Assignment 2 documentation
Queue based systems simulator

Litan Virgil

Group 30425

TA: Claudia Pop

Contents

1. Functional Requirements

The application should simulate a series of tasks arriving to a server, entering queues, waiting and finally being computed. It tracks the time the tasks spend waiting in queues and outputs the average waiting time on each server. The arrival time and the service time depend on the individual clients – when they show up and how much service they need. The application should allow customization of the simulation parameter and display real-time evolution of the tasks.

Also, at the end of the simulation a set of reports should be presented in order to be analyzed. While running the application should also output logs.

An aspect of paramount importance is implementing the simulation by means of multi-threading.

2. Objectives

Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients' waiting time.

2.1 Main objectives

Developing and multi - threaded application with a real – time GUI (General User Interface) that would illustrate the evolution of the system.

2.2 Secondary objectives

Highly customizable simulation parameters which are set at the beginning of the simulation run are a very useful feature because only this way the behavior of the queue based system can be fully understood.

Another important aspect of the application is designing a comprehensible set of logs that would allow an advanced user to fully understand the mechanisms that underlie the dispatching and executing process.

Another objective would be designing the application such a way that it could be easily further developed by separating the models, the general user interface and the main controllers.

3. Problem analysis

The main use case is simulating multiple servers which receive multiple tasks that are dispatched according to simple predefined policies. The main actors being the user, the simulation manager, task generator, dispatcher and finally the server. After the users sets the desired simulation parameters he has to press the " run simulation " button in order to start the simulation. After this, the simulation manager sets up all the components creating the tasks array list that has to be processed, creating the dispatcher and configuring all the parameters that have been previously chosen by the user. Further, the general user interface that will display the real - time evolution is initiated and set to refresh periodically. The simulation manager sends, at random time intervals, tasks to the dispatcher that will assign each task to a server according to a user-chosen strategy. In the case that every server is loaded to a given threshold a new server will be launched. Finally, the servers execute sequentially each task and when there are no more tasks to be processed the server is closed and it returns a set of data that will be used for the final reports.

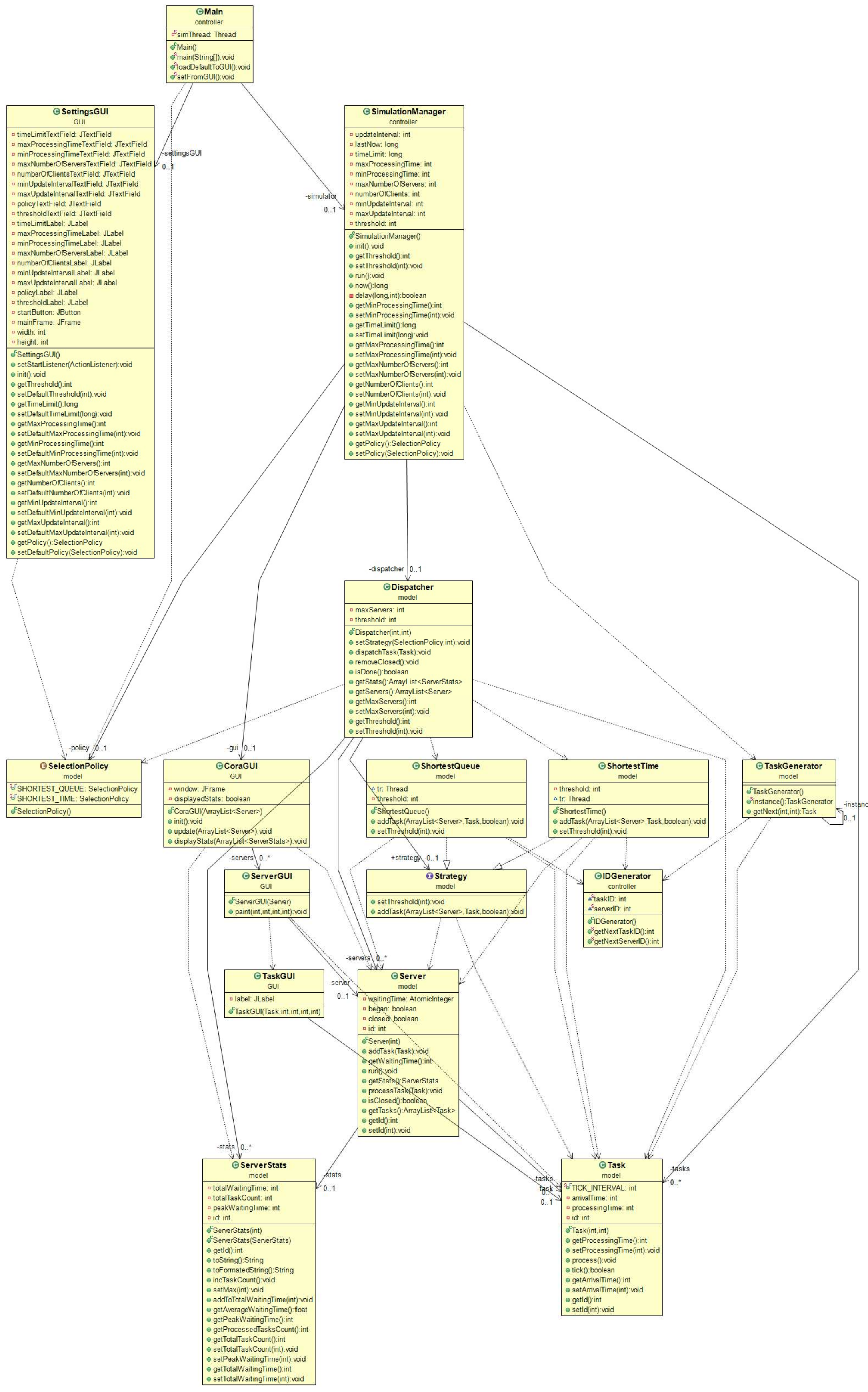
4. Design

4.1 Data structures

The used data structures are basic arrays, lists and queues. The main important aspect about data structures is the implementation of those. Given the fact that the application is developed in an multi – threaded manner special implementations were needed. Thus, the `BlockingQueue` was considered as a valuable alternative because it provides thread protection and it guarantees data integrity.

4.2 Class diagram

An important design pattern used in this application was Strategy. This was used in order to easily change the policy of the dispatcher regarding the task association (with respect to the number of tasks that a server already has or to the waiting time of a server).


