Third assignment
Programming Techniques

Student: Nagy Hanna

Group: 30421

1. Objective

Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients' waiting time.
Description
Queues are commonly seen both in real world and in the models. The main objective of a queue is to provide a place for a "client" to wait before receiving a "service". The management of queue based systems is interested in minimizing the time amount its "clients" are waiting in queues. One way to minimize the waiting time is to add more servers, i.e. more queues in the system (each queue is considered as having an associated processor) but this approach increases the costs of the supplier. When a new server is added the waiting clients will be evenly distributed to all current available queues.
The application should simulate a series of clients arriving for service, entering queues, waiting, being served and finally leaving the queue. It tracks the time the clients spend waiting in queues and outputs the average waiting time. To calculate waiting time we need to know the arrival time, finish time and service time. The arrival time and the service time depend on the individual clients – when they show up and how much service they need. The finish time depends on the number of queues, the number of other clients in the queue and their service needs.

Input data:
- Minimum and maximum interval of arriving time between clients;
- Minimum and maximum service time;
- Number of queues;
- Simulation interval;
- Other information you may consider necessary;

Minimal output:
- Average of waiting time, service time and empty queue time for 1, 2 and 3 queues for the simulation interval and for a specified interval;
- Log of events and main system data;
- Queue evolution;
- Peak hour for the simulation interval;

As the assignment requirements says this problem is seen in everyday life, even in the most common places like in the supermarkets when we are waiting our turn to be served in the queue.

The aim of this assignment is to simulate such a behavioral. To do this, we used the Thread theory , which means that we could simulate several queues being used my the customers in real time and in the same time as well.

2. Scenes. Analysis. Modelling. Cases of utilization.

We have a file, „input.txt" where we have defined the minimum Processing time which could be named as minimum service time and we have the maximum Processing time which could be said as maximum Processing time as well. We have also defined the number of queues we want to see and use in the simulation.

In our example, the minimum service time (processing time) is 1 and the maximum processing time is 12, the number of queues is 5.
These queues will be used, but just in the case when it is all needed. We said that a queue is almost full when it has more then 3 persons (tasks) waiting to be served.

3. Classes. Structure.

I have a Task class which behaves like a person which is shopping in the supermarket, or in our case a Task which will be given to a Server class.

```
public class Task {

	private int arrival Time;
	private int process Time;

	public Task( int current Time, int processingTime ) {
		this . arrivalTime = currentTime;
		this . processTime = processingTime;
	}

	public int getArrivalTime() {
		return arrivalTime;
	}

	public void setArrivalTime(int arrivalTime) {
		this . arrivalTime = arrivalTime;
	}

	public int getProcessTime() {
		return processTime;
	}

	public void setProcessTime(int processTime) {
		this . processTime = processTime;
	}

	public String toString() {
		return " arrival " + String . valueOf( arrivalTime ) + "
process " + String.valueOf( processTime );
	}
}
```

This class is just a model, as we can see, it has almost just getters and setters. Every task has an arrival time which represents the arrival time when it is given to a queue where it has to wait to be served or to be processed by the server. The toString method is just for the displaying in a right manner on the Jlists.

The class **Server** is similar to a check out in a supermarket. So a Server object gets times to time some tasks which will be scheduled in a right way.

```
import java . util . LinkedList;
import java . util . concurrent . BlockingQueue;
import java . util . concurrent . LinkedBlockingQueue;
import java . util . concurrent . atomic.AtomicInteger;
```

```
public class Server implements Runnable {

	private BlockingQueue<Task> blockingQueue;
	private AtomicInteger waitingTime;

	public Server( ) {
		blockingQueue = new LinkedBlockingQueue<Task>();
		waitingTime = new AtomicInteger(0);
	}

	@Override
	public void run( ) {

		while ( true ) {
			try {

				Task currentTask = blockingQueue.take();
				Thread.sleep(currentTask.getProcessTime() * 1000);
				this.waitingTime.addAndGet( (-1) *
currentTask.getProcessTime( ) );

			} catch (InterruptedException e1) {
				e1.printStackTrace( );
			}

		}

	}

	public void addTask( Task t ) {

		blockingQueue.add( t ); // put
		waitingTime.addAndGet(t.getProcessTime());
	}

	public Task[ ] getTasks( ) {

		Task[ ] tasks = new Task[blockingQueue.size( ) ];
		blockingQueue.toArray( tasks );
		return tasks;
	}
	public AtomicInteger getWaitingTime( ) {
		return waitingTime;
	}

}
```

