Delegate (these classes will inherit from Member).

AVAILABILITY CLASS



Availability
-days: List<String> -hours: List<int>
+Availability(d: List<String>, h: List<int>)

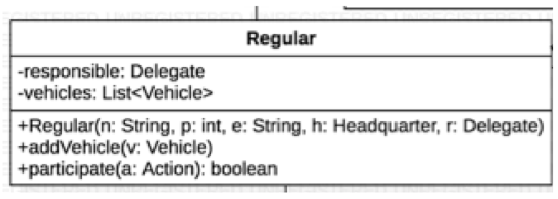


```java
public class Availability {

    LinkedList <String> days;
    LinkedList<Integer> hours;


    public Availability(LinkedList <String> d, LinkedList <Integer> h) {
        this.days=d;
        this.hours=h;
    }
```

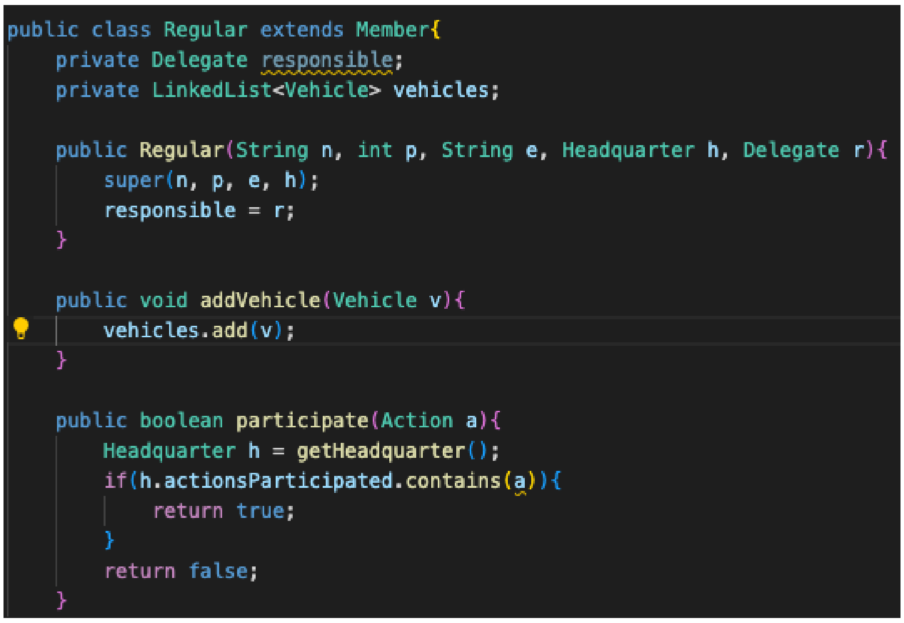
Availability just registers two LinkedLists of days and hours, and it’ll only be used (only one direct relation) for Members.

REGULAR CLASS

Laia Tomás
Daniela Quimis



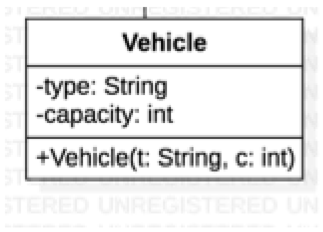
```
Regular
-responsible: Delegate
-vehicles: List<Vehicle>
+Regular(n: String, p: int, e: String, h: Headquarter, r: Delegate)
+addVehicle(v: Vehicle)
+participate(a: Action): boolean
```



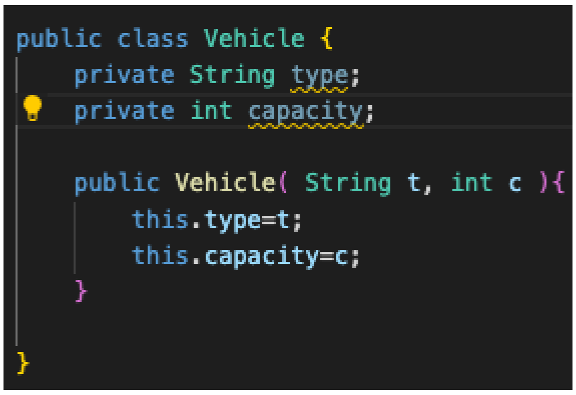
```java
public class Regular extends Member{
    private Delegate responsible;
    private LinkedList<Vehicle> vehicles;

    public Regular(String n, int p, String e, Headquarter h, Delegate r){
        super(n, p, e, h);
        responsible = r;
    }

    public void addVehicle(Vehicle v){
        vehicles.add(v);
    }

    public boolean participate(Action a){
        Headquarter h = getHeadquarter();
        if(h.actionsParticipated.contains(a)){
            return true;
        }
        return false;
    }
```

Regular Class inherits the attributes of Member, that are **name** , **email**, and **headquarter** We use keyword and function ***super*** to point this out.The rest of the methods are add(for adding a type vehicle of a regular) and a boolean for checking if an action is already in the Headquarter.

