CSE 443

Object Oriented Analysis and Design

Fall 2020– 2021

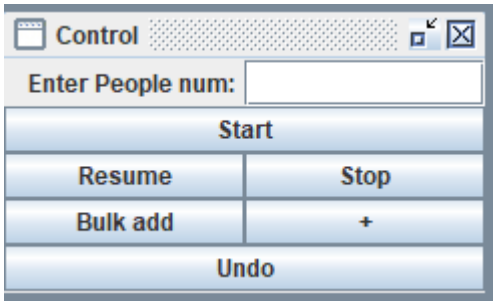
Final Report

Caner KARAKAŞ

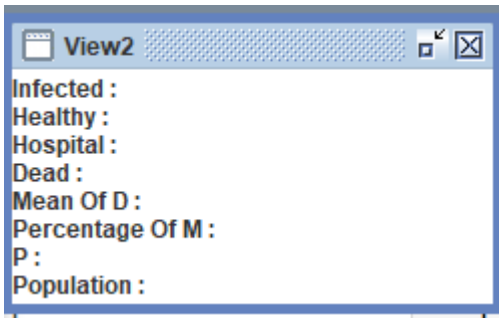
131044061

The Java Swing library was used for the visual simulation of an epidemic in human society. GUI was created from the Swing library.

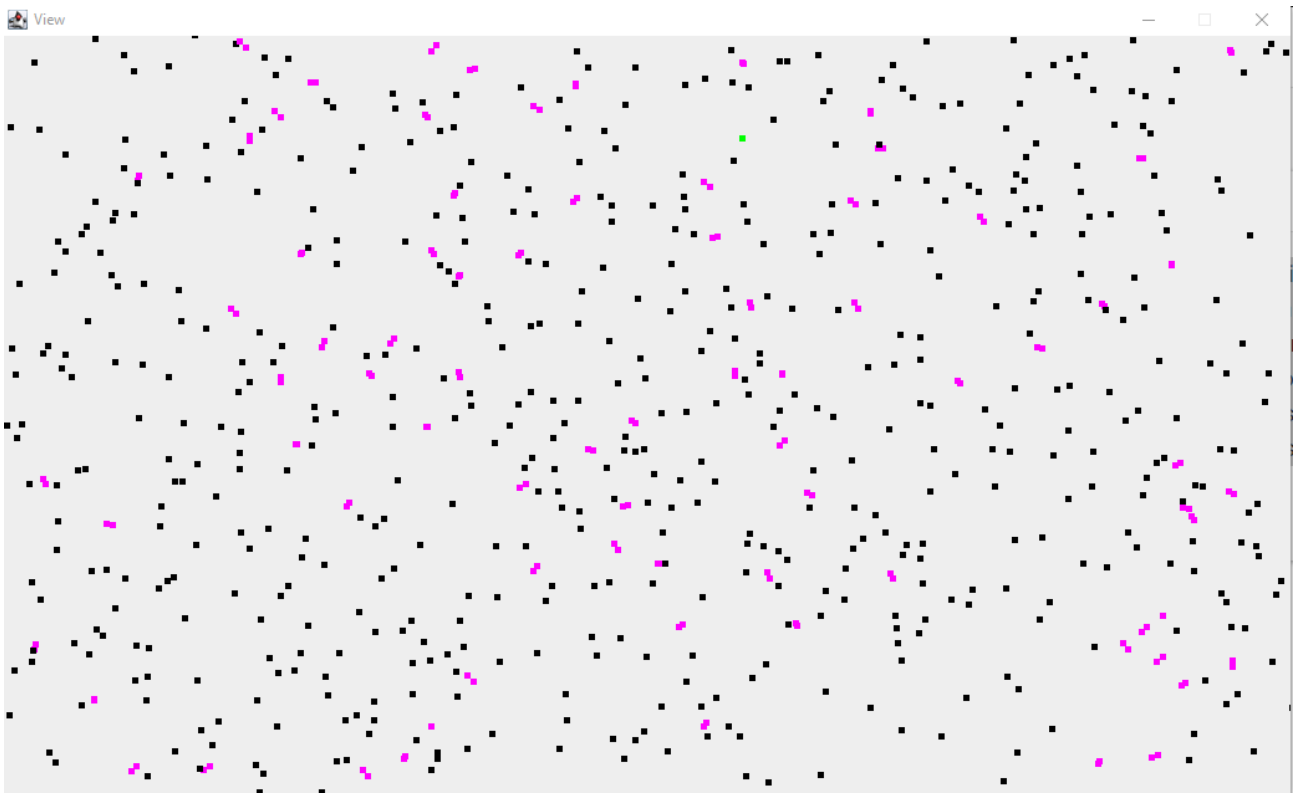
Starting the simulation in the interface presented to the user; Some buttons have been put in order to stop it, continue it again, and add people to the simulation. These buttons aim to receive and process the changes realized under the control of the user and display them on the graphical interface.



In order to give some statistics to the user during the simulation, a screen was created.



Person class was created to realize human society in simulation. This class hides the characteristics that are expected to be in people, such as position, whether they wear a mask, social distance, condition, color, speed, and the possibility of disease transmission. Many property values are randomly assigned. In the simulation to be started by the user, the view is as follows;



The user will be able to manage the simulation from the control unit. It is here that our goals that we need to realize begin. The first of our goals is to show the changes made to the user. Afterwards, it is to ensure that individuals can be formed as they wish, to be able to move individuals simultaneously, to be able to update the statistics and simulation continuously, but to do this in a way that is connected to a line, to be able to stop and replay at the desired time, to establish a structure that can synchronize and change instantly that can listen to the user continuously.

We basically used the MVC (Model, View, Controller) design model / architecture to achieve our goals. In this design, View is the part that shows the simulation to the user, the part that performs the commands to be received from the Controller through the Model, and the part that can make the updates made to the Model simultaneously. This model uses GUI components in the View part. manages these components with the Composite Design Pattern. In this way, a feature, interaction or function to be added can be added and managed very easily. Composite can do things on the back. Controller is used to process commands received from the user. One Controller object is authorized in the View. Controller uses Strategy Design Pattern. In this way, a dynamic transition between behaviors is provided. However, it reduces the relationship between Model and View. He wouldn't be interested in what was in the background and how. The model is actually the party that manages the data behind it. It has to keep the data up to date. So it needs to be independent. It uses Observer Design Pattern to keep up-to-date changes. This also allows us to use multiple Views. We can also manage more than one view at the same time. However, we have not put a separate view on our statistics screen in order to reflect the statistics more accurately in our application. Since setting up a synchronous and synchronous structure is one step before, we went through a single View so that what we will see in the screen map section and what we will see in the statistics section are synchronous and performance. But the Observer Design Pattern we use has the flexibility to update different View objects that can be added from now on. Because different views can define different interfaces and implement them, we can subscribe and update the classes we use.

