Ayesha Quinto Gammuac 241789
Minh Nguyen Phuong 241841

24292 - Object-Oriented Programming

LAB 2: Implementing a program design

REPORT

THE PROGRAM

1. MAIN PURPOSE AND DESCRIPTION

The aim of this report is to describe how we have done lab 2, including all the problems we encountered and our solutions. The objective of the program is to create a map of a world with different continents and regions which are created by using coordinates on the screen. There are in total five classes that we had to create: *Point.java, PolygonalRegion.java, TestPolygonalRegion.java, Continent.java* and *World.java*. There are also two classes given by the teacher that we need to use and/or modify: MyMap and MyWindow. In the following paragraphs we will describe each class and its implementation.

1.1. Point

This class has already been discussed and implemented in the previous lab, so we only had to make some modifications to adequate it for this lab's purpose. *Point.java* consists of the attributes x-coordinate *x* and y-coordinate *y*. We considered both attributes as integers since in class *Polygonalregion.java* we decided to use the *drawPolygon()* which takes two arrays of integers as arguments. The methods of Point are the constructor *Point()*, the getters *getx()* and *gety()* and a void method *printPoint()* to print the point in String format.

1.2. PolygonalRegion

The points from class Point() will be used to create geometric forms that will represent the regions of our desired map. This class' only attribute is a list of points *LinkedList<Point> ListP* with private visibility. We have used the java util LinkedList for all the lists we have in the whole program since it is more convenient and efficient to work with. As for the public methods, we need one to compute the area of a region and another one to draw the region.

Ayesha Quinto Gammuac 241789
Minh Nguyen Phuong 241841

$$\text{Area} = \frac{1}{2}\begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ \vdots & \vdots \\ x_n & y_n \\ x_1 & y_1 \end{vmatrix} = \frac{1}{2}\left[(x_1y_2 + x_2y_3 + x_3y_4 + \cdots + x_ny_1) - (y_1x_2 + y_2x_3 + y_3x_4 + \cdots + y_nx_1)\right]$$

To get the area of a polygonal form we must use the formula suggested by the teacher, assuming that this polygon is convex. We have divided this equation in three parts to code it more easily:

```
double area = 0.5 * (first - second);
```

To compute the first and second parts we use a for loop that iterates through all the list of points except for the last element since for each of them we need to use either *i* or *i+1*:

```
first += (ListP.get(i)).getx() * (ListP.get(i+1)).gety();
second += (ListP.get(i)).gety() * (ListP.get(i+1)).getx();
```

Then, outside of the loop, we only need to sum in the computation that requires the last and first element of the list:

```
first += (ListP.get(ListP.size()-1)).getx() * (ListP.get(0)).gety();
second += (ListP.get(ListP.size()-1)).gety() * (ListP.get(0)).getx();
```

For the method to draw the region, we decided to use the method from the Graphics g *drawPolygon()* which draws the entire polygon at once. This method requires the size of the list of points and two arrays of integers *xList[]* and *yList[]* as arguments, so we created these arrays and put each x and y coordinates in them using a for loop. The call *g.drawPolygon()* is made at the last line of the method.

In order to know that we have implemented this class correctly, we created another class called *TestPolygonalRegion.java*. Our test creates three points *p1*, *p2*, *p3* with the constructor *Point()* and puts each of them in a list *list* which would be used to create a polygonal region *Poly* using the constructor *PolygonalRegion()*. Then, it calls the method to compute the region's area *Poly.getArea()* and prints it. It also prints the tree points. We used the same points as the ones in the suggested web example so we could check that the area is correctly computed and it all worked. Apart from that, we also tested it in the graphical interface as stated in the pdf and it worked.