VEHICLE CLASS



```
Vehicle
-type: String
-capacity: int
+Vehicle(t: String, c: int)
```

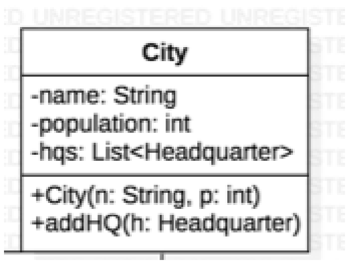


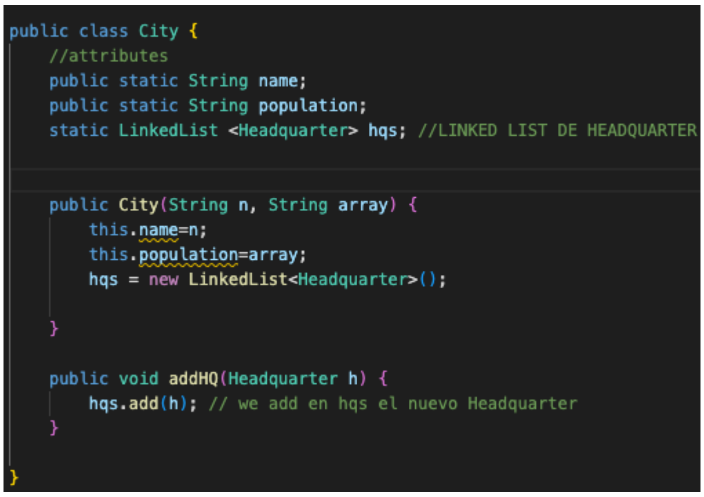
```java
public class Vehicle {
    private String type;
    private int capacity;

    public Vehicle( String t, int c ){
        this.type=t;
        this.capacity=c;
    }

}
```

Vehicle class just registers the type of vehicle and a capacity . Has association relation with Regular Class.

Laia Tomás
Daniela Quimis

CITY CLASS



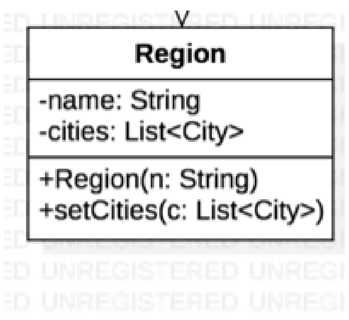


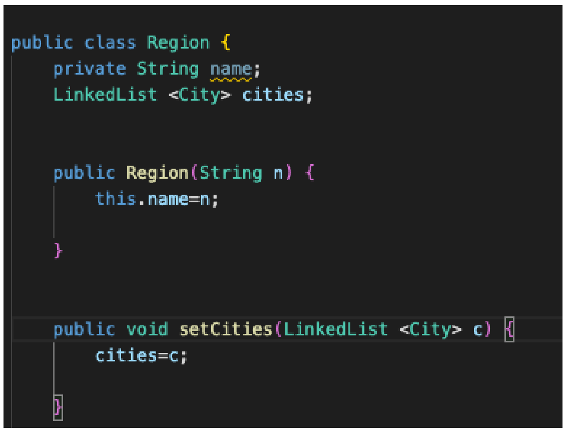
```
public class City {
    //attributes
    public static String name;
    public static String population;
    static LinkedList <Headquarter> hqs; //LINKED LIST DE HEADQUARTER


    public City(String n, String array) {
        this.name=n;
        this.population=array;
        hqs = new LinkedList<Headquarter>();

    }

    public void addHQ(Headquarter h) {
        hqs.add(h); // we add en hqs el nuevo Headquarter
    }

}
```

City class needs as attributes population and a name, then we Link a list of headquarters because in every city there's a headquarter. Then as methods, there's only one, for adding a new headquarter to this list.

City has an aggregation relationship with Region and association relation with Headquarter.

REGION CLASS



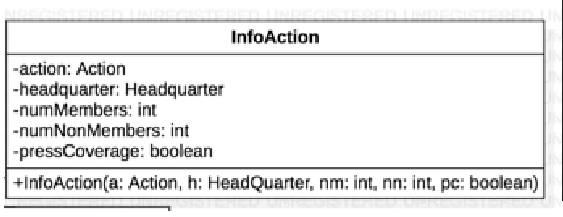


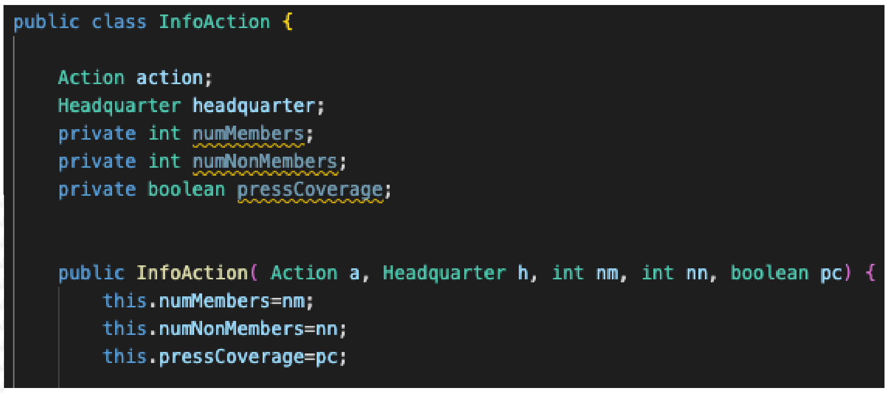
```
public class Region {
    private String name;
    LinkedList <City> cities;


    public Region(String n) {
        this.name=n;

    }


    public void setCities(LinkedList <City> c) {
        cities=c;

    }
```

