Aitana González Cárdenas - Álvaro Jiménez Barreno
U166651 - U186356

REPORT PRÀCTICA 4

1. ***An introduction where the problem is described. For example, what should the program do? Which classes do you have to define? Which methods do you have to define? Which methods do you have to implement for these classes?***

En aquesta quarta pràctica, s'ha hagut de donar un canvi d'idea a l'enfoc de l'objectiu. Ara, la idea és treballar exclusivament amb les regions. És a dir, l'objectiu principal ha estat dissenyar més varietat de regions amb diferents formes geomètriques, colors de fons i contorns; calcular les seves àrees, ser capaços de moure les regions a partir d'un input de coordenades i determinar si un punt pertany a una regió o no.

Per coordinar totes aquestes tasques, hem reutilitzat diferents classes i mètodes de la pràctica anterior, però també hem afegit de nous:

1. INTRODUCCIÓ: CLASSES I MÈTODES REUTILITZATS

La pràctica, inicialment, venia amb tres classes: EntityDrawer, DrawPanel i Entity. Aquestes tres classes ens van permetre tenir una idea inicial de quin era l'ordre que havíem de seguir per realitzar la pràctica. D'altra banda, de la pràctica anterior, teníem la classe Point, Region i PolygonalRegion. A partir d'aquestes classes inicials i reutilitzades, vam identificar, com les altres pràctiques, que la classe discreta és Point (tot es construeix a partir de la classe Point). Per tant, va ser la primera classe que vam modificar, què és el següent apartat que expliquem.

D'altra banda, Region no ens cal modificar-ho molt, ja que a la pràctica anterior també es tractava d'una classe abstracta. Tot i així, la classe PolygonalRegion sí l'hem hagut de redefinir amb altres mètodes i, un nou concepte que no havia aparegut abans, el concepte **Override**.

Explicarem la teoria d'override al punt 2, però de forma introductoria esmentem que l'utilitzem Override a les classe PolygonalRegion i EllipsoidalRegion, ja que es fan servir els mateixos mètodes però per a instàncies de diferents classes. Per tant, hem de "duplicar" els mètodes a les dues classes, definint-los de forma diferent en cadascuna.

2. MODIFICACIÓ DE POINT

La classe Point és una classe que portem utilitzant des de la primera pràctica. La definició de Point fins ara ha estat de dos atributs (coordenades), un constructor, i els getters i setters per cada atribut:

```
private double x;
private double y;
```

```
// Constructor
public Point(double x, double y){
    this.x = x;
    this.y = y;
}
```

```java
// Getters

// Obtenim el valor (double) de x.

public double getX() {
    return this.x;
}

// Obtenim el valor (double) de y.

public double getY() {
    return this.y;
}
```

```java
// Setters

// Establim el valor (double) de x.

public void setX(double x) {
    this.x = x;
}

// Establim el valor (double) de y.

public void setY(double y) {
    this.y = y;
}
```

Ara, a més d'aquests atributs i mètodes, hem implementat dos mètodes més:

a) **void translate (double axesX, double axesY):** Per fer un translate, només necessitem dues coordenades i, un cop passades per paràmetre, sumem el valor de cada coordenada a l'actual. Ho hem pensat amb una suma ja que la coordenada pot ser tant positiva com negativa.

```java
public void translate (double axesX, double axesY) {
    x += axesX;
    y += axesY;
}
```

b) **Vector difference (Point p):** Per definir aquest mètode, hem creat primer la classe Vector, però veiem que l'únic que fem és crear una instància de classe Vector (què té els mateixos atributs que Point, ja que Vector es tracta d'una classe que hereta d'aquesta), i restem les coordenades dels atributs de Point (ja que definim el mètode difference dins de la classe) amb les del punt que li passem per paràmetre.

```java
public Vector difference (Point p) {
    Vector dif = new Vector ((x - p.getX()), (y - p.getY()));
    return dif;
}
```

3. IMPLEMENTACIÓ DE VECTOR

Vector és una classe que hereta de Point, ja que l'hem enfocat com un punt amb una direcció. Per tant, hem afegit la classe Vector com una subclasse de la classe Point:

```java
public class Vector extends Point{...}
```

