Gebze Technical University

Computer Engineering Department

CSE443-Object Oriented Analysis and Design

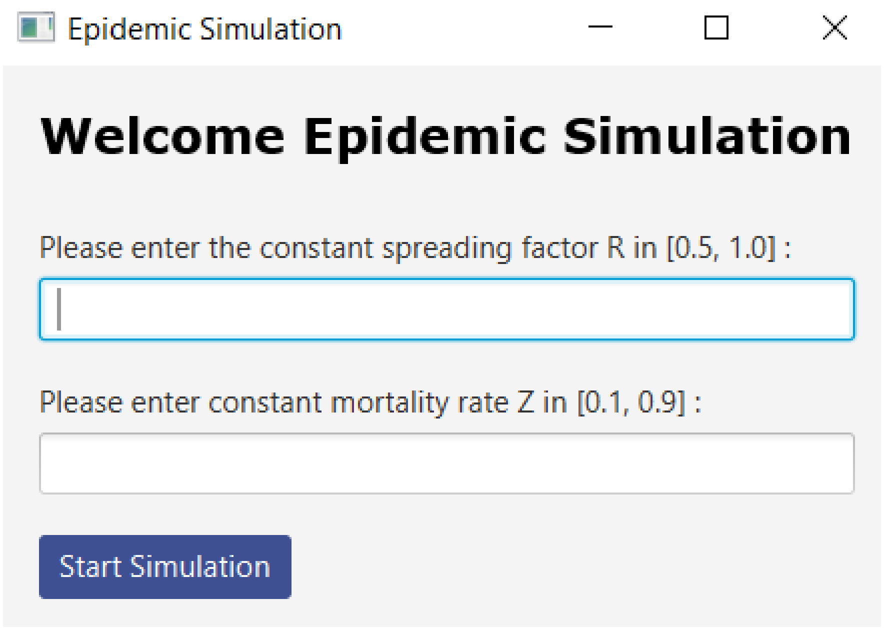
Fall 2020-2021

Final Project

Fatih Dural

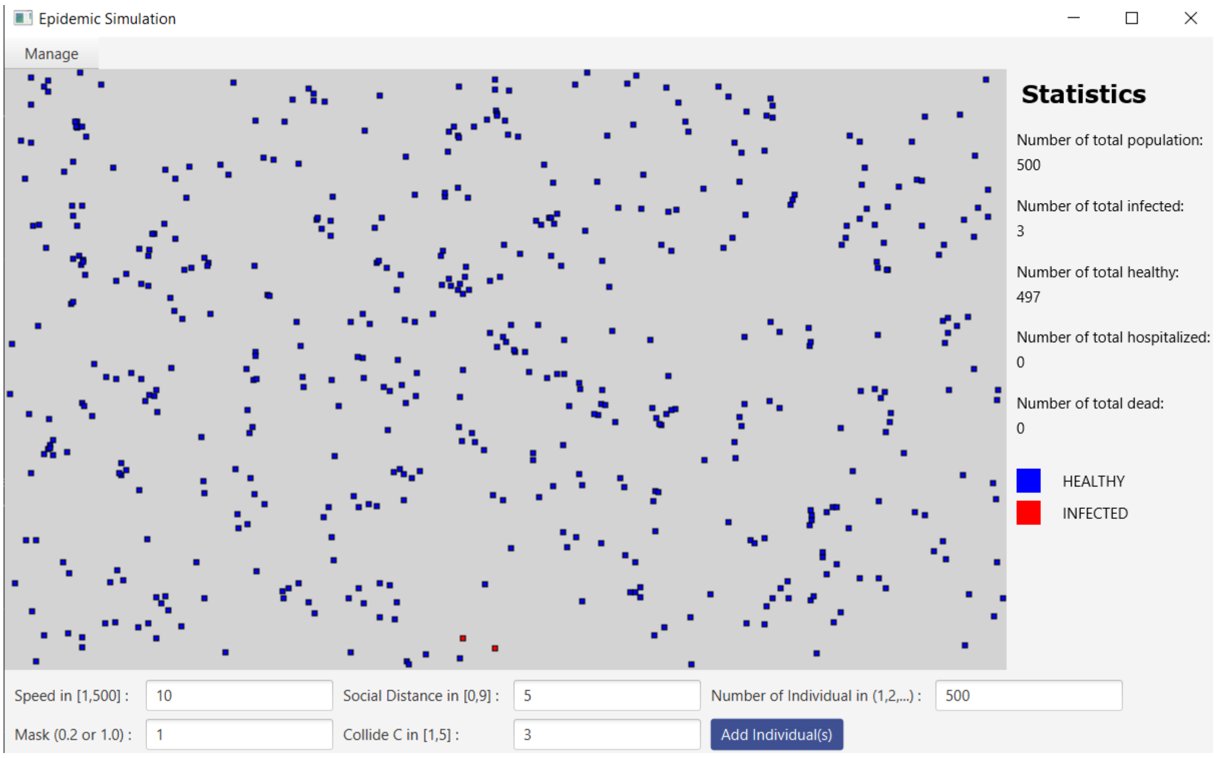
151044041

An **epidemic** was simulated with **Java FX** in this project. In a social community with many individuals, a random person is infected and spreads according to certain formulas.


