Laia Tomàs
Daniela Quimis

Lab exercise 1: Implementing a class

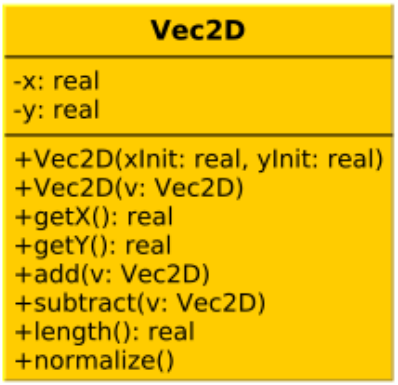
1 Introduction

The aim of this lab session is to implement the design of Seminar 1 consisting of the Agent and World classes.

The session is mandatory and you have to deliver the source code of the Java project and a document describing the implementation.

2- The Vec2D class

In the supplied code you will find a class Vec2D that represents 2-dimensional vectors, with the following design:



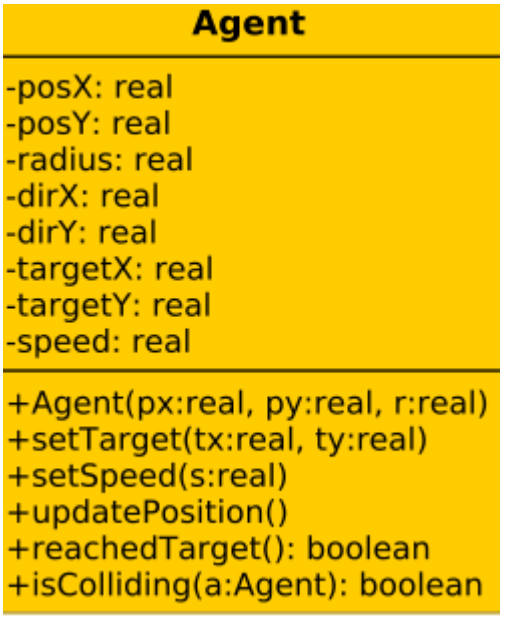
Study the source code and the comments and make sure that you understand the functionality. How could you use the Vec2D class to simplify the design of the Agent class? (Hint: several of the attributes of Agent are in fact positions or directions that can be represented using 2-dimensional vectors.)

Laia Tomàs
Daniela Quimis

3.1 The Agent Class

For defining the Agent class, we begin with their main attributes,

```java
public class Agent {
    // attributes
    private double radius;
    private Vec2D vector_target;
    private double speed;
    private Vec2D vector_agent;
    private Vec2D vector_direction;
```



We reduce (posX,posY) → vector_agent, (dirX,dirY) → vector_direction and (targetX,targetY) → vector_target. Moreover, we set the Agent's constructor, where it sets their radius and their initial position.

```java
//constructors
public Agent(double r, double posx, double posy){
    radius = r;
    vector_agent = new Vec2D(posx, posy);
}
```

Additionally, we define the Agent's methods, specifically the setters:

setTarget() → first initializing *vector_target* with their corresponding arguments as doubles, to acknowledge where our target is. This ''direction'' will be copied by *vector_direction* , where with our current position (vector_agent) we'll do a subtraction to obtain the corresponding vector and finally normalize it .

```java
//setters
public void setTarget(double tx, double ty){
    //1
    vector_target = new Vec2D(tx,ty);
    //2
    vector_direction = vector_target; // copy of target position
    vector_direction.subtract(vector_agent);
    vector_direction.normalize();
```

Laia Tomàs
Daniela Quimis

setSpeed() → Agent's speed is defined as a straightforward setter method. Just simply as…

```
public void setSpeed(double s){
    speed = s;
}
```

Another important methods:

UpdatePosition() → this method keeps us on track of the position of our Agent, this will be computed by adding the current direction multiplied by the main speed, in terms of vector notation. Hence, ends up being saved in a new vector, and then added to our *vector_agent*.

```
public void updatePosition(){

    double vx = speed*(vector_direction.getX()); // /vector_direction.length()
    double vy = speed*(vector_direction.getY());
    Vec2D vector = new Vec2D(vx, vy);

    vector_agent.add(vector);
```

TargetReached() → communicates us if our agent has reached their current target . In order to achieve this we define a *vectornew* which arguments are *(newX,newY)* that will save the difference between our target's position and agent's current position. If our Agent has accomplished their task, the length of *vectornew* must be less than our Agent's radius.

```
public boolean targetReached(){
    double newx = vector_target.getX() - vector_agent.getX();
    double newy = vector_target.getY() - vector_agent.getY();
    Vec2D vectornew = new Vec2D(newx, newy);
    //System.out.println((vectornew.length()));
    if (vectornew.length() < radius){
        return true;
    }
    return false;

}
```