The GUI class is implemented from the ActionListener interface and has a multi-threaded and always responsive structure. It contains the object of a class implemented from the ControllerInterFace interface and the object of a class implemented from the ModelInterface interface. An object of a class implemented from the ModelInterface interface subscribes for updates. An object of a class implemented from the ControllerInterFace interface executes commands.

```java
@Override
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == addBulk)
        controllerInterFace.addBulk(Integer.parseInt(inputTextField.getText()));
    else if (e.getSource() == increaseNumOfPerson)
        controllerInterFace.increaseNOP();
    else if (e.getSource() == start)
        controllerInterFace.start();
    else if (e.getSource() == resume)
        controllerInterFace.resume();
    else if (e.getSource() == stop)
        controllerInterFace.stop();
    else if (e.getSource() == undo)
        controllerInterFace.undo();
}
```

(Multi-threaded)

```java
@Override
public void updateMap() {
    Graphics g = viewPanel.getGraphics();
    g.clearRect( x: 0, y: 0, viewPanel.getWidth(), viewPanel.getHeight());
    modelInterface.getMutexPeople().lock();
    Vector<Person> people = modelInterface.getPeople();
    int maskUsed = 0; int sumOfD = 0; int healthy = 0; int dead = 0;
    int infected = 0; int atHospital = 0; double sumOfP = 0.0;
    for (Person person : people) {
        sumOfD += person.getD();
        if (person.getM()==0.2)
            maskUsed++;
        if (person.getSituation() == 0)
            healthy++;
        if (person.getSituation() == 1){
            sumOfP += person.getP();
            infected++;
        }
        if (person.getSituation() == 2)
            atHospital++;
        if (person.getDie())
            dead++;
        g.setColor(person.getColor());
        g.fillRect(person.getX_coordinate(),
                person.getY_coordinate(),  width: 5,  height: 5);
    }
    double meanOfD = (double)sumOfD / people.size();
    double meanOfM = (double)(100*maskUsed) / people.size();
    double meanOfP = sumOfP * 100;
    numOfHealthy.setText("Healthy : " + healthy);
    numOfDeadPerson.setText("Dead : " + dead);
    numOfInfected.setText("Infected : " + infected);
    numOfAtHospital.setText("Hospital : " + atHospital);
    this.meanOfD.setText("Mean Of D : " + meanOfD);
    this.meanOfM.setText("Percentage of M : " + meanOfM);
    this.population.setText("Population : " + people.size());
    p.setText("P : " + meanOfP);
    modelInterface.getMutexPeople().unlock();
}
```

(Updates)

The ControllerClass class is implemented from the ControllerInterFace interface. It contains a GUI object and an object implemented from the ModelInterface interface. GUI object is created in this class. The ModelInterface object is imported. A behavioral design pattern is used. The commands received from the user can be transferred here with the Multi-threading structure found on the GUI. Here, too, it can be processed with a class implemented from the ModelInterface interface.

```
@Override
public void start() {
    modelInterface.initialize(gui.getMemento());
}

@Override
public void stop() { modelInterface.stop(gui.getMemento()); }

@Override
public void resume() { modelInterface.resume(gui.getMemento()); }

@Override
public void increaseNOP() { modelInterface.increaseNOP(gui.getMemento()); }

@Override
public void addBulk(int NOP) { modelInterface.addBulk(NOP, gui.getMemento()); }

@Override
public void undo() { modelInterface.undo(gui.getMemento()); }
```

The ModelClass class is implemented from the ModelInterface interface. This class can perform the desired functions. For example, when the start button is pressed, the initialize method of an object from this class is called within the start method in the Controller. This allows us to add different meanings to the start button later. A maintenance or change to be made here can be made at much less cost and we have a flexible structure that is open to change.

```
public interface ModelInterface {
    void initialize(Memento memento);
    void resume(Memento memento);
    void stop(Memento memento);
    void undo(Memento memento);
    void addBulk(int NOP, Memento memento);
    void registerObserver(GUIMapObserver o);
    void removeObserver(GUIMapObserver o);
    void increaseNOP(Memento memento);
    Vector<Person> getPeople();
    ReentrantLock getMutexPeople();
}
```

In order to start our simulation, human input must be made into our database. In other words, the user cannot start the simulation without entering any data. An integer number is expected to be entered in the text space given before the start button. No data can be added except for integer number entry. To add data, a number is entered and the Add Bulk button is clicked. Add Bulk button allows us to enter data in batch. The only data entry is provided by the + button. The other buttons have no meaning unless data is entered with these two buttons. After entering the data, the simulation can be started by pressing the start button. After the simulation started, it is allowed to enter data collectively or individually at any time, thanks to a multi-thread structure and a GUI that always listens to the user. While the simulation is running, the database can be updated simultaneously. Likewise, the user can stop and replay the simulation at any time. This is exactly where we used another flexible structure that is easy to maintain and developable. This structure will keep us in its last state and perhaps even take it back when desired. Because a user who presses the stop button should be able to return to the desired position when he says resume. Due to our structure, we have based on the last position for now and allowed additions. However, if this place is flexible, it can easily change at another time. That's why we used the Memento Design Pattern here. In addition to keeping our position, this design pattern has enabled us to return. If we want, we can put an undo button. For now, our Person class is not clonable, so the undo button will not work well. It will keep our last position. When we press the resume button again, we can return to the last position we held in the Memento class

```java
import java.util.LinkedList;
import java.util.Vector;

public class Memento {
    private LinkedList<Vector<Person>> peopleBackups = new LinkedList<>();


    public void addBackup(Vector<Person> people){ peopleBackups.add(people); }

    public Vector<Person> getLastState(){
        Vector<Person> people = peopleBackups.getLast();
        peopleBackups.remove();
        return people;
    }
}
```

```java
@Override
public void resume(Memento memento) {
    if (stop){
        System.out.println("resume");
        mutexPeople.lock();
        people = memento.getLastState();
        mutexPeople.unlock();
        stop = false;
        undo = false;
    }
    else{
        if (start){
            System.out.println("hala resume");
        }
        else System.out.println("please start action dont click resume");
    }
}
```

```java
@Override
public void undo(Memento memento) {
    if (!undo){
        if (start){
            mutexPeople.lock();
            try {
                people = memento.getLastState();
                mutexPeople.unlock();
                notifyGuiMapObservers();
            }finally {
                mutexPeople.unlock();
            }
        }
        else System.out.println("please resume action dont click undo");
    }
    else System.out.println("resume bas");
}
```