Epidemic Simulation
Welcome Epidemic Simulation
Please enter the constant spreading factor R in [0.5, 1.0] :
Please enter constant mortality rate Z in [0.1, 0.9] :
Start Simulation

The first screen is the **welcome screen**. Before starting the simulation, constant spreading factor R and mortality rate Z **must be** entered according to certain intervals. If input is incorrect, user is warned and simulation starts.

The canvas on which the society moves is **1000x600** in size. Healthy individuals are colored **blue**, infected individuals are colored **red**. Each individual is represented **by 5x5 pixels**. One or more people can be added to the community from the bottom screen. Some information about the person to be added is **requested**. The screen on the right contains some data such as the number of people, the number of infected. In addition, simulation can be managed from the **top menu** (pause, resume). The first person or persons to enter are **randomly** infected. It should be kept in mind that the **simulation may** vary according to the **data entered** and there is **a lot of workload** in the background.



Statistics

Number of total population:
500

Number of total infected:
3

Number of total healthy:
497

Number of total hospitalized:
0

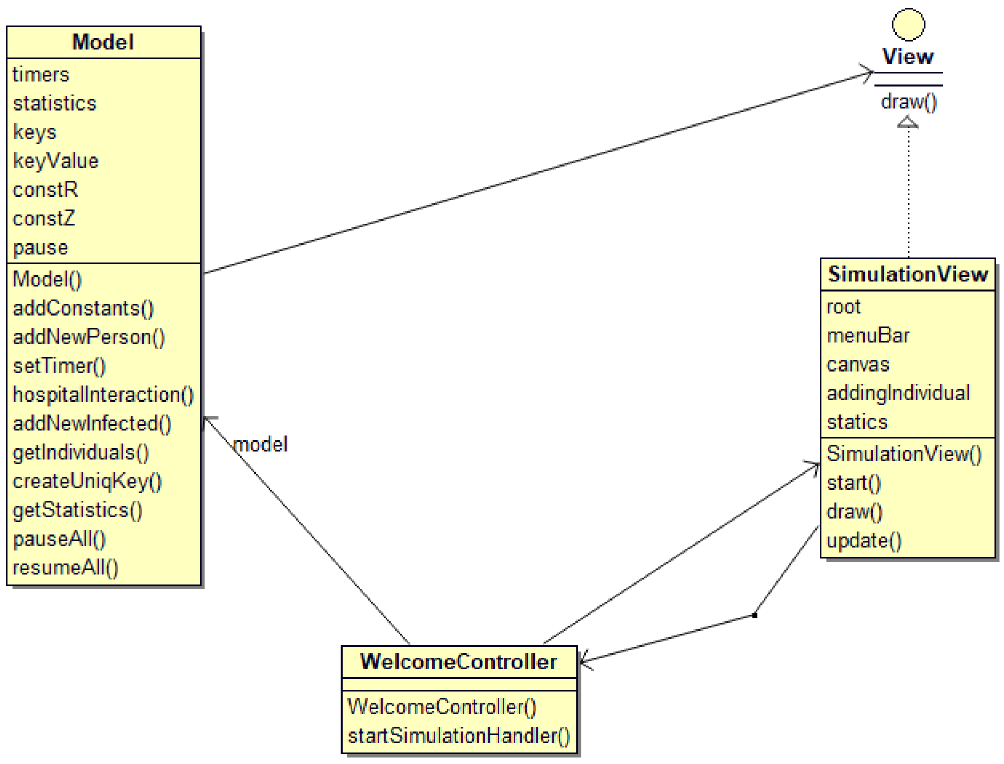
Number of total dead:
0

🟦 HEALTHY
🟥 INFECTED

Speed in [1,500] : 10
Social Distance in [0,9] : 5
Number of Individual in (1,2,...) : 500
Mask (0.2 or 1.0) : 1
Collide C in [1,5] : 3

