Finding Friends Project--Adithya Shastry

Summary:

The Finding Friends project was used to implement the Singly linked Node data structure and to implement a simulation about how groups group together. The solution for this project used inheritence between various classes such as the Agent,Grouper, and Categorized Grouper classes. These classes were implemented in the Landscape class where the classes were allowed to interact with one another. The classes interact with one another in different ways based on the number of neighbors that the groupers or categorized groupers have. The output of the simulation is finally outputted using a graphics class that provides a graphical user interface for the user to see the simulation.

Singly Linked List(Lab):

The singly linked list was fairly basic to implement because it was very similar to the code that we had written in class. The most interesting part was the implementation of the iterator in the class. The most interesing part of this was how easy it was to actually implement the iterator, and how useful it was throughout the project. The code can be seen below:

```
    }
    private class LLIterator implements Iterator<T>{
        private Node<T> ptr;
        public LLIterator(Node<T> head){
            ptr=head;
        }
        public boolean hasNext(){
//          return this.ptr.getNext()!=null;
            if (ptr == null){
                return false;
            }
            return true;

        }
        public T next(){
            T d= ptr.getThing();
            ptr=ptr.getNext();
            //this will return the next value
            return d;//There seems to be an error here for some reason
        }
    }
```

Agent:

The Agent class will serve as a basis for all other child classes. This class is an Abstract class, one that doesn't have to implement all the methods it specifies. This was done in this situation to make sure that the child classes had their own implementation of the method specific to the class, whereas all the general methods were implemented and did not have to be written in the subclasses.The code for this can be seen below:

```
public abstract class Agent {
    //This class will be used as a parent class for the different categories of groupers
    double x0;
    double y0;
    public Agent(double x0, double y0){
        this.x0=x0;
        this.y0=y0;

        //This will set the coordinates of the agent
    }
    public double getX0() {
        return x0;
    }
    public void setX0(double x0) {
        this.x0 = x0;
    }
    public double getY0() {
        return y0;
    }
    public void setY0(double y0) {
        this.y0 = y0;
    }
    public String toString(){
        return("("+this.getX0()+","+ this.getY0()+")");
    }

    //the below methods will be implemented differently in the child classes
    abstract public void updateState(Landscape scape);
    abstract public void draw(Graphics g);
    abstract public int getCategory();

}
```

Grouper:

The grouper class, as mentioned above, inherits from the Agent class, but is different in the way that it implements the updateState() class. The code can be seen below:

```
public void updateState(Landscape scape){
    int num=scape.getNeighbors(getX0(), getY0(), this.friendZone).size();//This will return the number of neighbors the cell has
    //This will create a random object that can be used to generate numbers used
    Random gen=new Random();
    //this will now be used for the logic of the program
    if(num>3){
        //If the agent has more than 3 neighbors
        if(gen.nextDouble()<0.01){
            //If there is a probability of less than 1 percent then
            this.move(this.getMoveRange(),scape);
            return;
        }
        else{
            return;//This will exit the method
        }
    }
    this.move(this.getMoveRange(),scape);//This will move
    return;
```

Landscape:

