Adria Soria, Carlo Marcello

PRÁCTICA 3 OOP

1. INTRODUCCIÓN

La práctica 3 de Programación Orientada a Objetos consiste en la creación y desarrollo de varias clases, entre ellas Headquarter, Organization, Delegate, Regular y Member como clases principales. Cuando se relacionan e implementan de forma eficiente y adecuada, respetando en todo momento los tipos de relaciones que existen entre ellas, crean una aplicación de administración de una Organización para acabar generando unos códigos QR para los miembros de esta organización dependiendo del tipo de miembro que sean, que pueden ser *regulars* o *delegates*. Para todas las clases previamente mencionadas deberemos crear e inicializar todos los métodos necesarios para que las clases se conecten entre ellas y tengan una funcionalidad en el programa. Para ello, nos hemos ayudado con el Lenguaje Unificado de Modelado (UML) que creamos en el seminario 3.

Tal y como hemos hecho en las dos últimas prácticas, hemos inicializado todos los atributos y métodos de cada una de las clases de nuestro programa. A continuación se muestra un ejemplo de una de las clases principales del programa:

```java
public class Headquarter {

    //atributes
    private String name;
    private String email;
    private LinkedList<Member> members;
    private Delegate head;
    private LinkedList<InfoAction> actionsParticipated;
    private Organization organization;
    private LinkedList<City> cities;

    //Contructor
    public Headquarter(String n, String e, Organization o){
        this.name = n;
        this.email = e;
        this.organization = o;
        this.actionsParticipated = new LinkedList<InfoAction>();
        this.members = new LinkedList<Member>();
        this.cities = new LinkedList<City>();
    }
```

En esta primera imagen podemos observar la declaración de todos los atributos de la clase *Headquarter* donde ya podemos ver variables de diferentes tipos de clases, como por ejemplo *delegate* o *organization*, y por lo tanto podremos decir que existe algún tipo de relación entre la clase *Headquarter* y las otras clases.
Por otro lado también tenemos el método constructor donde allí sí que inicializamos todos los atributos declarados previamente y les asignamos un valor determinado.

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

public void addCity(City c){
    this.cities.add(c);
}

public void setHead(Delegate d){
    this.head = d;
}
public Organization getOrganization(){
    return this.organization;

}
```

Aparte, en la clase *Headquarter* deberemos crear los métodos *getters* y *setters* que vayamos a utilizar en nuestro programa, ya que a la hora de leer nuestros ficheros XML deberemos llamar a estos métodos para obtener el resultado que queremos. Por otro lado podemos ver que también hemos creado unos métodos *add.* De aquí podemos decir que un *Headquarter* tiene uno o más Miembros y uno o más ciudades. La relación entre la clase *Headquarter* y la clase *Member* es una relación de agregación ya que un member sólo puede tener un *Headquarter* pero la relación entre *Headquarter* y la *City* es una relación de asociación ya que un *Headquarter* puede tener muchas ciudades pero una ciudad también puede tener varios *Headquarter.*

Una de las diferencias que debíamos crear respecto a otras prácticas era la relación de herencia que existía entre las clases *Member* y las clases *Regular* y *Delegate.* Para ello deberemos hacer un *extends* en ambas clases a la clase *Member* y en el método constructor hacer un super en referencia a la superclass *Member.* Esto nos proporciona una reutilización a la definición de la superclase *Member* y una flexibilidad de las nuevas clases subclases *Regular* y *Delegate.* A continuación vemos cómo hemos ejecutado esta relación de herencia entre estas tres clases:

```java
public class Delegate extends Member {

    //Atributes
    private LinkedList<Regular> dependents;
    private Headquarter headOf;

    //Constructor
    public Delegate(String n, int p, String e, Headquarter h){
        super(n, p, e, h);
        this.headOf = h;
        this.dependents = new LinkedList<Regular>();

        QRLib q = new QRLib();
        genDelegateQR(q);
    }
```

```java
public class Regular extends Member{

    //Atributes
    private Delegate responsible;
    private LinkedList<Vehicle> vehicles;

    //Constructor

    public Regular(String n, int p, String e, Headquarter h, Delegate r){
        super(n, p, e, h);
        this.responsible = r;
        this.vehicles = new LinkedList<Vehicle>();
    }
```

2. DESCRIPCIÓN DE POSIBLES ALTERNATIVAS Y RAZONAMIENTO DE NUESTRA DECISIÓN

La principal duda que tuvimos al empezar a crear nuestro programa fue cuando nos pusimos a programar la parte de la lectura de los ficheros XML. La primera que nos pusimos a hacer fue la lectura del fichero regions. Nos encontramos con el problema de que si bien sabíamos que para cada *region* debíamos leer el nombre de la región y luego el nombre de la ciudad con su respectiva población, se nos complicaba cuando una región tenía mas de

una ciudad ya que no podíamos únicamente llamar a la array pertinente sino que debíamos recorrer todas las ciudades de la array ciudades. Este problema lo teníamos ya que la relación que existe entre la clase *Region* y la clase *City* es una relación de agregación donde una *region* puede tener más de una ciudad. La solución que encontramos fue crear una *LinkedList* que llamamos *Cities_of_region* y recorrer mediante un for la longitud de la array entre dos (ya que cada city tiene una población) e ir añadiendo las *cities* en esta nueva *LinkedList.*