General Architecture

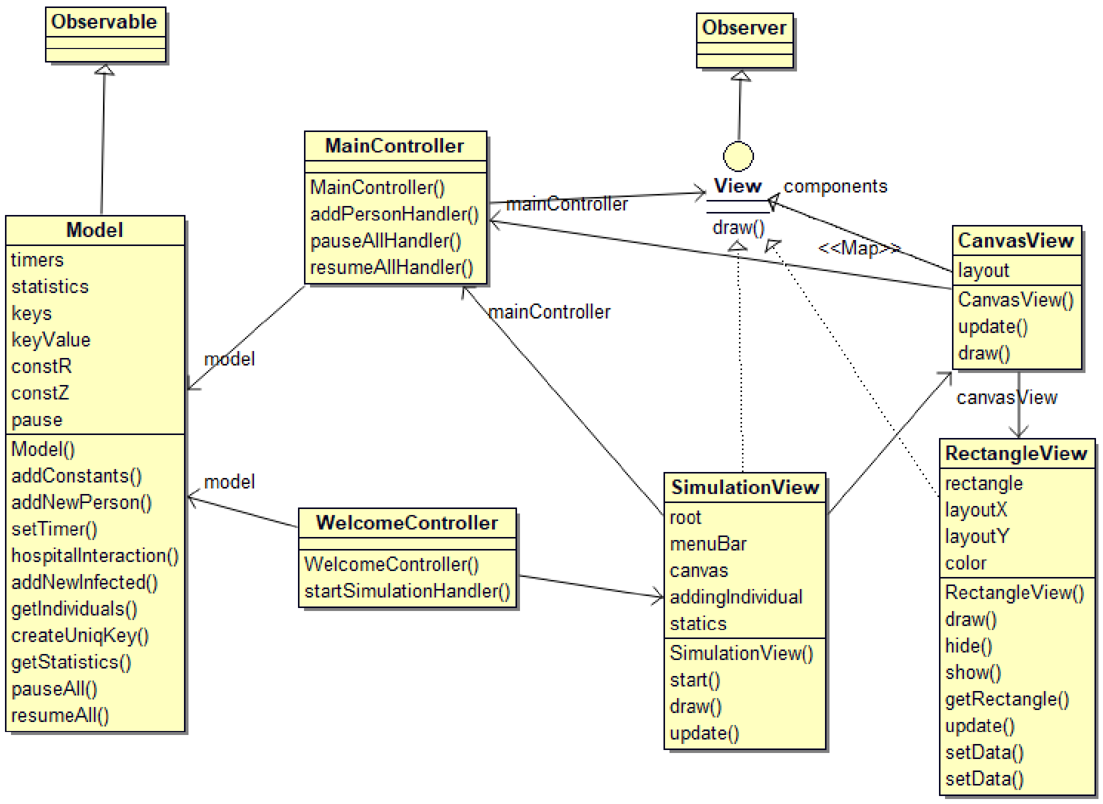
Compound design pattern was used in this simulation. In compound, a set of patterns work together. General architecture is made with MVC architecture, which is a good example of this. **Model-View-Controller** (MVC) consists of 3 layers and these layers work independently of each other. This provides great **flexibility and considerably** reduces maintenance costs.



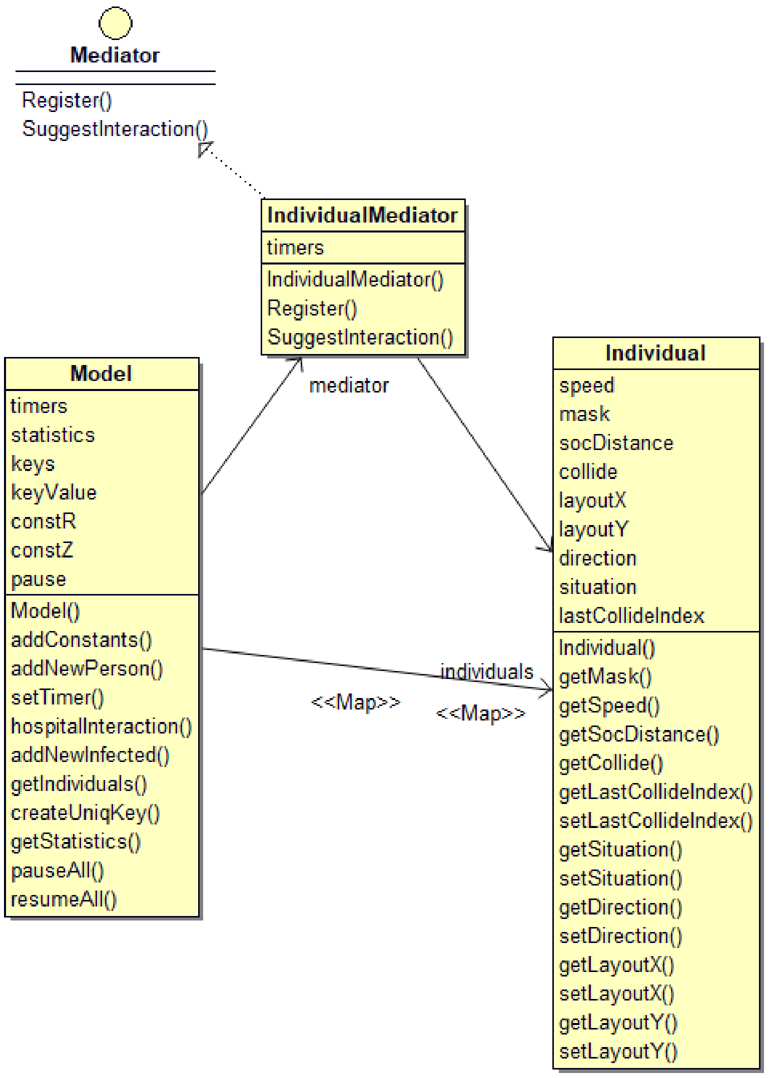
There is a **model-view-controller** structure that communicates with each other but is completely separated. Its views represent a general interface. **View interface** and sub views that implement it are available. Changes in the view are sent to the controllers and the **controller reports these changes** to the model. The model is where transactions are made and data is kept. Sends new stats to **views in case of change**.