Main
controller
simThread: Thread
Main()
main(String[]):void
loadDefaultToGUI():void
setFromGUI():void
-settingsGUI
0..1
-simulator
0..1
SettingsGUI
GUI
timeLimitTextField: JTextField
maxProcessingTimeTextField: JTextField
minProcessingTimeTextField: JTextField
maxNumberOfServersTextField: JTextField
numberOfClientsTextField: JTextField
minUpdateIntervalTextField: JTextField
maxUpdateIntervalTextField: JTextField
policyTextField: JTextField
thresholdTextField: JTextField
timeLimitLabel: JLabel
maxProcessingTimeLabel: JLabel
minProcessingTimeLabel: JLabel
maxNumberOfServersLabel: JLabel
numberOfClientsLabel: JLabel
minUpdateIntervalLabel: JLabel
maxUpdateIntervalLabel: JLabel
policyLabel: JLabel
thresholdLabel: JLabel
startButton: JButton
mainFrame: JFrame
width: int
height: int
SettingsGUI()
setStartListener(ActionListener):void
init():void
getThreshold():int
setDefaultThreshold(int):void
getTimeLimit():long
setDefaultTimeLimit(long):void
getMaxProcessingTime():int
setDefaultMaxProcessingTime(int):void
getMinProcessingTime():int
setDefaultMinProcessingTime(int):void
getMaxNumberOfServers():int
setDefaultMaxNumberOfServers(int):void
getNumberOfClients():int
setDefaultNumberOfClients(int):void
getMinUpdateInterval():int
setDefaultMinUpdateInterval(int):void
getMaxUpdateInterval():int
setDefaultMaxUpdateInterval(int):void
getPolicy():SelectionPolicy
setDefaultPolicy(SelectionPolicy):void
SimulationManager
controller
updateInterval: int
lastNow: long
timeLimit: long
maxProcessingTime: int
minProcessingTime: int
maxNumberOfServers: int
numberOfClients: int
minUpdateInterval: int
maxUpdateInterval: int
threshold: int
SimulationManager()
init():void
getThreshold():int
setThreshold(int):void
run():void
now():long
delay(long,int):boolean
getMinProcessingTime():int
setMinProcessingTime(int):void
getTimeLimit():long
setTimeLimit(long):void
getMaxProcessingTime():int
setMaxProcessingTime(int):void
getMaxNumberOfServers():int
setMaxNumberOfServers(int):void
getNumberOfClients():int
setNumberOfClients(int):void
getMinUpdateInterval():int
setMinUpdateInterval(int):void
getMaxUpdateInterval():int
setMaxUpdateInterval(int):void
getPolicy():SelectionPolicy
setPolicy(SelectionPolicy):void
-dispatcher
0..1
Dispatcher
model
maxServers: int
threshold: int
Dispatcher(int,int)
setStrategy(SelectionPolicy,int):void
dispatchTask(Task):void
removeClosed():void
isDone():boolean
getStats():ArrayList<ServerStats>
getServers():ArrayList<Server>
getMaxServers():int
setMaxServers(int):void
getThreshold():int
setThreshold(int):void
-policy
0..1
-gui
0..1
SelectionPolicy
model
SHORTEST_QUEUE: SelectionPolicy
SHORTEST_TIME: SelectionPolicy
SelectionPolicy()
CoraGUI
GUI
window: JFrame
displayedStats: boolean
CoraGUI(ArrayList<Server>)
init():void
update(ArrayList<Server>):void
displayStats(ArrayList<ServerStats>):void
ShortestQueue
model
tr: Thread
threshold: int
ShortestQueue()
addTask(ArrayList<Server>,Task,boolean):void
setThreshold(int):void
ShortestTime
model
threshold: int
tr: Thread
ShortestTime()
addTask(ArrayList<Server>,Task,boolean):void
setThreshold(int):void
TaskGenerator
model
TaskGenerator()
instance():TaskGenerator
getNext(int,int):Task
-instance
0..1
-servers
0..*
+strategy
0..1
ServerGUI
GUI
ServerGUI(Server)
paint(int,int,int,int):void
Strategy
model
setThreshold(int):void
addTask(ArrayList<Server>,Task,boolean):void
IDGenerator
controller
taskID: int
serverID: int
IDGenerator()
getNextTaskID():int
getNextServerID():int
-servers
0..*
-server
0..1
TaskGUI
GUI
label: JLabel
TaskGUI(Task,int,int,int,int)
Server
model
waitingTime: AtomicInteger
began: boolean
closed: boolean
id: int
Server(int)
addTask(Task):void
getWaitingTime():int
run():void
getStats():ServerStats
processTask(Task):void
isClosed():boolean
getTasks():ArrayList<Task>
getId():int
setId(int):void
-stats
0..*
-stats
0..1
-tasks
-task
0..*
0..1
-tasks
0..*
ServerStats
model
totalWaitingTime: int
totalTaskCount: int
peakWaitingTime: int
id: int
ServerStats(int)
ServerStats(ServerStats)
getId():int
toString():String
toFormatedString():String
incTaskCount():void
setMax(int):void
addToTotalWaitingTime(int):void
getAverageWaitingTime():float
getPeakWaitingTime():int
getProcessedTasksCount():int
getTotalTaskCount():int
setTotalTaskCount(int):void
setPeakWaitingTime(int):void
getTotalWaitingTime():int
setTotalWaitingTime(int):void
Task
model
TICK_INTERVAL: int
arrivalTime: int
processingTime: int
id: int
Task(int,int)
getProcessingTime():int
setProcessingTime(int):void
process():void
tick():boolean
getArrivalTime():int
setArrivalTime(int):void
getId():int
setId():void

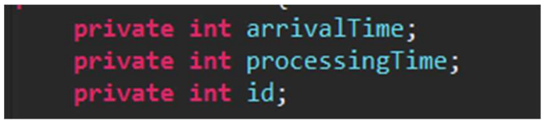
4.3 Algorithms

Several dispatching algorithms were considered. On one side, the server with the least amount of tasks was considered to be suitable for accepting another task and on the other side the total processing time of a server (the sum of all processing times of the tasks).

5. Implementation

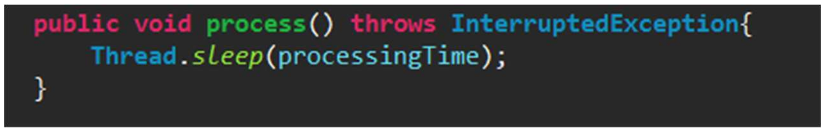
5.1 Task

This class represents the tasks that servers have to process. The “ processingTime ” field is used in order to represent the time each task needs in order to be processed. Also, an ID is associated to each task so it could be easily identified during simulation.



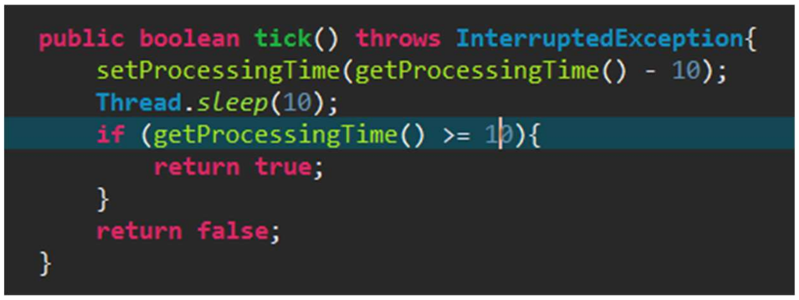
```
    private int arrivalTime;
    private int processingTime;
    private int id;
```

In order to simulate the processing of a task, it implements the “ process ” method that takes no parameters and stops the current thread for the amount of time that is needed for the task to be processed.



```
public void process() throws InterruptedException{
    Thread.sleep(processingTime);
}
```

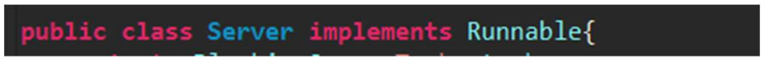
This method, comes with limitations from the point of view of the real – time display of the evolution. So, a new processing method is needed. The “ tick ” method stops the thread for ten milliseconds and decreases the processing time by 10. The return value consists of whether or not the task was fully processed.



```
public boolean tick() throws InterruptedException{
    setProcessingTime(getProcessingTime() - 10);
    Thread.sleep(10);
    if (getProcessingTime() >= 10){
        return true;
    }
    return false;
}
```

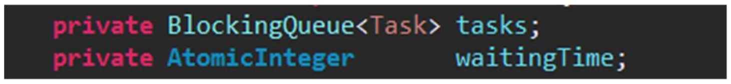
5.2 Server

The “ Server ” class represents the servers that receive and process tasks. In order to create a more realistic simulation each server will run in a separate thread. To achieve this the class “ Server “ implements the interface “ Runnable ”. This approach was chosen, instead of extending the “ Thread ” class in order to contain only the functionality we want in the run method and to achieve a loosely coupled code.



```
public class Server implements Runnable{
```

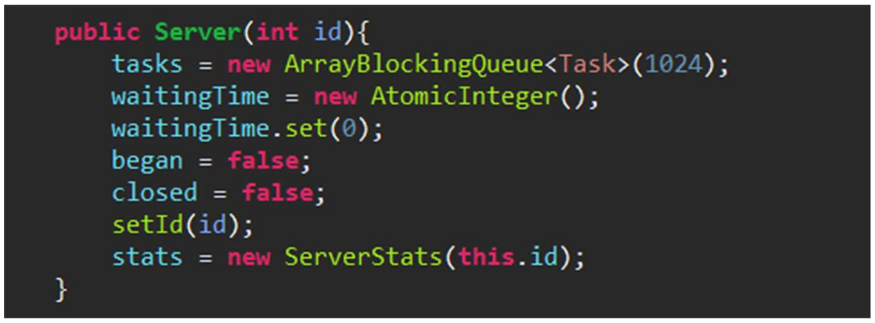
Given the fact that each server will run in a separate thread, special attention is needed for the data types that interact with external components, from different threads.



```
private BlockingQueue<Task> tasks;
private AtomicInteger       waitingTime;
```

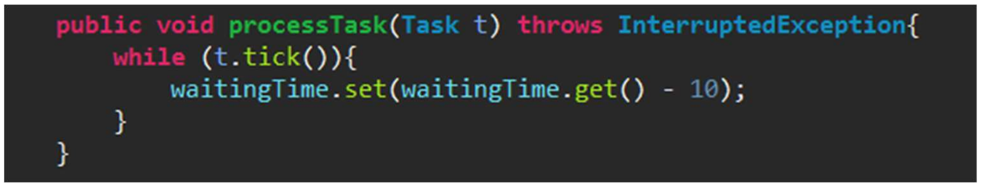
Other important fields are used in order to get information about the state of the current object and to receive data about it.

