PROGRAMACIÓN ORIENTADA A OBJETOS

PRÁCTICA 4 – DOCUMENTACIÓN

1. Introducción

El objetivo de esta práctica era implementar un diseño a una aplicación prestando mucha atención en el uso del concepto de la herencia y el movimiento de regiones. Respecto a las prácticas anteriores hemos podido reutilizar las clases **PolygonalRegion**, **EllipsoidalRegion** de las prácticas 2 y tres respectivamente, aunque hemos añadido más métodos para el funcionamiento de la práctica. Además de estas clases, en la declaración del laboratorio también se pueden encontrar más clases que hemos implementado en nuestro programa: **DrawApp, Entity, Region, TriangularRegion, RectangularRegion, CircularRegion.**

Nuestra tarea ha residido en entender lo requerido por la práctica e implementar las clases y sus relaciones. En esta documentación, hemos explicado brevemente las clases y sus métodos que hemos utilizado. Hemos finalizo con una pequeña conclusión que detalla cómo nos hemos organizado para llevar adelante el trabajo de esta práctica.

2. Explicando el código

a. PolygonalRegion

```
import java.util.LinkedList;
import java.awt.*;

public class PolygonalRegion extends Region{
    //Atributos
    protected LinkedList< Point > points;
    //Ctor
    public PolygonalRegion(LinkedList< Point > l, Color linecolor, Color fillColor) {
        super(linecolor, fillColor);
        this.points = l;
    }
    //Metodos
    //Calculamos el area de la region usando la formula pertinente.
    public double getArea(){
        double det1 = 0;
        double det2 = 0;
        for(int i = 0; i < points.size() - 1; i++){
            det1 = det1 + (points.get(i).getX()*points.get(i+1).getY());
            det2 = det2 + (points.get(i).getY()*points.get(i+1).getX());
        }
        det1 = det1 + (points.getLast().getX()*points.getFirst().getY()) ;
        det2 = det2 + (points.getLast().getY()*points.getFirst().getX());
        double det = det1 det2;
        double area = 0.5 * det;
        return area;
    }
    @Override
    //Sobreescribimos el metodo abstracto en la clase padre, definiendolo aqui
    public void draw(Graphics g){
        int pointsx [] = new int[points.size()];//Pasamos la linked list a array para poder definir luego drawPolygon y fillPolygon
        int pointsy [] = new int[points.size()];
        //Cada punto se unira con el siguiente creado
        for(int i = 0; i < points.size(); i++){
            pointsx[i] = ((int)points.get(i).getX());
            pointsy[i] = ((int)points.get(i).getY());
        }
```

```
    g.setColor(lineColor);
    g.drawPolygon( pointsx, pointsy, points.size());
    g.setColor(fillColor);
    g.fillPolygon(pointsx, pointsy, points.size());


}
@Override
//Sobreescribimos el metodo isPointInside definiendolo aqui
public boolean isPointInside(Point p) {
    int counter_neg = 0;
    int counter_pos = 0;
    //Miramos si el signo de (q2-q1) x (p-q1) es el mismo para todos los puntos, en ese caso devolveremos true, en cualquier otro false
    for(int i = 0; i < points.size()-1; i++){
        if((points.get(i+1).difference(points.get(i))).crossProduct(p.difference(points.get(i))) < 0){
            counter_neg++;
        }
        if((points.get(i+1).difference(points.get(i))).crossProduct(p.difference(points.get(i))) > 0){
            counter_pos++;
        }
    }
    if(points.getFirst().difference(points.getLast()).crossProduct(p.difference(points.getLast()))< 0){
        counter_neg++;
    }
    if(points.getFirst().difference(points.getLast()).crossProduct(p.difference(points.getLast()))> 0){
        counter_pos++;
    }
    if(counter_neg == points.size() || counter_pos == points.size()){
        return true;
    }
    return false;
}
```

Nos encontramos con un atributo que es una Linked List de Point. En el constructor, como lo hemos hecho en anteriores prácticas en la clase, inicializamos la Linked List que esta vacía. En el apartado de los métodos tenemos un getter **getArea()**, para calcular el área de la región usando la formula pertinente, y devolvemos el valor de los atributos en forma de área.