This class has a method addTask(task t) which will be used by the Scheduler object which will compute the right place where the task to be placed in order to not to have heavy loading in one of the queues while others are empty.
The method getTasks is used is in the Simulator class where it has the role to getTasks for a specific Server. In other words, it gets back a list (array = techically) which consists of the current tasks in that queue of the Server from which was called in the Simualtor class.

Scheduler class.
This class has the role of schedule by a certain computation and logic the right place of the task which is going to be given to a certain Server.

```
public class Scheduler {

	private List<Server> servers;
	private List<Thread> threads;
	private Reader reader;
	private int nrOfServers;
	private int sumWaitingTime;
	private int sumAllWaitingTime = 0;
	private int howManyTasks = 0;
	private int peakHourMaxWT = Integer.MIN_VALUE;
	// private List<Integer> peakHours;
	private int time = 0;

	public Scheduler() {
		reader = new Reader ( );
		// peakHours = new ArrayList < Integer > ( );
		nrOfServers = reader.getNrOfServers ( );

		servers = new ArrayList < Server > ( );
		threads = new ArrayList < Thread > ( );

		for (int i = 0; i < nrOfServers; i++) {
			servers.add( new Server ( ) );
			Thread thread = new Thread ( servers.get (i ) );
			threads.add ( thread  ) ;
		}

		for ( Thread thread : threads ) {
			thread.start ( ) ;
		}
	}

	public int min( int[] size ) {
		int i = 0 ;
		int j = 0 ;
		int min = Integer.MAX_VALUE;
		while (i < size.length ) {
			if (min > size [  i ] ) {
				min = size [ i ] ;
				j = i ;
			}
			i++ ;
```