Makes model state changes with the help of **Observer Design Pattern**. The model is observable. It notifies the observers who have registered with him of the **situation changes**.

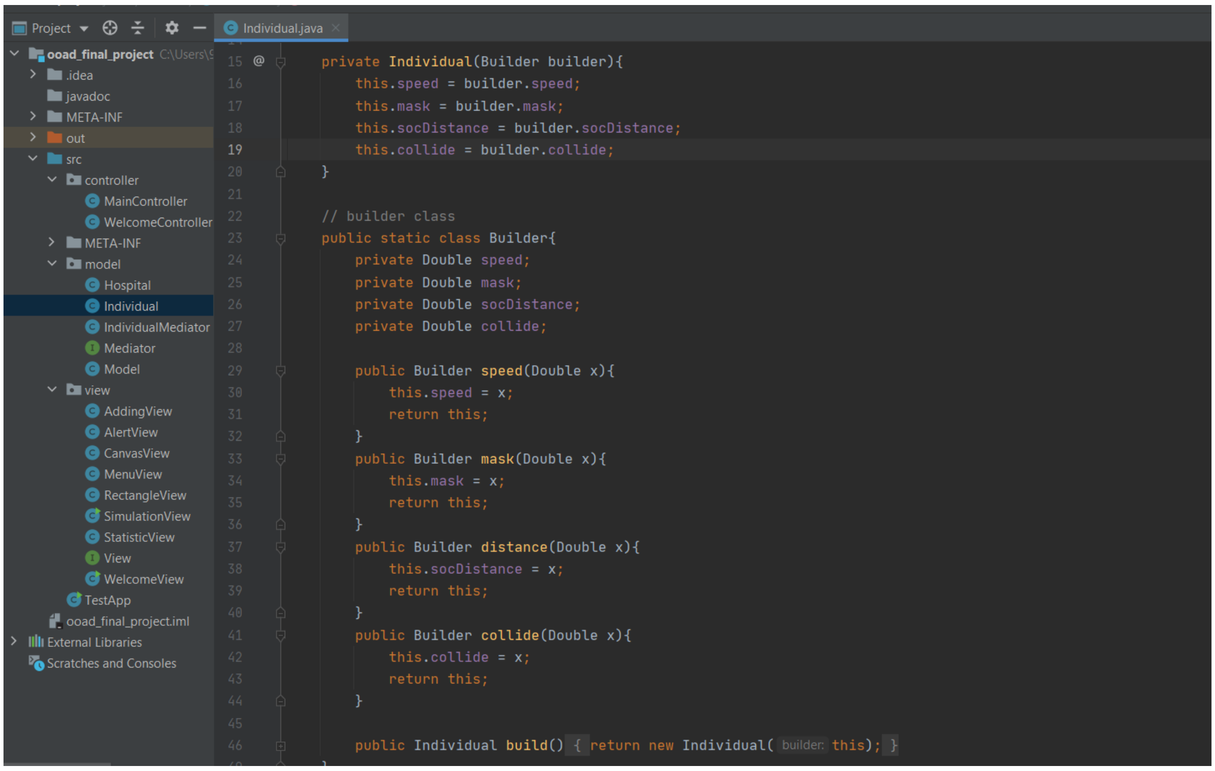
However, another subject is that it has a complex structure in views. In order to minimize this confusion and to add flexibility, **Composite Design Pattern** was used in Views. In fact, the simulation itself is a **view** in structures such as canvas and square that make up it. With Composite Design Pattern, all views **implement a top view interface**. In this way, view can be used in view. In addition, this structure is very **compatible with observer**. Model broadcasts only to **top view**. Forwards updates to its components in the top view. The model does not have to deal with **every component**.



