GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING

CSE 443
OBJECT ORIENTED ANALYSIS AND DESIGN
FINAL PROJECT REPORT

GÖKHAN HAS
161044067

INSTRUCTOR : PROF.DR. ERCHAN APTOULA

Visual Simulation Of An Epidemic Within A Human Society

Important Notes:

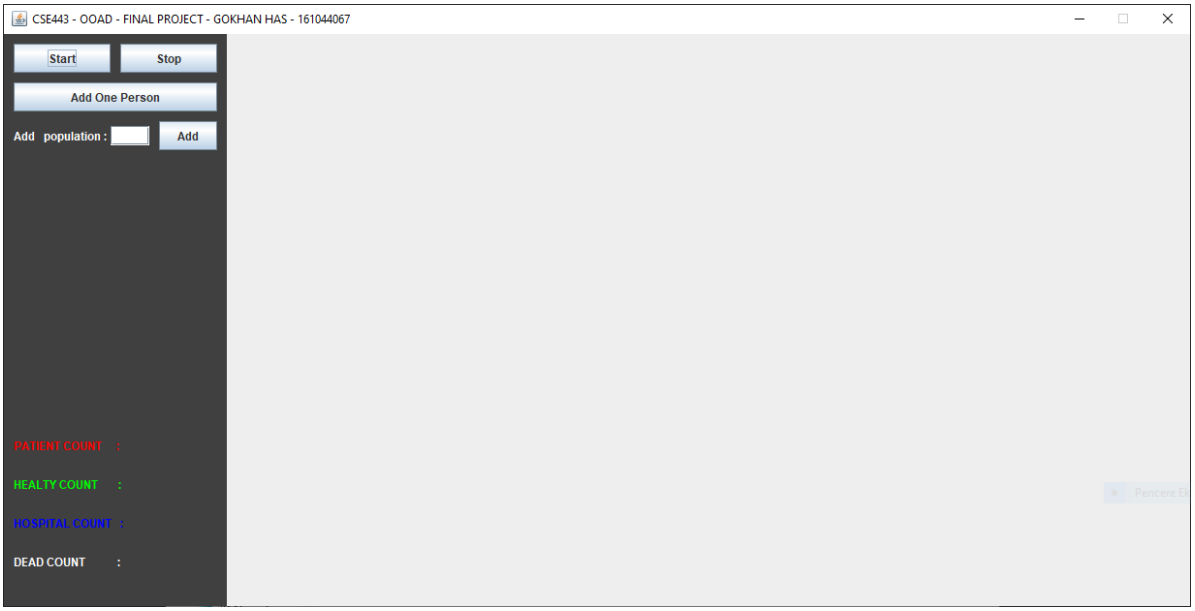
When the program is run, the first start button must be pressed. When the Start button is pressed, a society is created with 100 individuals. Then other components can be used to add individuals one by one or collectively. The reason I do this is to ensure that there is at least one ventilator in the hospital.

In the panel, individuals move only up, down, right and left. I did this as there is no information about it in the pdf.

In the panel, healthy individuals are shown in black and injected individuals in red.

When individuals go out of the panel after their speed reaches 500, a second may seem as if they are not moving.

To view the UML diagram and screenshots in detail, you can view the screenshots and diagram folder.



As seen in the application, there are 4 buttons and an area where statistics will be printed.

Model-View-Controller

The MVC pattern consists of 3 layers and its layers work independently (without affecting each other). For this reason, it is mostly preferred in large-scale projects to ensure the management and control of projects more easily. In projects developed with MVC, many people can easily work simultaneously according to the details of the project.

I started by coding the project in accordance with the MVC architecture. Because I could completely separate the GUI side from the background processes and reduce the maintenance costs.

View is the section where the interfaces of the project are created in MVC. This section contains the GUI files of the project that will be presented to the users.

Here is the view package. This class extends the JPanel compenent, implements the ViewObserverPopulationInterface, and holds a reference with both SimulationControllerInterface and SimulationModelInterface interfaces. Interfaces with JComponenet objects. This object also subscribes itself in the constructor with the registerObserver method in SimulationModel.

When the buttons are pressed, the actionPerformed function is overridden and the corresponding function of the corresponding button SimulationController is called.

One of the important functions here is the updatePopulation function. SimulationView is notified as a result of the calculations to be made in SimulationModel. This function is executed in the notification. And it updates the panel. Their places are arranged according to the knowledge of the individuals and different colors are given according to their situation. Information that needs to be displayed in the GUI is also updated here. Since some values are taken from the SimulationModel reference here, it is used synchronized in order to avoid confusion in the function.

The work that must be done by the application and the work that must be shown to the user is separated. It has already been invented to avoid maintenance costs in visual applications. The algorithm of the design and the application should be different.

Composite design pattern is used in SimulationView. Subcomponents of JComponents are used in the tree structure. It can also be used when creating new components.

```
public void updatePopulation() {
    synchronized (simulationModel.getSemaphore()) {
        int patient = 0, healt = 0, dead = 0, hospital = 0;
        maskUsageRatio = 0.0;
        meanSocialDistance = 0.0;
        Graphics g = populationPanel.getGraphics(); // YENIDEN HARITA GUNCELLENECEK ...
        g.clearRect( x: 0, y: 0 ,populationPanel.getWidth(), populationPanel.getHeight()); // TEMIZLEME
        ArrayList populationList = simulationModel.getPopulationList();
        int hospitalUsedSize = (simulationModel.getHospital()).getSize();
        for(int i = 0; i < populationList.size(); i++) {
            Individual temp = (Individual) populationList.get(i);
            if(temp.getIndividualState() == IndividualState.DEAD) {
                dead++;
            }
            else {
                if(temp.isIfInfected()) {
                    g.setColor(Color.RED);
                    patient++;
                    g.fillRect(temp.getxCoord(), temp.getyCoord(), width: 5, height: 5);
                }
                else if(temp.getIndividualState() == IndividualState.INHOSPITAL) {
                    hospital++;
                }
                else {
                    g.setColor(Color.BLACK);
                    g.fillRect(temp.getxCoord(), temp.getyCoord(), width: 5, height: 5);
                    healt++;
                }
            }
            meanSocialDistance += temp.getSocialDistance();
            if(temp.getMaskValue() == 1.0) {
                maskUsageRatio++;
            }
        }
        patientCount.setText(String.valueOf(patient));
        healthyCount.setText(String.valueOf(healt - hospitalUsedSize));
        hospitalCount.setText(String.valueOf(hospitalUsedSize));
        deadCount.setText(String.valueOf(dead));

        maskUsageRatio = 100.0 - (100.0 * maskUsageRatio / (double) populationList.size());
        meanSocialDistance = meanSocialDistance / (double) populationList.size();
    }
```

The controller package contains the class that implements the Controller interface with this interface. Controller is the part that controls the internal processes of the project in MVC. In this section, the connection between View and Model is established. Requests from users are evaluated in Controllers, and the Model is notified which actions will be taken according to the details of the request. Strategy design pattern is used here. It is ensured that a behavior happens dynamically with different algorithms. The realization of the functions of the start / stop buttons is thanks to the controller. The behaviors that the application can do are written in separate functions. Examples include start / stop simulation and add user.