Al constructor només passem per paràmetre els atributs de Point i els incloem mitjançant un super().

```java
// Constructor
public Vector (double x, double y){
    super(x, y);
}
```

D'altra banda, només creem un mètode exclusiu per a la classe Vector, el mètode que calcula el producte creuat. Aquest mètode ens servirá més endavant per calcular si un punt en concret es troba dins d'una regió o no. Per definir el mètode, passem per paràmetre una instància de tipus Vector, i fem el producte vectorial amb la fórmula proporcionada a les instruccions de la pràctica amb les coordenades de la superclasse Point i les coordenades del punt que li passem per paràmetre.

```
public double crossProduct(Vector u) {
    double product = (getX() * u.getY()) - (getY() * u.getX());
    return product;
}
```

4. INHERITANCE D'ENTITY: MODIFICACIÓ DE REGION

Un cop redefinides les dues classes Point i Vector, passem a la classe Region. Aquesta, com s'ha esmentat abans, ja estava creada. Tot i així, hem implementat altres dos altres mètodes:

- isPointInside
- translate

Aquests mètodes s'han implementat a Region de la mateixa forma que vam fer a l'altra pràctica, amb un **públic abstract**, ja que aquesta classe és una classe abstracta i no crearem instàncies de classe Region.

```
public abstract boolean isPointInside(Point p);
public abstract void translate(int dx, int dy);
```

D'altra banda, Region hereta d'Entity, pel que afegim l'*extends*:

```
public abstract class Region extends Entity{...}
```

Això implica que hereta els mètodes i atributs definits a Entity. A més, com veiem a Entity, l'atribut de Region haurà de ser protected i no private, perquè pugui ser visible a les altres subclasses (PolygonalRegion i EllipsoidalRegion).

```
protected Color fillColor;
```

5. INHERITANCE DE REGION: MODIFICACIÓ DE POLYGONALREGION

PolygonalRegion, juntament amb EllipsoidalRegion és la classe en la que hem hagut de definir tots els mètodes relacionats amb el que es demana sobre les regions.

Primer sabem que PolygonalRegion hereta de Region, per tant, afegim l'extends com sempre:

```
public class PolygonalRegion extends Region{...}
```

Els atributs de PolygonalRegion es mantenen com totes les altres pràctiques, doncs afegim una llista de punts que delimitarà la regió poligonal:

```
protected LinkedList<Point> points;
```

En aquest cas, points també ha de ser protected perquè utilitzem els atributs en les subclasses (per exemple a getPoints() de RectarngularRegion). Si fós privat, no seria visible en aquestes classes i, per tant, no es podria utilitzar.

El constructor, com sempre, necessita per paràmetre els seus atributs i els de tota classe que sigui superclasse de PolygonalRegion. Per tant, afegim amb un super els atributs d'Entity i Region, i actualitzem la LinkedList de punts.

```
// Constructor
public PolygonalRegion(LinkedList<Point> points, Color lineColor, Color fillColor) {
    super(lineColor, fillColor);
    this.points = points;
}
```

Addicionalment, hem definit i implementat quatre mètodes més:

a. **getArea()**: Aquest mètode és el mateix que en les altres pràctiques. La funció calcula l'àrea independentment dels costats que tingui aquest polígon (inclòs el triangle). En veure que totes les àrees dels diferents polígons generats les calculava correctament, vam decidir no modificar aquest mètode.

b. **draw(Graphics g):** Aquest mètode ens permet imprimir per pantalla els diferents polígons i també estava implementat d'anteriors pràctiques. A diferència de l'anterior, ara hem afegit els colors de la funció setColors amb els atributs de les superclasses Region i Entity que, com són protected, els podem utilitzar.

```
@Override
public void draw(Graphics g) {
    int nPoints = points.size();
    int xPoints[] = new int[nPoints];
    int yPoints[] = new int[nPoints];

    for (int i = 0; i < points.size(); i++) {
        xPoints[i] = (int) points.get(i).getX();
        yPoints[i] = (int) points.get(i).getY();
    }
    g.setColor(lineColor);
    g.drawPolygon(xPoints, yPoints, nPoints);
    g.setColor(fillColor);
    g.fillPolygon(xPoints, yPoints, nPoints);
}
```