Laia Tomàs
Daniela Quimis

isColliding() → To check if our Agent is colliding with another, we'll calculate the difference between their respective positions and then save these values into *vector_new.* If the length of this new vector is less than the sum of our Agents radius, they are colliding otherwise will return false.

```
public boolean isColliding(Agent b){
    double newx = vector_agent.getX() - b.vector_agent.getX();
    double newy = vector_agent.getY() - b.vector_agent.getY();
    Vec2D vectornew = new Vec2D(newx, newy);
    double agentsum = radius + b.radius;
    if (vectornew.length() < agentsum){
        System.out.println(x: "colliding");
        return true;
    }
    return false;
}
```

3.2 The World Class

We'll start defining all the World attributes:

```
public class World {
    //attributes
    private int width;
    private int height;
    private Agent[] agents;
    private int numAgents;
    private double margin = 30;
```

We keep in mind that a World has a specific capacity for Agents, in this world the capacity is 10 Agents. Therefore we define an array for agents (as an attribute) and also a limit that Agents cannot cros in this World, we add *margin()* with default value of 30.

Laia Tomàs
Daniela Quimis

On top of that, we'll create a random radius , random target position and set a speed of 1 pixel/s to each Agent (until the 10th), with auxiliary methods like randomPos() and randomRadius() that will pick a random value.

```
//constructor
public World(int w, int h, int cap){
    width = w;
    height = h;
    cap = numAgents;
    agents = new Agent[10];
    for(int i = 0; i<10; i++){
        double radius = randomRadius();
        Vec2D vectorAgent = randomPos();
        double posx = vectorAgent.getX();
        double posy = vectorAgent.getY();
        agents[i] = new Agent(radius, posx, posy);
        agents[i].setSpeed(s: 1);
        Vec2D vectorTarget = randomPos();
        agents[i].setTarget(vectorTarget.getX(), vectorTarget.getY());
    }
}
```

For the purpose of not generating positions outside or too close to the edge (margin), we'll say to our randomPos()...

```
private Vec2D randomPos(){
    double x = margin + Math.random() * ( width - 2 * margin ) ;
    double y = margin + Math.random() * ( height - 2 * margin ) ;
    return new Vec2D ( x , y ) ;
}
```

Same to our randomRadius()...

```
private double randomRadius(){
    return 5 + Math.random() * ( margin - 5 ) ;
}
```

Laia Tomàs
Daniela Quimis

simulationStep() → This function iterates over all agents and if an agent has reached to its target, a new random target is generated and set. If not, its position is updated. The function manage collisions is commented because it doesn't work perfectly with all agents, but it attempts to do so.

```
public void simulationStep(){
    for(int i = 0; i<10; i++){
        if (agents[i].targetReached()){
            Vec2D newtarget = randomPos();
            agents[i].setTarget(newtarget.getX(), newtarget.getY());
        }else{
            //manageCollisions();
            agents[i].updatePosition();
        }
    }
}
```

manageCollisions() → This function iterates over all pair of agents and compares if both agents are colliding. If so, it sets another random target and updates both positions so that they change directions. It doesn't work perfectly, but if you want to see it, you need to uncomment the manageCollisions() line in the simulationStep function above.

```
//aux methods
public void manageCollisions(){
    for(int i = 0; i<10; ++i){
        for(int j = i+1; j<10; ++j){
            if(agents[i].isColliding(agents[j])){
                //if both are colliding, change direction
                Vec2D vectorTarget = randomPos();
                agents[i].setTarget(vectorTarget.getX(), vectorTarget.getY());
                Vec2D vectorTarget2 = randomPos();
                agents[j].setTarget(vectorTarget2.getX(), vectorTarget2.getY());
            }
        }
    }
}
```

paintWorld() → It iterates over all agents and paints each of them using paintAgent function.

```
public void paintWorld(Graphics g) {
    for (int i = 0; i < 10; i++){
        agents[i].paintAgent(g);
    }
}
```

Laia Tomàs
Daniela Quimis

3.3 Other solutions

We thought of other possible implementations in some functions. For example, in setTarget function we could have also used this code, but it was cleaner the way we finally implemented.

```
//double newposx = tx - vector_agent.getX();
//double newposy = ty - vector_agent.getY();
//vector_direction = new Vec2D(newposx,newposy);
//vector_direction.normalize();
```

The same thing occurs with function updatePosition(). We could have computed first the components taking into account the vector agent position instead of adding the vector later.

```
//double vx = vector_agent.getX() + speed*(vector_direction.getX());
//double vy = vector_agent.getY() + speed*(vector_direction.getY());
//vector_agent = new Vec2D(vx, vy);
```

3.4 Conclusions

To sum up, we've view how des it really implement a class and how they interact with other ones, in order to built a system that works and achieve our goal. Essentially, we had some problems on how to traduce the solution into real code, but as we kept programming, finding other ways and commenting as a teamwork, we've learn how does a class work and java's language.

As a result, our world with their agents co-exist and works fine, we may had some difficulties with the aux method "ManageCollisions()", as you can see some Agents had trouble dealing with collisions, but without this method our World is computed perfectly.