The constructor sets all values to default and also creates a new “ ServerStats ” object (see 5 . 3) with the same id as the server.



```
public Server(int id){
    tasks = new ArrayBlockingQueue<Task>(1024);
    waitingTime = new AtomicInteger();
    waitingTime.set(0);
    began = false;
    closed = false;
    setId(id);
    stats = new ServerStats(this.id);
}
```

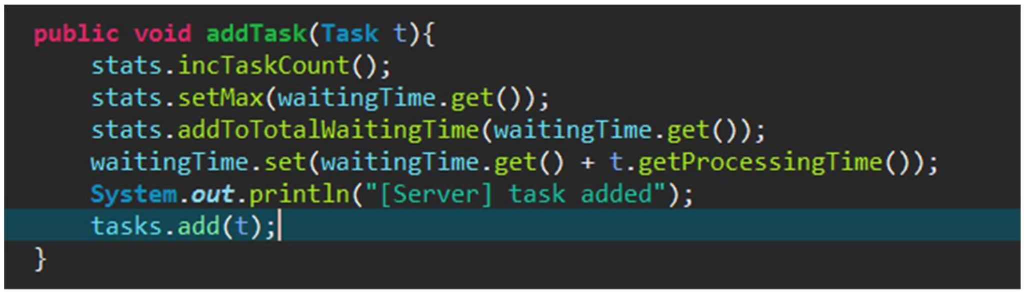
In order to process a task, the “ processTask ” method used. It receives as parameter the task to be processed and it makes use of the “ tick ” method to process and update the “ waitingTime ” value.



```
public void processTask(Task t) throws InterruptedException{
    while (t.tick()){
        waitingTime.set(waitingTime.get() - 10);
    }
}
```

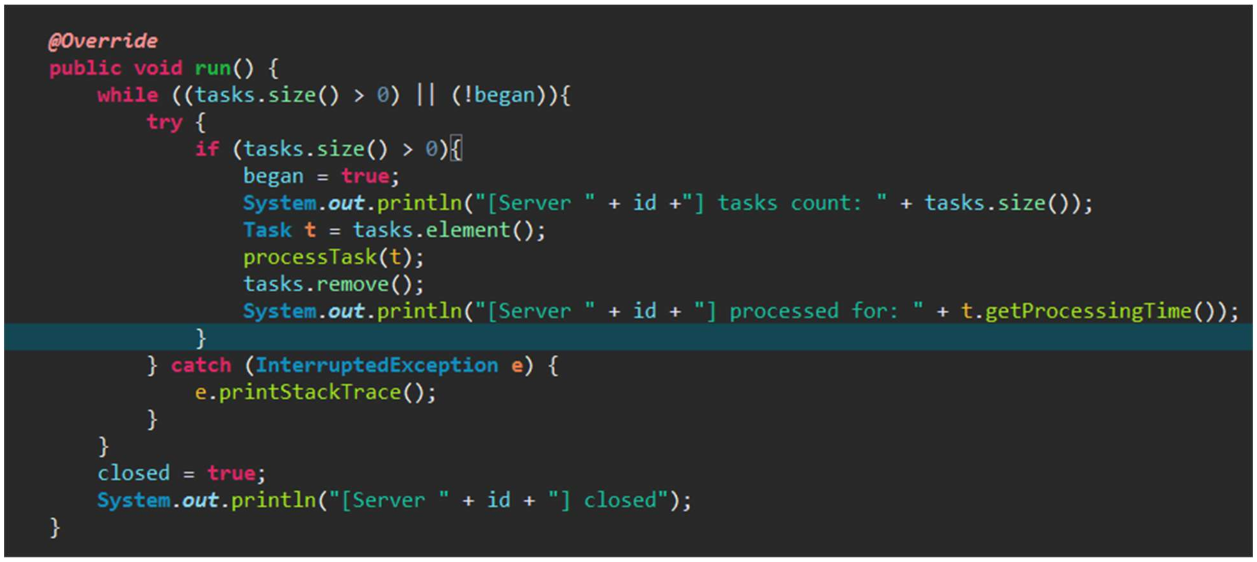
Another important method of the “ Server ” class is the “ addTask ” method that allows an external class to add a new task to the current queue.

Firstly, the stats of the current server are updated, then the waiting time is increased. Finally, the a log message is printed and the task is added in the queue.



```
public void addTask(Task t){
    stats.incTaskCount();
    stats.setMax(waitingTime.get());
    stats.addToTotalWaitingTime(waitingTime.get());
    waitingTime.set(waitingTime.get() + t.getProcessingTime());
    System.out.println("[Server] task added");
    tasks.add(t);
}
```

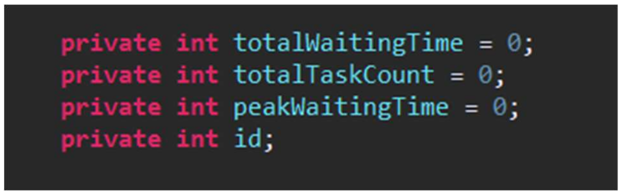
The most important method is “ run ”. This method is run only once when the thread starts. The while will run until there exist tasks in the queue. The “ began ” flag ensures the fact that the while will run only once. After the while has finished, the flag “ closed ” is set and a log message is printed to the console. Inside the while loop the next task is retrieved from the queue, being processed and the it is being removed from the queue. A try - catch is needed because of the " processTask " call that throws an exception.



```
@Override
public void run() {
    while ((tasks.size() > 0) || (!began)){
        try {
            if (tasks.size() > 0){
                began = true;
                System.out.println("[Server " + id +"] tasks count: " + tasks.size());
                Task t = tasks.element();
                processTask(t);
                tasks.remove();
                System.out.println("[Server " + id + "] processed for: " + t.getProcessingTime());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    closed = true;
    System.out.println("[Server " + id + "] closed");
}
```

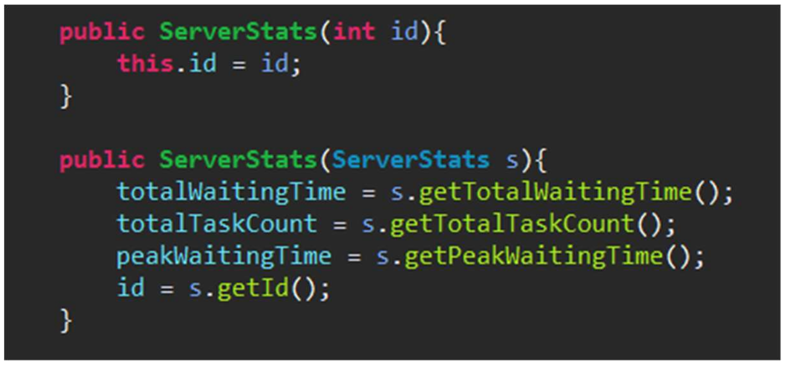
5.3 ServerStats

The class " ServerStats " is needed in order to log data about the evolution of a thread such as peak waiting time, average waiting time and total task count, across the lifetime of a server and the id of the respective server.