SimulationController has references with both Model and View interfaces. The method that creates the GUI, the View, is the Controller's contructor method. In other functions, the method to perform the necessary operation is called from the model reference.

```java
public interface SimulationControllerInterface {
    /**
     * It is the function that starts the simulation or resumes it after stopping it.
     */
    void startSimulation();

    /**
     * It is a function for stopping the simulation.
     */
    void pauseSimulation();

    /**
     * It is the function that will work on the GUI side when the user wants to add an individual to the population.
     * @param number
     */
    void increasePopulation(int number);
}
```

There are many classes in the model package. The processes that the application does in the background, many Threads are found in these files. Model is the section in which the business logic of the project is created in MVC. Along with the business logic, validation and data access operations are also performed in this section.

There is an implemented SimulationModel class from SimulationModelInterface. This class is the main Model function. In other words, it is the class that receives requests from the Controller, processes them and sends them to the View using the Observer design pattern.

Variables in this class are explained in Javadoc or source code. There are data fields such as individuals list, classes required to apply Mediator, semaphore, hospital reference.

The most important function here is the CalculateMove inner class. Because this class gets extends from TimerTask. And the run () method, which we know from Threads, is overridden. Here, the timer calls the run method in this class every second. Here, it means that new information is sent to View every second.

```java
/**
 * It is the class that will be executed by the timer every second. Gets extends from the TimerTask class.
 * And in this class, it is necessary to override the run () method in the Runnable interface.
 * So there is Thread in it.
 */
private class CalculateMove extends TimerTask {
    @Override
    public void run() {
        synchronized (getSemaphore()) {
            if(simulationState == SimulationState.RUN) {
                for (Individual individual : populationList) {
                    try {
                        if((individual.getIndividualState() != IndividualState.DEAD) && !(individual.getIndividualState() == IndividualState.INCOLLISION))
                            individual.changedCoord();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                notifyPopulationObservers();
            }
        }
    }
}
```

It allows those who use the observer design pattern to make loosely-coupled applications. Subject and Observer are loosely-coupled with each other. This design pattern has been used because it is desired that an object automatically affect more than one object.

Mediator Design Pattern

The process of whether individuals collide when their coordinates change or not is performed using this design pattern. The main philosophy, as used in the game we learn in the lesson, tells the Mediator that his place has changed, saying that my individual has changed. Mediator checks if there is an overlap where this changes. Calls related functions, if any. In the lesson, we said that the soldier fired at the following coordinates in the game and asked the Mediator if there was someone there, and if there were, Mediator called the methods that reduce the lives of the relevant people.

The pattern is usually used for the number of classes of the program written. Logic and compotation are distributed among these classes. In addition, in highly developed programs, communication between classes becomes more and more complex and legibility is affected by the progress of the code. When using the mediator pattern, communication is not provided directly between classes. There is a Mediator pattern between classes and manages the classes. Thus, the dependency between objects is reduced and the program becomes more convenient to be managed and developed. In short, the Mediator design pattern is used to reduce dependencies between classes and facilitate communication between them.

There is only one Mediator reference in the SimulationModel class.

```java
/**
 * The reference of the Mediator object to be given to each individual is kept.
 */
private MediatorIndividual mediatorIndividual;
```

Once individuals are created, this same reference is sent to all individuals.

```java
/**
 * It is a helper function that enables the creation of an individual object.
 * @param number
 */
private void addIndividualArray(int number) {
    synchronized (this.getSemaphore()) {
        for(int i = 0; i < number; i++) {
            Individual temp = new Individual(mediatorIndividual, hospital);
            if(isFirst) {
                temp.setIfInfected(true);
                isFirst = false;
            }
            populationList.add(temp);
        }
        hospital.setHospitalSize(populationNumber / 100);

        for(int i = 0; i < populationList.size(); i++) {
            populationList.get(i).ID = i;
        }
    }
}
```

The mediator class hires an arraylist within itself. And an individual object can be added to this list by using the add method.

It is the notify method that provides interaction between individuals. Compares the person calling this method with other individuals. Social distance between individuals is calculated. If the distance between individuals is smaller than the social distance, these individuals are considered to have collided. Then it is checked whether the individuals are injected or not. If present and the other individual is healthy, the healthy individual P is likely to be injected P is calculated in probability.

```java
/**
 * It is the function that is necessary to understand which individual is called
 * and whether there is a collision or not.
 * @param individual
 */
@Override
public void notify(Individual individual)  {
    for(int i=0; i < individuals.size(); i++) {
        Individual temp = (Individual) individuals.get(i);
        if(temp != individual && (temp.getIndividualState() != IndividualState.INCOLLISION && individual.getIndividualState() != IndividualState.INCOLLISION)
                && (temp.getIndividualState() != IndividualState.DEAD && individual.getIndividualState() != IndividualState.DEAD)) {

            double distance = Math.sqrt((Math.pow(individual.getxCoord() - temp.getxCoord(), 2) + Math.pow(individual.getyCoord() - temp.getyCoord(), 2)));
            int minD = min(individual.getSocialDistance(), temp.getSocialDistance());

            if(distance <= minD) {
                individual.setIndividualState(IndividualState.INCOLLISION);
                temp.setIndividualState(IndividualState.INCOLLISION);

                int maxC = Math.max(individual.getcTime(), temp.getcTime());
                double P = min(this.R * (1 + maxC / 10) * temp.getMaskValue() * individual.getMaskValue() * (1 - minD / 10), 1);
                if(individual.isIfInfected()) {
                    temp.setIfInfected(true); // temp enfekte oldu ...
                    temp.setProbability(P);
                }
                else if(temp.isIfInfected()) {
                    individual.setIfInfected(true); // mediatore haber veren enfekte oldu ...
                    individual.setProbability(P);
                }
```