Region class has a really basic attribute, a name, and we link our cities list, due to the fact every region has a city. So the only method needed is a setter of the cities for the Region.

Laia Tomás
Daniela Quimis

INFOACTION CLASS

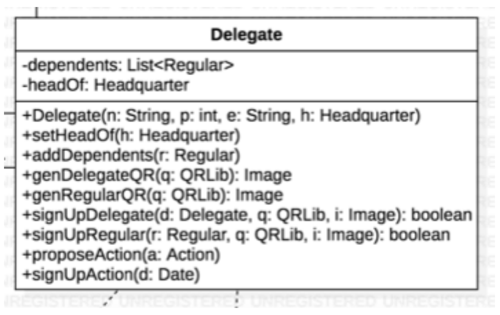

InfoAction
-action: Action
-headquarter: Headquarter
-numMembers: int
-numNonMembers: int
-pressCoverage: boolean
+InfoAction(a: Action, h: HeadQuarter, nm: int, nn: int, pc: boolean)

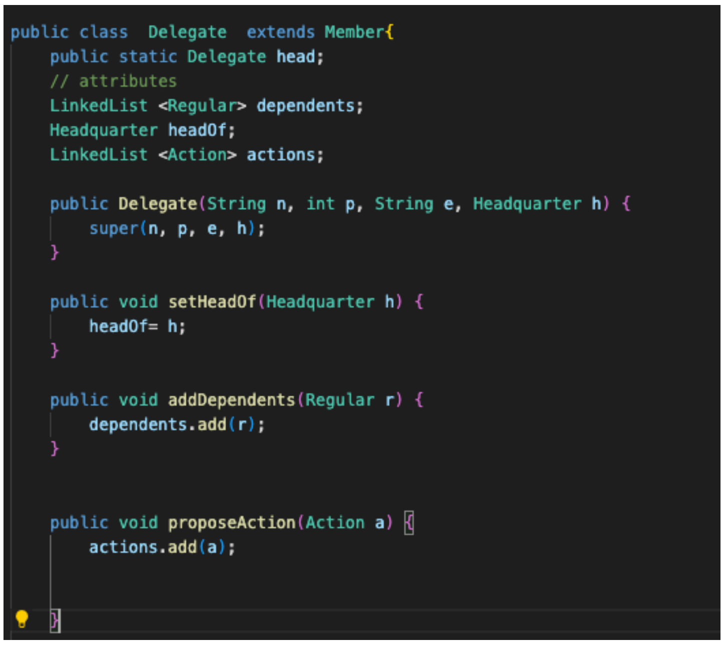


```java
public class InfoAction {

    Action action;
    Headquarter headquarter;
    private int numMembers;
    private int numNonMembers;
    private boolean pressCoverage;


    public InfoAction( Action a, Headquarter h, int nm, int nn, boolean pc) {
        this.numMembers=nm;
        this.numNonMembers=nn;
        this.pressCoverage=pc;
```

InfoAction is made because of the relation between Action and Headquarter, it contains the action, the headquarter, the number of members and not members of the Action and if it's press coverage involved.

DELEGATE CLASS


Delegate
-dependents: List<Regular>
-headOf: Headquarter
+Delegate(n: String, p: int, e: String, h: Headquarter)
+setHeadOf(h: Headquarter)
+addDependents(r: Regular)
+genDelegateQR(q: QRLib): Image
+genRegularQR(q: QRLib): Image
+signUpDelegate(d: Delegate, q: QRLib, i: Image): boolean
+signUpRegular(r: Regular, q: QRLib, i: Image): boolean
+proposeAction(a: Action)
+signUpAction(d: Date)



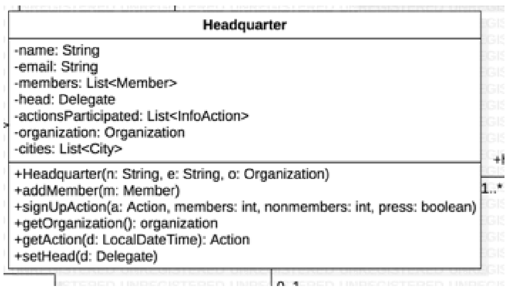
```java
public class  Delegate  extends Member{
    public static Delegate head;
    // attributes
    LinkedList <Regular> dependents;
    Headquarter headOf;
    LinkedList <Action> actions;

    public Delegate(String n, int p, String e, Headquarter h) {
        super(n, p, e, h);
    }

    public void setHeadOf(Headquarter h) {
        headOf= h;
    }

    public void addDependents(Regular r) {
        dependents.add(r);
    }


    public void proposeAction(Action a) {
        actions.add(a);


    }
```