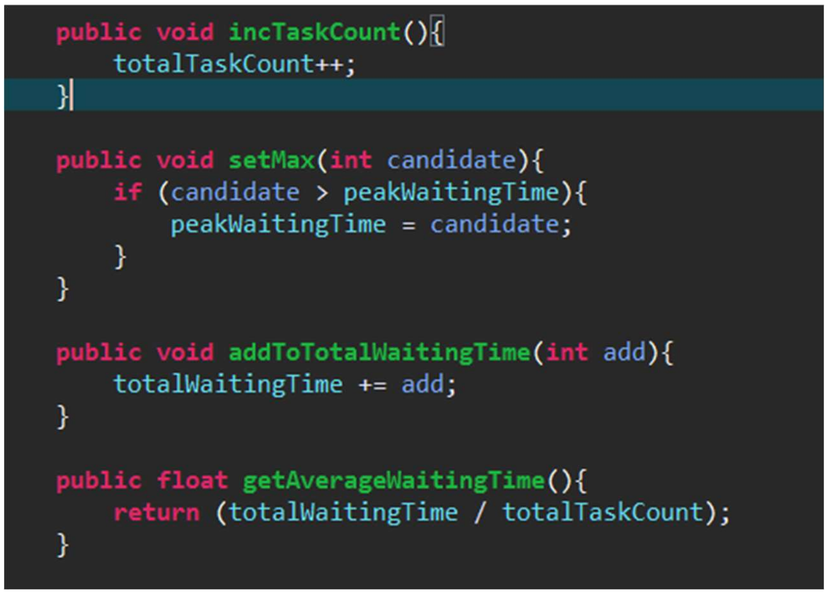
```
private int totalWaitingTime = 0;
private int totalTaskCount = 0;
private int peakWaitingTime = 0;
private int id;
```

Two kinds of constructors were needed : a classical constructor that receives only the ID and a copy – constructor.



```
public ServerStats(int id){
    this.id = id;
}

public ServerStats(ServerStats s){
    totalWaitingTime = s.getTotalWaitingTime();
    totalTaskCount = s.getTotalTaskCount();
    peakWaitingTime = s.getPeakWaitingTime();
    id = s.getId();
}
```

Further a set of useful methods was considered that will ease the logging process.



```
public void incTaskCount(){
    totalTaskCount++;
}

public void setMax(int candidate){
    if (candidate > peakWaitingTime){
        peakWaitingTime = candidate;
    }
}

public void addToTotalWaitingTime(int add){
    totalWaitingTime += add;
}

public float getAverageWaitingTime(){
    return (totalWaitingTime / totalTaskCount);
}
```

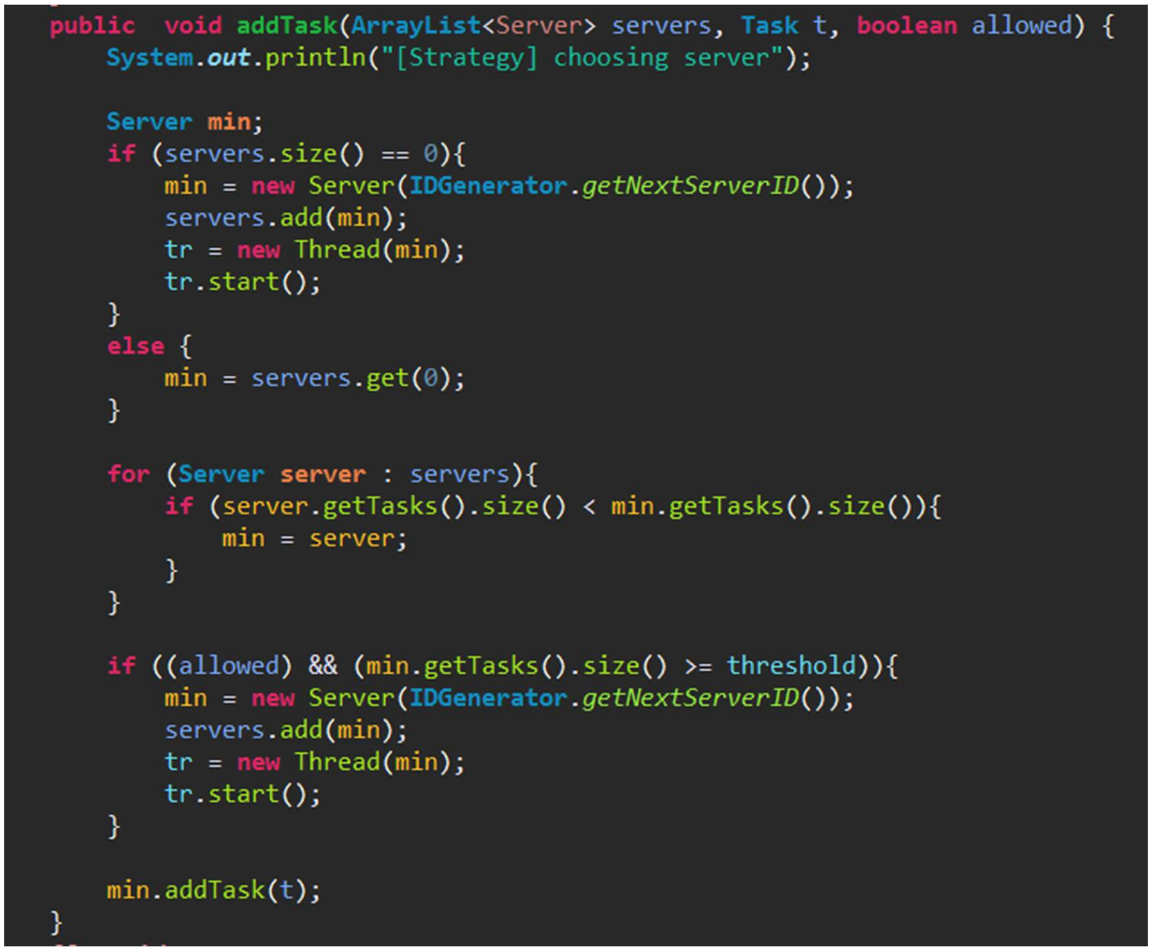
This way the task counter is incremented, the maximum peak time is set and the total waiting time is increased. The last method computes the average waiting time (the ration between the total waiting time and the task counter) and returns it.

Useful “ toString ” and “ toFormatedString ” were considered. The last one returns a string formatted in html style in order to ease the display of the data in a JLabel (see 5.12) .

5.4 ShortestQueue

The “ ShortestQueue “ object implements a policy of choosing the server that should receive a given task at an arbitrary moment. It takes into account a given threshold and if all servers are overloaded (have more tasks than the threshold) a new server will be open. In the same time, there exists a maximum number of servers that can be opened at a given time, and whether or not a opening a new server is allowed is transmitted through a boolean parameter.

The “ ShortestQueue ” implementation of the selection policy chooses the server with the minimum size of the queue (hence the name).



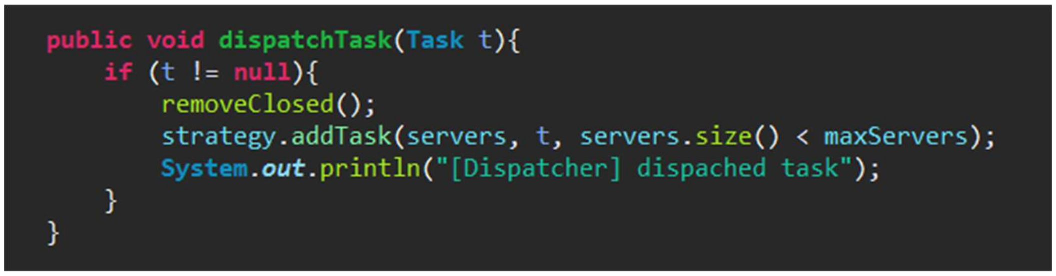
```java
public  void addTask(ArrayList<Server> servers, Task t, boolean allowed) {
    System.out.println("[Strategy] choosing server");

    Server min;
    if (servers.size() == 0){
        min = new Server(IDGenerator.getNextServerID());
        servers.add(min);
        tr = new Thread(min);
        tr.start();
    }
    else {
        min = servers.get(0);
    }

    for (Server server : servers){
        if (server.getTasks().size() < min.getTasks().size()){
            min = server;
        }
    }

    if ((allowed) && (min.getTasks().size() >= threshold)){
        min = new Server(IDGenerator.getNextServerID());
        servers.add(min);
        tr = new Thread(min);
        tr.start();
    }

    min.addTask(t);
}
```

5.5 ShortestTime

Identical with “ ShortestQueue ” the only difference being the selection criteria, in this implementation the minimum is chosen based on the waiting time of a server.