Then, by creating two different threads, each individual is waited for the time it should wait, and the necessary methods are called.

```
Thread sleepThread1 = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            // The standby function of the interacting individual is called. This function should run in a separate thread.
            individual.waiting(maxC);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

Thread sleepThread2 = new Thread(new Runnable() {
    @Override
    public void run() {
        try {
            // The standby function of the interacting individual is called. This function should run in a separate thread.
            temp.waiting(maxC);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});
```

The function these threads are called initially wait C seconds to simulate the collision time. Then their directions change. A random new direction is given. They can go in the same direction and collide again. Then, if there is a patient among them for the first time, the infectedThread must be run in order to simulate the time of going to the hospital. If the individual is already sick, this thread has been pre-worked. It should not be restarted. If two healthy individuals collide, after C seconds they continue as if nothing had happened.

```
/**
 * It is the function in which the collision is simulated and runs in a separate thread.
 * @param c
 * @throws InterruptedException
 */
public void waiting(int c) throws InterruptedException {

    Thread.sleep( millis: c * 1000); // c saniye bekleyin ...
    this.direction = IndividualDirections.values()[getRandom(0,3)]; // yönü değişsin ...

    // CARPISMA BITTI !!!
    // HASTA OLAN VAR MI KONTROLU ...
    if(this.isIfInfected() && this.isIfFirstInfected()) { // ben hastalandım mı ?
        this.setIndividualState(IndividualState.INFECTED); // durumumu hasta olarak değiştir..
        Thread goHospitalThread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    // İyileşme, ölme fonksiyonu çalışsın
                    infectedThreadMethod();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        goHospitalThread.start();
    } else {
        // Demekki iki sağlıklı birey çarpışmış, INCOLLISION olan durumumu NORMAL yap
        this.setIndividualState(IndividualState.NORMAL);
    }
}
```

Procedure-Consumer Paradigm

If the individual is infected, infectedThreadMethod is called as the continuation of the code above. In this method, the procedure-consumer paradigm is applied. The procedures applied here are the same as what we learned in the System Programming course last semester.

First 25 seconds to this Thread. Because individuals can infect the disease for about 25 seconds after the disease starts.

```
private void infectedThreadMethod() throws InterruptedException {
    // System.out.println("THREAD : " + this.ID);
    this.setIfFirstInfected(false);
    Thread.sleep( millis: 25 * 1000); // 25 saniye gezmem lazım, 25 saniye sonra en erken tedavi olabilirim

    Thread thread = new Thread(new Runnable() {
        @Override
        public void run() {
            Hospital.lock.lock(); // hastaneye girebiliyom mu, ortak değişkenleri kullanabilmek için
            //System.out.println("HOSPITAL SIZE : " + hospital.getHospitalSize() + " LIST SIZE : " + hospital.getSize());
            while (hospital.getHospitalSize() == hospital.getSize()) {
                try {
                    //
                    if(Hospital.empty.await( time: deadTime - 25, TimeUnit.MILLISECONDS)) {
                        // 25 saniye geçti + ölüm sayısı kadar geçti, hastanede yer yok, öldüm :(
                        setIndividualState(IndividualState.DEAD); // durumumu ölü olarak ayarla
                        Hospital.lock.unlock();
                        return;
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            hospital.addElement(tempIndividual); // hastane sırasına girebiliyorum, beni hastaneye götür.
            Hospital.full.signal();
            Hospital.lock.unlock();
        }
    });
```

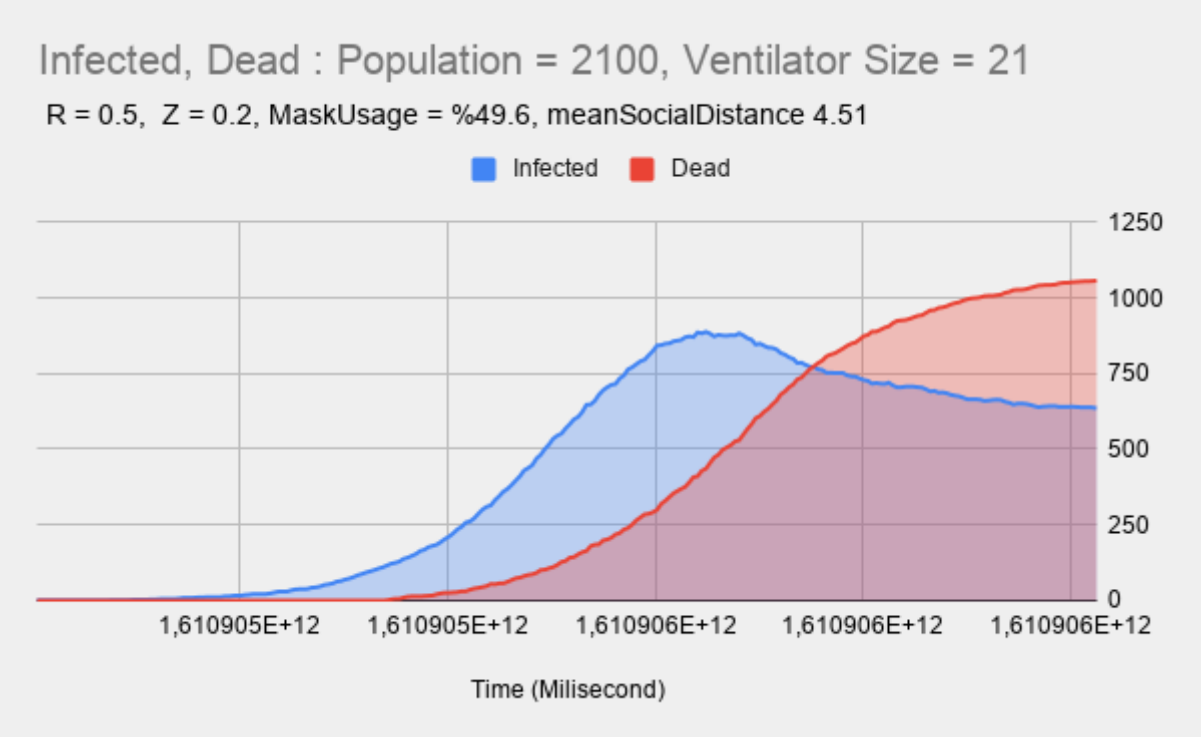
Later, Lock variable and condition variables defined in Hospital class are defined. These variables are defined as stratic. Because it should be used jointly for all individuals.