c. **isPointInside (Point p):** Amb aquest mètode podem declarar, especificat un punt, si es troba dins de l'àrea d'una regió o si es troba fora. Partim de la fórmula següent:

$$(q_2 - q_1) \times (p - q_1)$$

Al mètode comparem els resultats que ens dona l'equació a partir de les coordenades de dos punts. Si són menors que 0, retornarà **FALSE** (punt no inclòs), sinó, **TRUE** (punt inclòs). Abans del bucle for in, calculem la diferència del primer punt amb l'últim, per evitar-nos problemes d'out of range. Així, a més, podíem anar comparant els valors de cada equació obtinguda a les iteracions per poder acabar la funció en quant trobéssim dos vectors el qual el seu producte creuat donés menor que 0.

```java
@Override
public boolean isPointInside(Point p) {
    int nPoints = points.size();

    Point q1 = points.get(nPoints-1);
    Point q2 = points.get(0);

    Vector dif1 = q2.difference(q1);
    Vector dif2 = p.difference(q1);
    double equation1 = dif1.crossProduct(dif2);
```

```java
    for(int i=0; i < nPoints-1; i++){
        q1 = points.get(i);
        q2 = points.get(i+1);
        dif1 = q2.difference(q1);
        dif2 = p.difference(q1);
        double equation2 = dif1.crossProduct(dif2);

        if (equation1 * equation2 < 0) {
            return false;
        }
    }
    return true;
}
```

d. **translate(int dx, int dy):** En aquest mètode obtenim la llargada de les instàncies de la llista de punts, que ens servirà per fer un loop for. Per cada iteració, cridem al mètode translate, definit a Point i així traslladar tots els punts que formen el polígon.

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

Tots els mètodes esmentats (excepte el constructor) necessiten d'un @Override, ja que els duplicarem en la classe EllipsoidalRegion. A més, com les dues classes no són abstract, però sí inheritance d'una classe abstract, s'ha de fer un override de tots els mètodes heretats de la superclasse.

6. INHERITANCE DE REGION: CREACIÓ DE ELLIPSOIDALREGION

Tal com hem esmentat abans, la classe EllipsoidalRegion també definim els quatre mètodes que hem fet a PolygonalRegion, amb el seu constructor. Tot i així, tant els atributs com els mètodes estan definits de diferent forma.

EllipsoidalRegion també hereta de Region, per tant ho declarem amb *extend*:

```java
public class EllipsoidalRegion extends Region{...}
```

Els atributs que hem declarat per aquesta classe són un punt (c) i dos diàmetres (r1, r2).

```java
    private Point c;
    private double r1, r2;
```

Al constructor, passem per paràmetre els tres atributs de la classe i els dos de les seves superclasses. Is incloem amb el super() i actualitzem els valors dels atributs de la classe.

```java
// Constructor
public EllipsoidalRegion(Point c, double r1, double r2, Color lineColor, Color fillColor) {
    super(lineColor, fillColor);
    this.c = c;
    this.r1 = r1;
    this.r2 = r2;
}
```

Seguidament, definim els altres quatre mètodes, tal com hem fet a PolygonalRegion:

a. **getArea():** Utilitzem la fórmula del càlcul de l'àrea d'una elipse $A = \pi \cdot a \cdot b$, calculem l'àrea de la regió elipsoidal. A més, com es tracta d'una regió circular, pot comptar amb molts decimals. Per tant, utilitzem la funció round() i arrodonim a 4 decimals.

```
@Override
public double getArea() {
    double area = Math.PI * (r1/2) * (r2/2);
    return Math.round(area*10000.0)/10000.0;
}
```