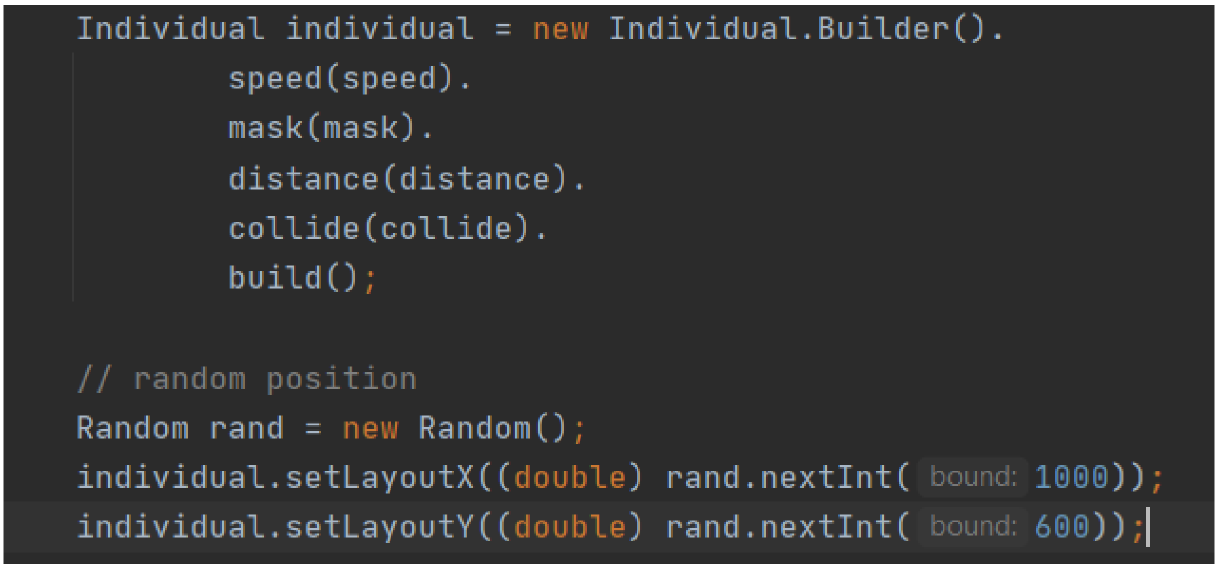
The model includes many **complex operations**. It uses a **mediator** to get out of this mess. Rather than examining all relationships between contacts, a new **contact location request** and mediator does analysis. Actually, this is like an airport tower. Rather than each plane communicating individually, an auxiliary tower guides them.



We mentioned that the model does many complex operations and uses the mediator class. The focus of these works is the **Individual class**. This class represents a person and his many characteristics. This class was created with **builder design pattern.** Builder design makes it **easier to create** people with different characteristics. Some personal features are available by default, some are added later. This variability is solved with the builder design pattern. Builder, a static inner class, helps object creation.