How will person objects, that is, the interaction between our individuals and their continuity? How will individuals who collide with these interactions, who are hospitalized, who died, be handled? How will the updates and transactions be made with a scheduler? We analyzed the interaction, the first of these questions, with the Mediator Design Pattern as requested. Our purpose of using this design pattern is to examine each individual one by one, and save the burden of taking other objects of other classes one by one that can be added later. Because Mediator does this flexibly and with low maintenance costs. For example, when we add a tank object, the change we will make only on this class for the human event with the tank will be much easier and applicable. In addition, the interaction between isosclass objects is also facilitated. For example, it is difficult to look at what will happen when you collide with Ali individual and Ayşe individual and to examine this with all individuals one by one. But if Ali informs a manager like Mediator in which direction he is moving and his interaction with anyone there is left to the Mediator object, this process becomes easier. Here we are doing exactly that. First of all, we create a Mediator object in our Model class. We give this to all Person objects we create so they can recognize it. Then we add those objects to the Mediatore as if they were subscribers (similar to an Observer relationship). Then, when the walk method is called on an object from Person class, the operations are done and the Mediator is informed in the walk method, that is, the walk method of the Mediator is called. The mediator also manages the interaction there. He sends what will go to the hospital, and makes those who have to wait wait.

```java
@Override
public void walk(Person person) {
    for (Person p: people) {
        if (p != person && !p.getInvisible()) {
            if (isCollision(p, person)){
                int maxC = Math.max(p.getC(), person.getC());
                setInvisiblePeople(p, person);
                if (p.getSituation()==1){
                    if (person.getSituation()!=1){
                        setInfectPeople(p, person);
                        person.setP(collisionInfectProb(person, p, maxC));
                        if (!p.isQueueAtHospital())
                            p.hospitalWaiting(maxC);
                        else p.collisionWaiting(maxC);
                        person.hospitalWaiting(maxC);
                    }
                    else{
                        p.collisionWaiting(maxC);
                        person.collisionWaiting(maxC);
                    }
                }
                else{
                    if(person.getSituation()==1){
                        setInfectPeople(p, person);
                        p.setP(collisionInfectProb(person, p, maxC));
                        p.hospitalWaiting(maxC);
                        if (!person.isQueueAtHospital())
                            person.hospitalWaiting(maxC);
                        else person.collisionWaiting(maxC);
                    }
                    else{
                        p.collisionWaiting(maxC);
                        person.collisionWaiting(maxC);
                    }
                }
                break;
            }
        }
    }
}
```

```java
public void walk(){
    switch (direction) {
        case 0:
            if (y_coordinate - S <= 10)
                newDirection();
            y_coordinate -= S;
            break;
        case 1:
            if (x_coordinate + S >= 1000)
                newDirection();
            x_coordinate += S;
            break;
        case 2:
            if (y_coordinate + S >= 600)
                newDirection();
            y_coordinate += S;
            break;
        case 3:
            if (x_coordinate - S <= 10)
                newDirection();
            x_coordinate -= S;
            break;
        default:
            break;
    }
    mediator.walk( person: this);
}
```

```java
public void setMediator(Mediator mediator) {
    this.mediator = mediator;
    this.mediator.addPerson(this);
}
```

```java
@Override
public void addPerson(Person person) { people.add(person); }
```

```java
p.setMediator(mediator);
```

The collision situations we are realizing on the mediator are realized as desired. Person objects calling Mediator's walk method are compared with individual members. Members are expected to be visible and different. If this condition is met, a collision check is made. It checks whether these two objects are in the same location and are touching each other. Objects that pass this condition are made invisible. Then their status is checked. In other words, it is checked whether an object is sick or healthy. If two individuals colliding are sick, they are held and released for a maximum of their waiting times. If only one is sick, it infects the other. In addition, the probability of being sick is calculated and this value is recorded for the new patient. Then he is sent on hold for the newly patient hospital. The other patient goes on waiting for the hospital if he / she has not attended the hospital wait before, and if he / she attends the normal wait. It is here that a new design pattern was needed. As we noticed here, there are two different waiting behaviors. Besides these, different waiting behaviors can be added or update or maintenance can be done with these waiting behaviors. Here Strategy Design Pattern has been applied to standby behaviors to increase flexibility and reduce maintenance cost. The Waiting interface was created and two classes that were implemented from this interface were created. These classes are the CollisionWaiting class and the HospitalWaiting class.

We keep two Waiting objects that can perform 2 different waits in the Person class and when we create Person objects, we also create them. Because an object must dynamically execute two different waits.

```
private Waiting waitingCollision; waitingCollision = new CollisionWaiting( person: this);
private Waiting waitingHospital;  waitingHospital = new HospitalWaiting( person: this);
```

When we tried to use them by writing two different functions first, we could not apply the design pattern. We tried to fix it and call it with a single method. Here, we update the waiting object with which waiting object we will use and call our waiting method. We did this easily with an extra Waiting object.

```
public void setWaiting(Waiting waiting) {
    this.waiting = waiting;
}
```

```
public void waiting(int waitingTime) { waiting.waiting(waitingTime); }
```

```
        p.setWaiting(p.getWaitingHospital());
        if (!person.isQueueAtHospital())
            person.setWaiting(person.getWaitingHospital());
        else
            person.setWaiting(person.getWaitingCollision());
    }
    else{
        p.setWaiting(p.getWaitingCollision());
        person.setWaiting(person.getWaitingCollision());
    }
}
p.waiting(maxC);
person.waiting(maxC);
break;
```

The CollisionWaiting class holds Person object in it. This object tells who to wait. Override the waiting method because it is implemented from the Waiting interface. creates a thread in it. It calls the waitingCollision method with this thread. This method finishes the job by making the Person object visible again with random direction finding after the sleeping process. Here he was able to work simultaneously in interfering with the work flow using thread.