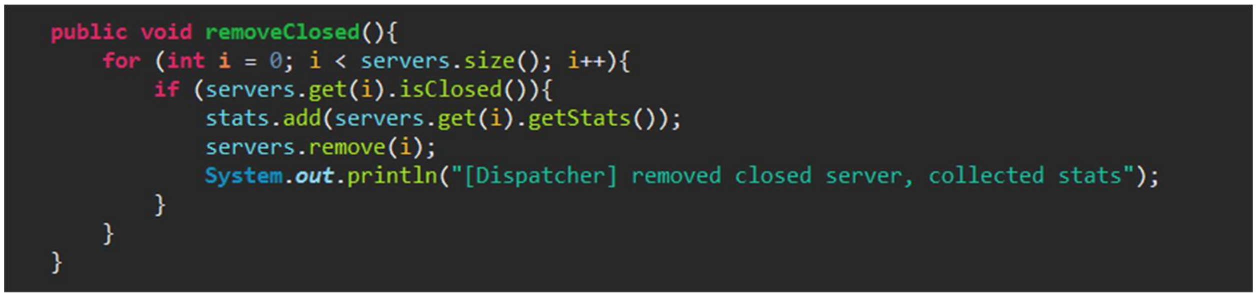
5.6 Dispatcher

The “ Dispatcher ” class is of paramount importance. It manages the distribution of tasks between servers based on a chosen strategy. The method that does this is “ dispatchTask ” which receives a task to be dispatched in the current servers.



```java
public void dispatchTask(Task t){
    if (t != null){
        removeClosed();
        strategy.addTask(servers, t, servers.size() < maxServers);
        System.out.println("[Dispatcher] dispached task");
    }
}
```

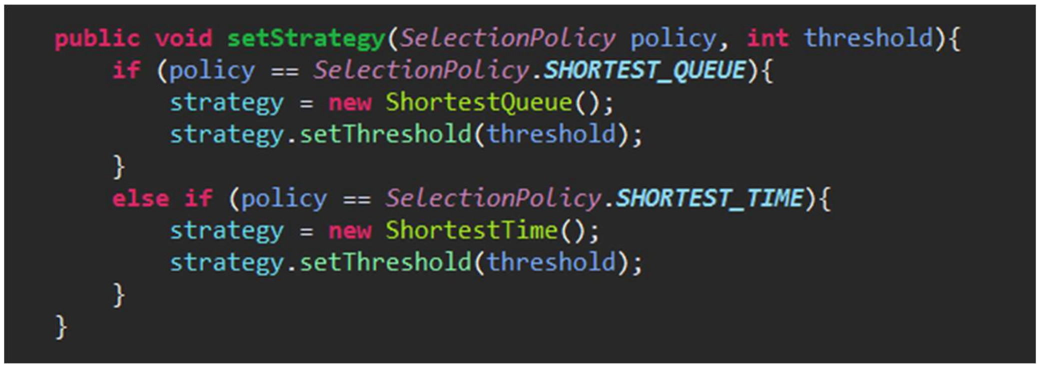
Firstly the closed servers are removed from the array using the “ removeClosed ” method



```java
public void removeClosed(){
    for (int i = 0; i < servers.size(); i++){
        if (servers.get(i).isClosed()){
            stats.add(servers.get(i).getStats());
            servers.remove(i);
            System.out.println("[Dispatcher] removed closed server, collected stats");
        }
    }
}
```

This method, besides removing closed servers it collects data about each run and stores it in an array of stats.

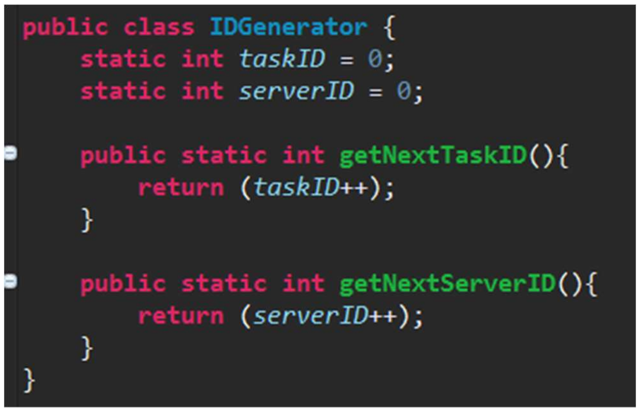
Also, the dispatcher allows an external class to chose the desired strategy for dispatching through the method " setStrategy ".



```java
public void setStrategy(SelectionPolicy policy, int threshold){
    if (policy == SelectionPolicy.SHORTEST_QUEUE){
        strategy = new ShortestQueue();
        strategy.setThreshold(threshold);
    }
    else if (policy == SelectionPolicy.SHORTEST_TIME){
        strategy = new ShortestTime();
        strategy.setThreshold(threshold);
    }
}
```

5.7 IDGenerator

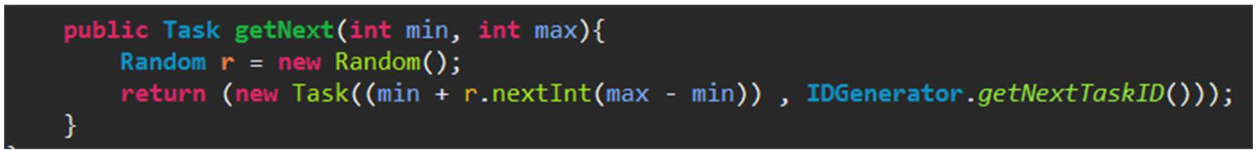
The class " IDGenerator " contains two static methods that generate ids for tasks and servers. The need of static variables and static methods can be justified by the fact that the ids have to be unique and independent of instances.



```java
public class IDGenerator {
    static int taskID = 0;
    static int serverID = 0;

    public static int getNextTaskID(){
        return (taskID++);
    }

    public static int getNextServerID(){
        return (serverID++);
    }
}
```

5.8 TaskGenerator

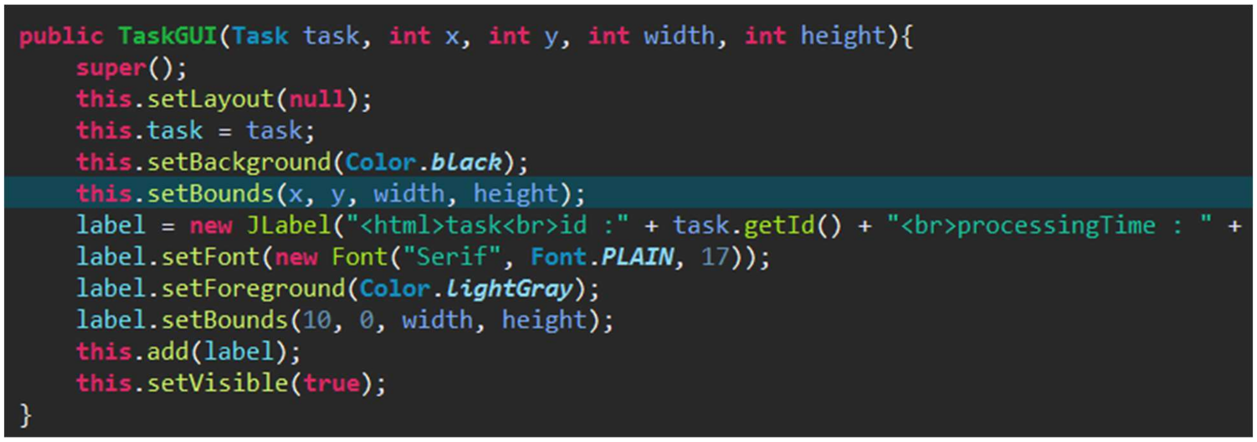
The " TaskGenerator " class is implemented following the singleton approach. The main method is " getNext " which returns a new task with a new ID and a random processing time between the two values received as inputs.



```java
public Task getNext(int min, int max){
    Random r = new Random();
    return (new Task((min + r.nextInt(max - min)) , IDGenerator.getNextTaskID()));
}
```

5.9 TaskGUI

The “ TaskGUI ” class represents a view of a task object. It receives a task as the single parameter of a constructor and also it extends JPanel, in order to have better GUI functionality. On the panel a label is added which displays real – time data about the task.