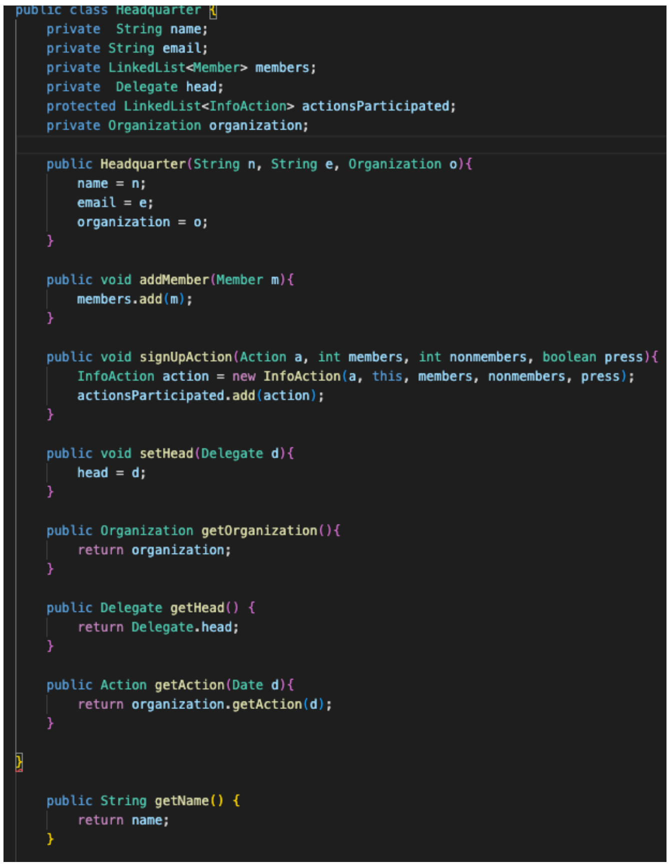
(Delegate class without QR METHODS , they are commented below)

Delegate Class inherits from Member, so name, phone, email and headquarter are inherited in their class. As we see in the constructor we use keyword super() to point out the inheritance. It has useful and basic methods, a setter for the head of the Headquarter and add regulars to our list Dependents, and last but not least a list for proposal actions, when a new action is proposed, we'll add to this list.

Laia Tomás
Daniela Quimis

HEADQUARTER CLASS

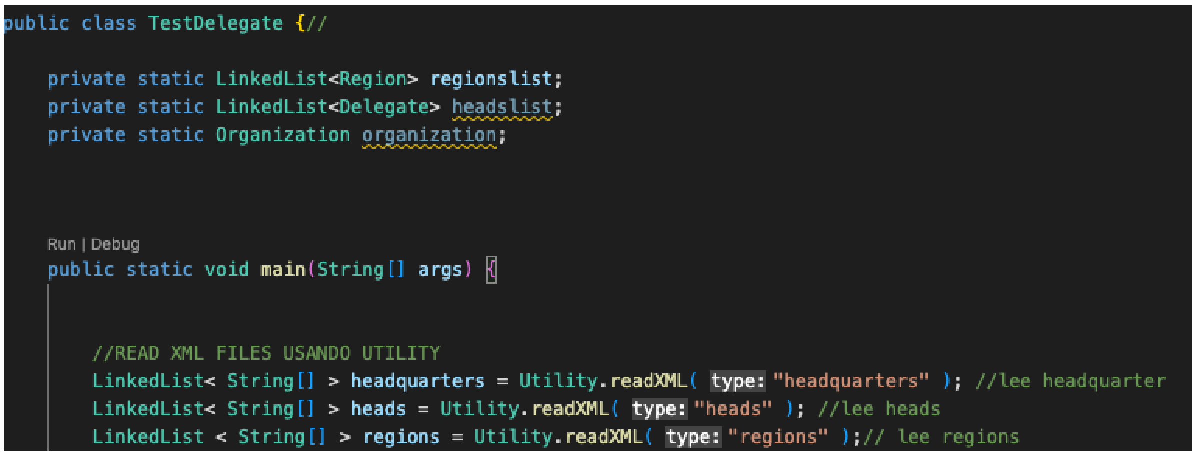

Headquarter
-name: String
-email: String
-members: List<Member>
-head: Delegate
-actionsParticipated: List<InfoAction>
-organization: Organization
-cities: List<City>
+Headquarter(n: String, e: String, o: Organization)
+addMember(m: Member)
+signUpAction(a: Action, members: int, nonmembers: int, press: boolean)
+getOrganization(): organization
+getAction(d: LocalDateTime): Action
+setHead(d: Delegate)
1..*
0..1



```java
public class Headquarter {
    private  String name;
    private String email;
    private LinkedList<Member> members;
    private  Delegate head;
    protected LinkedList<InfoAction> actionsParticipated;
    private Organization organization;

    public Headquarter(String n, String e, Organization o){
        name = n;
        email = e;
        organization = o;
    }

    public void addMember(Member m){
        members.add(m);
    }

    public void signUpAction(Action a, int members, int nonmembers, boolean press){
        InfoAction action = new InfoAction(a, this, members, nonmembers, press);
        actionsParticipated.add(action);
    }

    public void setHead(Delegate d){
        head = d;
    }

    public Organization getOrganization(){
        return organization;
    }

    public Delegate getHead() {
        return Delegate.head;
    }

    public Action getAction(Date d){
        return organization.getAction(d);
    }

}

    public String getName() {
        return name;
    }
```