The HospitalWaiting class also holds Person object. In addition, it maintains a LinkedList where it can hold another Person. Because during this waiting period, entrance and exit are made to the hospital and here we are asked to approach this with a producer-consumer relationship. Primarily the retained LinkedList is required for this. Let's start to explain our approach to this relationship. The idea we think of in our hospital waiting algorithm, which we approach like the producer-consumer problem, is; Let the Producer act as an ambulance. The duty of this ambulance will pick up patients who call the hospital from their home and leave them to the hospital garden. However, the garden has a capacity here. We determine this as requested from us. All he has to do is go and bring the patients with the order given to him as the hospital garden is empty or empty. So, how will the patients who are brought to the hospital garden enter with their sick conditions? Here the consumer comes into play. Consumer, like a caregiver, hospital attendant, nurse, is to carry patients left by the ambulance to the doctor. Of course, if the garden is empty, they should wait. When he is sick in the garden, they are told to take an order and take it. In line with this logic, we created a Producer class. This class is implemented as Runnable implement. Because there will not be an ambulance, the hospital should also work while he goes and takes patients. Likewise, a Consumer class has been created. This class also has the same Runnable implement. We will use these classes in the producerConsumer method called in the Waiting method in the HospitalWaiting class. This method first puts the patient on the waiting queue of the hospital. Then it makes it visible and holds it for 25 seconds like everyone else. He then checks if his life time has expired. If it's still alive, it creates a producer object with a thread. So he creates and calls an ambulance. Likewise creates consumer object. So a nurse creates and engages. Producer puts the patient in line, that is, in the garden, if there is an empty place in the row, in accordance with the

above logic. The consumer, that is, the nurse, takes if there is a patient and continues on his way. In the meantime, it informs each other in cases of putting it on the list and taking it from the list. He tells each other to wait while they work. He tells that they have to work while waiting. A classic producer consumer adaptation is made.

```
private void producerConsumer(){
    person.setQueueAtHospital(true);
    if (person.getDie())
        return;
    person.setInvisible(false);
    try {
        Thread.sleep( millis: 25000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    person.setWaitingHospitalTime(person.getWaitingHospitalTime() - 25);
    if (person.getWaitingHospitalTime()<=0)
        person.setDie(true);
    else {
        Thread producer = new Thread(new Producer(person));
        producer.start();
        Thread consumer = new Thread(new Consumer());
        consumer.start();
    }
}
```

```
public void produce(){
    lock.lock();
    while (HospitalWaiting.personLinkedList.size() == ModelClass.hospitalSize) {
        try {
            if (!full.await(person.getWaitingHospitalTime(), TimeUnit.MILLISECONDS))
                person.setDie(true);
                lock.unlock();
                return;
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    HospitalWaiting.personLinkedList.add(person);
    Consumer.empty.signal();
    lock.unlock();
}
```