b. **draw(Graphics g):** Per imprimir per pantalla les regions elipsoidals, creem dos variables amb les coordenades del punt, i fem servir la funció drawOval() i fillOval() amb les coordenades x i y del punt i els dos diàmetres (r1 i r2). Com hem fet també a PolygonalRegion, per la funció setColor() establim els atributs de Regió i Entity, així el contorn de la regió és diferent al color de dins de la regió.

```
@Override
public void draw(Graphics g) {
    int pointX = (int) c.getX();
    int pointY = (int) c.getY();

    g.setColor(lineColor);
    g.drawOval(pointX, pointY, (int) r1, (int) r2);
    g.setColor(fillColor);
    g.fillOval(pointX, pointY, (int) r1, (int) r2);
}
```

c. **isPointInside(Point p):** Per saber si un punt està dins de la regió elipsoidal, ha de seguir la següent fórmula. $$\frac{(p_x - c_x)^2}{a^2} + \frac{(p_y - c_y)^2}{b^2} \leq 1,$$

Veiem al nostre mètode, que dividim la equació en diverses variables que, en ser sumades, ha de ser menor o igual que 1 (retornem **TRUE**) i, si no compleix la igualtat, retornem **FALSE**.

```
@Override
public boolean isPointInside (Point p) {
    double beforesum = (Math.pow((p.getX() - c.getX()), 2)) / (Math.pow(r1, 2));
    double aftersum = (Math.pow((p.getY() - c.getY()), 2)) / (Math.pow(r2, 2));
    double equation = beforesum + aftersum;
    if(equation <= 1){
        return true;
    }
    return false;
}
```

d. **translate(int dx, int dy):** Per traslladar els punts de l'elipse, creem dues variables que siguin les coordenades del punt c amb els getters. Un cop obtinguda, establim que la nova coordenada del punt c serà la obtinguda amb el getX, getY més el valor que li passem per paràmetre a la funció, establint-ho amb els setters.

```
@Override
public void translate(int dx, int dy) {
    double px = c.getX();
    double py = c.getY();
    c.setX(px + dx);
    c.setY(py + dy);
}
```

En aquest mòdul també hem fet Override, per la mateixa explicació que hem esmentat amb PolygonalRegion.

7. INHERITANCE DE POLYGONALREGION: CLASSES RECTANGULARREGION I TRIANGULARREGION

Veiem que PolygonalRegion és una superclasse de dues altres classes més:

7.1. RECTANGULARREGION

En aquesta classe, que hereta de PolygonalRegion, no necessitem cap atribut, doncs és necessària només la llista de punts que ja passem a PolygonalRegion. Per tant, només utilitzarem un extend.

```
public class RectangularRegion extends PolygonalRegion{...}
```

Els mètodes d'aquesta classe només són el constructor, el mètode de càlcul d'àrea i el mètode per obtenir tots els punts d'un rectangle:

a. **Constructor:** En constructor només cal passar per paràmetre els atributs de PolygonalRegion, Region i Entity, definits amb un super().

```
// Constructor
public RectangularRegion(LinkedList<Point> points, Color lineColor, Color fillColor) {
    super(points, lineColor, fillColor);
}
```

b. **getPoints(LinkedList<Points> points):** Aquest és un mètode essencial perquè el rectangle faci totes les funcions que ha der fer. Això és així ja que la llista de punts que passem a RectangularRegion només conté dos punts. Per tant, hem creat un mètode per obtenir els altres dos i que sigui més fàcil treballar amb instàncies d'aquesta classe.

   Els dos primers són els punts que podem guardar en variables auxiliars. Aquests representen la diagonal del rectangle. El tercer punt no és més que punt 1 coordenada X i punt 2 coordenada Y. I finalment, el quart és l'oposat: punt 1 coordenada Y i punt 2 coordenada X.

