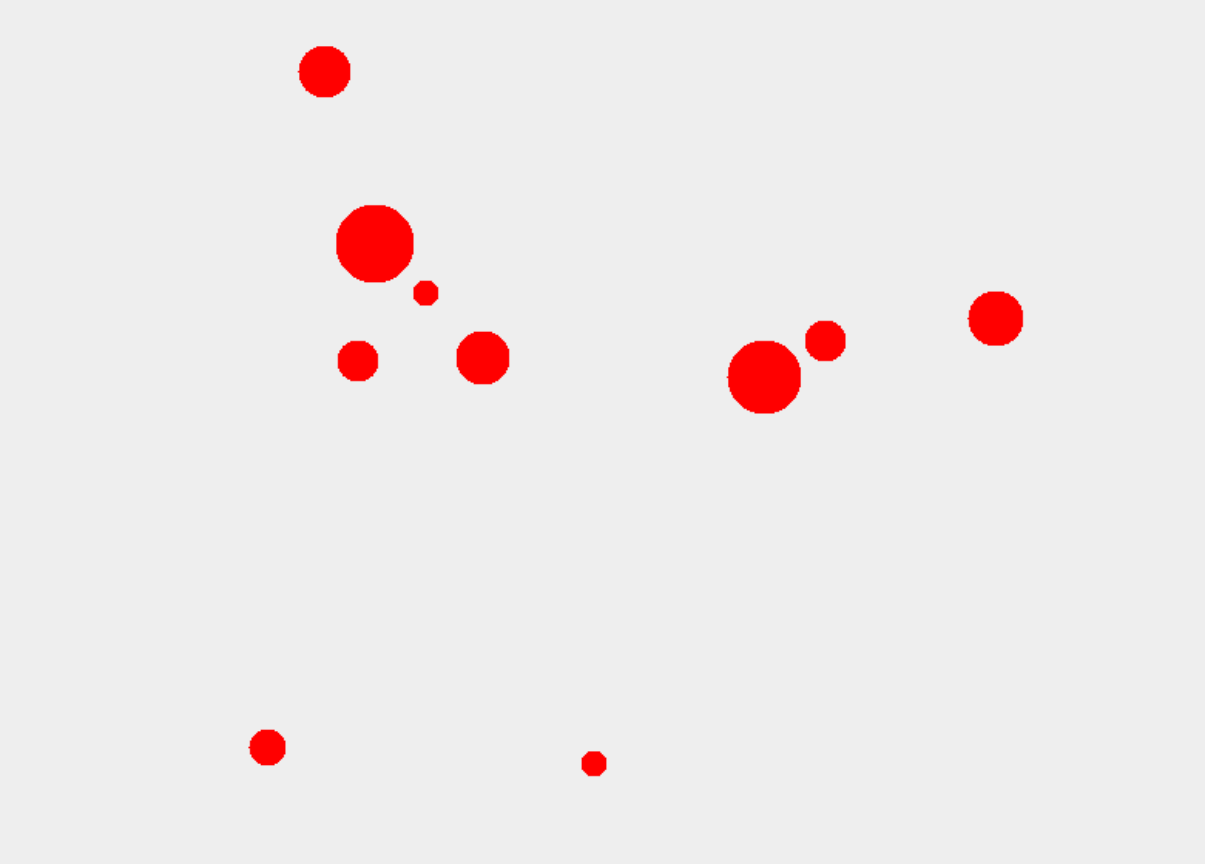
PRACTICA 1 OOP

1. INTRODUCCIÓN

La práctica 1 de Programación Orientada a Objetos consiste en la creación y desarrollo de varias clases (Agent, World, WorldGUI, Vec2D) que cuando se relacionan e implementan de forma efeciente y adecuada, éstas crean un mundo gráfico de forma rectangular dónde formas esféricas de color rojo, también llamados ajentes en el código del programa, ejecutan movimientos hacia un objetivo definido aleatoriamente, a la vez que evitan las colisiones con otras esferas en el mundo.



Así pues, el programa lo conforman cinco clases distintas: TestWorld, WorldGUI, Vec2D, estas tres primeras casi ya estaban completamente desarrolladas, junto a Agent y World, que sí requerían de implementación.