Headquarter Class is the main class and the rest of classes have a relation with them. Members have an aggregation relation with headquarters and association relation with City ,Action and Delegate. It has very basic information of the headquarters(name, email and organization) and also several methods like the setters and getters, ***signupAction***() that uses attributes of ***InfoAction***().

Once all of this is done, we create a TestDelegate that includes a main method, first this test will read the provided XML files of some of the classes (regions, heads and headquarters), using Utility class functions.

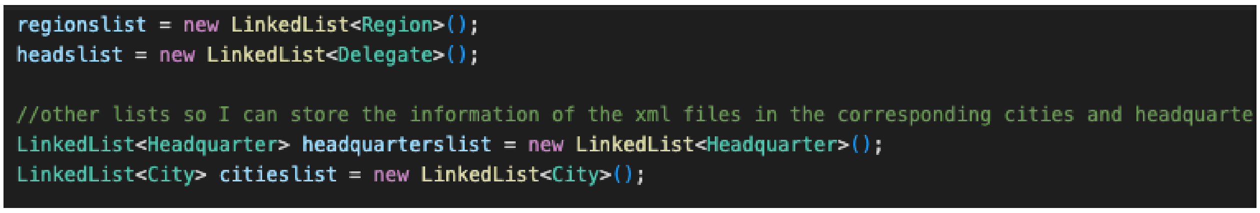


```java
public class TestDelegate {//

    private static LinkedList<Region> regionslist;
    private static LinkedList<Delegate> headslist;
    private static Organization organization;



    Run | Debug
    public static void main(String[] args) {

        //READ XML FILES USANDO UTILITY
        LinkedList< String[] > headquarters = Utility.readXML( type: "headquarters" ); //lee headquarter
        LinkedList< String[] > heads = Utility.readXML( type: "heads" ); //lee heads
        LinkedList < String[] > regions = Utility.readXML( type: "regions" );// lee regions
```

Laia Tomás
Daniela Quimis

We chose to have regions and delegates and an organisation as attributes, and then inside create another 2 more lists to be filled with headquarters and cities, in order to be able to access it anytime, rather than only inside regions or delegates with utility.getObject.

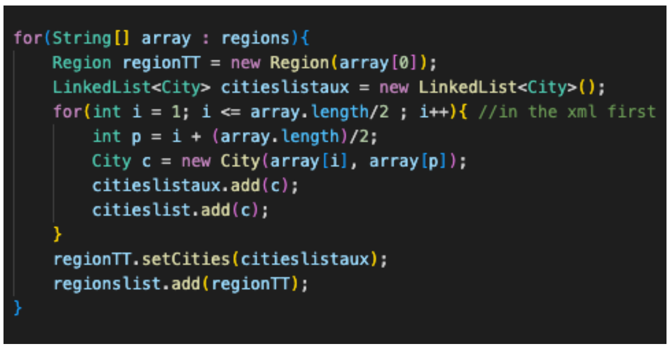
After this, it should create an instance for Region, City , Organization, Headquarter and finally Head Delegate, in order to perform this we create as an attribute a list of Region, Heads and an Organization.



```
regionslist = new LinkedList<Region>();
headslist = new LinkedList<Delegate>();

//other lists so I can store the information of the xml files in the corresponding cities and headquarte
LinkedList<Headquarter> headquarterslist = new LinkedList<Headquarter>();
LinkedList<City> citieslist = new LinkedList<City>();
```

Therefore we create two more lists for saving headquarters and city types of data, with all of these store lists, we read the XML files and pass all the data to their main lists. Consequently, we need to check every xml file to see where the information we have to subtract.

For regions, we run first their file, and the regions name is in the first position (i[0]) , in each region we'll create a linkedlist of cities where each city belonging to that region will be stored, population number is on the second half of the array, so when it's all ran and stored in our lists previously defined, we use them for setting the corresponding city to their main region and consecutively to the main regions list.