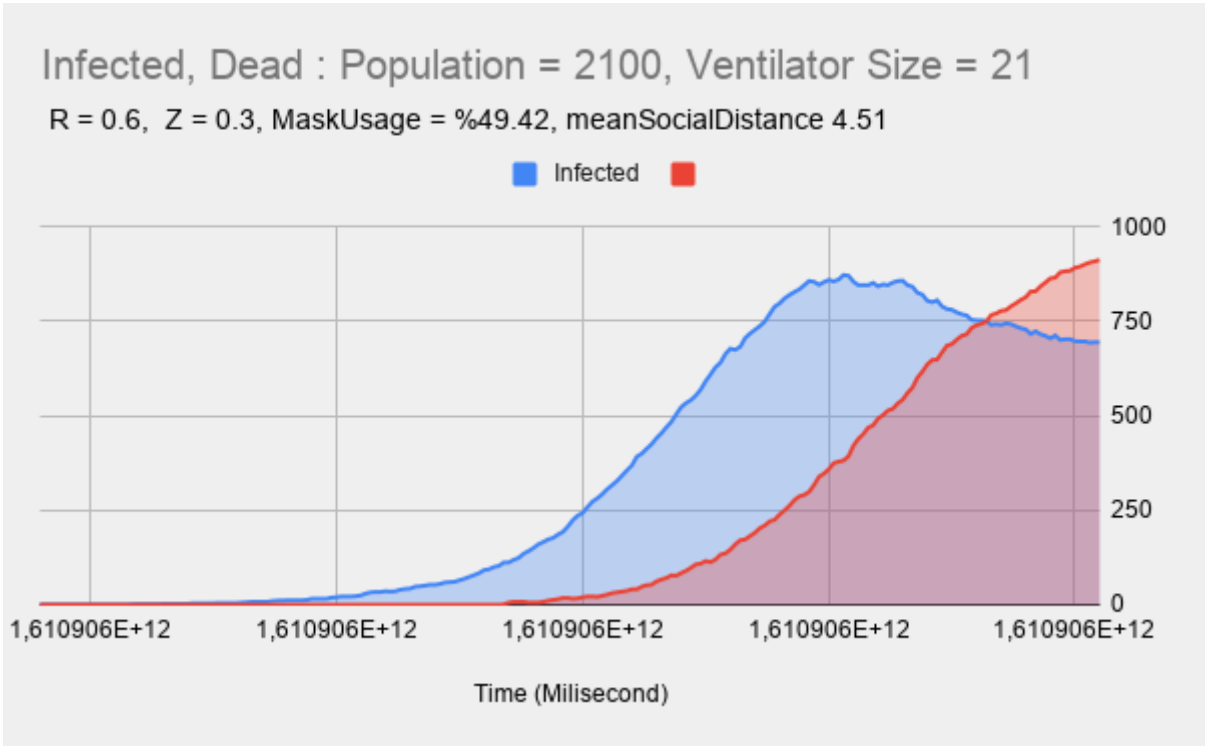
One more thread is used to simulate the hospital. Every individual uses this thread to improve itself, if time is sufficient.

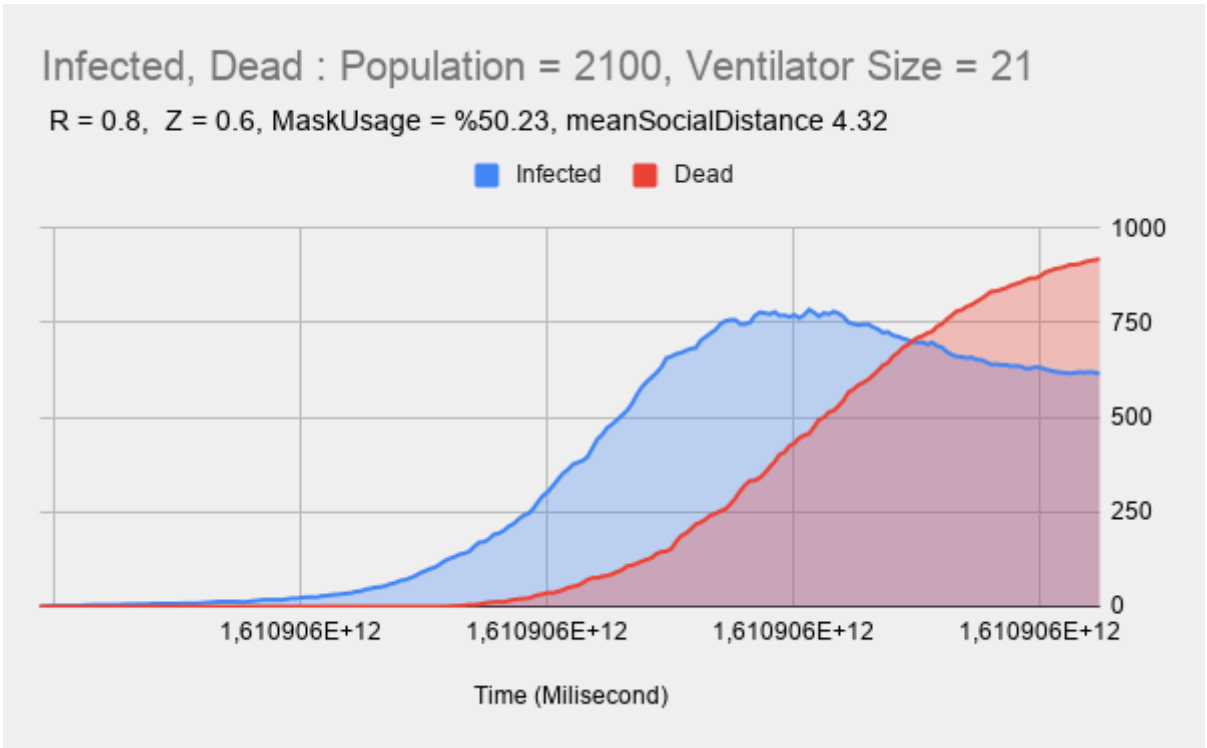
The individual can recover after 10 seconds. Here it is important to use lock and condition variables correctly. Because when the individual heals, it sends a signal to wake up other threads that are sleeping, that is, queuing.