Consecuentemente, nos basamos en los gráficos para desenvolver los atributos y métodos de estas dos clases. A continuación podemos observar los atributos definidos en la clase Agent y la clase World:

```java
public class Agent {

    //Atributes
    private Vec2D position;
    private double radious;
    private Vec2D direction;
    private Vec2D target;
    private float speed;
```

```java
public class World {

    //Atributes
    private int width;
    private int height;
    private Agent[] agents;
    private int numAgents;
    private int margin;
```

A lo que los métodos se refiere, si indagamos en el programa veremos cómo a partir de comentarios de texto y anotaciones se definen y complementan las técnicas e implementación de nuestro código. A continuación algunos de los métodos más esenciales del programa:

```java
    //Constructor
    public World(int w, int h){
        width = w;
        height = h;
        margin = 30;
        numAgents = 10;
        agents = new Agent[numAgents];

        //Agent initialisation using 'for'
        for(int i = 0; i < numAgents; i++){
            agents[i] = new Agent(randomPos(),randomRadius());
            agents[i].setTarget(randomPos());
            agents[i].setSpeed(s: 1);
        }
    }
```

```java
    //Constructors
    public Agent(Vec2D initPos, double initRad) {
        position = initPos;
        radious = initRad;
    }
```

2. DESCRIPCIÓN DE POSIBLES ALTERNATIVAS Y RAZONAMIENTO DE NUESTRA DECISIÓN

La principal alternativa que nos generó dudas a la hora de diseñar nuestro programa fue cuando creamos los agentes en nuestra clase *World*. Relacionado con la teoría que estamos haciendo en clase, sería esta interacción entre las dos clases *Agent* y *World*, que como sabemos, es una relación de agregación ya que el *World* puede tener varios *Agents*. A la hora de programar, nos dimos cuenta que cuando creamos los agentes mediante un for y luego volvíamos a hacer otro for para establecer un objetivo, una posición y un radio para cada agente estos dos for 's se solapan y hacía que los agentes tenían valores *null*. Por lo que creamos en un mismo for todos los agentes con sus atributos en la clase *World*.

```
for(int i = 0; i < numAgents; i++){
    agents[i] = new Agent(randomPos(),randomRadius());
    agents[i].setTarget(randomPos());
    agents[i].setSpeed(s: 1);
}
```

Otra de las alternativas que propusimos fue la de trabajar siempre con vectores en todos los métodos que hemos creado en la clase agente. Esto nos proporciona un código más entendible y menos extenso a la hora de programar. Creemos que la alternativa de programar sin vectores era una opción más sencilla de programar pero hacía que el código fuera más extenso y costoso. Aquí vemos un ejemplo de dos métodos de la clase agent trabajando con vectores. Vemos lo telegráfico que nos queda.

```
public void updatePosition(){
    Vec2D product = new Vec2D(direction.getX() * speed, direction.getY() * speed);
    position.add(product);
}

public boolean targetReached(){
    Vec2D vec = new Vec2D(target.getX()-position.getX(),target.getY()-position.getY());

    if(vec.length() < radious){
        return true;
    }
    return false;
}
```

Otra alternativa más que hemos probado es en el método *simulationStep* cuando hacemos el apartado extra de la práctica, teniendo en cuenta las colisiones entre agentes. En este método debemos comprobar si el agente ha llegado al objetivo que tiene marcado y seguidamente si está colisionado con otro agente. Primero habíamos hecho el código teniendo en cuenta primero si colisionaban y luego si llegaban al objetivo, pero vimos que a la hora de ejecutar el programa era mejor si tenemos en cuenta primero el objetivo y luego las colisiones. Simplemente cambiamos el orden.

```
public void simulationStep(){

    for(int i = 0; i < numAgents; i++){


        if(agents[i].targetReached() == true){
            agents[i].setTarget(randomPos());
        } else {
            for(int j = 0; j < numAgents; j++){
                if(j != i){
                    if(agents[i].isColliding(agents[j])){
                        agents[i].setTarget(randomPos());
                    }
                }
            }
            agents[i].updatePosition();
        }
    }
}
```

3. CONCLUSIÓN

Una vez terminado el programa estamos satisfechos con el resultado obtenido. Sin embargo, hemos encontrado algún fallo en el programa. Por ejemplo, cuando dos agentes colisionan y se les asigna otro objetivo, es posible que este objetivo esté en una dirección similar a la anterior, por lo que los agentes volverían a colisionar. El fallo que encontramos es que al ser los agentes de forma circular hace que cuando colisionan sea muy probable que los mismos dos agentes vuelvan a colisionar y entren en un bucle de colisiones entre ellos. Creemos que si les asignamos otra forma que no sea circular a los agentes haría que no se produjeran tantas colisiones. Otra solución que hemos visto posible sería la de que cuando dos agentes colisionan, enviar cada agente a una posición opuesta a la de otro agente. No obstante, esto haría que el objetivo no fuera tan aleatorio y por eso hemos pensado que ésta no acabaría siendo una solución viable.