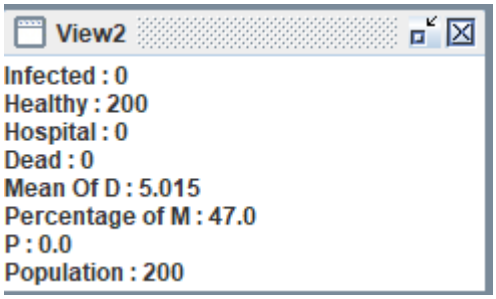
```java
private void consumer(){
    Producer.lock.lock();
    try {
        while (HospitalWaiting.personLinkedList.isEmpty())
            empty.await();
        Person person = HospitalWaiting.personLinkedList.removeFirst();
        Producer.full.signal();
        Producer.lock.unlock();
        person.goHospital();
        try {
            sleep( millis: 10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Random random = new Random();
        person.setX_coordinate(random.nextInt( bound: 1000));
        person.setY_coordinate(random.nextInt( bound: 600));
        person.setDirection(random.nextInt( bound: 4));
        person.setSituation(0);
        person.setQueueAtHospital(false);
        person.setInvisible(false);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

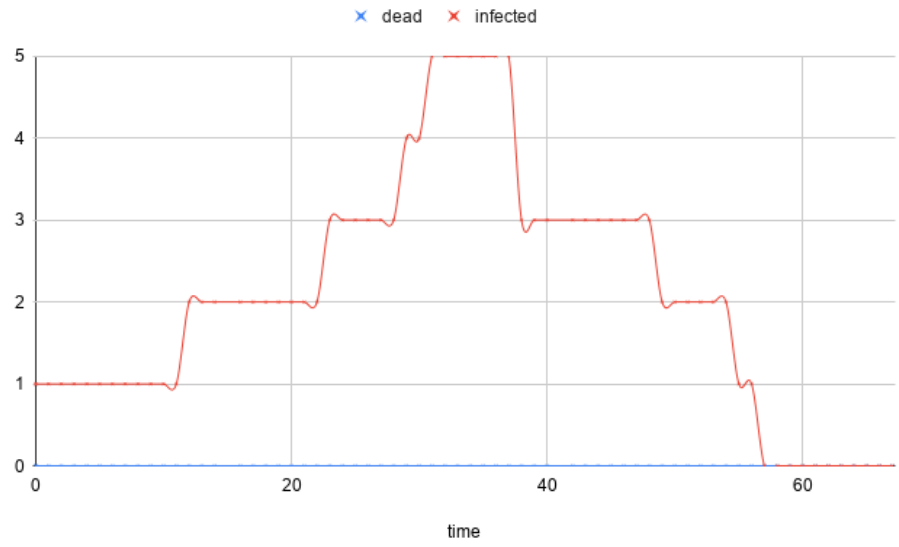
In our simulation, we implemented the desired continuity or updates based on a time schedule with the Worker class. This class is implemented from the Runnable interface. In this way, we can implement our own timer. This creates a more flexible working area for us. We can add the functionality we want here. In addition, we cannot fully establish a structure that can operate continuously at the back. With this class that holds a model object, we provide continuity in different models at the same time. With the run method coming from the runnable interface, we can work on threads without any trouble. We call the walk method of our Person objects together with the goPeople method. We call this method inside the run method. This method, which will operate in the back, works in an infinite loop and intervals are obtained by waiting for the desired time.