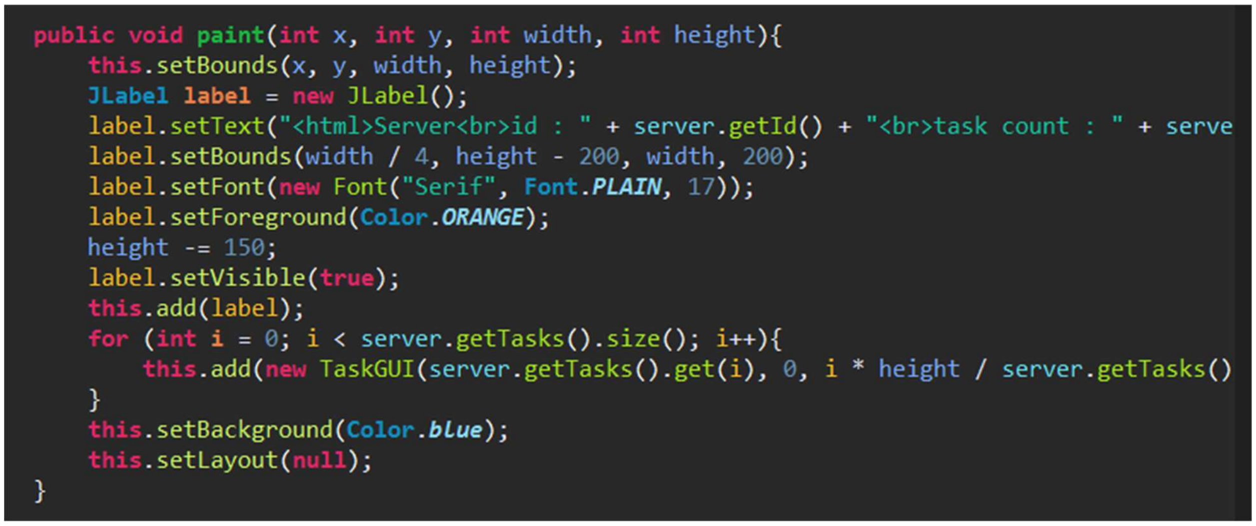


```java
public TaskGUI(Task task, int x, int y, int width, int height){
    super();
    this.setLayout(null);
    this.task = task;
    this.setBackground(Color.black);
    this.setBounds(x, y, width, height);
    label = new JLabel("<html>task<br>id :" + task.getId() + "<br>processingTime : " +
    label.setFont(new Font("Serif", Font.PLAIN, 17));
    label.setForeground(Color.lightGray);
    label.setBounds(10, 0, width, height);
    this.add(label);
    this.setVisible(true);
}
```

5.10 ServerGUI

In the same manner as above, the class “ ServerGUI ” is a view of a Server object aggregating a server and also extending a panel. This is why, it is very easy to add other components on the server like a label to display real-time info and all the tasks belonging to a server. More precisely TaskGUIs.

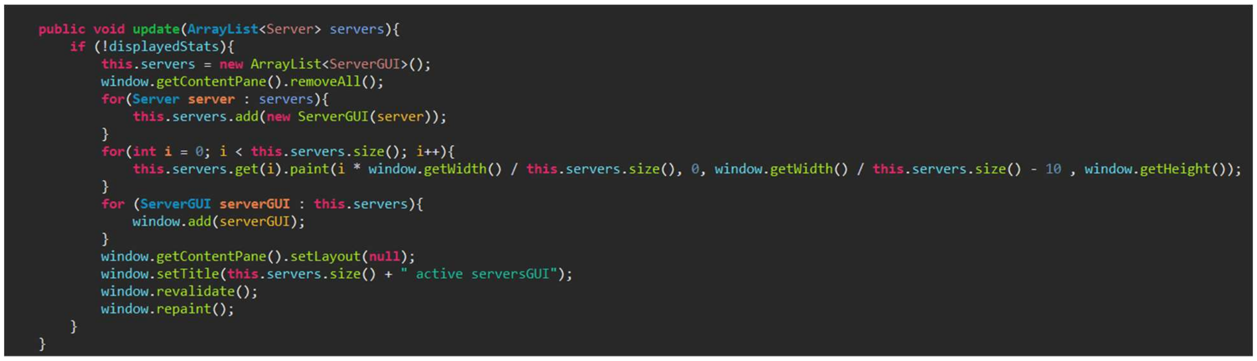
The method “ paint ” redraws all the tasks of the corresponding server.



```java
public void paint(int x, int y, int width, int height){
    this.setBounds(x, y, width, height);
    JLabel label = new JLabel();
    label.setText("<html>Server<br>id : " + server.getId() + "<br>task count : " + serve
    label.setBounds(width / 4, height - 200, width, 200);
    label.setFont(new Font("Serif", Font.PLAIN, 17));
    label.setForeground(Color.ORANGE);
    height -= 150;
    label.setVisible(true);
    this.add(label);
    for (int i = 0; i < server.getTasks().size(); i++){
        this.add(new TaskGUI(server.getTasks().get(i), 0, i * height / server.getTasks()
    }
    this.setBackground(Color.blue);
    this.setLayout(null);
}
```

5.11 CoraGUI

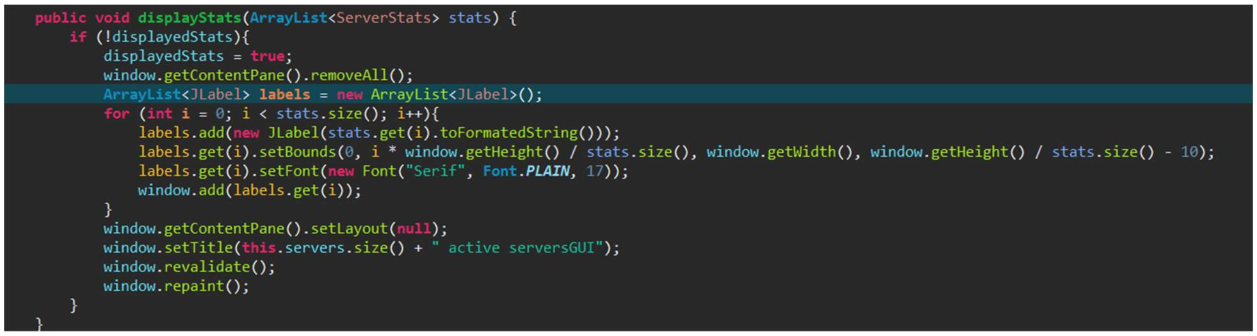
The “ CoraGUI ” class is the top level class that manages the General User Interface for real-time displaying of the simulation evolution but as well as displaying the final report about the simulation.



```
public void update(ArrayList<Server> servers){
    if (!displayedStats){
        this.servers = new ArrayList<ServerGUI>();
        window.getContentPane().removeAll();
        for(Server server : servers){
            this.servers.add(new ServerGUI(server));
        }
        for(int i = 0; i < this.servers.size(); i++){
            this.servers.get(i).paint(i * window.getWidth() / this.servers.size(), 0, window.getWidth() / this.servers.size() - 10 , window.getHeight());
        }
        for (ServerGUI serverGUI : this.servers){
            window.add(serverGUI);
        }
        window.getContentPane().setLayout(null);
        window.setTitle(this.servers.size() + " active serversGUI");
        window.revalidate();
        window.repaint();
    }
}
```

The update method takes as input parameter an arrayList of servers and updates the main window. If the simulation hasn’t ended (i.e. the final stats were not displayed) all content is removed and the new serverGUIs are created, populated and added to the window.