```
for (String[] array : _regions){
    Region r = new Region(array[0]);

    LinkedList<City> cities_of_region = new LinkedList<City>();

    int phone_i = (array.length / 2)+1;

    for (int i = 1; i <= array.length/2; i++){
        int x = Integer.parseInt(array[phone_i]);
        City c = new City(array[i],x);
        this.cities.add(c);
        cities_of_region.add(c);
        phone_i++;
    }

    r.setCities(cities_of_region);

    this.regions.add(r);
}
```

Otra de las alternativas que hemos encontrado a los problemas que hemos ido encontrando ha sido cuando queríamos leer el fichero *head,* donde nos hemos encontrado que los *delegates* tenían una *availability* expresada mediante puntos y nosotros teníamos que pasar esa String separada por puntos a una *LinkedList* de toda la *availability* de cada *delegate*. Para ello creamos dos *LinkedLists,* una para los días y otra para las horas, en la que mediante un for y el método .split íbamos añadiendo cada String separada por un punto a nuestras nuevas *LinkedLists*, como se muestra en la imagen:

```
LinkedList<String[]> _delegates = Utility.readXML( type: "head");

this.delegates = new LinkedList<Delegate>();

for(String[] array : _delegates){
    int x = Integer.parseInt(array[1]);
    Headquarter h = Utility.getObject(array[3], headquarters);
    Delegate d = new Delegate(array[0], x, array[2], h);

    LinkedList<String> days = new LinkedList<String>();
    LinkedList<Integer> hours = new LinkedList<Integer>();


    //Days and hours selection and implementation by '.split'
    for(int i = 0; i<=countElements(array[4]); i++){
        String day = array[4].split(regex: "\\.")[i];

        days.add(day);
    }


    for(int i = 0; i<=countElements(array[5]); i++){
        String hour_ = array[5].split(regex: "\\.")[i];
        int hour = Integer.parseInt(hour_);

        hours.add(hour);
    }
```

Aparte teníamos que crear un método auxiliar para que el for fuera de i hasta el número de elementos que había en la string separada por puntos:

```java
//Extra method to count days and hours of the XML
private int countElements(String s){
    int ret = 0;
    for(int i = 0; i<s.length(); i++){
        if(s.charAt(i) == '.') ret++;
    }
    return ret;
}
```

A la hora de generar los QR 's únicamente nos generó dudas como hacer la verificación de que el QR se haya generado correctamente. Para ello, generamos el método *signUpRegular* y *signUpDelegate* que les entra un *delegate, un qr y una image* y mediante el método *decodeQRCodeImage* verifican si el *QR* devolvía el mismo texto que el que se pedía en el enunciado de la práctica con el *delegate*/*regular* correspondiente.
A continuación se muestra el código de la verificación con algún comentario extra de lo que está haciendo cada línea de código y otra imágen con el resultado que hemos obtenido:

```java
//Now we are going to test if the Qr repartition has been successfully done. To make the checking we will simulate a regular
//member signup into their headquarter. So first we load the member qr from their path and then, as we have taken the qr for the
//Clara Doménech department (first department), we will print the pertinent verification of such a member in an especific
//headquarter wants to access to their department using his Qr code.

QRLib q = new QRLib();
Image image = new Image(path: "QrRegularDataBase/QrRegular for Clara Domènech department.png");//Clara Domènech headquarter regular Qr code
image.load();

Delegate delegate = myOrganization.delegates.get(index: 0); //Taking a headquarter delegate (Clara Domènech) to then take a member

if(delegate.signUpRegular(delegate.getDependents().get(index: 3), q, image)){ //(true)
    System.out.println("Welcome to "+ delegate.getHeadquarter().toString()+" headquarter!");
}else{
    System.out.println("Access denied. Wrong Qr for "+delegate.getHeadquarter().toString()+" headquarter.");
}
System.out.print(s: "\n");
```

```
Welcome to Santsenc headquarter!
```

3. CONCLUSIÓN

Una vez terminado el programa estamos satisfechos con el resultado obtenido. Los puntos donde hemos tenido alguna duda han sido cuando debíamos implementar relaciones entre clases que no habíamos hecho aún, pero con la ayuda que nos proporcionó la profesora y con la indagación en Internet, hemos sido capaces de implementar todo el código necesario para la herencia de las dos clases *Regular* y Delegate a la clase *Member*, y la agregación entre algunas clases como la de *City* con *Region.* Aparte, hemos aprendido a generar

códigos QR mediante la utilización de algunos métodos ya creados y vemos una aplicación directa de estos códigos a las diferentes relaciones que hemos hecho entre las clases de una aplicación.

En conclusión, llegados a este punto damos la práctica por concluida conscientes de que, durante el desarrollo de ésta, hemos logrado mejorar en el entorno ‘java’ y a su vez familiarizarnos un poco más con el paradigma de programación orientado a objetos el cual hace unas pocas semanas apenas éramos conscientes de su existencia.