Por otro lado, en el método **draw(Graphics g)**, sobreescribimos el método abstracto en la clase padre. Además, pasamos la LInked List a array para poder definir drawPolygon y fillPolygon. Mediante un for loop vamos a unir cada punto con el siguiente creado. Además, también tenemos nuestro setter, **setColor()**, que establecen el valor a nuestros atributos.

A parte de estos también tenemos otros tantos métodos, **isPointInside(Point p),** lo sobreescribimos el método isPointInside, inicializamos los dos contadores a cero y mediante un for loop miramos

```java
@Override
//sobreescribimos translate haciendo que se trasladen todos los puntos de la region usando el metodo translate de Point.
public void translate(int dx, int dy) {
    for (int i = 0; i < points.size(); i++){
        points.get(i).translate(dx, dy);;
    }
}
```

si el signo de (q2-q1) x (p-q1) es el mismo para todos los puntos, en ese caso devolveremos true, en cualquier otro false. Para finalizar hemos implementado el método **Translate (int dx, int dy),** hemos sobreescito el método translate haciendo que se transladeb todos los puntos de la región usando el método translate de Point.

b. EllipsoidalRegion

```java
import java.awt.*;
public class EllipsoidalRegion extends Region{
    private Point c;//Centro
    //Los dos radios de la elipsoide
    private double r1;
    private double r2;
    //Ctor
    public EllipsoidalRegion(Point c, double r1, double r2, Color linecolor, Color fillColor) {
        //Ejecutamos el ctor de la clase padre en este caso Region y le pasamos los argumentos de esta, en este caso el color de la linea
        super(linecolor, fillColor);
        //Inicializamos los 3 atributos de la clase
        this.c = c;
        this.r1 = r1;
        this.r2 = r2;
    }
    //Getter del area
    public double getArea(){
        return Math.PI*r1*r2;
    }
    @Override
    //Sobreescribimos el draw de region el cual era abstracto
    public void draw(Graphics g){
        g.setColor(fillColor);
        g.fillOval((int)c.getX(),(int) c.getY(),(int) r1,(int) r2);
        g.setColor(lineColor);
        g.drawOval((int)c.getX(), (int) c.getY(), (int) r1, (int) r2);
    }
```

```java
    @Override
    //Sobreescribimos el metodo isPointInside de Region el cual era abstracto, por lo que lo definimos ahora
    public boolean isPointInside(Point p) {

        double numerador1 = Math.pow(p.getX()-this.c.getX(),2);//(px-cx)^2
        double denominador1 = Math.pow(r1, 2);//a^2
        double numerador2 = Math.pow(p.getY()-this.c.getY(), 2);//(py-cy)^2
        double denominador2 = Math.pow(r2, 2);//b^2
        if((numerador1/denominador1)+(numerador2/denominador2) <= 1){//Lo juntamos todo para formar la formula, si da menor que 1,
            return true;                                            //estara dentro, en cualquier otro caso, el punto estara fuera
        }
        return false;
    }
    @Override
    //Sobreescribimos el metodo abstracto translate y lo definimos aquí llamando a translate del centro, lo cual hara que se mueva toda la elipse
    public void translate(int dx, int dy) {
        this.c.translate(dx, dy);
    }
}
```

Tenemos una clase llamada **EllipsoidalRegion** que tiene una interfaz extends que hereda los métodos de **Region.** Está compuesta por tres atributos que son **r1**, **r2**, **c. R1** y **r2** son los radios de la elipsoide y la **c** es el centro. En el constructor ejecutamos el constructor de la clase padre en este caso Región y le pasamos los argumentos de esta, ene este caso el color de la línea e inicializamos los valores **c**, **r1**, **r2.**Utilizamos un getter, **getArea** para devolver el valor del área. Para finalizar sobre escribimos el método **draw** de la región el cual era abstracta. También sobre escribimos los métodos **isPointInside(Point p)** y **translate(int dx, int dy)**. El primero de ellos calcula los dos enumeradores y denominadores y los junta para formar la formula. Dependiendo del resultado hará una cosa u otra. Sin embargo, el segundo método sobre escrito hará que la elipse se mueva.