Also, this class is responsible for displaying the final stats so the method “ displayStats “ was created in order to fulfil this need.



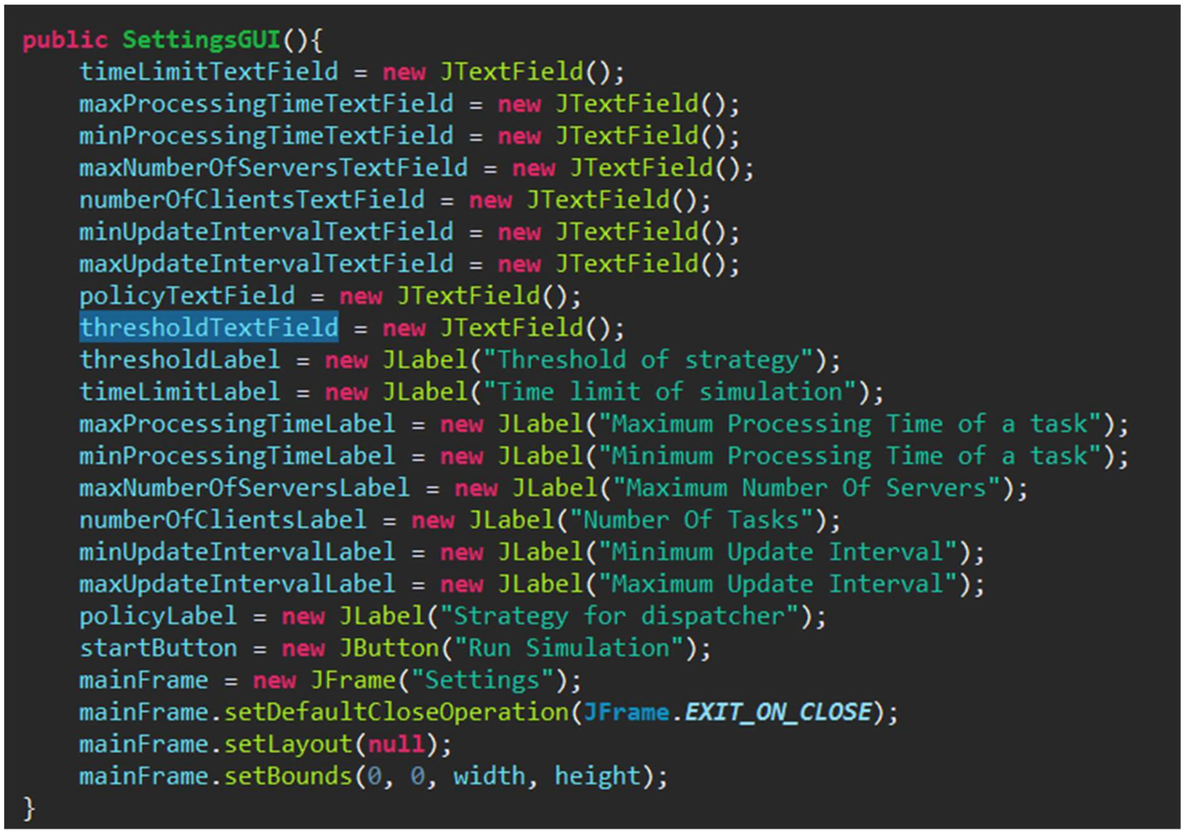
```
public void displayStats(ArrayList<ServerStats> stats) {
    if (!displayedStats){
        displayedStats = true;
        window.getContentPane().removeAll();
        ArrayList<JLabel> labels = new ArrayList<JLabel>();
        for (int i = 0; i < stats.size(); i++){
            labels.add(new JLabel(stats.get(i).toFormatedString()));
            labels.get(i).setBounds(0, i * window.getHeight() / stats.size(), window.getWidth(), window.getHeight() / stats.size() - 10);
            labels.get(i).setFont(new Font("Serif", Font.PLAIN, 17));
            window.add(labels.get(i));
        }
        window.getContentPane().setLayout(null);
        window.setTitle(this.servers.size() + " active serversGUI");
        window.revalidate();
        window.repaint();
    }
}
```

The method takes as input parameter and ArrayList of “ ServerStats ” object which will be displayed in JLabels.

5.12 SettingsGUI

The “ SettingsGUI “ class is managing the first window that shows up and it allows the user to set the parameters of the simulation and to start it using the “ run ” button.
The default values are loaded from the simulation manager and when the button is pressed the current values are sent to the simulation manager. This approach was considered in order to make possible further development of the application without a setup GUI.

The ActionListener of the start button is set from the main controller in order to link the front end and the back end of the application without a hard connection between the too, ensuring independence.
The constructor creates the interface components.

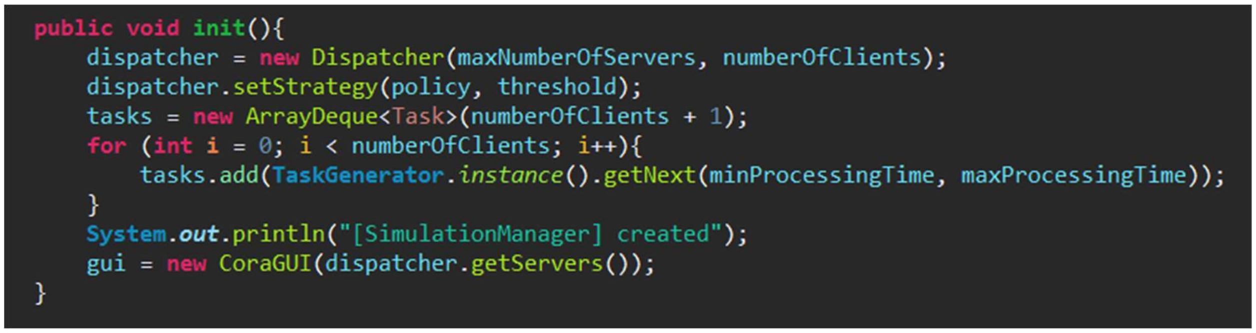


```java
public SettingsGUI(){
    timeLimitTextField = new JTextField();
    maxProcessingTimeTextField = new JTextField();
    minProcessingTimeTextField = new JTextField();
    maxNumberOfServersTextField = new JTextField();
    numberOfClientsTextField = new JTextField();
    minUpdateIntervalTextField = new JTextField();
    maxUpdateIntervalTextField = new JTextField();
    policyTextField = new JTextField();
    thresholdTextField = new JTextField();
    thresholdLabel = new JLabel("Threshold of strategy");
    timeLimitLabel = new JLabel("Time limit of simulation");
    maxProcessingTimeLabel = new JLabel("Maximum Processing Time of a task");
    minProcessingTimeLabel = new JLabel("Minimum Processing Time of a task");
    maxNumberOfServersLabel = new JLabel("Maximum Number Of Servers");
    numberOfClientsLabel = new JLabel("Number Of Tasks");
    minUpdateIntervalLabel = new JLabel("Minimum Update Interval");
    maxUpdateIntervalLabel = new JLabel("Maximum Update Interval");
    policyLabel = new JLabel("Strategy for dispatcher");
    startButton = new JButton("Run Simulation");
    mainFrame = new JFrame("Settings");
    mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    mainFrame.setLayout(null);
    mainFrame.setBounds(0, 0, width, height);
}
```

5.13 SimulationManager

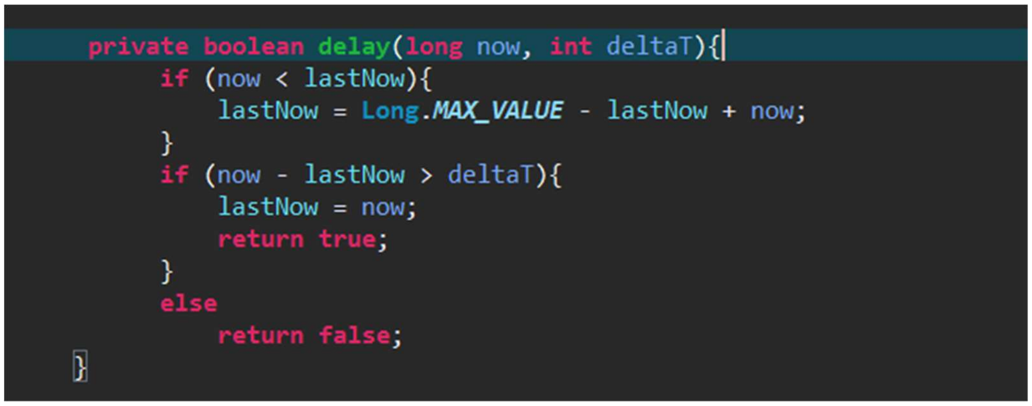
The “ SimulationManager ” class is responsible for creating and running all the tasks, servers, dispatcher and link those to the simulation GUI.