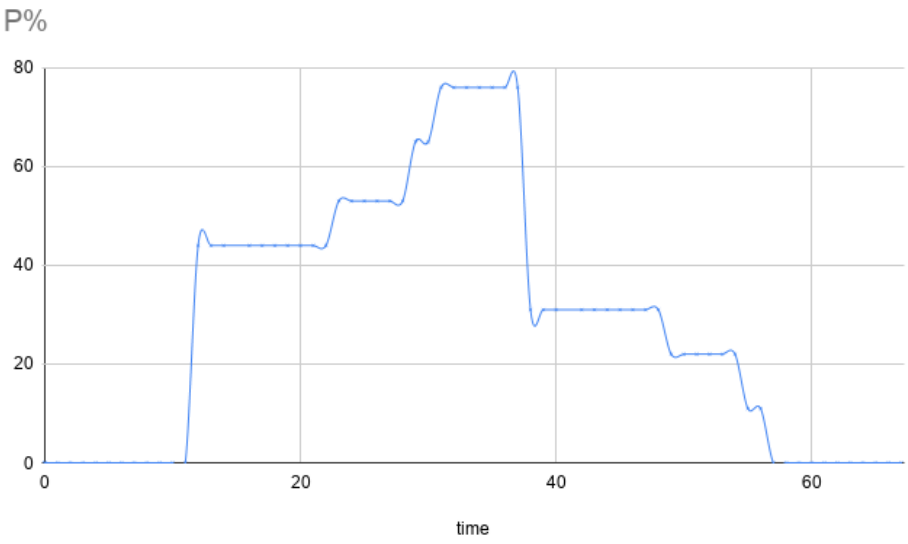
Finally, we plotted the effect of R and Z values on death and infection. In addition, we have drawn some graphs to see the disease transmission effect of mask wearing rate and average social distance. We also compared their impact on the changing population.

R = 0.5, 0.8, 1.0 Z = 0.1, 0.5, 0.9 po = 200, 1500

It has changed in the form.

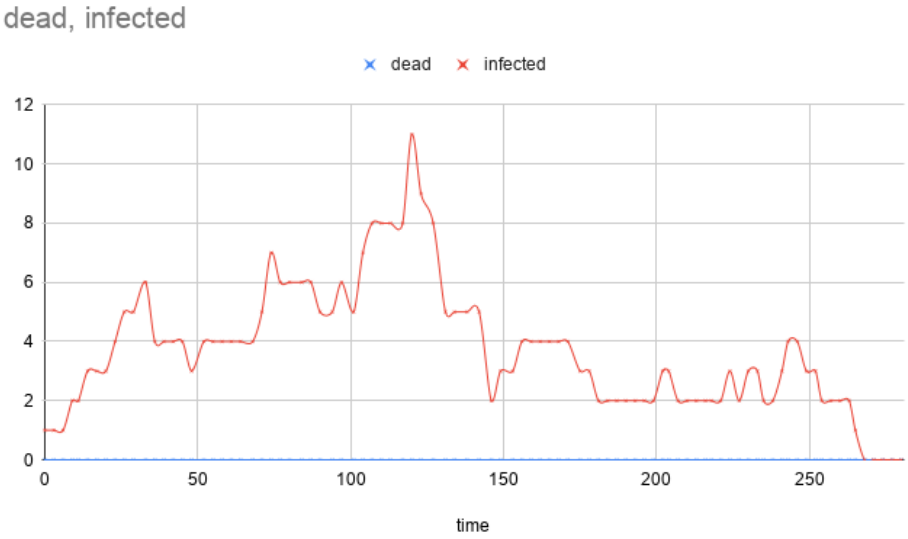


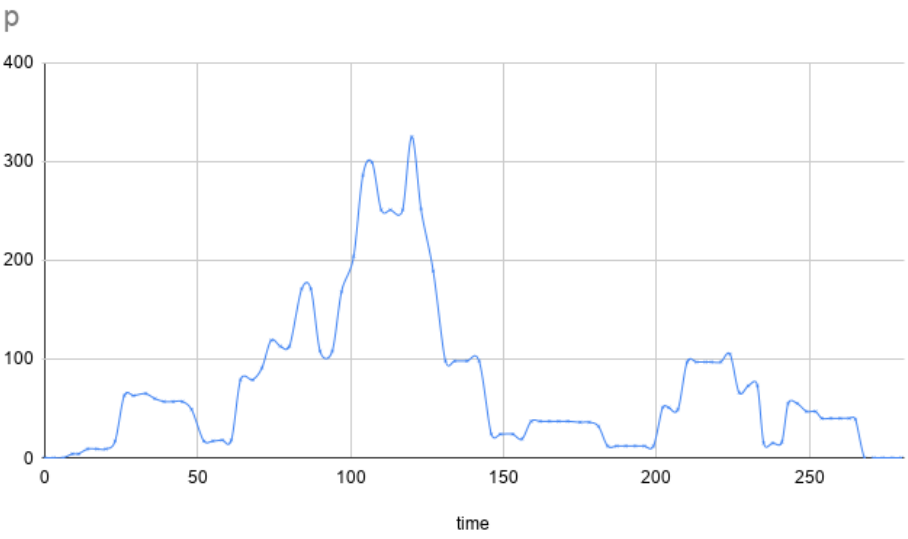


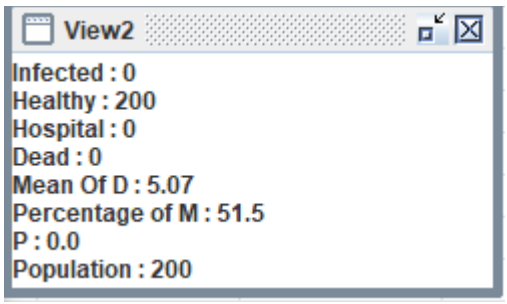


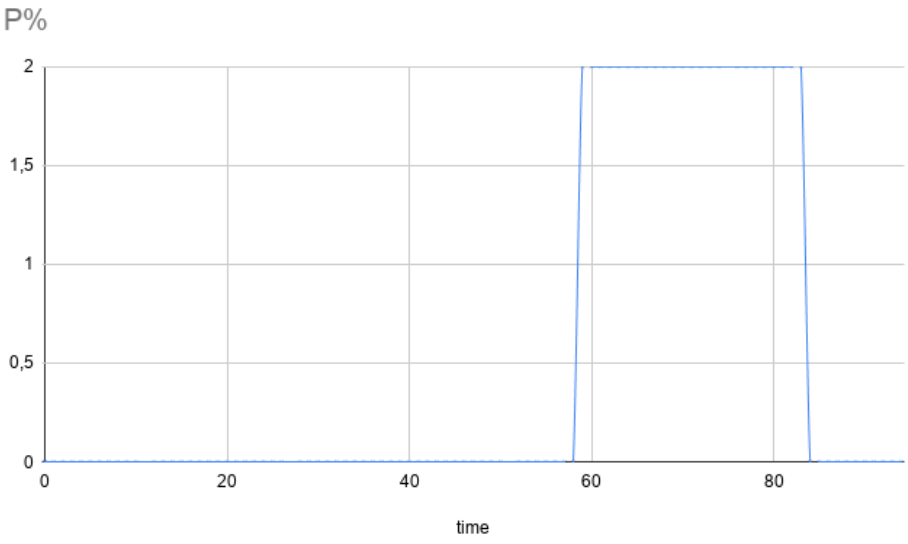
View2

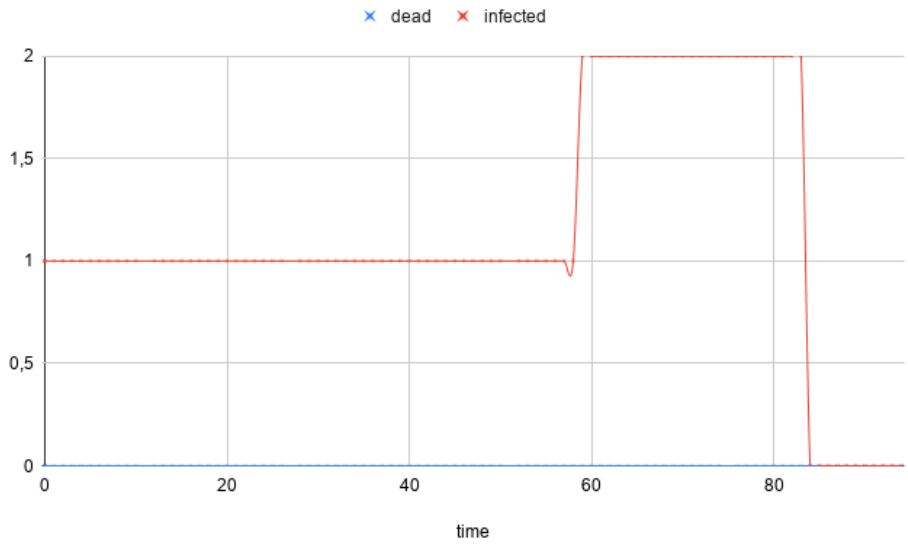
Infected : 0
Healthy : 1500
Hospital : 0
Dead : 0
Mean Of D : 4.8966666666666665
Percentage of M : 51.86666666666667
P : 0.0
Population : 1500


dead, infected
dead
infected
12
10
8
6
4
2
0
0
50
100
150
200
250
time


p
400
300
200
100
0
0
50
100
150
200
250
time

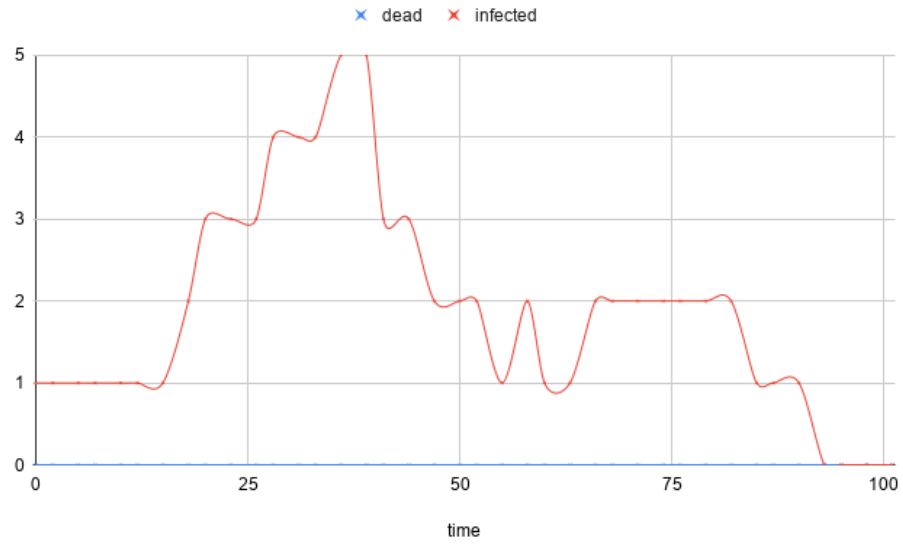

View2
Infected : 0
Healthy : 200
Hospital : 0
Dead : 0
Mean Of D : 5.07
Percentage of M : 51.5
P : 0.0
Population : 200

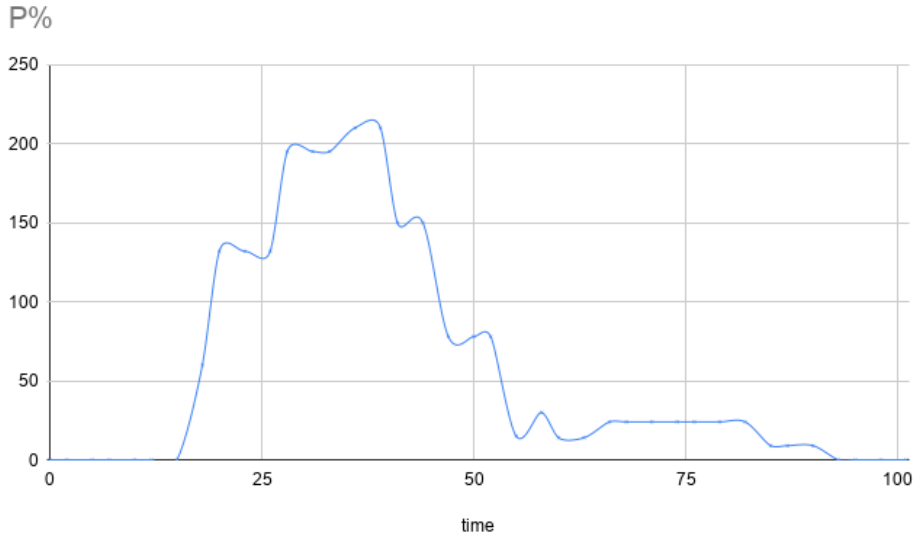




View2

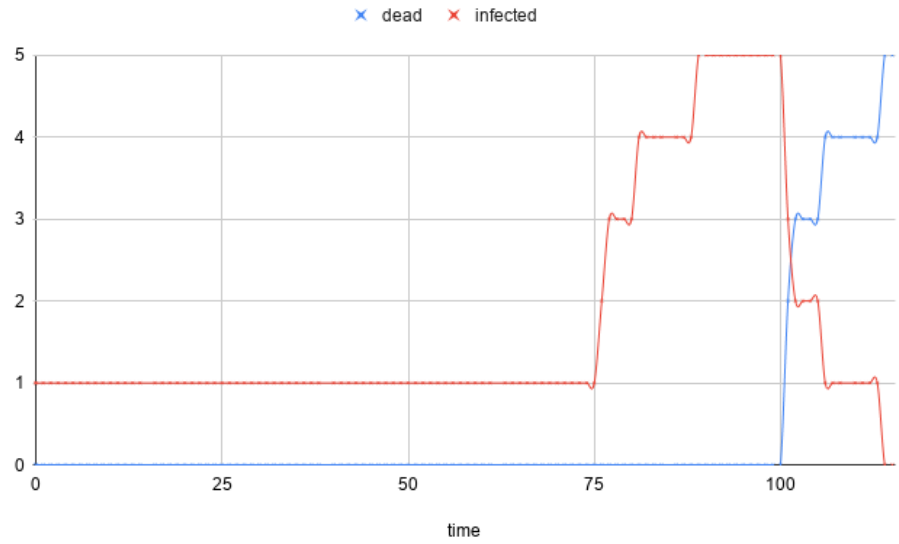
Infected : 0
Healthy : 1500
Hospital : 0
Dead : 0
Mean Of D : 4.932666666666667
Percentage of M : 51.8
P : 0.0
Population : 1500

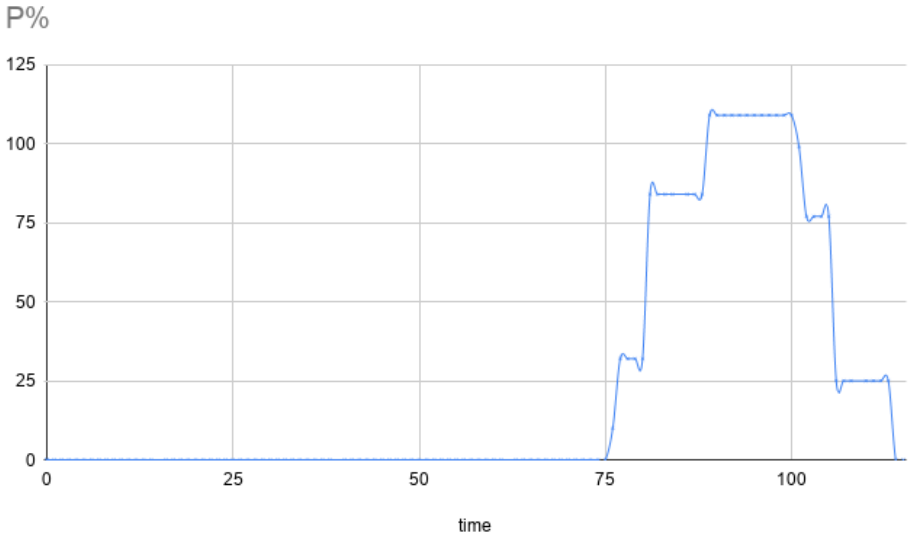




View2

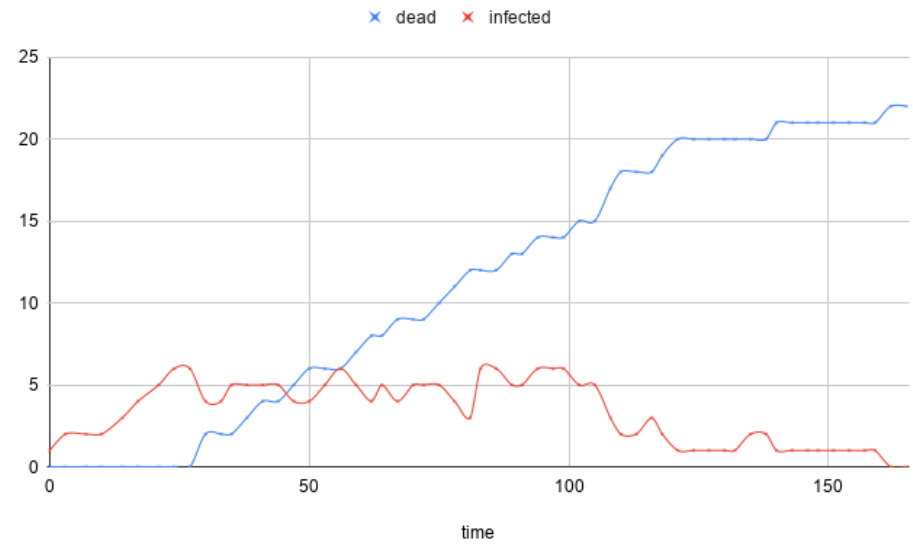
Infected : 0
Healthy : 195
Hospital : 0
Dead : 5
Mean Of D : 5.2
Percentage of M : 49.5
P : 0.0
Population : 200

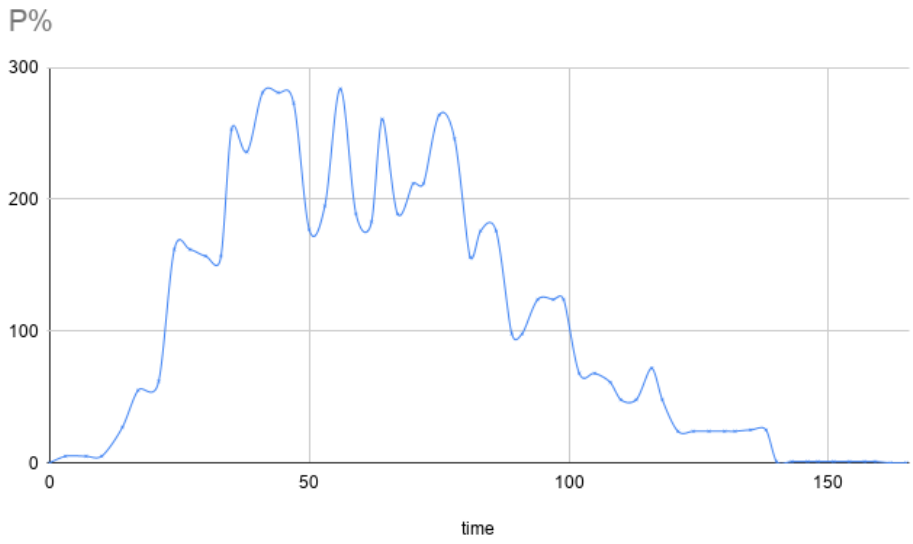




View2

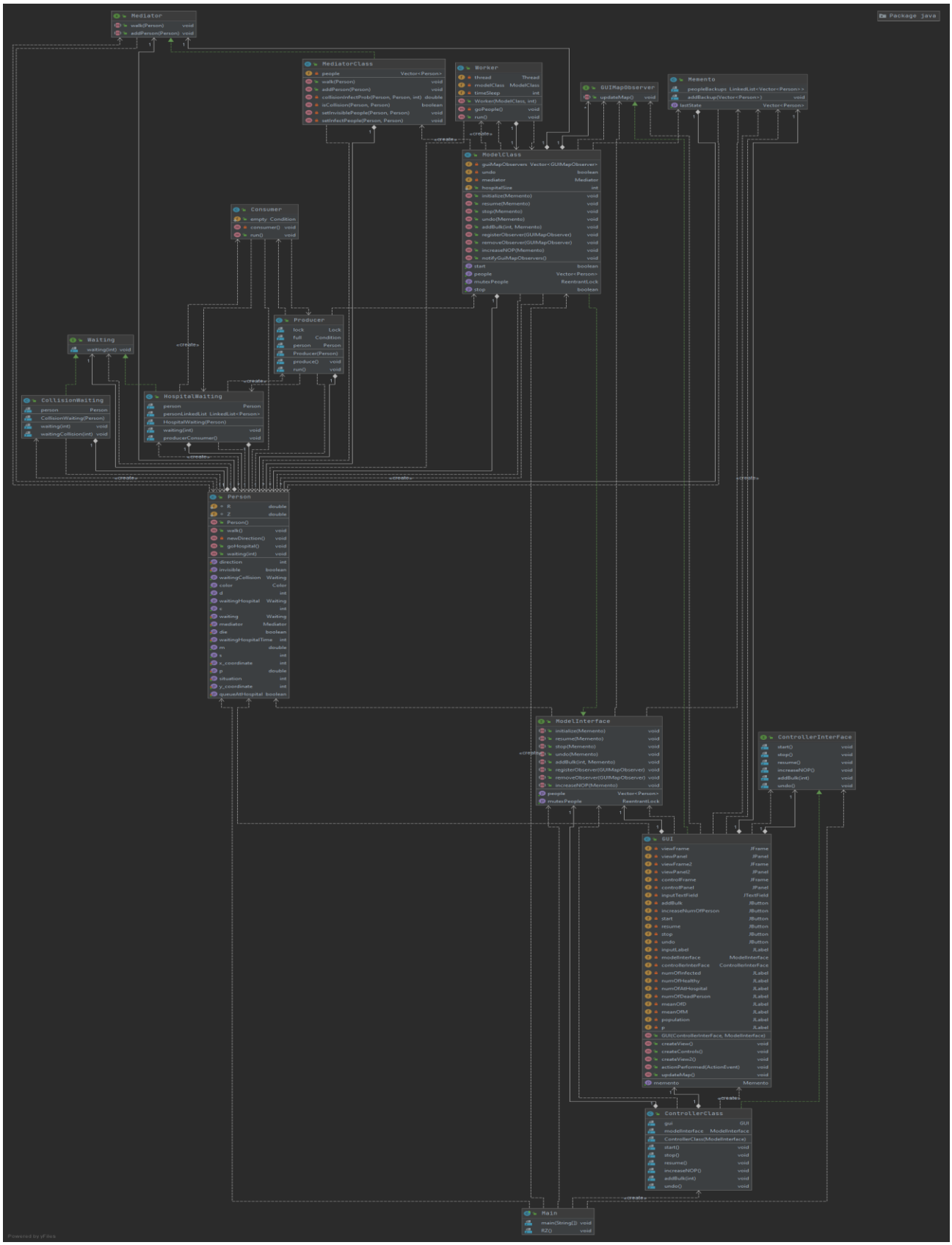
Infected : 0
Healthy : 1478
Hospital : 0
Dead : 22
Mean Of D : 4.946
