Queue Simulator

Fejér Alpár

Group 30424

Objective

The goal of this project is to develop an application that simulates queues in a store with multiple registers .

The project works with a graphical user interface , which allows the user to input the data necessary for the simulation . It also provides the user with a representation of the queues in the registers .

Problem analysis

The main problem is figuring out how to make the queues work in a concurrent fashion . Threads are used to make this happen .

The graphical user interface is an important component of the project , as the user only needs to see the progress of the queues . It shows when a customer arrives at a register queue , the time of his / her arrival , the time he / she needs to scan his / her own products . The graphical user interface also shows when a register opens or closes due to the number of customers in queue .

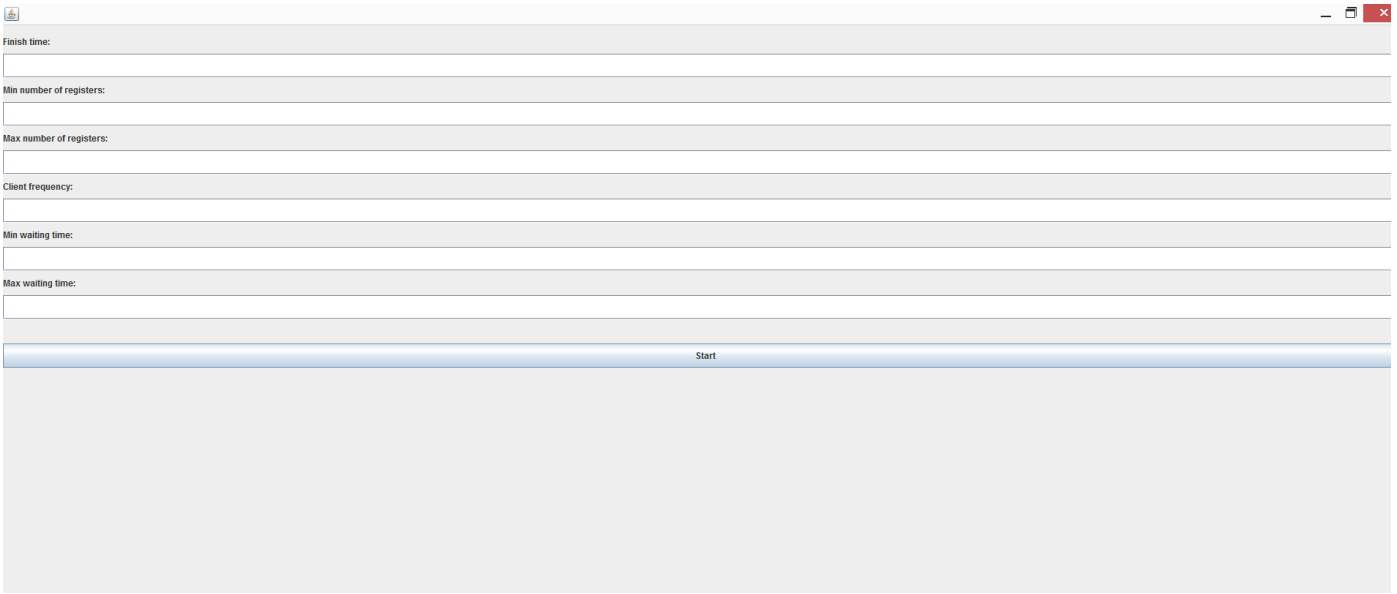
Modeling

The problem is modeled after a scheduler having servers that execute tasks of its own .

The tasks are the customers , the servers are the registers in the store and the scheduler is the schedule of the cashiers . To resemble real life , cashiers only open their registers if the number of the clients at other registers grow . The customers always choose the registers with the shortest . There is a minimum and maximum number of registers that can be open at any time .

Because the registers are modeled separately , there would be no point in uniquely defining the tasks , but the store has to store information about the daily traffic .