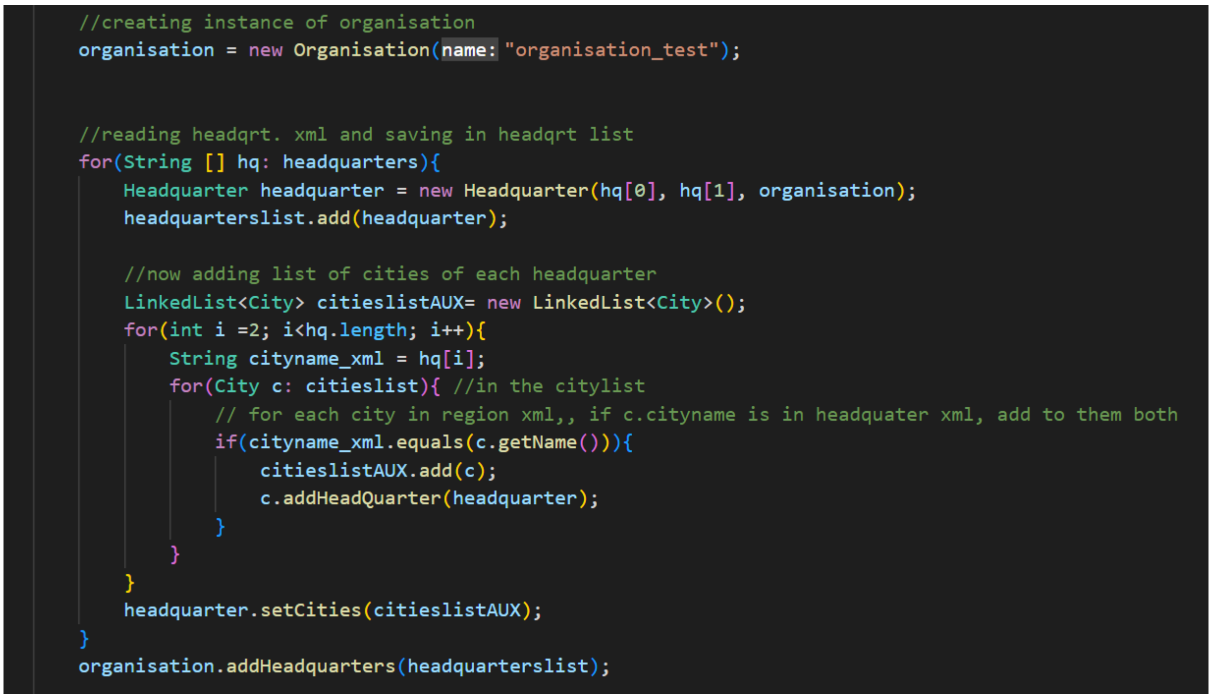
```
for(String[] array : regions){
    Region regionTT = new Region(array[0]);
    LinkedList<City> citieslistaux = new LinkedList<City>();
    for(int i = 1; i <= array.length/2 ; i++){ //in the xml first
        int p = i + (array.length)/2;
        City c = new City(array[i], array[p]);
        citieslistaux.add(c);
        citieslist.add(c);
    }
    regionTT.setCities(citieslistaux);
    regionslist.add(regionTT);
}
```

Now we create the instance for the organization and then we'll read the headquarters file, same process as Regions, we create the headquarters, for saving the name and email of the headquarters and the main organization,then we'll add this info inside of our previous list (*headquarterslist*).

Laia Tomás
Daniela Quimis

For the cities of each headquarter, we again run the headquarters file, the cities position is in third position (index 2), so we save the name of that city, and then using the citieslist we previously filled in region, if the city name is in the citieslist, we add the city to the auxiliar citylist and then update the city information for the general citylist.

Finally, when all the cities of a headquarter are in the list, we add that list to the headquarter using setCities method, and then we add all of the headquarters to the organization.