c. DrawPanel

```java
import java.awt.*;
import javax.swing.*;
import java.util.*;

public class DrawPanel extends JPanel {
    protected LinkedList< Entity > drawables;

    public DrawPanel() {
        drawables = new LinkedList< Entity >();
    }

    public void addDrawable( Entity entity ) {
        drawables.add( entity );
    }

    protected void paintComponent( Graphics g ) {
        super.paintComponent(g);
        for ( int i = 0; i < drawables.size(); ++i )
            drawables.get( i ).draw( g );
    }

    public void translate( int dx, int dy ) {
        for ( int i = 0; i < drawables.size(); ++i )
            drawables.get( i ).translate( dx, dy );
    }
}
```

Tenemos una clase llamada DrawnPanel, donde tenemos un atributo que es una **Linked List de entity.** Está compuesta por el constructor, donde inicializamos la Linked List. Y los métodos **addDrawable**, añade a la entity un drawable, **paintComponent**, pinta el componente mediante un for loop hasta que la i sea más grande que el tamaño del componente, **translate**, hace que las entitys de la lista sean movidas.

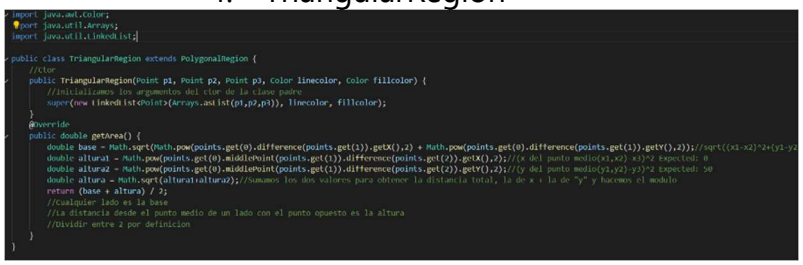
d. Entity

Es una clase abstracta llamada **Entity**. Tiene un atributo protected llamada **lineColor**. Inicializamos el constructor de la clase y le pasamos el parámetro color. Utilizamos getters, **getLIneColor(),** obtenemos el valor ya asignado al atributo que en este caso es lineColor y setters, **setLineColor()**, establece un valor a nuestro atributo. También encontramos métodos abstractos, por lo que harán que la clase sea abstracta. Estos métodos son el método **draw (java.awt.Graphics g)** y **translate (int dx, int dy).**

e. Region

La clase Region también es una clase abstracta que es extendida por la clase Entity. Tiene un atributo protegido llamada **fillColor.** En el constructor inicializamos los atributos de la clase padre y los de esta clase. Creamos un setter del atributo, **setFillColor(Color fillColor)**, para establecer un valor a nuestro atributo. Como en la clase de Entity, creamos dos métodos abstractos, que son las culpables de que la clase sea abstracta. Estos métodos son **getArea()** y **isPointInside(Point p).**

f. TriangularRegion



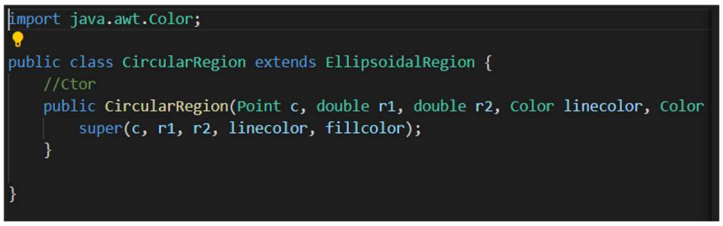
Tenemos una clase llamada **TriangularRegion** extendida de PolygonalRegion. En el constructor de esta clase inicializamos los argumentos del constructor de la clase padre, el cual es una Linked List de Point. A parte de esto, tenemos un método que es sobre escrito que es **getArea(),** en el cual coge cualquier lado, y la convierte en la base. Calcula la distancia desde el punto medio de un lado con el punto opuesto que es la altura. Después de hacer todo esto lo divide entre dos.

g. RectangularRegion

```
import java.awt.Color;
import java.util.Arrays;
import java.util.LinkedList;

public class RectangularRegion extends PolygonalRegion{

    public RectangularRegion(Point p1, Point p2, Point p3, Point p4, Color linecolor, Color fillColor) {
        super(new LinkedList<Point>(Arrays.asList(p1,p2,p3,p4)), linecolor, fillColor);

        //TODO Auto-generated constructor stub
    }
    @Override
    public double getArea() {
        double base = Math.sqrt(Math.pow(points.get(0).difference(points.get(1)).getX(),2)+ Math.pow(points.get(0).difference(points.get(1)).getY(),2));
        double altura = Math.sqrt(Math.pow(points.get(1).difference(points.get(2)).getX(),2)+ Math.pow(points.get(1).difference(points.get(2)).getY(),2));
        double area = base * altura;
        //Lado por lado, cualquiera.
        return area;
    }
}
```