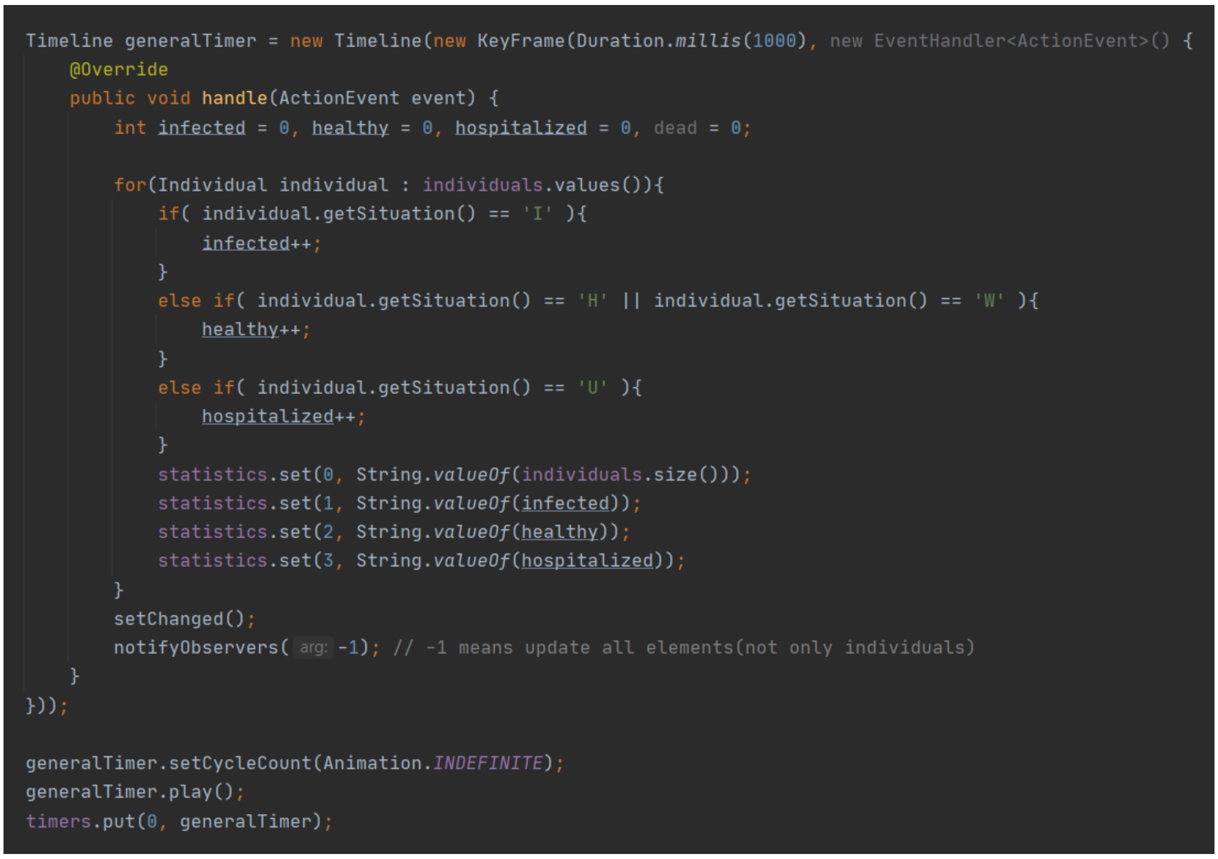


```java
    private Individual(Builder builder){
        this.speed = builder.speed;
        this.mask = builder.mask;
        this.socDistance = builder.socDistance;
        this.collide = builder.collide;
    }

    // builder class
    public static class Builder{
        private Double speed;
        private Double mask;
        private Double socDistance;
        private Double collide;

        public Builder speed(Double x){
            this.speed = x;
            return this;
        }
        public Builder mask(Double x){
            this.mask = x;
            return this;
        }
        public Builder distance(Double x){
            this.socDistance = x;
            return this;
        }
        public Builder collide(Double x){
            this.collide = x;
            return this;
        }

        public Individual build() { return new Individual( builder: this); }
```



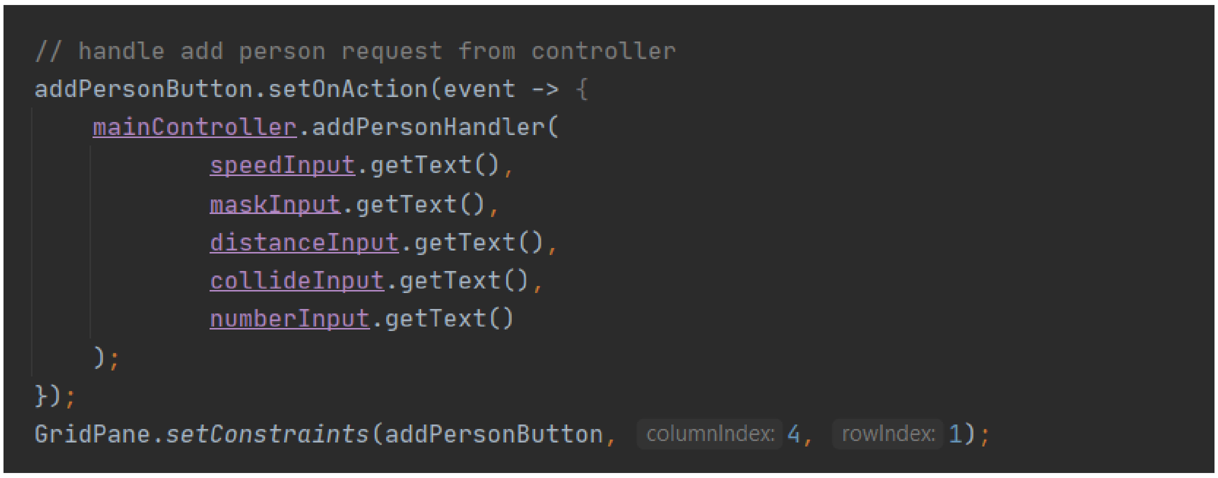
```java
Individual individual = new Individual.Builder().
        speed(speed).
        mask(mask).
        distance(distance).
        collide(collide).
        build();

// random position
Random rand = new Random();
individual.setLayoutX((double) rand.nextInt( bound: 1000));
individual.setLayoutY((double) rand.nextInt( bound: 600));
```

The simulation uses a **multi-thread structure** as an interface and as a process. While a general timer sends updates every second, **each added person is updated at their own pace**.