   Un cop trobats els quatre punts, fem un clear de la llista de punts i els afegim. És una forma d'actualitzar-la amb tots els punts del rectangle sense repetir cap.

```
public void getPoints(LinkedList<Point> points) {
    Point p1 = points.get(0);
    Point p3 = points.get(1);
    Point p2 = new Point(p1.getX(), p3.getY());
    Point p4 = new Point(p3.getX(), p1.getY());

    points.clear();
    points.add(p1);
    points.add(p2);
    points.add(p3);
    points.add(p4);
}
```

c. **getArea():** Per calcular l'àrea del rectangle, com hem definit el mètode per obtenir tots els punts, cridem a getPoints i fem un super del càlcul de l'àrea, ja que ara només es tracta d'una llista de punts que genera un polígon de quatre costats, i ho podem interpretar com una PolygonalRegion qualsevol.

```
public double getArea() {
    getPoints(points);
    return super.getArea();
}
```

7.2. TRIANGULARREGION

Per a TriangularRegion ens passa una mica el mateix, però més senzill, ja que la llista que li passem conté exactament els nombre de punts d'un triangle. Com hereta de PolygonalRegion, incloem l'*extends*.

```
public class TriangularRegion extends PolygonalRegion{...}
```

Pel que fa al constructor, només afegim els atributs de les superclasses com a paràmetres i amb un super(), tal com ho fem amb el constructor de RectanularRegion.

```
// Constructor
public TriangularRegion(LinkedList<Point> points, Color lcinit, Color fillColor) {
    super(points, lcinit, fillColor);
}
```

Pel que fa a getArea(), cridem directament a la funció getArea() de PolygonalRegion amb un super().

```
public double getArea() {
    return super.getArea();
}
```

8. INHERITANCE DE ELLIPSOIDALREGION: CLASSE CIRCULARREGION

La classe CircularRegion hereta d'EllipsoidalRegion:

```
public class CircularRegion extends EllipsoidalRegion{...}
```

Passa el mateix amb les classes TriangularRegion i RectangularRegion. Com es tracta d'una el·lipse amb els dos diàmetres iguals, pot heretar els atributs d'EllipsoidalRegion.

Per tant, al constructor només es passarà per paràmetre els atributs d'EllipsoidalRegion, Region i Entity, que declararem dins amb un super().

```
// Constructor
public CircularRegion(Point c, double r1, Color lineColor, Color fillColor) {
    super(c, r1, r1, lineColor, fillColor);
}
```

El mateix passa amb getArea(). Ja hem definit com es calcula l'àrea d'una el·lipse de qualsevol valor r1 i r2. Per tant, només cal cridar a getArea() d'EllipsoidalRegion amb un super().

```
public double getArea() {
    return super.getArea();
}
```

9. PROVEM EL CODI: MÒDUL TESTLAB

Un cop definides totes les classes necessàries i implementat tots els mètodes que ens fan falta perquè el codi ens funcioni correctament, creem el mòdul TestLab i fem totes les proves necessàries per demostrar que el codi ens funciona. Per fer-ho:

