Ayesha Quinto Gammuac 241789
Minh Nguyen Phuong 241841

24292 - Object-Oriented Programming

LAB 3: Implementing a design based on inheritance

REPORT

THE PROGRAM

1. MAIN PURPOSE AND DESCRIPTION

The aim of this report is to describe how we have done lab 3, including all the problems we encountered and our solutions. The objective of the program is to create a graphical application which draws and moves regions. In this lab, we were given the Java code that includes a graphical interface as well as the abstract Entity class: *DrawPanel.java, Entity.java* and *EntityDrawer.java.* We made a copy of the class *Point.java* from lab 2 and *PolygonalRegion.java* from lab 3. We also created new classes: *Vector.java, Region,java, EllipsoidalRegion.java, TriangularRegion.java, RectangularRegion.java, CircularRegion.java, Color.java*, *RGBColor.java, HSVColor.java* and *Color.java,* and some testing classes.

1.1. Points and vectors

To begin, we copied the class *Point.java* we had implemented from lab 2 and created a new class called *Vector.java.* In class *Point.java,* we added a function *translate()* to move a point to another position, and *difference()*, which is a vector or we can say the distance between 2 points calculated by: $p - q = (p_x - q_x, p_y - q_{y)}$. The class *Vector.java* is similar. Besides the basic attributes and functions as in class *Point.java,* we also implemented another function called *crossProduct(),* which returns a real value calculated by the following equation: $u \times v = u_x v_y - u_y v_x$.

1.2. Entities and Regions

The elements that will be drawn with the application are called entities. An entity can be a line, a text or a region, but in this lab we only implement regions as entities. The entity and the region classes are abstract since we will create instances of them in the following subclasses. Their methods *translate(), draw(), isSelected* and *isPointInside()* will be overridden in their subclasses.

1.3 Polygonal Regions

Having done the class *PolygonalRegion.java* in the previous lab, we just reused the code and added more functions. As in inheritance, it extends *Region.java*. In addition to the methods we already had, we implemented the function *translate(),* which received the values of x and y coordinates and moved the region to another position. We used a loop to move the region point by point. We then implemented two boolean functions which are *isSelected()* and *isPointInside()* to check if the point is inside of the region and if yes then is it selected. Following the instructions of the lab, to see if a point is inside a convex polygon the following should hold. For this, we used the classes Point and Vector and their methods *difference()* and *crossProduct()*.

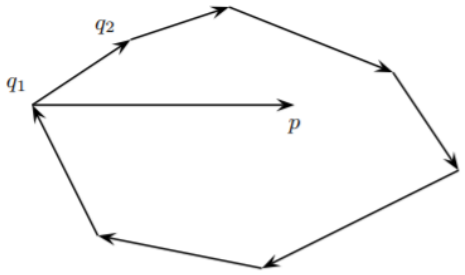

Figure 1: Determine whether a point p is inside a convex polygon.

The point p is inside of a convex polygon if the *sign* of the cross-product $(q_2 - q_1) \times (p - q_1)$ is the same for all pairs of consecutive points q_1 and q_2 of the polygon. To obtain the vectors $(q_2 - q_1)$ and $(p - q_1)$ you can simply use the new method `difference` of the Point class. Recall that the sign of a number n is $+1$ if n is positive and -1 if n is negative.

This is how we implemented it in the code:

```
public boolean isPointInside( Point p ) {
    double cp = 0; //cross-product initialized as 0 (no sign)

    //check for each pair of vectors of the polygon
    for ( int i = 0; i < ListP.size()-1; i++ ) {
        Vector v1 = ListP.get(i+1).difference( ListP.get(i) ); //(q2-q1)
        Vector v2 = p.difference( ListP.get(i) );              //(p -q1)
        double newcp = v1.crossProduct(v2);            //(q2-q1)x(p -q1)

        // check the sign of the cross-product
        if ( cp < 0 & newcp > 0 ) { return false; }
        else if ( cp > 0 & newcp < 0 ) { return false; }
        else cp = newcp;
    }
    //check the last vector (qn-q1):
    Vector v1 = ListP.get(0).difference( ListP.get( ListP.size()-1 ) ); //(qn-q1)
    Vector v2 = p.difference( ListP.get(0) );                           //(p -q1)
    double newcp = v1.crossProduct(v2);                         //(qn-q1)x(p -q1)

    // check the sign of the cross-product
    if ( cp < 0 & newcp > 0 ) { return false; }
    else if ( cp > 0 & newcp < 0 ) { return false; }
    return true;
}
```