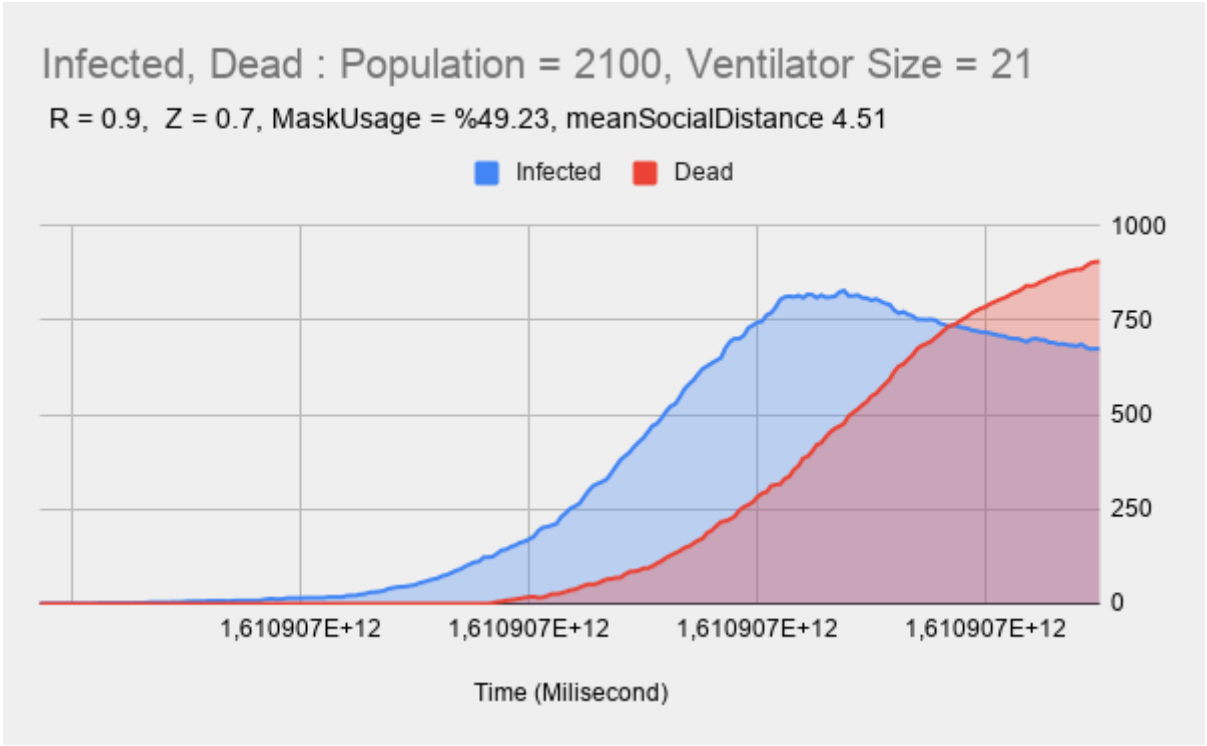
```java
Thread thread_2 = new Thread(new Runnable() {
    @Override
    public void run() {
        Hospital.lock.lock();
        while(hospital.getSize() == 0) {
            try {
                // Hastane sırasında kimse yoksa uyumam lazım.
                Hospital.full.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // Bireyi aldım 10 saniye sonra tedavi edeceğim.
        Individual x = hospital.getList().get(0);
        //System.out.println("--> HOSPITAL SIZE : " + hospital.getHospitalSize() + " LIST SIZE : " + hospital.getSize());
        x.setIndividualState(IndividualState.INHOSPITAL); // durumunu hastanede diye güncelledim
        inHospitalIndividual(x);
        Hospital.empty.signal(); // başka sırada bekleyen varsa uyansın girebilirler
        Hospital.lock.unlock();
        try {
            Thread.sleep( millis: 10 * 1000); // 10 saniye bekle
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Hospital.lock.lock();
        hospital.getList().remove(x); // listeden bireyi kaldır, çünkü birey artık tedavi oldu ve iyileşti :)
        Hospital.lock.unlock();
        x.setIndividualState(IndividualState.NORMAL); // durumumu normal olarak güncelle
        x.setIfInfected(false); // birey artık başka hastatık kapabilir, umarım kapmaz
        x.setIfFirstInfected(true);
    }
});
```