User Interface



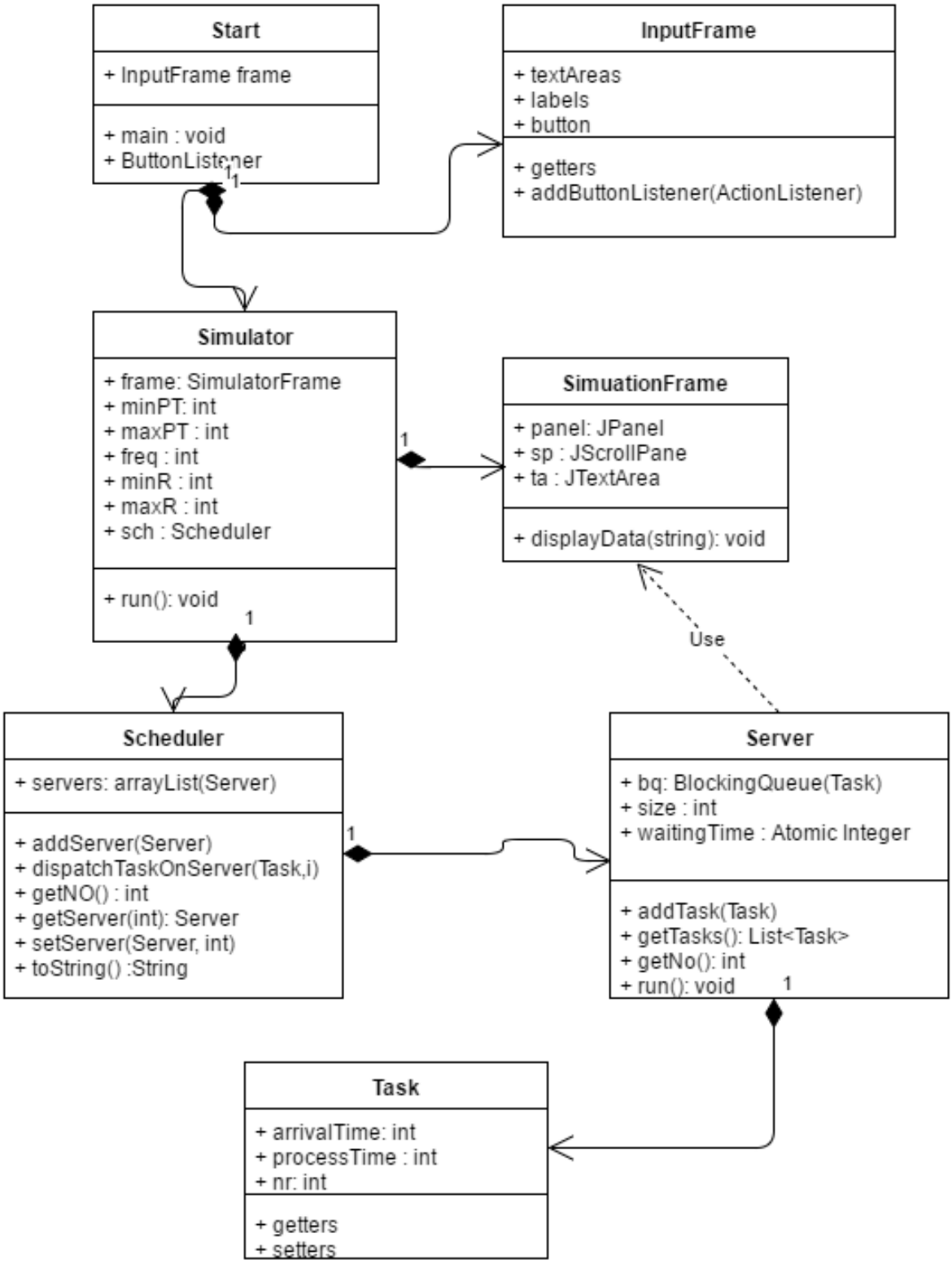
The graphical user interface is made to be simple , easy to use and to understand , and with as few distractions as possible . First , an input panel is opened , that the user can use to give the parameters of the simulation . Once the user inputs the parameters , the simulation can be started with the press of a button .