- Creem una instància de classe Entity.
- Creem llistes de punts que definiran diferents instàncies de les subclasses de PolygonalRegion (RectangularRegion i TriangularRegion).
- També declarem diverses variables per crear instàncies de classe EllipsoidalRegion i de la seva subclasse, CircularRegion.
- Per cada instància, calculem l'àrea i comprovem si un punt aleatori es troba dins o fora de la regió.
- Després de la comprovació de l'àrea i el punt, afegim la instància a EntityDrawer perquè, un cop fem Run del programa, apareixi la instància per pantalla. D'aquesta forma, comprovem que sí existeix i es mostra per pantalla, que té els colors establerts i, per tant, la funció draw és correcte; i que podem traslladar les regions amb un input per la coordenada X i la coordenada Y.

```
// Creació d'una EntityDrawer per poder mostrar per pantalla les regions que formin part d'aquesta.
EntityDrawer draw1 = new EntityDrawer();
```

```
// Creació llista de punts p1 (figura --> rectangle)
LinkedList<Point> p1 = new LinkedList<Point>();
p1.add(new Point(10,10));
p1.add(new Point(110,510));

// Creació de RectangularRegion rectangle, amb les seves coordenades, color de vora i color de "relleno".
RectangularRegion rectangle = new RectangularRegion(p1, Color.magenta, Color.blue);
// Imprimim l'àrea del rectangle.
System.out.println("\nArea of the Rectangular Region: " + rectangle.getArea() + " u^2.");
// Afegim la regió rectangular a la EntityDrawer per poder dibuixar-la.
draw1.addDrawable(rectangle);

// Creació de dos punts aleatoris.
Point testPoint1 = new Point(10,300);
Point testPoint2 = new Point(111,300);
// Mirem si els punts creats pertanyen o no al rectangle (pertany = true, no pertany = false).
System.out.println("Point (" + testPoint1.getX() + "," + testPoint1.getY() + ") is in the rectangle: " + rectangle.isPointInside(testPoint1) + ".");
System.out.println("Point (" + testPoint2.getX() + "," + testPoint2.getY() + ") is in the rectangle: " + rectangle.isPointInside(testPoint2) + ".");
```

```
// Creació llista de punts p2 (figura --> triangle)
LinkedList<Point> p2 = new LinkedList<Point>();
p2.add(new Point(300, 50));
p2.add(new Point(500, 200));
p2.add(new Point(300, 250));

// Creació de TriangularRegion triangle, amb les seves coordenades, color de vora i color de "relleno".
TriangularRegion triangle = new TriangularRegion(p2, Color.green, Color.gray);
// Imprimim l'àrea del triangle.
System.out.println("\nArea of the Triangular Region: " + triangle.getArea() + " u^2.");
// Afegim la regió triangular a la EntityDrawer per poder dibuixar-la.
draw1.addDrawable(triangle);

// Creació de dos punts més aleatoris.
Point testPoint3 = new Point(350,150);
Point testPoint4 = new Point(250,100);
// Mirem si els punts creats pertanyen o no al triangle (pertany = true, no pertany = false).
System.out.println("Point (" + testPoint3.getX() + "," + testPoint3.getY() + ") is in the triangle: " + triangle.isPointInside(testPoint3) + ".");
System.out.println("Point (" + testPoint4.getX() + "," + testPoint4.getY() + ") is in the triangle: " + triangle.isPointInside(testPoint4) + ".\n");
```

```
// Creació de dos radis (r1 i r2) i un centre (centre 1) (figura --> ellipsoide).
// Creació d'un radi (r1) i un centre (centre2) (figura --> circle).
double r1 = 100;
double r2 = 300;
Point centre1 = new Point(150, 150);
Point centre2 = new Point(300, 300);

// Creació de EllipsoidalRegion ellipsoide, amb el seu centre, els seus dos radis, color de vora i color de "relleno".
EllipsoidalRegion ellipsoide = new EllipsoidalRegion(centre1, r1, r2, Color.black, Color.green);
// Imprimim l'àrea de l'elipse.
System.out.println("\nArea of the Ellipsoidal Region: " + ellipsoide.getArea() + " u^2.");
// Afegim la regió elipsoidal a la EntityDrawer per poder dibuixar-la.
draw1.addDrawable(ellipsoide);

// Creació de dos punts més aleatoris.
Point testPoint5 = new Point(170, 250);
Point testPoint6 = new Point(250, 350);
// Mirem si els punts creats pertanyen o no a l'elipse (pertany = true, no pertany = false).
System.out.println("Point (" + testPoint5.getX() + "," + testPoint5.getY() + ") is in the ellipse: " + ellipsoide.isPointInside(testPoint5) + ".");
System.out.println("Point (" + testPoint6.getX() + "," + testPoint6.getY() + ") is in the ellipse: " + ellipsoide.isPointInside(testPoint6) + ".\n");

// Creació de CircularRegion circle, amb el seu centre, els seus radi, color de vora i color de "relleno".
CircularRegion circle = new CircularRegion(centre2, r1, Color.blue, Color.magenta);
// Imprimim l'àrea de la circumferència.
System.out.println("\nArea of the Circular Region: " + circle.getArea() + " u^2.");
// Afegim la regió circular a la EntityDrawer per poder dibuixar-la.
draw1.addDrawable(circle);

// Creació d'un punt més aleatori.
Point testPoint7 = new Point(400, 300);
// Mirem si els punts creats pertanyen o no a la circumferència (pertany = true, no pertany = false).
System.out.println("Point (" + testPoint7.getX() + "," + testPoint7.getY() + ") is in the circle: " + circle.isPointInside(testPoint7) + ".");
System.out.println("Point (" + centre1.getX() + "," + centre1.getY() + ") is in the circle: " + circle.isPointInside(centre1) + ".\n");
```