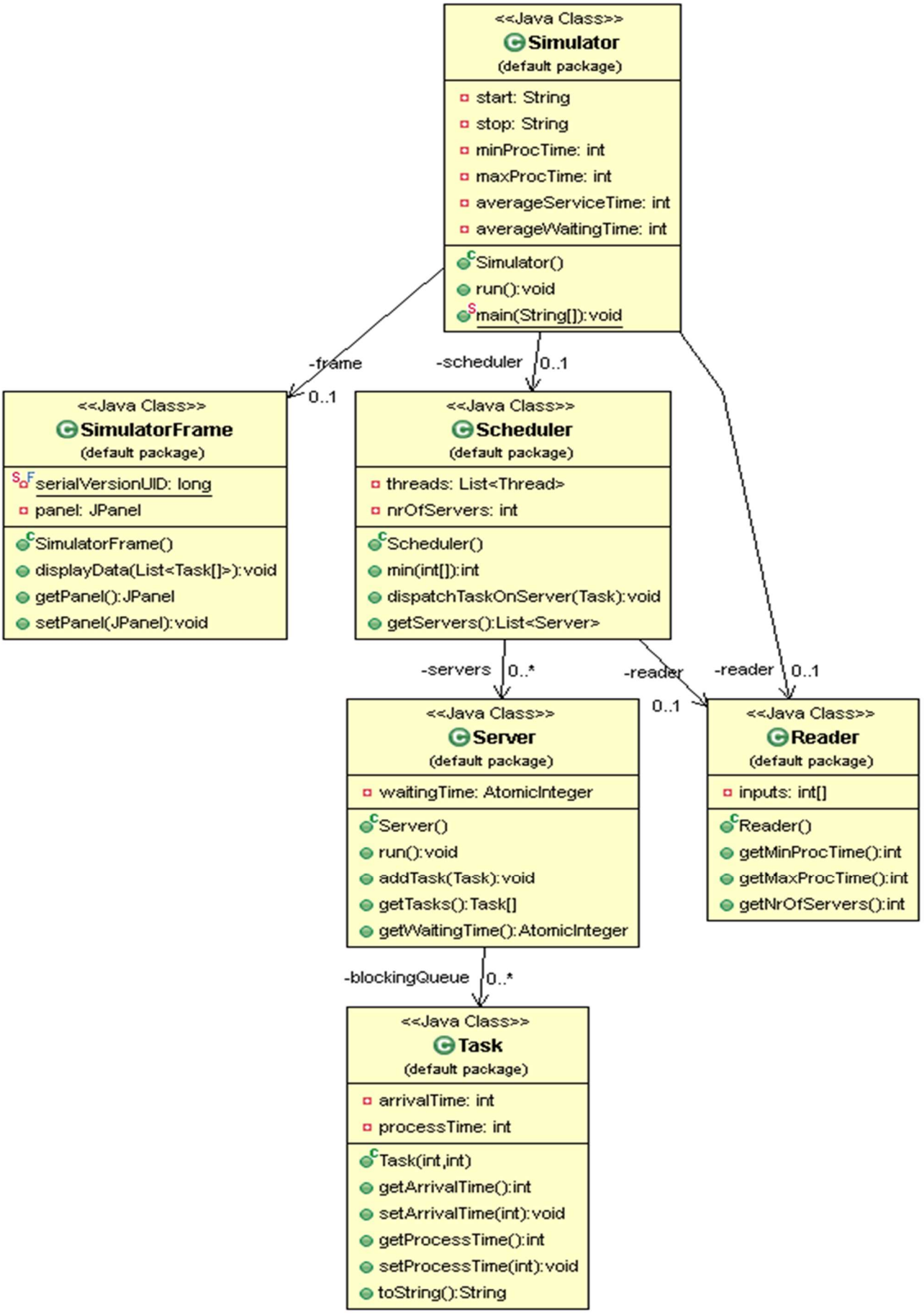
```
		}
		return j ;


	public void dispatchTaskOnServer(Task t, int currentTime) {

		int[] sizes = new int[servers.size()];
		for (int i = 0; i < servers.size(); i++) {
			sizes[i] = servers.get(i).getTasks().length;
		}
		int i = 0;
		howManyTasks++;
		boolean isAdded = false;
		while (!isAdded && i < servers.size()) {

			if (sizes[i] <= 2) {

				int waitingTime = calculateWaitingTimePerOneQueue(i,
sizes[i]);
				if (peakHourMaxWT < waitingTime) {
					peakHourMaxWT = waitingTime;
					setTime(currentTime);
				}
				System.out.println("currentTimein Sch:" + currentTime);

				servers.get(i).addTask(t);
				System.out.println("task added in the queue: " + i);
				isAdded = true;
				break;
			}
			if (i == servers.size() - 1) {

				int waitingTime = calculateWaitingTimePerOneQueue(i,
sizes[i]);

				if (peakHourMaxWT < waitingTime) {
					peakHourMaxWT = waitingTime;
					setTime(currentTime);
				}
				servers.get(min(sizes)).addTask(t);
				System.out.println("task added in the queue: " +
min(sizes));

			}

			i++;
		}
```

```
	}

	public int calculateAverageWaitingTime() {
		return sumAllWaitingTime / howManyTasks;
	}

	public int calculateWaitingTimePerOneQueue(int i, int size) {

		sumWaitingTime = 0;

		Task[] tasks = new Task[size];
		tasks = servers.get(i).getTasks();

		int j = 0;
		while (j < tasks.length) {
			sumWaitingTime += tasks[j].getProcessTime();
			sumAllWaitingTime += sumWaitingTime;
			j++;
		}
		System.out.println("waiting time " + sumWaitingTime + " for queue: " +
i);
		return sumWaitingTime;
}
```

This class it can be said to be the core of the application, because the main role of the application is to schedule every task depending on the loading of the servers, the waiting times of the tasks and the size of the queues or the number of the tasks which are present in that certain Server.