1.3. Continent

Once the Point and PolygonalRegion classes have been implemented, we need to group the regions in continents and that is done in the class *Continent.java*. The attribute is the list of polygonal regions *LinkedList<PolygonalRegion> ListR* with private visibility. The necessary public methods are the following: *Continent()*, *getContinentArea()*, *drawContinent()*. Since the total area of a continent is the sum of the areas of its regions, we use a for loop iterating over the whole *ListR* to get their areas and storing them in the variable *totalArea* that we will return.

```
public double getContinentArea() {
    double totalArea = 0;
    for ( int i = 0; i < ListR.size(); i++) {
        totalArea += (ListR.get(i)).getArea();
    }
    return totalArea;
}
```

In order to draw the continent, we used the same pattern we have been using often in the methods: a for loop that iterates through all the list *ListR* calling the method *drawRegion()* for each element.

```
public void drawContinent( Graphics g ) {
    for ( int i = 0; i < ListR.size(); i++) {
        (ListR.get(i)).drawRegion(g);
    }
}
```

1.4. World

Now that the continents can be created, we only need to group the whole thing in the class *World.java*. Again, its attribute is the list of all the continents *LinkedList<Continent> ListC*. Also, apart from the constructor *World()*, we will need the same methods as in the *Continent.java* class: *getWorldArea()*, *drawWorld()*. The only modifications needed were to change the list used to *ListC* and the name of the methods that are called inside of them, such that:

```
    public double getWorldArea() {
        double worldArea = 0;
        for ( int i = 0; i < ListC.size(); i++) {
            worldArea += (ListC.get(i)).getContinentArea();
        }
        return worldArea;
    }
    public void drawWorld( Graphics g ) {
        for ( int i = 0; i < ListC.size(); i++) {
            (ListC.get(i)).drawContinent(g);
        }
    }
```