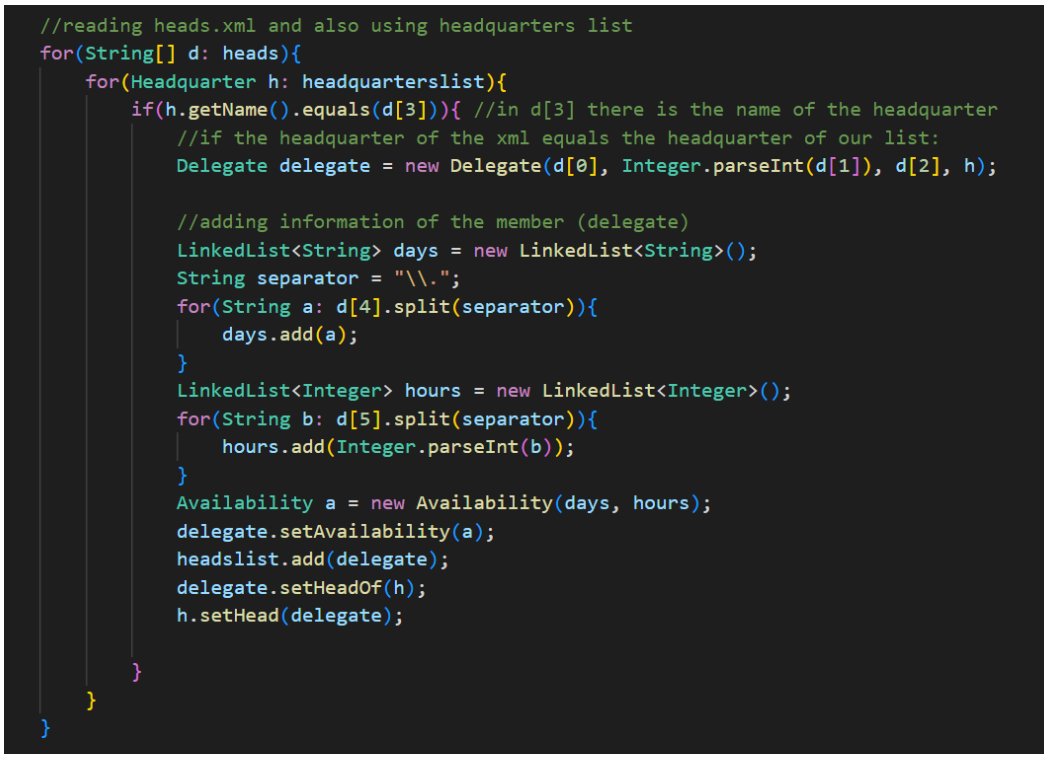


```java
//creating instance of organisation
organisation = new Organisation(name: "organisation_test");


//reading headqrt. xml and saving in headqrt list
for(String [] hq: headquarters){
    Headquarter headquarter = new Headquarter(hq[0], hq[1], organisation);
    headquarterslist.add(headquarter);

    //now adding list of cities of each headquarter
    LinkedList<City> citieslistAUX= new LinkedList<City>();
    for(int i =2; i<hq.length; i++){
        String cityname_xml = hq[i];
        for(City c: citieslist){ //in the citylist
            // for each city in region xml,, if c.cityname is in headquater xml, add to them both
            if(cityname_xml.equals(c.getName())){
                citieslistAUX.add(c);
                c.addHeadQuarter(headquarter);
            }
        }
    }
    headquarter.setCities(citieslistAUX);
}
organisation.addHeadquarters(headquarterslist);
```

Laia Tomás
Daniela Quimis

For heads file, we need to check if the headquarter of the general list coincides with the headquarter of the corresponding head delegate with the help of the function *getName*() and *equals*(). In the case they are equal, we create the instance of the delegate with the accurate information (name, phone,email and headquarter).

Then, for setting the availability given in the file, we create two different store lists, and use *split*() function to save the data correctly. We have to say that we needed to use the separator "\\." because "." alone didn't work. Once this is done, we can set the availability with the two lists and use *setAvailability*() for each delegate and add the final delegate to headslist, set its corresponding headquarter and set the delegate as the head of that headquarter.

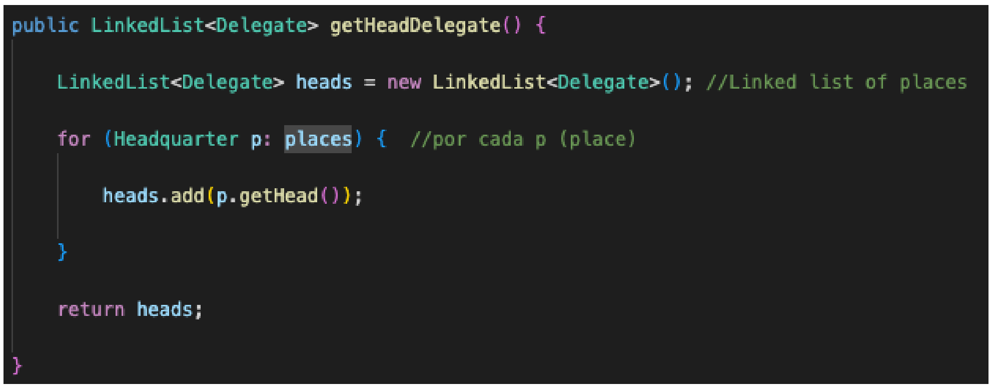


```java
//reading heads.xml and also using headquarters list
for(String[] d: heads){
    for(Headquarter h: headquarterslist){
        if(h.getName().equals(d[3])){ //in d[3] there is the name of the headquarter
            //if the headquarter of the xml equals the headquarter of our list:
            Delegate delegate = new Delegate(d[0], Integer.parseInt(d[1]), d[2], h);

            //adding information of the member (delegate)
            LinkedList<String> days = new LinkedList<String>();
            String separator = "\\.";
            for(String a: d[4].split(separator)){
                days.add(a);
            }
            LinkedList<Integer> hours = new LinkedList<Integer>();
            for(String b: d[5].split(separator)){
                hours.add(Integer.parseInt(b));
            }
            Availability a = new Availability(days, hours);
            delegate.setAvailability(a);
            headslist.add(delegate);
            delegate.setHeadOf(h);
            h.setHead(delegate);

        }
    }
}
```

Laia Tomás
Daniela Quimis

Increasing the organization structure

This part asks us to create a new method in Organization, this method will return the head delegate of every headquarter.