The landscape class was relatively straightforward except for the implementation of the getNeighbor() method that used a fair amount of math, namely the distance formula. The idea here was to add neighbors to an arraylist which were within a specified radius. The code can be seen below:

```java
import java.awt.Graphics;
import java.util.ArrayList;

public class Landscape {
    private int w;
    private int h;
    private LinkedList<Agent> agents;
    public Landscape(int w,int h){
        this.w=w;
        this.h=h;
        this.agents=new LinkedList<Agent>();
    }
    public int getWidth() {
        return w;
    }
    public int getHeight() {
        return h;
    }
    public void addAgent(Agent a){
        agents.addFirst(a);//This is meant to add the agent to the beginning of the list
    }
    public String toString(){
        return "There are "+ this.agents.getCounter()+"Agents in the landscape";
    }
    //This method will get the neighbors of the specified agent and will add it to an array list
    public ArrayList<Agent> getNeighbors(double x0,double y0,double radius){
        ArrayList<Agent> neighbors=new ArrayList<Agent>();
        //Now I will iterate through all the agents in the linked list and see where they are situated
        for(Agent a:this.agents.toArrayList()){//This will run through the agents in the landscape
            //The distance between this cell and the original cell will be calculated now
            //The square root will not be used in order to make the program more efficient instead the output will be compared to the radius squared
            //the formula being used is the distance formula
            double distance=(Math.pow((a.getX0()-x0), 2))+(Math.pow(a.getY0()-y0, 2));
            if(distance<=Math.pow(radius, 2)){
                neighbors.add(a);//This will add the agent if it is within the radius
            }
            //There is no else statement because we only want to add the number of agents within the radius of the current agent
            //This will return the ArrayList


        }
        return neighbors;
    }
    /* draw all the agents */
    public void draw (Graphics g) {
        for (Agent a: this.agents) {
            a.draw(g);
        }
    }

    public void updateAgents(){
        //this method will update the agents in landscape in a random order
        for(Agent a:this.agents.toShuffledList()){
            //This will iterate through the entire shuffled list
            a.updateState(this);
        }
    }
}
```

Visualization:

The LandscapeDisplay class was pretty easy to implement because it just involved changing a couple of things from the original.

updateState():

The updateState() method was pretty difficult to implement because it meant finding a random value for the radius, x, and computing the value for y that would satisfy the radius requirement. The radius requirement stipulates that the agent only move within a radius of 5 units. The overall logic for the method is to find out if the situation of the agent meets its required standards for movement, or non movement, and acting accordingly. The code for this can be seen below:

```
public void updateState(Landscape scape){
    int num=scape.getNeighbors(getX0(), getY0(), this.friendZone).size();//This will return the number of neighbors the cell has
    //This will create a random object that can be used to generate numbers used
    Random gen=new Random();
    //this will now be used for the logic of the program
    if(num>3){
        //If the agent has more than 3 neighbors
        if(gen.nextDouble()<0.01){
            //If there is a probability of less than 1 percent then
            this.move(this.getMoveRange(),scape);
            return;
        }
        else{
            return;//This will exit the method
        }
    }
    this.move(this.getMoveRange(),scape);//This will move
    return;

}
public void move(double r,Landscape scape){
    Random gen=new Random();
    //Here I will write the code that will make sure the agent is moving the correct distance
    double x1=(gen.nextDouble()*10)-5;//the idea here is to shift the minimum value down to -5 and generate a double between 0 and 1 and multiply it by 10.
    double y1=(gen.nextDouble()*10)-5;
    //Now the value of x1 and y1 will be added to x0 and y0

    this.setX0(this.getX0()+x1);
    this.setY0(this.getY0()+y1);
    if(Math.abs(this.x0)>scape.getWidth()||Math.abs(this.y0)>scape.getHeight()){
        //this will check to see if the new value for x and y is greater than the width and height of the
        //If this is the case, the move method must be run again
        this.move(r, scape);
    }

}
```

updateAgents():

This method was pretty easy to implement because it involved using the shuffledArray() method from the LinkedList class and iterating through all the agents and calling update state on them. The code can be seen below:

```
public void updateAgents(){
    //this method will update the agents in landscape in a random order
    for(Agent a:this.agents.toShuffledList()){
        //This will iterate through the entire shuffled list
        a.updateState(this);
    }
}
```

GrouperSimulation:

This was also relatively easy to set up because it meant just implementing the update methods from the landscape class.This code can be seen below:

```
public class GrouperSimulation {
    public void run(int height,int width,int time,int scale) throws InterruptedException{
        Landscape l=new Landscape(width,height);
        LandscapeDisplay Land=new LandscapeDisplay(l,scale);
        //Create a Random object
        Random gen=new Random();
        gen.setSeed(System.currentTimeMillis());
        //this will populate the landscape
        int i=0;
        while(i<=2000){
            Grouper a =new Grouper(gen.nextDouble()*height,gen.nextDouble()*width);
            l.addAgent(a);

            i++;
        }


        //Run the actual simulation
        int x=0;
        while(x<=time){

            l.updateAgents();
            Land.repaint();
            Thread.sleep(250);


            x++;
        }
    }

public static void main(String[] args) throws InterruptedException{
    GrouperSimulation gs=new GrouperSimulation();
    gs.run(200, 200, 100, 5);

}
}
```