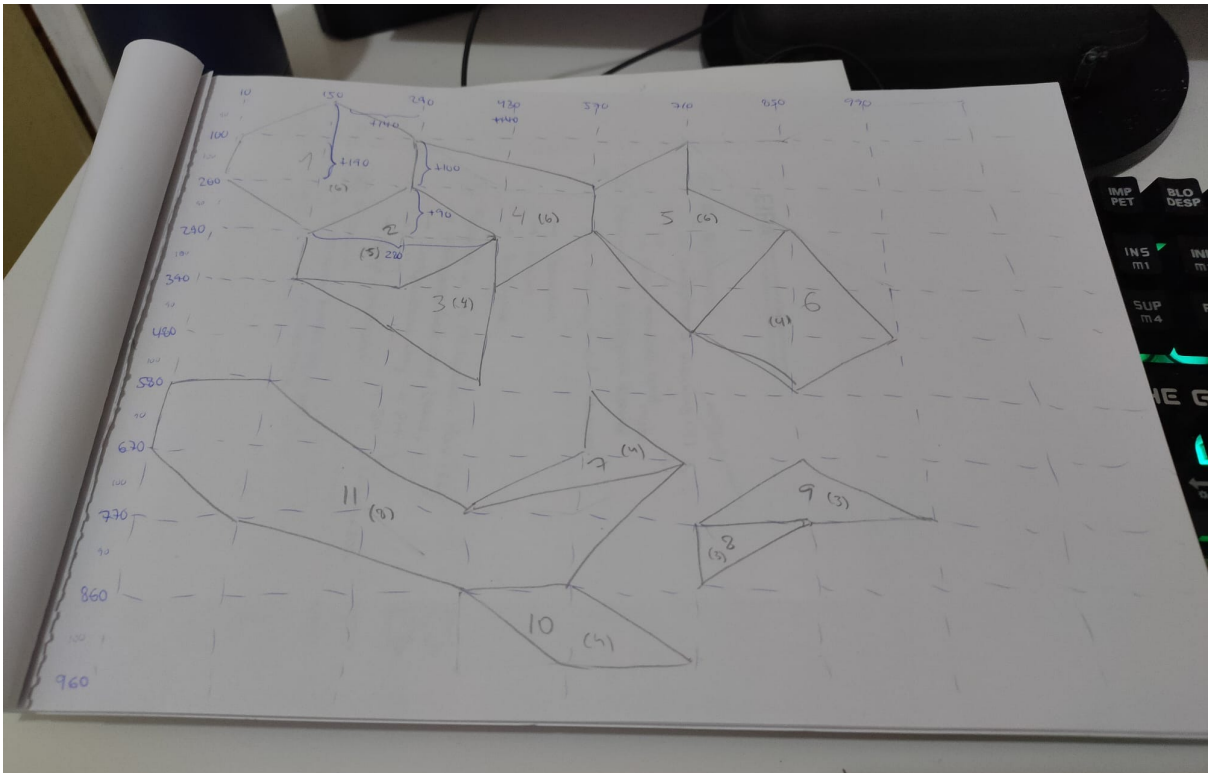
1.5. MyMap & MyWindow

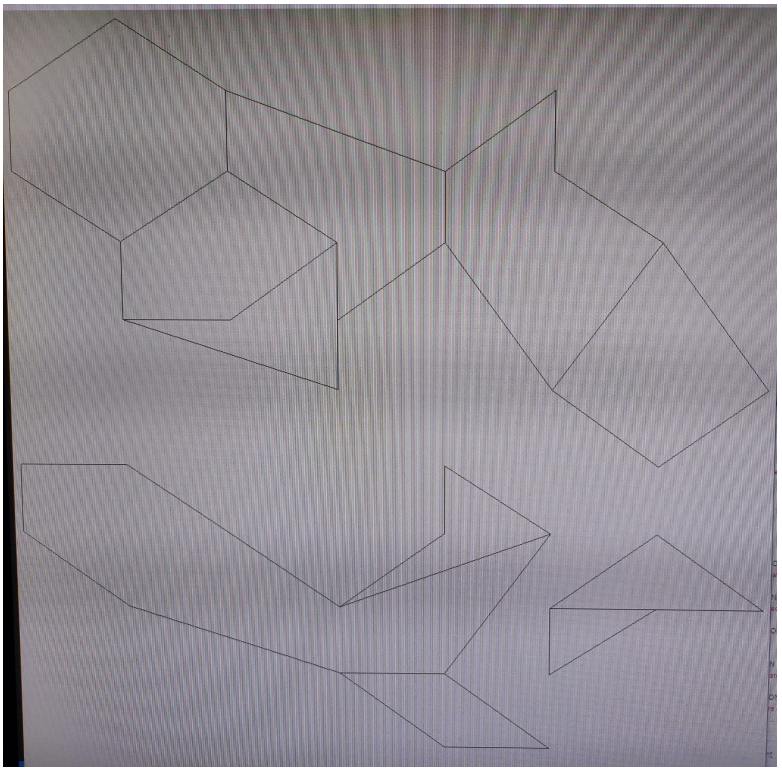
Finally, what is left is to create all the world and show it in the window. For that purpose, we needed to modify the *MyMap.java* class such that its attribute was a World and therefore that it could draw it.

To create the world, we had to start with the Point class and finish with the World one. First, using *Point()* we created lists of points in the interval [0, 1000] that would represent our regions and using *PolygonalRegion()* we could generate the regions with these lists. Grouping some regions in lists, we created the continents with *Continent()* and finally a list of all the continents so that using World() our created world was generated. In our case we have 11 regions named from *reg1* to *reg11*, 3 continents named from *cont1* to *cont3* and the world named *myWorld*.

At first, trying to create our world, we had problems with the graphical outcome since the points we used were random numbers we thought, so the world was not a decent world but a messy drawing. Therefore, after thinking about it, we came up with the idea of designing the map manually beforehand. That is, in a paper we drew squares with the corresponding coordinates and formed geometric shapes that simulated an imaginary world. After designing the whole world, we only needed to create the points with the coordinates we had in the paper, so it was way easier and organized.

Ayesha Quinto Gammuac 241789
Minh Nguyen Phuong 241841





2. CONCLUSION

First of all, we are really appreciative that you are still reading this long and detailed report.

For us, this lab was easier than the previous one since we are more familiar with the coding functions and we also knew from the seminar about which attributes and methods we need to implement in each class. Furthermore, we also reused a lot of methods. As we mentioned before, the only struggle we had was to make the world look fine. After this lab, we learned one new thing is the method *draw* from Graphics in Java.

Thank you very much.