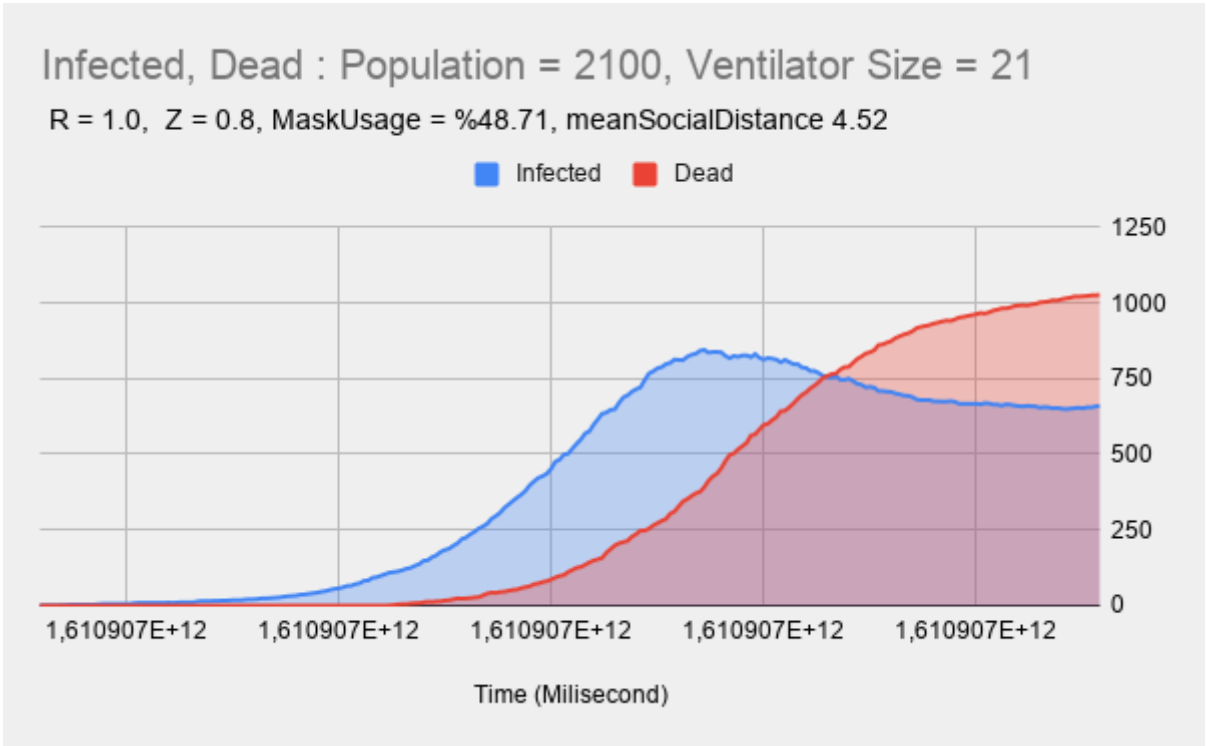
Graphics

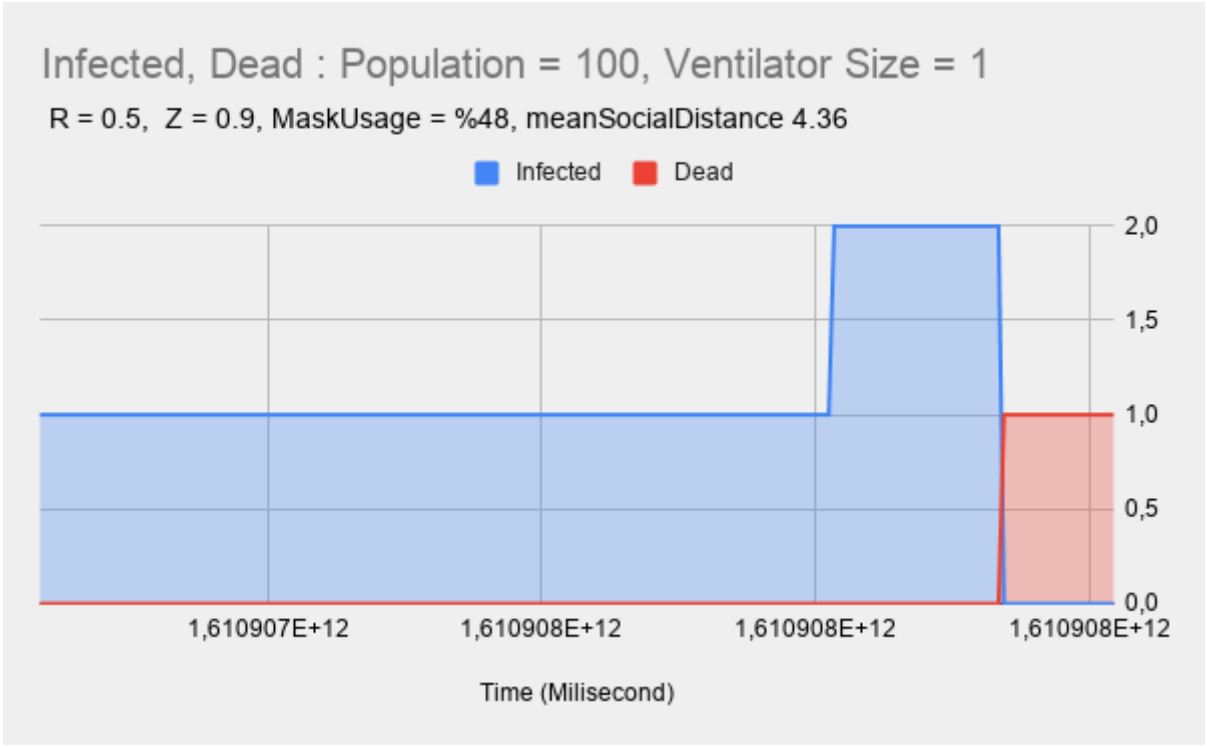


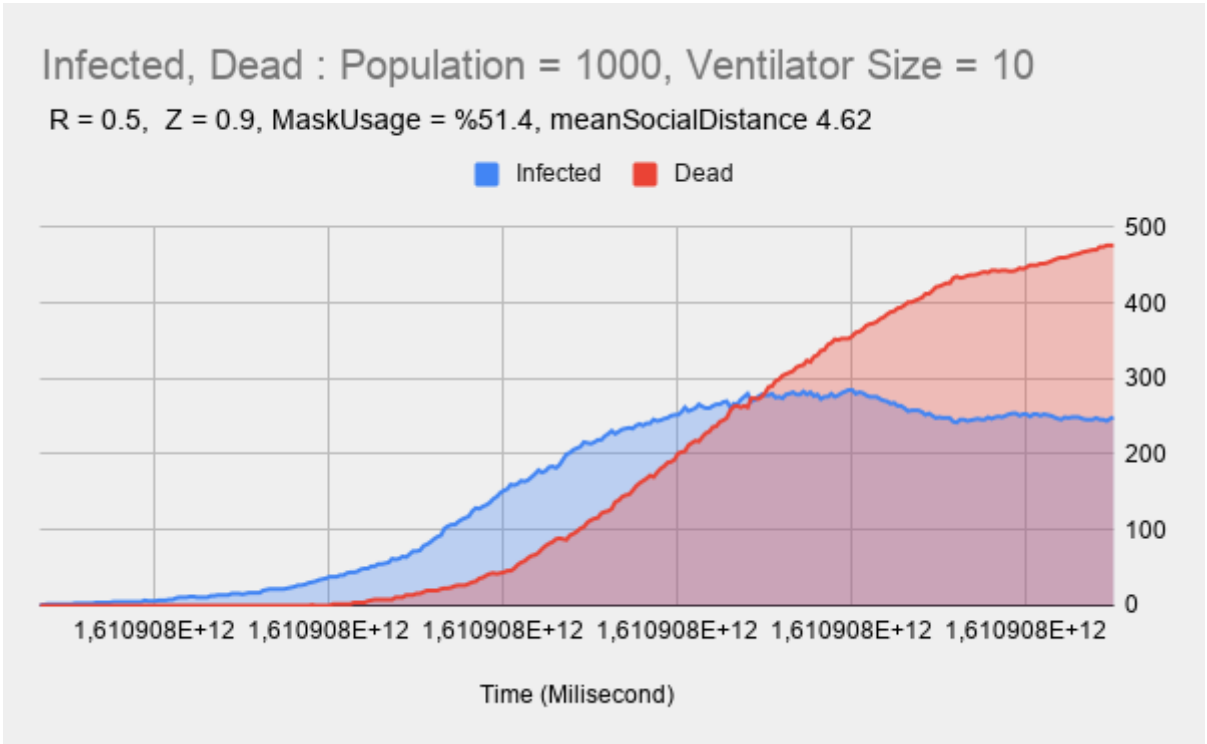

Infected, Dead : Population = 2100, Ventilator Size = 21
R = 0.6, Z = 0.3, MaskUsage = %49.42, meanSocialDistance 4.51
Infected
1000
750
500
250
0
1,610906E+12
1,610906E+12
1,610906E+12
1,610906E+12
1,610906E+12
Time (Milisecond)


Infected, Dead : Population = 2100, Ventilator Size = 21
R = 0.8, Z = 0.6, MaskUsage = %50.23, meanSocialDistance 4.32
Infected
Dead
1000
750
500
250
0
1,610906E+12
1,610906E+12
1,610906E+12
1,610906E+12
Time (Milisecond)


Infected, Dead : Population = 2100, Ventilator Size = 21