Tenemos una clase llamada **RectangularRegion** extendida de PolygonalRegion. En el constructor de esta clase inicializamos los argumentos del constructor de la clase padre, el cual es una Linked List de Point. A parte de esto, tenemos un método que es sobre escrito que es **getArea(),** en el cual coge la base y la altura y la multiplica para obtener el área.

h. CircularRegion



```
import java.awt.Color;

public class CircularRegion extends EllipsoidalRegion {
    //Ctor
    public CircularRegion(Point c, double r1, double r2, Color linecolor, Color
        super(c, r1, r2, linecolor, fillcolor);
    }

}
```

Tenemos una clase llamada **CircularRegion** extendida de EllipsoidallRegion. En el constructor de esta clase inicializamos los argumentos del constructor de la clase padre, EllipsoidalRegion.

i. Point

```
public class Point {
    //Atributos
    private double x;
    private double y;
    //Ctor
    public Point(double xi, double yi) {
        this.x = xi;
        this.y = yi;
    }
    //Metodos, getters y setters en este caso
    public double getX() {
        return x;
    }
    public double getY() {
        return y;
    }
    public void setX(double x) {
        this.x = x;
    }
    public void setY(double y) {
        this.y = y;
    }
    //Metodo para mover el punto
    public void translate(double x, double y){
        this.x += x; //Sumamos el valor de x que nos pasan a la x que ya teniamos
        this.y += y; //Sumamos el valor de y que nos dan a la y que ya teniamos
    }
    //Metodo para calcular la diferencia entre dos puntos
    public Point difference(Point p){
        Point difference = new Point(this.x-p.getX(), this.y-p.getY());//P= (x1-x2, y1-y2)
        return difference;
    }
    //Metodo para calcular el producto escalar
    public double crossProduct(Point p){
        return (this.x*p.getY() - this.y*p.getX());//Devolemos x1*y2 - y1*x2
    }
```

```
    //Metodo para calcular el punto medio, creado para mas facilidad en la clase Triangular Region
    public Point middlePoint(Point p){
        Point point = new Point(this.x, this.y);
        point.setX((this.x + p.getX())/2);
        point.setY((this.y + p.getY())/2);
        return point;//Devolvemos P = ((x1+x2)/2, (y1+y2)/2)
    }
}
```

Esta clase llamada **Point,** está compuesta por dos atributos que son números reales llamados **x** e **y** e indica las coordenadas de un punto. Y como hemos visto en clase, los atributos han de ser privados.

Además, también tenemos nuestros setters **setX y setY**, que establecen el valor a nuestros atributos. Sin embargo, también utilizamos nuestros getters **getX, getY** para devolver el valor de los atributos. Además, tenemos cuatro métodos: **translate(double x, double y)**, lo utilizamos para mover el punto, **difference(Point p),** lo utilizamos para calcular la diferencia entre dos puntos, **crossProduct(Point p),** con este método calculamos el producto escalar, **middlePoint(Point p),** lo utilizamos para calcular el punto medio(lo hemos creado para que la clase de TriangularRegion sea más fácil).

3. Conclusión

Desde el primer momento que comenzamos a trabajar en este proyecto sabíamos que el hecho de ir comprobando el programa a medida que íbamos avanzando iba a ser la clave para llevar a cabo nuestro proyecto limpio, funcional y entendible. Por el método fallo y error, hemos modulado nuestras funciones y es por eso que, hemos corregido estas impurezas que tuvimos después del primer esbozo de la práctica.

Nos hemos esforzado para seleccionar el código más apropiado para cada clase, aunque en alguna que otra clase como, EllipsoidalRegion o PolygonalRegion nos han llevado a rompernos la cabeza en varias ocasiones. Estas han sido los pocos casos que hemos tenido dificultades y errores graves que solucionar.

Esta práctica ha sido todo un reto debido a las dificultades de la práctica, pero hemos sabido sobreponernos, gracias a la buena organización de los dos participantes de la práctica.

Para acabar, podemos decir que hemos producido un código de calidad, producto del trabajo de calidad, del trabajo en equipo y el método fallo y error.