CategorizedGrouper:

The categorized grouper was relatively easy to implement because it inherited from the grouper class. The update state method had to be changed however, because the categorized grouper had more stringent restrictions. The categorized grouper only stayed in place if it had more friends of its category than any other category. This was rather difficult to implement because it involved finding the mode of the categories of the neighbors that surrounded the the agent. The code is shown below and the overall logic is also explained in the code.

```java
@Override//This should override the original method in the java
public void updateState(Landscape scape){
    ArrayList<Agent> neighbors= scape.getNeighbors(this.getX0(),this.getY0(),this.moveRange);
    //here I will take neighbors and only take their categories
    ArrayList<Integer> categories=new ArrayList<Integer>();
    for(Agent x:neighbors){
        categories.add(x.getCategory());
    }

    //here I will create an algorithm that will find the mode of the arraylist of categories
    //These values will be required to run the method.
    int maxCount=0;//this will hold the frequency of the most common category in the list
    int maxCountValue=0;//This will hold the actually value of the mode
    int count=0;//This will be an intermediary variable that will be used to compare to previous values.
    //Now I will start the actual algorithm

    for(int a:categories){
        for(int b:categories){
            if(a==b){
                count++;//this will add one to the count if the values are equal

            }

        }
        if(maxCount<count){
            //If the maxCount is less than count then make the max count equal to the count
            maxCount=count;
            //We also want to return the value being checked
            maxCountValue=a;
        }

    }
    /*
     * The general idea here is to take all the values of the arraylist and compare them to all the other values of the arraylist.
     * Then based on the frequency of the category values, a new max count will be created. When this variable is changed the value
     * of the mode must also change because there is a new mode.This algorithm should effectively find the mode of the list.
     */

    //Create a random object
    Random gen=new Random();

    //Now I will actually mode the categorizedGrouper based on the rules
    if(maxCountValue==this.category){
        //If the category is the most common one, then it will move with a chance of 0.01
        if(gen.nextDouble()<=0.01){
            this.move(this.moveRange,scape);
        }

    }

    else{
        this.move(this.moveRange,scape);
    }

}
```

The very interesting thing about this implementation, as explained in the extension section is that this algorithm can take into account as many categories as the user wants to simulate.

CategorizedGrouperSimulation:

This was very similar to the GrouperSimulation and therefore will not be discussed in an effort to show more important parts!The code can be seen below:

```
public class GrouperSimulation {
    public void run(int height,int width,int time,int scale) throws InterruptedException{
        Landscape l=new Landscape(width,height);
        LandscapeDisplay Land=new LandscapeDisplay(l,scale);
        //Create a Random object
        Random gen=new Random();
        gen.setSeed(System.currentTimeMillis());
        //this will populate the landscape
        int i=0;
        while(i<=2000){
            Grouper a =new Grouper(gen.nextDouble()*height,gen.nextDouble()*width);
            l.addAgent(a);

            i++;
        }


        //Run the actual simulation
        int x=0;
        while(x<=time){

            l.updateAgents();
            Land.repaint();
            Thread.sleep(250);


            x++;
        }
    }

public static void main(String[] args) throws InterruptedException{
    GrouperSimulation gs=new GrouperSimulation();
    gs.run(200, 200, 100, 5);

}
}
```

Extensions:

Implementing multiple Categories:

I thought it would be interesting to be able to include more categories into the mix.The only way to do this is to create a better way of choosing color for the categories and also to create a better way of sorting the categories in the updateState() method.