Percentage of M : 50.13333333333333
P : 0.0
Population : 1500





Deaths increase when R and Z values increase. Since random D and M values are given, they are close to each other and their direct effect cannot be observed. However, when the population increased, a faster spread was observed. Therefore, the disease lasts longer.

Graphics are not covering. It has not even been fully interpreted.


Package Java
Mediator
walk(Person) void
addPerson(Person) void
MediatorClass
people Vector<Person>
walk(Person) void
addPerson(Person) void
collisionInfectProb(Person, Person, int) double
isCollision(Person, Person) boolean
setInvisiblePeople(Person, Person) void
setInfestPeople(Person, Person) void
Worker
thread Thread
modelClass ModelClass
timeSleep int
Worker(ModelClass, int)
goPeople() void
run() void
GUIMapObserver
updateMap() void
Memento
peopleBackups LinkedList<Vector<Person>>
addBackup(Vector<Person>) void
lastState Vector<Person>
ModelClass
guiMapObservers Vector<GUIMapObserver>
undo boolean
mediator Mediator
hospitalSize int
initialize(Memento) void
resume(Memento) void
stop(Memento) void
undo(Memento) void
addBulk(int, Memento) void
registerObserver(GUIMapObserver) void
removeObserver(GUIMapObserver) void
increaseNOP(Memento) void
notifyGuiMapObservers() void
start boolean
people Vector<Person>
mutexPeople ReentrantLock
stop boolean
Consumer
empty Condition
consumer() void
run() void
Producer
lock Lock
full Condition
person Person
Producer(Person)
produce() void
run() void
Waiting
waiting(int) void
CollisionWaiting
person Person
CollisionWaiting(Person)
waiting(int) void
waitingCollision(int) void
HospitalWaiting
person Person
personLinkedList LinkedList<Person>