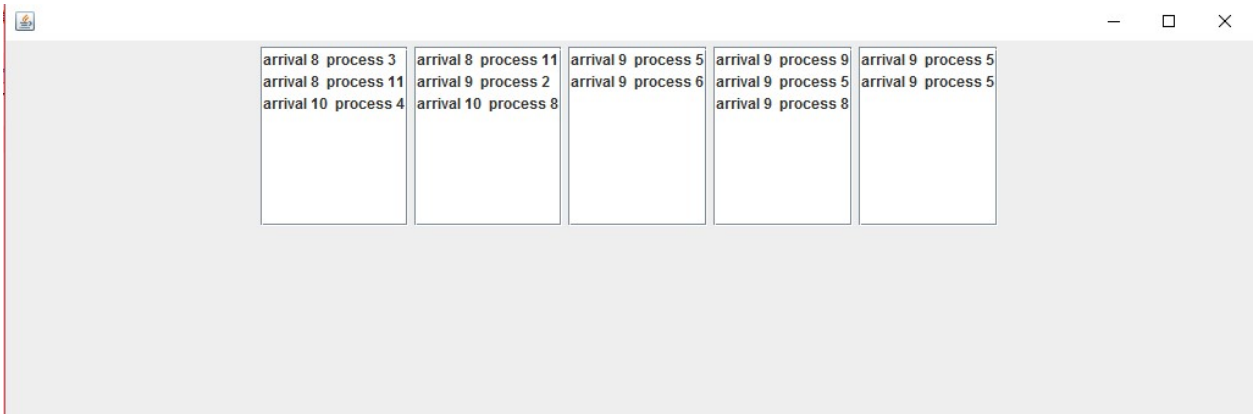
The Scheduler class has a method named:

public void dispatchTaskOnServer(Task t, int currentTime) which computer the the right place (the right server where the task should be inserted). It follows the idea of a queue of a certain server is considered almost full when it has more than 3 elements in this case a new Server is opened.


<<Java Class>>
Simulator
(default package)
start: String
stop: String
minProcTime: int
maxProcTime: int
averageServiceTime: int
averageWaitingTime: int
Simulator()
run():void
main(String[]):void
-frame
0..1
-scheduler
0..1
<<Java Class>>
SimulatorFrame
(default package)
serialVersionUID: long
panel: JPanel
SimulatorFrame()
displayData(List<Task[]>):void
getPanel():JPanel
setPanel(JPanel):void
<<Java Class>>
Scheduler
(default package)
threads: List<Thread>
nrOfServers: int
Scheduler()
min(int[]):int
dispatchTaskOnServer(Task):void
getServers():List<Server>
-servers
0..*
-reader
0..1
-reader
0..1
<<Java Class>>
Server
(default package)
waitingTime: AtomicInteger
Server()
run():void
addTask(Task):void
getTasks():Task[]
getWaitingTime():AtomicInteger
<<Java Class>>
Reader
(default package)
inputs: int[]
Reader()
getMinProcTime():int
getMaxProcTime():int
getNrOfServers():int
-blockingQueue
0..*
<<Java Class>>
Task
(default package)
arrivalTime: int
processTime: int
Task(int,int)
getArrivalTime():int
setArrivalTime(int):void
getProcessTime():int
setProcessTime(int):void
toString():String

4. Simualtion

We have a class named Simulator which creates or technically said it intantiate the Simulator frame where we can display the queues in real time.



Here can be seen the graphical user interface provided and implemented by this application. The Simulator class instantiate the SimulatorFrame class which is actually this frame/ window with a panel where we add as many Jlists as we needs (as many servers and threads we have).

5.Conclusions

Achieving such a program may be hard since we had to synchronize with threads.

For a better performance there should be implemented all cases where exceptions can occur and the application stops working due to an error made by the user.

Another thing that could be improved is the display so that it would be more interactive the displaying of the queues simulation.

6.References

http://www.javabeginner.com/learn-java/java-threads-tutorial

http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html

http://javabeginnerstutorial.com/core-java-tutorial/java-thread-tutorial/

http://stackoverflow.com/