The code foe the color implementation can be seen below:

```
public void draw(Graphics g){
        Random gen=new Random();
        Color c=new Color(5*((this.category)),5*+((this.category)),5*((this.category)));
        // if(this.category==1){

            g.setColor(Color.RED);
        }
        else if(this.category==2){
            g.setColor(Color.BLUE);
        }
        else{
            g.setColor(Color.ORANGE);
        }
        g.setColor(c);
        g.fillOval((int)this.getX0(),(int) this.getY0(), 5, 5);//This will draw the object with the specified colour
        System.out.println(g.getColor());
}
```

The code for the frequency finder can be found above in the Categorized grouper class section.

Implementing Command Line Arguments:

This will use the scanner class to ask the user to input certain values that will run the game. the code can be seen below:

```
public static void main(String[] args) throws InterruptedException{
    Controller c=new Controller();
    Scanner reader=new Scanner(System.in);
    int height;
    int width;
    int time;
    int numAgents;
    System.out.println("Please input the Height:\n");
    height=reader.nextInt();
    System.out.println("Please input the Width:\n");
    width=reader.nextInt();
    System.out.println("Please input the Time:\n");
    time=reader.nextInt();
    System.out.println("Please input the Number of Agents:\n");
    numAgents=reader.nextInt();

    c.gameType(height, width, time, numAgents);
}
```

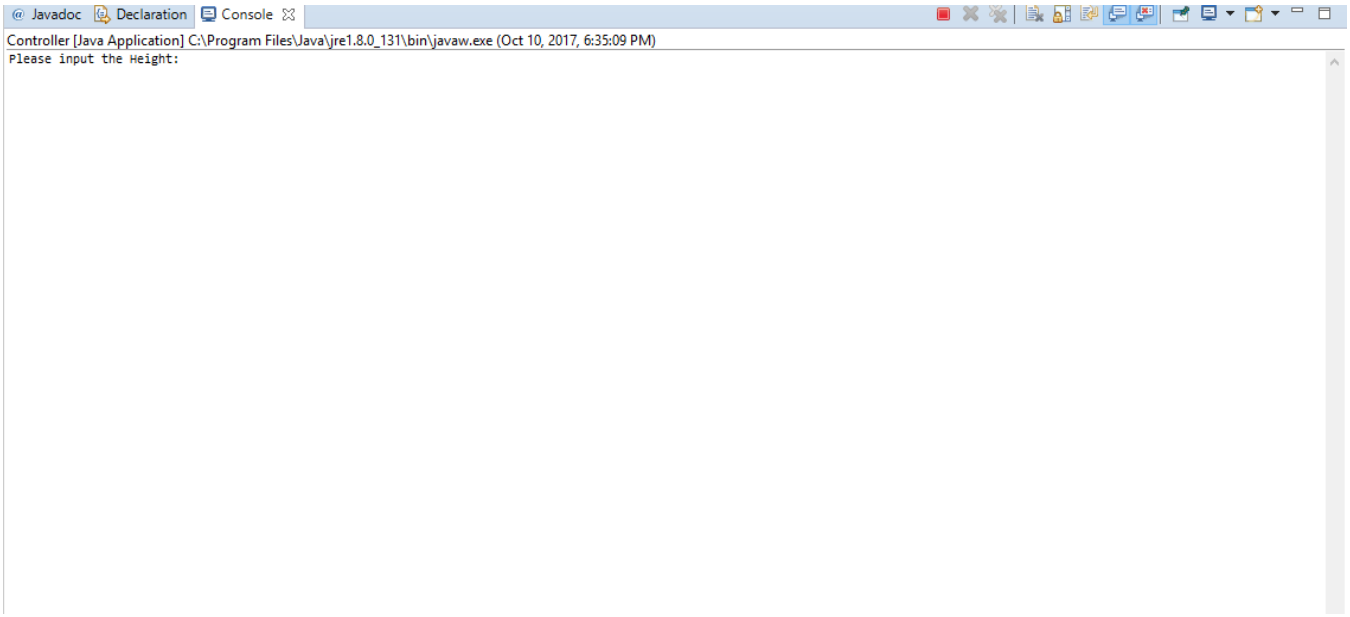
Implementing a command line argument that will run a specific version of the game:

This was mainly done using the game type method which can be seen below. The method also takes into account a user error if the user enters a value that is not directly specified int he instructions.Most people would follow the instructions, but my dad would always find ways to break my programs to help me learn, so this is me trying to solve user errors.