HospitalWaiting(Person)
waiting(int) void
producerConsumer() void
Person
R double
Z double
Person()
walk() void
newDirection() void
goHospital() void
waiting(int) void
direction int
invisible boolean
waitingCollision Waiting
color Color
d int
waitingHospital Waiting
v int
waiting Waiting
mediator Mediator
die boolean
waitingHospitalTime int
m double
s int
x_coordinate int
p double
situation int
y_coordinate int
queueAtHospital boolean
ModelInterface
initialize(Memento) void
resume(Memento) void
stop(Memento) void
undo(Memento) void
addBulk(int, Memento) void
registerObserver(GUIMapObserver) void
removeObserver(GUIMapObserver) void
increaseNOP(Memento) void
people Vector<Person>
mutexPeople ReentrantLock
ControllerInterFace
start() void
stop() void
resume() void
increaseNOP() void
addBulk(int) void
undo() void
GUI
viewFrame JFrame
viewPanel JPanel
viewFrame2 JFrame
viewPanel2 JPanel
controlFrame JFrame
controlPanel JPanel
inputTextField JTextField
addBulk JButton
increaseNumOfPerson JButton
start JButton
resume JButton
stop JButton
undo JButton
inputLabel JLabel
modelInterface ModelInterface
controllerInterface ControllerInterface
numOfInfected JLabel
numOfHealthy JLabel
numOfAtHospital JLabel
numOfDeadPerson JLabel
meanOfD JLabel
meanOfM JLabel
population JLabel
p JLabel
GUI(ControllerInterface, ModelInterface)
createView() void
createControls() void
createView2() void
actionPerformed(ActionEvent) void
updateMap() void
memento Memento
ControllerClass
gui GUI
modelInterface ModelInterface
ControllerClass(ModelInterface)
start() void
stop() void
resume() void
increaseNOP() void
addBulk(int) void
undo() void
Main
main(String[]) void
RZ() void
«create»