1.3.1 Triangular and Rectangular Regions

Two basic polygons are triangles and rectangles, with 3 and 4 points each respectively. We created these two subclasses of PolygonalRegion with the keyword *extends*. In both cases, we pass as the list of points an array of points converted to LinkedList<Point> using the imports *java.util.Arrays* and *java.util.LinkedList*.

1.4. Ellipsoidal Regions

For ellipsoidal regions, we reused some of the structure of PolygonalRegion. However, in this case it has as attributes a Point *center* and the integers *width* and *height*. The methods *translate()* and *isSelected()* are very similar. As for the *isPointInside()* method, we applied the given formula, where *p* is the Point we are given, the *a* and *b* are the width and height respectively and *c* the center point of the ellipsoid.

$$\frac{(p_x - c_x)^2}{a^2} + \frac{(p_y - c_y)^2}{b^2} \leq 1$$

In the draw() method we use the Graphics functions we implemented in the previous lab to draw cities and lakes: *fillOval()* and *drawOval()*. However, these functions will draw the oval given the upper left corner so we had to make modifications.

1.4.1 Circular Regions

A specific subclass of EllipsoidalRegion is the CircularRegion which is an ellipsoid but with the same values in width and height. Therefore, this class extends the *EllipsoidalRegion.java* and calls super() with the same radius.

1.5. Color

For this lab, we had to implement a class *Color.java* that would have methods to convert from rgb model to hsv model and vice versa. To do that, we decided to create three classes: abstract *Color.java*, *RGBColor.java* and *HSVColor.java*. The latter two inherit from Color and the first one extends the already existing java.awt.Color. In order to correctly implement each method, we searched about the Java library Color and its methods and about the conversion of rgb and hsv models. Specifically, we had to be careful with what we sent as parameters when calling the constructor of java.awt.Color since it could receive integers [0,255] for rgb models or floats [0,1] for hsv models.

1.6. Graphical Interface

In order to use the graphical application, we use the class *EntityDrawer.java* and create another class for testing it called *TestApp.java*. In this testing, we create an instance of EntityDrawer, then create instances of regions (Polygonal, Triangular, Rectangular, Ellipsoidal and Circle regions) and store them in our EntityDrawer using the method addDrawable so that these entities will be in the list drawables. When launching the application, the entities added are drawn. In addition, we have the Move tool, which given x and y coordinates it will move the entities accordingly.

2. OPTIONAL PART: Selection tool

We have also implemented the selection tool which allows the user to select entities and move only those selected. To make this possible, as the instructions of the teacher stated, we used the Java library *java.awt.event Mouselistener* and *MouseEvent* in our *DrawPanel.java*. We added the attribute *LinkedList<Entity> selection* where the selected entities will be stored and we overrode the methods of MouseListener. The method that we had to build its body is the *mousePressed()* which selects a subset of regions. We do it with a for loop with all the entities in the list, if the entity is a region and isPointInside() (which receives the Point where the mouse is selecting) returns true, then it will be added in the selection list.

Apart from implementing these methods in DrawPanel, we needed to modify the *translate()* method and implement the isSelected() methods. An entity is selected if *isPointInside(Point p)* holds for that specific entity. The *translate()* method has been modified so that only the entities in the list selection are moved.

3. CONCLUSION

This lab was longer and a lot more difficult compared to the previous labs. We were struggling with the constructors of the Triangular, Rectangular and Circular Regions. However, we did a lot of research on the error and could finally solve it using *java.util.Arrays* Arrays.asList(...) instead of new Point[] {...}. We also had some problems implementing the class *Color.java* since there is already a Java library Color and it had different constructors. But we could manage it and the graphical application works!