The input panel consists only of JTextAreas for the user to input the parameters , labels for the user to understand the panel , and the starting button , with an abstract ActionListener class .

```
void addButtonListener(ActionListener bal) {

        start.addActionListener(bal);
    }

class ButtonListener implements ActionListener{


            @Override
            public void actionPerformed(ActionEvent e) {
                  Simulator sim = new
Simulator(frame.getFT(),frame.getFreq(),frame.getMin(),frame.getMax(),frame.g
etMinReg(),frame.getMaxReg());
                  Thread th = new Thread(sim);
                  th.start();

            }
```

Classes

This section contains the description of the used classes in more detail .



Task

This class is for modeling the customers in the store . It contains information about the customers , namely the time of arrival , the time required for the customers to scan their products and the number of the customer .

```
public class Task {

	private int arrivalTime;
	private int processTime;
	private int nr;
	public Task(int aT, int pT,int nr){
		this.arrivalTime = aT;
		this.processTime = pT;
		this.setNr(nr);
	}
```

Server

This class if for modeling the registers in the store . It implemets Runnable , as it needs to run in its own thread . It holds a queue of Tasks , modeling real world lines at a register .

```
public class Server implements Runnable {

	private BlockingQueue<Task> bq;
	private AtomicInteger waitingTime;
	private int size;
	public Server() {
		bq = new LinkedBlockingQueue<>();
		waitingTime = new AtomicInteger(0);
	}
```

Scheduler

This class is for modeling the schedule of the cashiers in the store . It opens the minimum number of registers at start , and can open more registers if needed . It also starts the threads on the servers when creating them .

```
public class Scheduler {

	private List<Server> servers;//list

	public Scheduler(int n){
		servers = new ArrayList<Server>(n);
		for (int i=0; i<n; i++) servers.add(new Server());
		for (Server s : servers){
			Thread th = new Thread(s);
			th.start();
		}
	}

	public void addServer(){
		Server s = new Server();
```

```
            servers.add(s);
            Thread th = new Thread(s);
            th.start();
      }

      public void dispatchTaskOnSever(Task t, int i){
            Server s = servers.get(i);
            s.addTask(t);
      }
```

Main

The main (start) class holds the input frame from which it gets the parameters for the simulations , and the ActionListener for the start button that starts the simulation .

```
public class Start {
      private InputFrame frame;
      public Start(){
            frame = new InputFrame();
            frame.addButtonListener(new ButtonListener());

      }
      public static void main(String[] args){
            new Start();

      }
      class ButtonListener implements ActionListener{


            @Override
            public void actionPerformed(ActionEvent e) {
                  Simulator sim = new
Simulator(frame.getFT(),frame.getFreq(),frame.getMin(),frame.getMax(),frame.g
etMinReg(),frame.getMaxReg());
                  Thread th = new Thread(sim);
                  th.start();

            }
      }
}
```

Simulation

```
public void run() {
            // TODO Auto-generated method stub
            int currentTime = 0;
            int nr = 1;
            while (currentTime < finishedTime){
                  currentTime++;


                  if ((currentTime % freq == 0)){
                        int i = 0;
                        for(int j = 1; j<sch.getNo(); j++) if
(sch.getServer(j).getNo()<sch.getServer(i).getNo()) i=j;
```

```
                        int processTime =
(int)(Math.random()*(maxProcessTime-minProcessTime)+minProcessTime);
                        Task t = new Task(currentTime,processTime,nr);
                        nr++;
                        if (sch.getServer(i).getNo()>3 &&
sch.getNo()<maxReg){
                              sch.addServer();
                              i++;
                              frame.displayData("Register "+sch.getNo()+"
opened.\n");
                              sch.dispatchTaskOnSever(t,i);
                        }
                        else{
                              sch.dispatchTaskOnSever(t,i);
                        }
                        i++;

                        frame.displayData("Client "+t.getNr()+" arrived to
register "+i+". Arrival time: "+t.getArrivalTime()+". Process time:
"+t.getProcessTime()+".\n");

                  }
                  try {
                        Thread.sleep(1000);
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }


            }

      }
```

The simulation class contains all the logic for the problem . It adds a new customer based on the frequency given by the user to the register with the shortest line and opens registers if needed and possible . It also closes registers if they are not in use .

GUI

This holds the user interface . It helps the user use the program without understanding it . It is a crucial part of the project .

The user interface is extremely easy to use , with no unnecessary buttons or complications.

Bibliography

http://stackoverflow.com/