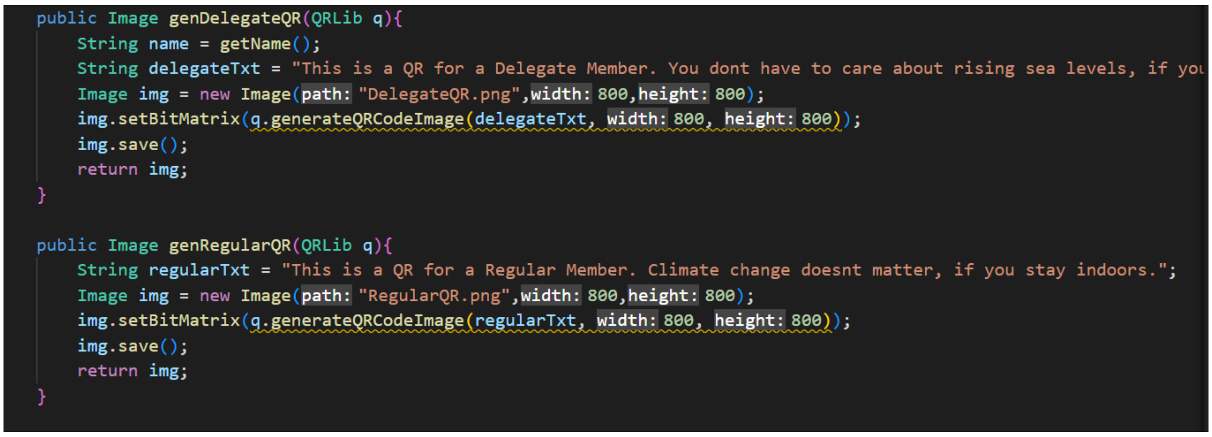
```
public LinkedList<Delegate> getHeadDelegate() {

    LinkedList<Delegate> heads = new LinkedList<Delegate>(); //Linked list of places

    for (Headquarter p: places) {  //por cada p (place)

        heads.add(p.getHead());

    }

    return heads;

}
```

We use *getHead*() in places list, the heads will be added to the previously created list. When it finishes, return it.

Now we begin to work with the QR functionalities, we go to the Delegate class where they are created, for generating functions (Regular and Member).

Following the instructions, the two generating functions have a different phrase and also the delegate one includes its head delegate name, so checking the Image and QRLib class, we know which instances and functions we'll use.

We first need to create variables for saving the corresponding text, then create a new instance of image. After that, we need to use the methods of the image and qr class, and finally we save the image. Additionally, in the delegate one, we retrieve the name of the head delegate, and add it to the string delegateText.



```
public Image genDelegateQR(QRLib q){
    String name = getName();
    String delegateTxt = "This is a QR for a Delegate Member. You dont have to care about rising sea levels, if you
    Image img = new Image(path: "DelegateQR.png",width: 800,height: 800);
    img.setBitMatrix(q.generateQRCodeImage(delegateTxt, width: 800, height: 800));
    img.save();
    return img;
}

public Image genRegularQR(QRLib q){
    String regularTxt = "This is a QR for a Regular Member. Climate change doesnt matter, if you stay indoors.";
    Image img = new Image(path: "RegularQR.png",width: 800,height: 800);
    img.setBitMatrix(q.generateQRCodeImage(regularTxt, width: 800, height: 800));
    img.save();
    return img;
}
```

Laia Tomás
Daniela Quimis

CONCLUSION AND DIFFICULTIES

In closing in this lab, we've again seen types of relations and how to code them in VS code as their behavior between them. Similar to lab 2 some parts were trivial, like the creation of the classes even so we learned new types like Date type and a new library, also how complex can be linking lists and how are they connected. We can even say the most challenging part was the one involving the functions of QR and Image due to the fact they were tricky to understand, and also populating all the organization in the TestDelegate.

We chose to have *regionslist* and *headslist* as an attribute, and then inside the main we added the headquarters and city list for also storing its information, rather than only having them inside the blocks of regions and heads using getObject. We thought it would be clearer. Thus, we needed to distinguish the citieslist form the citieslist auxiliary inside regions.

Moreover, notice that we created many getters and print methods that are not explained in the report because they are trivial, but they helped us to make sure all the structure was correct. Note that some of them are commented in the TestDelegate so you can uncomment the ones you need to correct our code and check that it works correctly. (Lines 36-37 and from 140 in the TestDelegate).

We also had some trouble when creating the cities list because even though in the xml file we can see name and population interleaved, they weren't in fact. First they were all the names in the first half, and then all the populations. So we had to take that into account. Another problematic was the split method, because it didn't split the array when the separator was ".", but it finally did when it was "\\." .

Regarding the optional part, we only created the classes as they were part of the diagram, but we didn't finish them at the end.

Nonetheless we manage to finish the lab and also their execution being successful, after many attempts with tests. Whether, we found the seminar 3 more understandable and a little bit easier than the previous, so that means we're getting better at this, and also thanks to this lab more subjects were clarified.