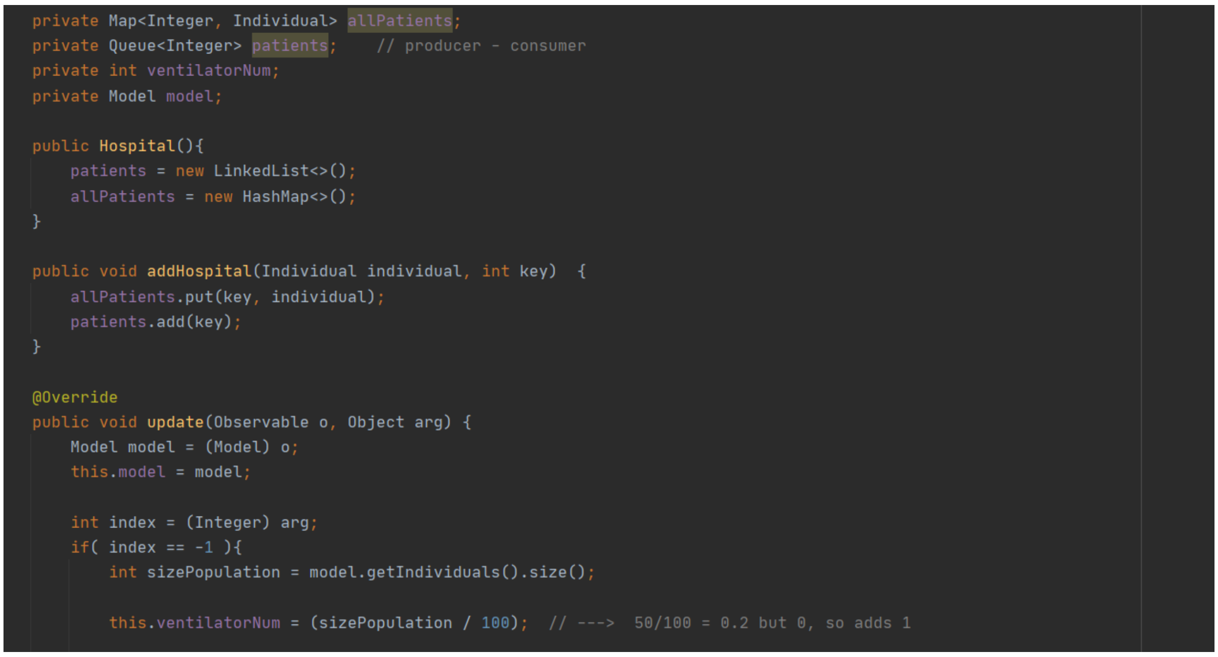
```java
Timeline generalTimer = new Timeline(new KeyFrame(Duration.millis(1000), new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        int infected = 0, healthy = 0, hospitalized = 0, dead = 0;

        for(Individual individual : individuals.values()){
            if( individual.getSituation() == 'I' ){
                infected++;
            }
            else if( individual.getSituation() == 'H' || individual.getSituation() == 'W' ){
                healthy++;
            }
            else if( individual.getSituation() == 'U' ){
                hospitalized++;
            }
            statistics.set(0, String.valueOf(individuals.size()));
            statistics.set(1, String.valueOf(infected));
            statistics.set(2, String.valueOf(healthy));
            statistics.set(3, String.valueOf(hospitalized));
        }
        setChanged();
        notifyObservers( arg: -1); // -1 means update all elements(not only individuals)
    }
}));

generalTimer.setCycleCount(Animation.INDEFINITE);
generalTimer.play();
timers.put(0, generalTimer);
```

The multi-thread list methods **offered by Java** also reinforce this. Requests are listened to, information is received, transmitted to controllers and processed in the model. All threads are kept in memory and the user has the right to **stop and continue** these threads at any time.