2. ***A description of possible alternative solutions that were discussed, and a description of the chosen solution and the reason for choosing this solution rather than others. It is also a good idea to mention the related theoretical concepts of object-oriented programming that were applied as part of the solution.***

Com a concepte nou teòric, ens hem enfocat principalment en l'Override:

- **Override:** La definició d'override principalment és el concepte de sobreescriure un mètode existent de la superclasse en la subclasse. També es pot fer override per tractar dues versions del mateix mètode, què és precisament pel que ho hem utilitzat. Molts mètodes els hem definit amb un Override tant a PolygonalRegion com a EllipsoidalRegion, ja que la tasca era la mateixa però no es podia calcular de la mateixa forma.

Addicionalment, també veiem que les relacions d'herència entre classes també tenen importància, doncs veiem una gran cadena de relacions d'herència. Això comporta una relació d'atributs i mètodes concreta, i l'hem vist molt sobretot en les classes més específiques, com RectangularRegion, TriangularRegion i CircularRegion.

D'altra banda, com a solucions alternatives a la resolució del nostre programa, vam pensar alternativament els constructors de les classes RectangularRegion i TriangularRegion indicant la quantitat de punts (amb un array de punts), i no passar una llista de punts. D'aquesta forma, no hauria fet falta el mètode de getPoints() a RectangularRegion i podríem haver treballat directament amb els 4 o 3 punts en els altres mètodes (getArea(), isPointInside() i draw()).

```
// Constructor
public RectangularRegion(Point p1, Point p2, Color lineColor, Color fillColor) {
    super(new LinkedList<Point>(Arrays.asList(p1, new Point(p2.getX(), p1.getY()), p2, new Point(p1.getX(), p1.getY()))), lineColor, fillColor);
}
```

```
// Constructor
public TriangularRegion(Point p1, Point p2, Point p3, Color lcinit, Color fillColor) {
    super(new LinkedList<Point>(Arrays.asList(p1, p2, p3)), lcinit, fillColor);
}
```

Tot i així, vam decidir deixar-ho amb una llista de punts, ja que ens semblava més genèric i les explicacions proporcionades a la classe introductòria de la pràctica ens van especificar que era correcte passar-li la llista de PolygonalRegion.

3. ***A conclusion that describes how well the solution worked in practice, i.e. did the tests show that the classes vere correctly implemented? You can also mention any difficulties during the implementation as well as any doubts you might have had.***

Com a conclusions generals de la pràctica, ens hem vist més solvents i coordinats relacionats amb les altres. Aquest fet ha sigut més significatiu ja que hem trobat que aquesta pràctica era una mica més complicada que les anteriors, a més, se li ha de sumar que hem tingut una setmana menys per realitzar-la, per tant, ens hem hagut d'organitzar molt millor. No obstant, creiem que ens n'hem sortit força bé tot i no haver tingut temps de realitzar la part opcional.

D'altra banda, la redacció d'un informe després d'aquesta pràctica ens ha ajudat a assimilar tant els continguts teòrics, en aquest cas, l'herència múltiple, com els continguts pràctics realitzats a les diferents classes de l'assignatura.

Tanmateix, volem destacar un parell d'inconvenients que s'han generat al llarg de la realització de la pràctica que podem dir que hem resolt.