R = 0.9, Z = 0.7, MaskUsage = %49.23, meanSocialDistance 4.51
Infected
Dead
1000
750
500
250
0
1,610907E+12
1,610907E+12
1,610907E+12
1,610907E+12
Time (Milisecond)


Infected, Dead : Population = 2100, Ventilator Size = 21
R = 1.0, Z = 0.8, MaskUsage = %48.71, meanSocialDistance 4.52
Infected
Dead
1250
1000
750
500
250
0
1,610907E+12
1,610907E+12
1,610907E+12
1,610907E+12
1,610907E+12
Time (Milisecond)

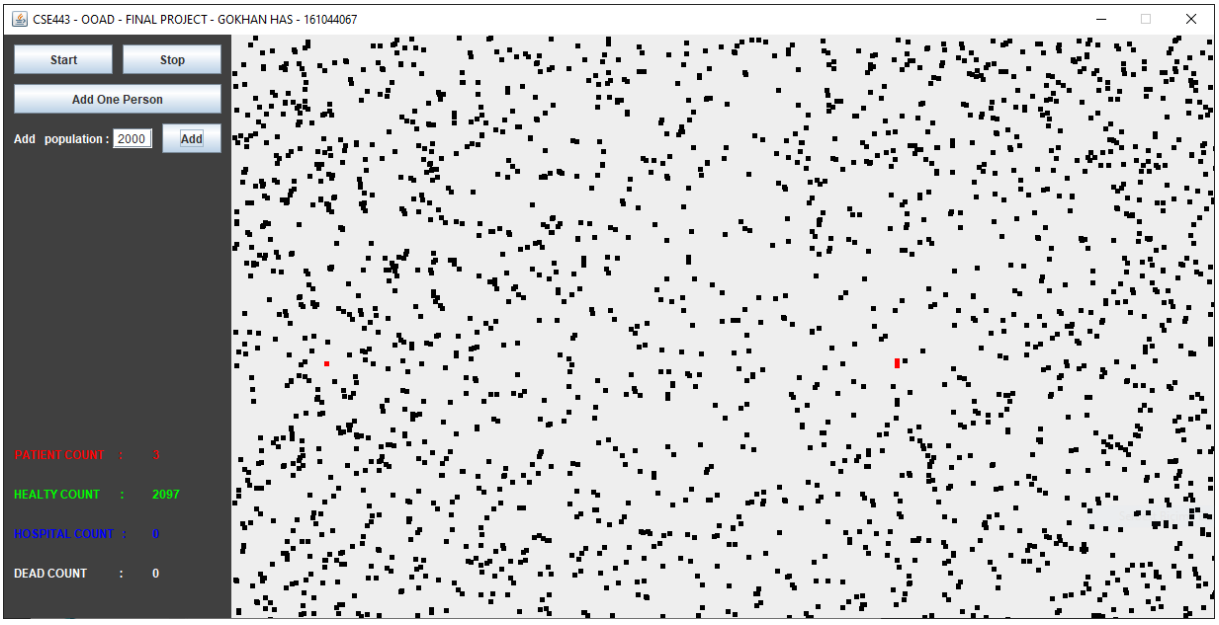

Infected, Dead : Population = 100, Ventilator Size = 1
R = 0.5, Z = 0.9, MaskUsage = %48, meanSocialDistance 4.36
Infected
Dead
2,0
1,5
1,0
0,5
0,0
1,610907E+12
1,610908E+12
1,610908E+12
1,610908E+12
Time (Milisecond)