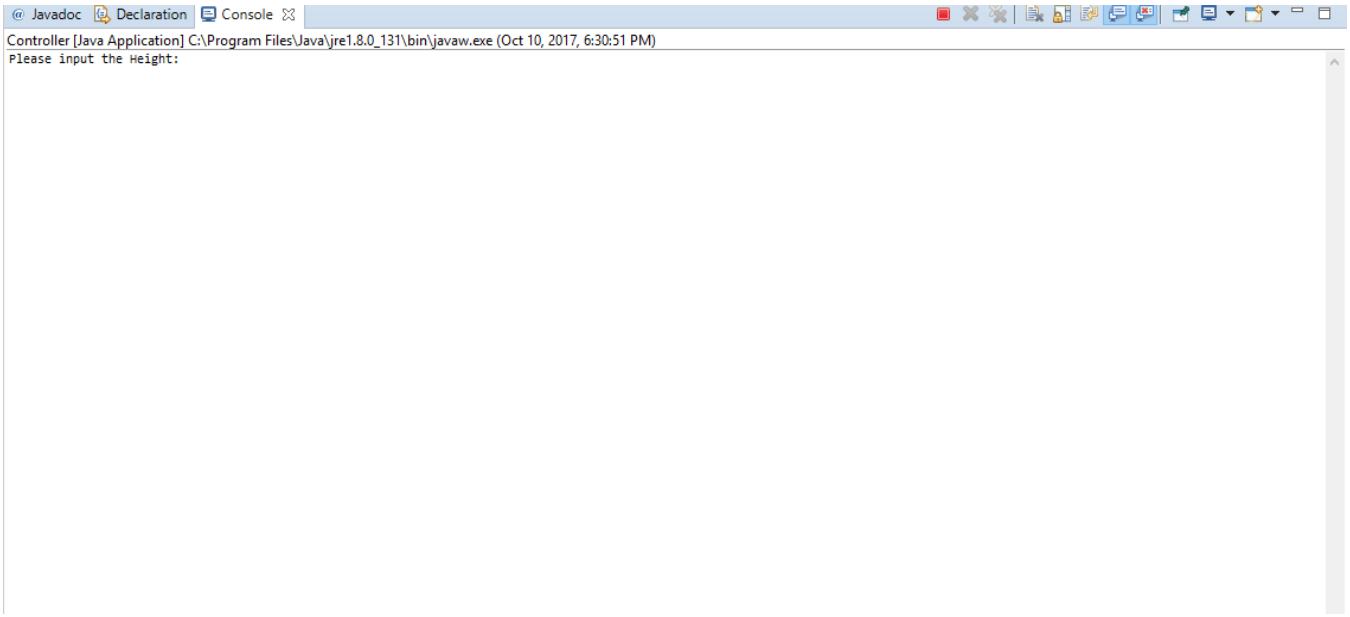
```
public void gameType(int h,int w,int t, int num) throws InterruptedException{
    System.out.println("Please indicate what game you would like to simulate:(1)Catogrized grouper or (2) Grouper");
    Scanner reader=new Scanner(System.in);
    int game=reader.nextInt();
    if(game==1){
        CategorizedGrouperSimulation gs=new CategorizedGrouperSimulation();
        gs.run(h, w, t, 5,num);
    }
    else if(game==2){
        GrouperSimulation gs=new GrouperSimulation();
        gs.run(h, w, t, 5,num);
    }
    else{
        System.out.println("That wasn't a valid imput!");
        this.gameType(h, w, t, num);
    }
}
```

Final Output:

CategorizedGrouper:



Grouper:



Conclusion:

This project really helped me understand the inns and outs of using a node based data structure and dealing with inheritance! I also got a lot of practice debugging code, which is always a good thing in my opinion!