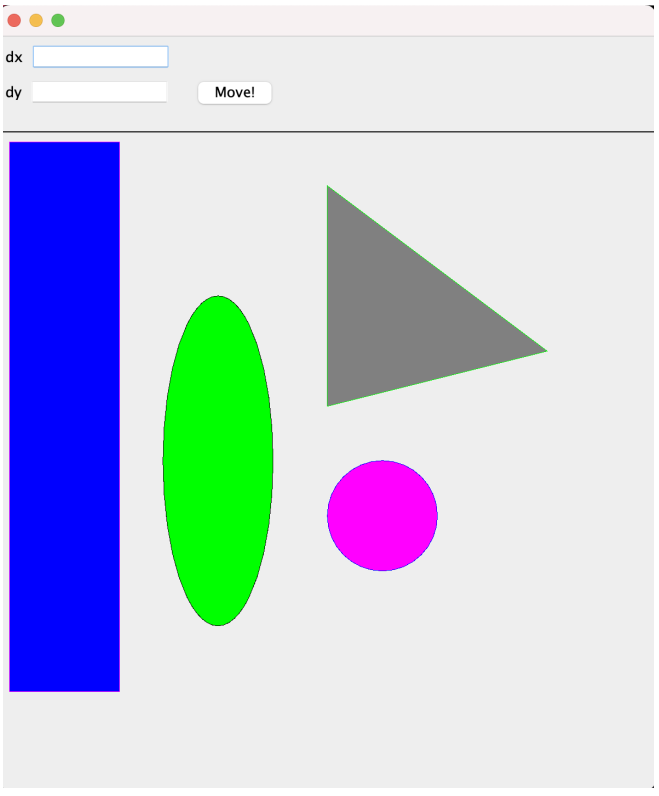
En un principi, vam crear la instància de rectangle pero creada a partir de 4 punts. No obstant, a classe es va dir que construíssim el rectangle a partir de 2 punts (els que formen una diagonal). D'aquesta manera, vam tenir el problema d'haver d'implementar un nou mètode, getPoints(), per aconseguir els altres dos punts. Un cop creat aquest mètode no sabíem ben bé on implementar-lo. Per aquest motiu vam provar d'implementar-lo en els diferents mètodes de RectangularRegion. En un primer moment, vam decidir cridar-lo a la funció isPointInside(), no obstant, quan mostravem per pantalla les nostres instàncies, només se'ns dibuixava la diagonal del rectangle. Ens va passar una cosa semblat a la funció draw(). Finalment, vam decidir implementar-la al mètode getArea(), i en aquest cas, ens va funcionar.

Altrament, volíem demostrar que les diferents funcions getArea() i isPointInside() funcionaven correctament. Vam crear les funcions (amb override) i, un cop acabat el programa, vam crear instàncies dels diferents tipus de regions. Per una banda, vam printejar les àrees (calculades al codi) i, simultàniament, vam calcular-les de forma manual. Així doncs, quan vam tenir diversos exemples que ens coincidien, vam acabar concloent que les funcions estaven correctes. D'altra banda, vam cridar a la funció isPointInside() per, donat un punt (testPoint), comprovar si aquest estava dins o fora del nostre polígon / regió. D'aquesta manera, ens vam dibuixar les figures en un paper i vam fer els càlculs necessaris per veure si el punt pertenyia o no a la regió, així mateix, quan vam veure que ens coincidien les respostes, vam poder determinar que les funcions estaven correctament implementades.

```
Area of the Rectangular Region: 50000.0 u^2.
Point (10.0,300.0) is in the rectangle: true.
Point (111.0,300.0) is in the rectangle: false.

Area of the Triangular Region: 20000.0 u^2.
Point (350.0,150.0) is in the triangle: true.
Point (250.0,100.0) is in the triangle: false.


Area of the Ellipsoidal Region: 23561.9449 u^2.
Point (170.0,250.0) is in the ellipse: true.
Point (250.0,350.0) is in the ellipse: false.


Area of the Circular Region: 7853.9816 u^2.
Point (400.0,300.0) is in the circle: true.
Point (150.0,150.0) is in the circle: false.
```

Finalment, el resultat del nostre programa apareix de la següent forma:

PRINT DE LA NOSTRA ENTITY



Podem observar que cada polígon, elipsoide o regió té diferent color:

- Rectangle: fillColor = blue
- Triangle: fillColor = gray
- Ellipsoid: fillColor = green
- Circle: fillColor = magenta

Així mateix, cadascuna té un color de vora diferent:

- Rectangle: lineColor = magenta
- Triangle: lineColor = green
- Ellipsoid: lineColor = black
- Circle: lineColor = blue

PRINT DE LA NOSTRA ENTITY MOGUDA (dx = 60, dy = 20)