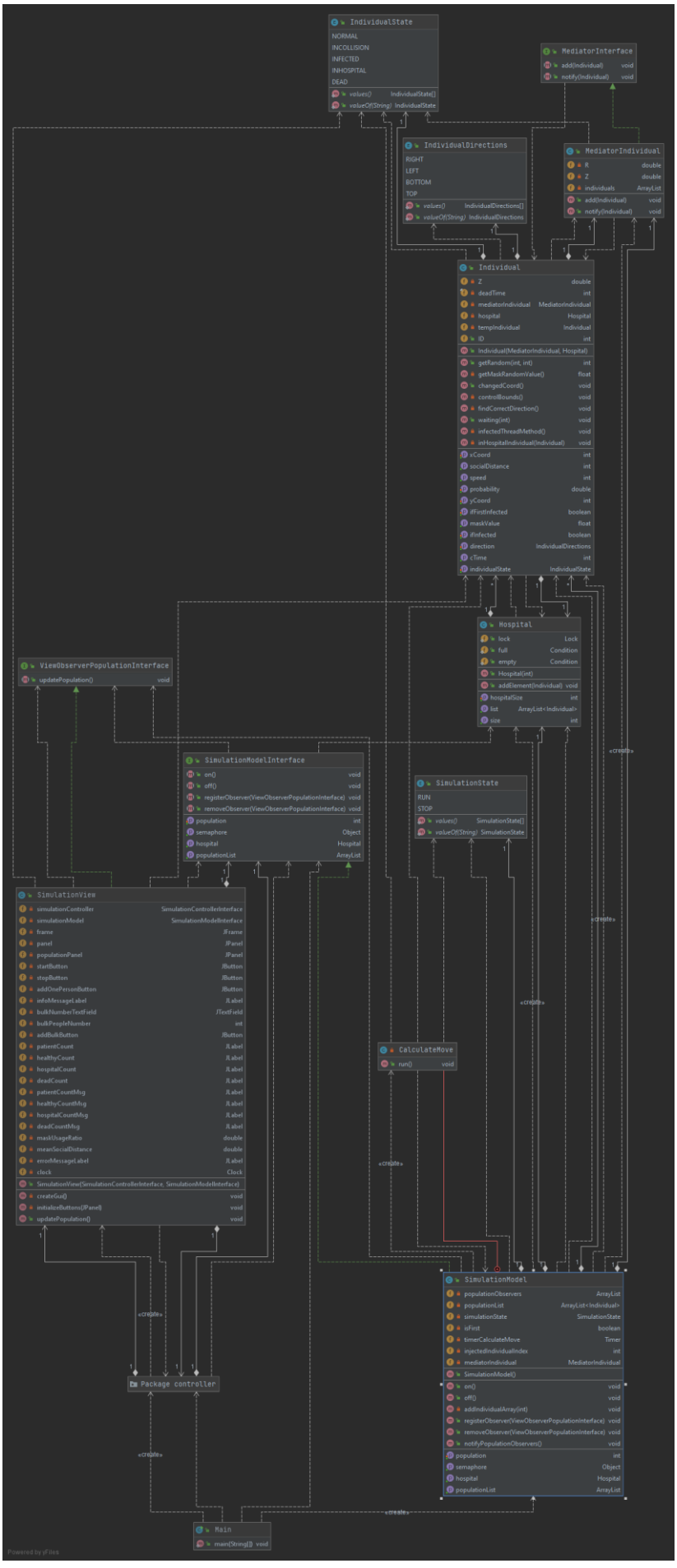

Infected, Dead : Population = 1000, Ventilator Size = 10
R = 0.5, Z = 0.9, MaskUsage = %51.4, meanSocialDistance 4.62
Infected
Dead
500
400
300
200
100
0
1,610908E+12
1,610908E+12
1,610908E+12
1,610908E+12
1,610908E+12
1,610908E+12
Time (Milisecond)

At the beginning of the graphs, I kept the population number constant and increased the R and Z values. I tried to run it at the same time while doing my tests. However, sometimes the contagion between individuals is not very rapid. I ignored this issue and continued the tests. I observed that the number of deaths increased as the values of R and Z increased.

This increase exponentially increases first, then remains a little constant, and finally, if we have enough space, it is possible to see that it decreases again exponentially.

Then I haven't tried the population count to be 100 and 100. Other parameters remained constant as in the 2100 population. When the population number was 100, it could infect a maximum of 2 people. Since only one of them can receive treatment at the hospital (because there is only one ventilator in the hospital), the other individual died at the same time. I observed that as the population number increased, the death toll increased in the same period.




IndividualState
NORMAL
INCOLLISION
INFECTED
INHOSPITAL
DEAD
values() IndividualState[]
valueOf(String) IndividualState
MediatorInterface
add(Individual) void
notify(Individual) void
IndividualDirections
RIGHT
LEFT
BOTTOM
TOP
values() IndividualDirections[]
valueOf(String) IndividualDirections
MediatorIndividual
R double
Z double
individuals ArrayList
add(Individual) void
notify(Individual) void
Individual
Z double
deadTime int
mediatorIndividual MediatorIndividual
hospital Hospital
tempIndividual Individual
ID int
Individual(MediatorIndividual, Hospital)
getRandom(int, int) int
getMaskRandomValue() float
changedCoord() void
controllBounds() void
findCorrectDirection() void
waiting(int) void
infectedThreadMethod() void
inHospitalIndividual(Individual) void
xCoord int
socialDistance int
speed int
probability double
yCoord int
ifFirstInfected boolean
maskValue float
ifInfected boolean
direction IndividualDirections
cTime int
individualState IndividualState
Hospital
lock Lock
full Condition
empty Condition
Hospital(int)
addElement(Individual) void
hospitalSize int
list ArrayList<Individual>
size int
ViewObserverPopulationInterface
updatePopulation() void
SimulationModelInterface
on() void
off() void
registerObserver(ViewObserverPopulationInterface) void
removeObserver(ViewObserverPopulationInterface) void
population int
semaphore Object
hospital Hospital
populationList ArrayList
SimulationState
RUN
STOP
values() SimulationState[]
valueOf(String) SimulationState
SimulationView
simulationController SimulationControllerInterface
simulationModel SimulationModelInterface
frame JFrame
panel JPanel
populationPanel JPanel
startButton JButton
stopButton JButton
addOnePersonButton JButton
infoMessageLabel JLabel
bulkNumberTextField JTextField
bulkPeopleNumber int
addBulkButton JButton
patientCount JLabel
healthyCount JLabel
hospitalCount JLabel
deadCount JLabel
patientCountMsg JLabel
healthyCountMsg JLabel
hospitalCountMsg JLabel
deadCountMsg JLabel
maskUsageRatio double
meanSocialDistance double
errorMessageLabel JLabel
clock Clock
SimulationView(SimulationControllerInterface, SimulationModelInterface)
createGui() void
initializeButtons(JPanel) void
updatePopulation() void
CalculateMove
run() void
SimulationModel
populationObservers ArrayList
populationList ArrayList<Individual>
simulationState SimulationState
isFirst boolean
timerCalculateMove Timer
injectedIndividualIndex int
mediatorIndividual MediatorIndividual
SimulationModel()
on() void
off() void
addIndividualArray(int) void
registerObserver(ViewObserverPopulationInterface) void
removeObserver(ViewObserverPopulationInterface) void
notifyPopulationObservers() void
population int
semaphore Object
hospital Hospital
populationList ArrayList
Package controller
Main
main(String[]) void
«create»
Powered by yFiles