Firstly all objects are created and the array of tasks is generated in the “ init ” method.



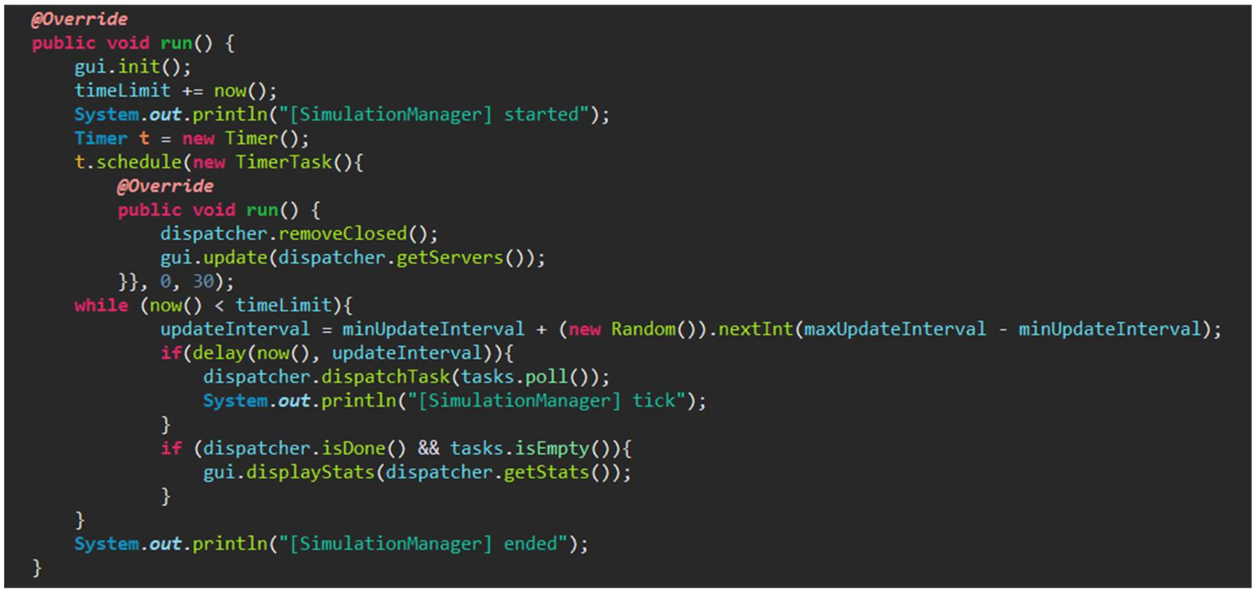
```java
public void init(){
    dispatcher = new Dispatcher(maxNumberOfServers, numberOfClients);
    dispatcher.setStrategy(policy, threshold);
    tasks = new ArrayDeque<Task>(numberOfClients + 1);
    for (int i = 0; i < numberOfClients; i++){
        tasks.add(TaskGenerator.instance().getNext(minProcessingTime, maxProcessingTime));
    }
    System.out.println("[SimulationManager] created");
    gui = new CoraGUI(dispatcher.getServers());
}
```

Further, the need of time management is obvious (the simulation has to run a limited amount of time, and a new task has to be dispatched at a random amount of time). In order to prevent the thread to block using a call of the “ sleep ” method the “ delay ” method was created. It returns true if a given amount of time has elapsed since the last call.



```java
    private boolean delay(long now, int deltaT){
        if (now < lastNow){
            lastNow = Long.MAX_VALUE - lastNow + now;
        }
        if (now - lastNow > deltaT){
            lastNow = now;
            return true;
        }
        else
            return false;
    }
```

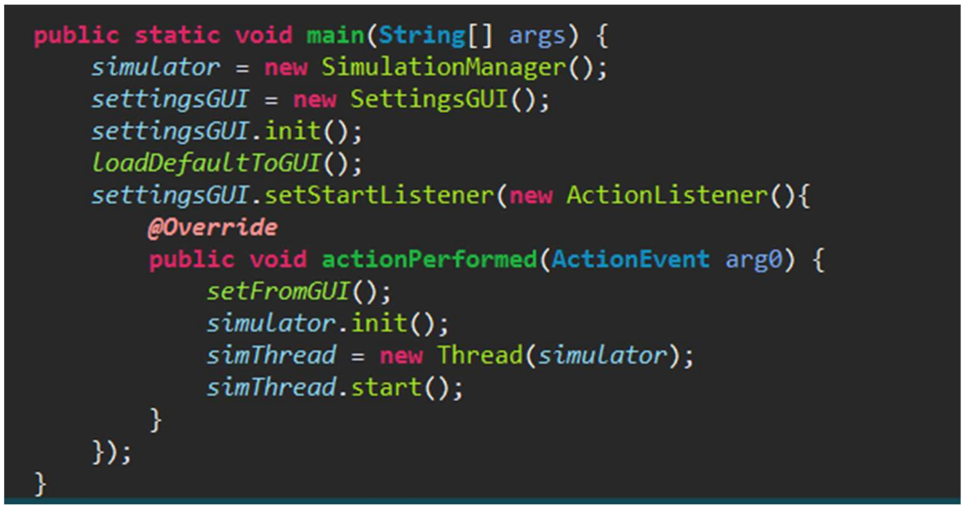
Finally, the “ run ” method initializes the gui and sets up a timer to call the update method once every 30 milliseconds in order to have an independent update of the general user interface.
The main while runs a limited amount of time, constantly checking if enough time has elapsed. When an update has to be made and a new task has to be dispatched a log message is printed. The task is polled from the pre-generated queue of tasks and sent to the dispatcher.



```java
@Override
public void run() {
    gui.init();
    timeLimit += now();
    System.out.println("[SimulationManager] started");
    Timer t = new Timer();
    t.schedule(new TimerTask(){
        @Override
        public void run() {
            dispatcher.removeClosed();
            gui.update(dispatcher.getServers());
        }}, 0, 30);
    while (now() < timeLimit){
            updateInterval = minUpdateInterval + (new Random()).nextInt(maxUpdateInterval - minUpdateInterval);
            if(delay(now(), updateInterval)){
                dispatcher.dispatchTask(tasks.poll());
                System.out.println("[SimulationManager] tick");
            }
            if (dispatcher.isDone() && tasks.isEmpty()){
                gui.displayStats(dispatcher.getStats());
            }
    }
    System.out.println("[SimulationManager] ended");
}
```

5.14 Main

In the main function the settings GUI created and default values are loaded from the simulation. After this, the action listener of the start button is set. Firstly to load the values from the GUI into the simulation and the run the simulation.



```
public static void main(String[] args) {
    simulator = new SimulationManager();
    settingsGUI = new SettingsGUI();
    settingsGUI.init();
    loadDefaultToGUI();
    settingsGUI.setStartListener(new ActionListener(){
        @Override
        public void actionPerformed(ActionEvent arg0) {
            setFromGUI();
            simulator.init();
            simThread = new Thread(simulator);
            simThread.start();
        }
    });
}
```

6 Conclusions and Further Development

The development of this application was a very good exercise of OOP design but much more important it offered an insight of multi-thread application. Further development consists of more complex server behavior (dynamically passing tasks to more free servers) and a more comprehensive GUI (meaningful colors).