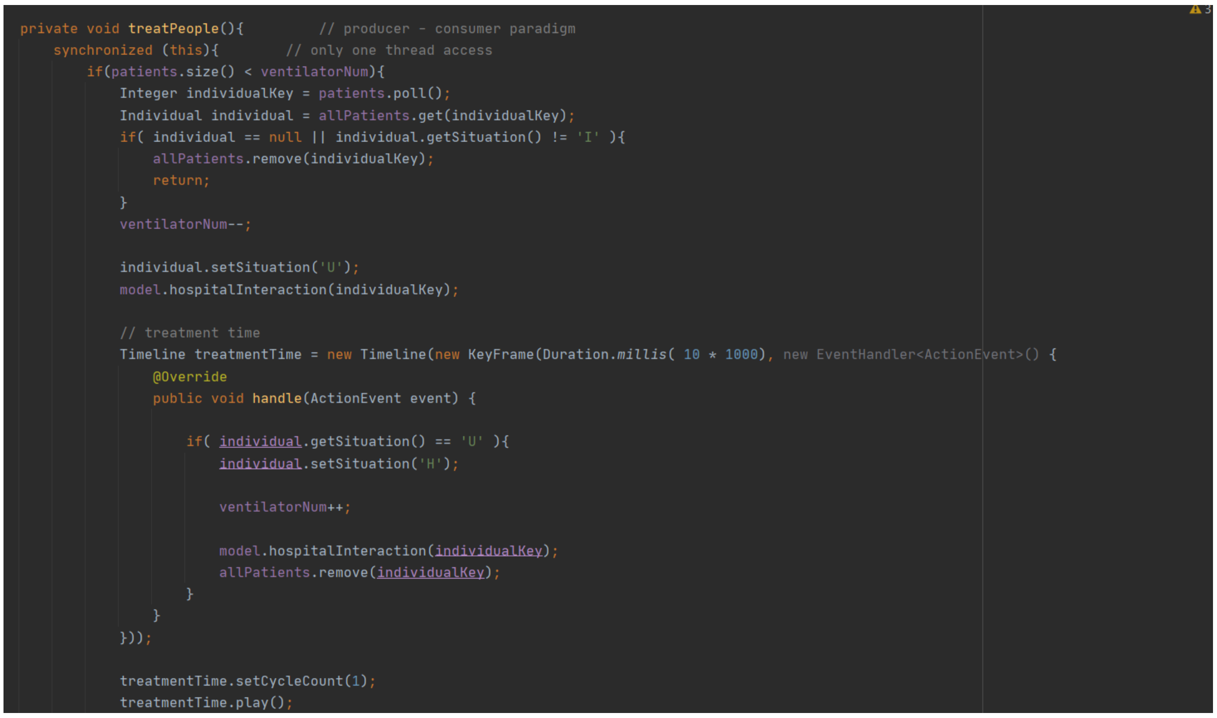
```java
// handle add person request from controller
addPersonButton.setOnAction(event -> {
    mainController.addPersonHandler(
            speedInput.getText(),
            maskInput.getText(),
            distanceInput.getText(),
            collideInput.getText(),
            numberInput.getText()
    );
});
GridPane.setConstraints(addPersonButton, columnIndex: 4, rowIndex: 1);
```

Some people who come into contact with infected people in the community become infect and go to the hospital. This structure is provided with the **producer-consumer paradigm.** The number of ventilators is limited and needs to be checked. Patients line up and wait for their turn.



```java
private Map<Integer, Individual> allPatients;
private Queue<Integer> patients;    // producer - consumer
private int ventilatorNum;
private Model model;

public Hospital(){
    patients = new LinkedList<>();
    allPatients = new HashMap<>();
}

public void addHospital(Individual individual, int key)  {
    allPatients.put(key, individual);
    patients.add(key);
}

@Override
public void update(Observable o, Object arg) {
    Model model = (Model) o;
    this.model = model;

    int index = (Integer) arg;
    if( index == -1 ){
        int sizePopulation = model.getIndividuals().size();

        this.ventilatorNum = (sizePopulation / 100);  // --->  50/100 = 0.2 but 0, so adds 1
```

Synchronous optimization **solves incorrect access** and queue issues. The person who comes to this function starts the process, the others wait. In the time it takes to **complete the treatment**, they can make a request and **their order** is maintained.



```java
private void treatPeople(){         // producer - consumer paradigm
    synchronized (this){        // only one thread access
        if(patients.size() < ventilatorNum){
            Integer individualKey = patients.poll();
            Individual individual = allPatients.get(individualKey);
            if( individual == null || individual.getSituation() != 'I' ){
                allPatients.remove(individualKey);
                return;
            }
            ventilatorNum--;

            individual.setSituation('U');
            model.hospitalInteraction(individualKey);

            // treatment time
            Timeline treatmentTime = new Timeline(new KeyFrame(Duration.millis( 10 * 1000), new EventHandler<ActionEvent>() {
                @Override
                public void handle(ActionEvent event) {

                    if( individual.getSituation() == 'U' ){
                        individual.setSituation('H');

                        ventilatorNum++;

                        model.hospitalInteraction(individualKey);
                        allPatients.remove(individualKey);
                    }
                }
            }));

            treatmentTime.setCycleCount(1);
            treatmentTime.play();
```