FUNDAMENTAL PROGRAMMING TECHNIQUES

ASSIGNMENT 2 – SUPPORT PRESENTATION

Problem and solution

PROBLEM: “Improper queue management leads to high waiting times for clients and inefficient usage of resources”

SOLUTION: Queue management system implementing efficient queue allocation mechanisms

How to design and implement the solution?

1. Clearly state the main objective and the sub-objectives required to reach it.

2. Analyze the problem and define the functional and non-functional requirements.

3. Design the solution

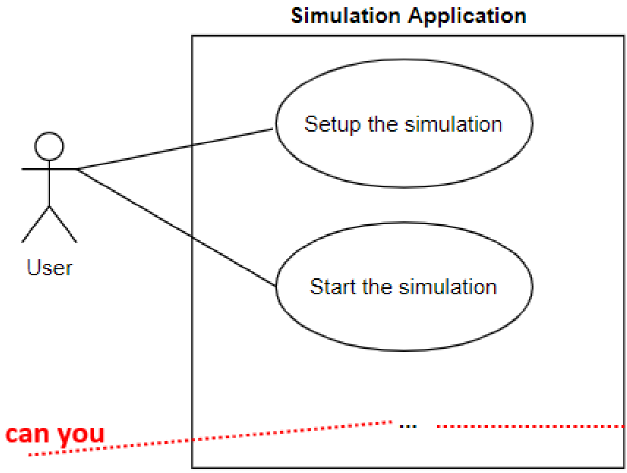
4. Implement the solution

5. Test the solution

Objectives

- Main objective
  - Design and implement an application aiming to analyze queuing-based systems by (1) simulating a series of N clients arriving for service, entering Q queues, waiting, being served and finally leaving the queues, and (2) computing the average waiting time, average service time and peak hour .
- Sub-objectives
  - Analyze the problem and identify requirements
  - Design the simulation application
  - Implement the simulation application
  - Test the simulation application

Analysis



What other use cases can you identify?

Can you identify use cases connected with "extend", "include" and "generalize" relationships?

Use Case: setup simulation
Primary Actor: user
Main Success Scenario:

1. The user inserts the values for the: number of clients, number of queues, simulation interval, minimum and maximum arrival time, and minimum and maximum service time
2. The user clicks on the validate input data button
3. The application validates the data and displays a message informing the user to start the simulation

Alternative Sequence: Invalid values for the setup parameters

- The user inserts invalid values for the application's setup parameters
- The application displays an error message and requests the user to insert valid values
- The scenario returns to step 1

Define requirements

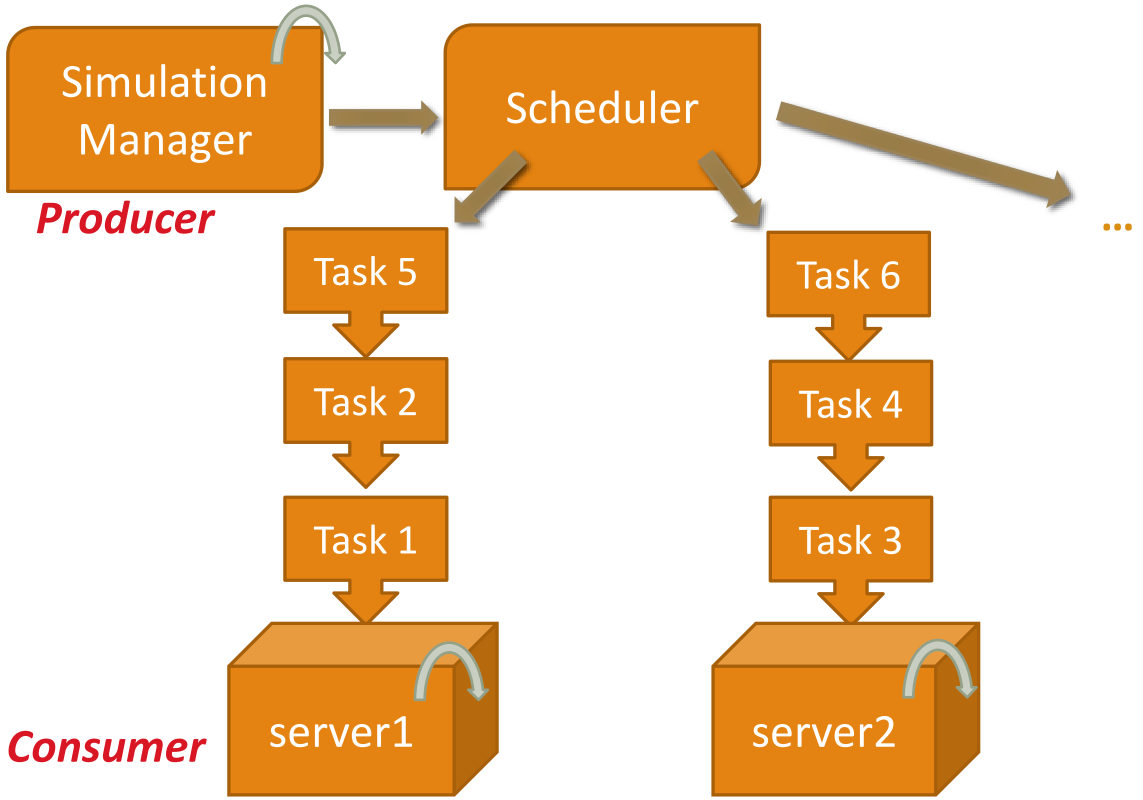
Functional requirements:

- The simulation application should allow users to setup the simulation
- The simulation application should allow users to start the simulation
- The simulation application should display the real-time queues evolution
- … what other functional requirements can you define? …

Non-Functional requirements:

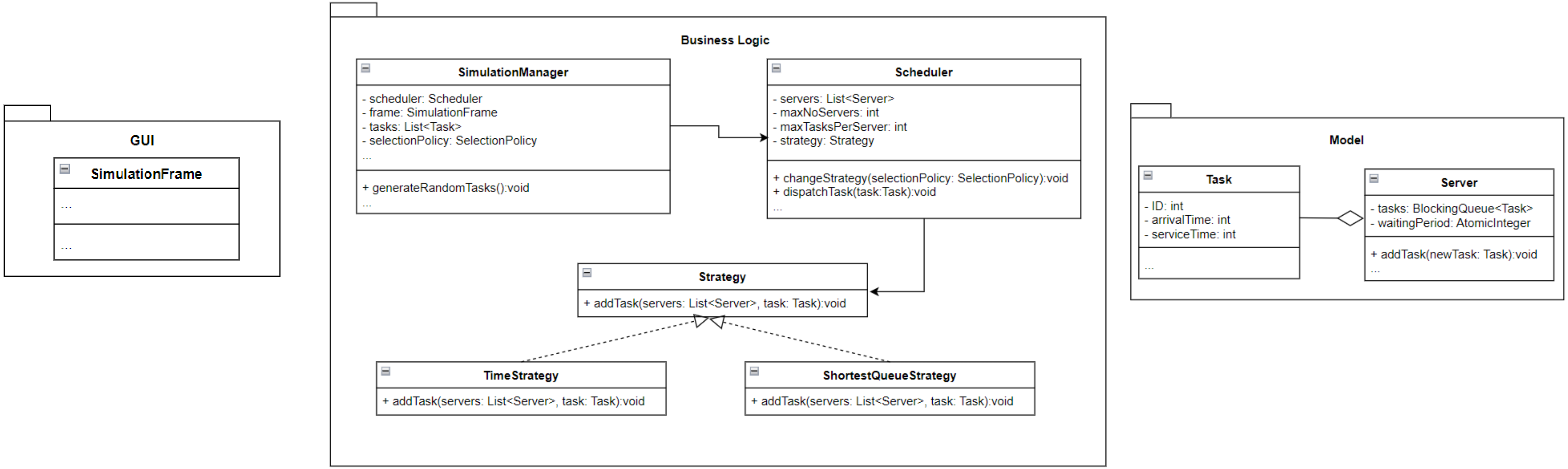
- The simulation application should be intuitive and easy to use by the user
- … what other non-functional requirements can you define? …

Design – Conceptual Architecture



GUI (show servers state & waiting queue)

Design



GUI

SimulationFrame

...

...

Business Logic

SimulationManager

- scheduler: Scheduler
- frame: SimulationFrame
- tasks: List<Task>
- selectionPolicy: SelectionPolicy

...

+ generateRandomTasks():void

...

Scheduler

- servers: List<Server>
- maxNoServers: int
- maxTasksPerServer: int
- strategy: Strategy

+ changeStrategy(selectionPolicy: SelectionPolicy):void
+ dispatchTask(task:Task):void

...

Strategy

+ addTask(servers: List<Server>, task: Task):void

TimeStrategy

+ addTask(servers: List<Server>, task: Task):void

ShortestQueueStrategy

+ addTask(servers: List<Server>, task: Task):void

Model

Task

- ID: int
- arrivalTime: int
- serviceTime: int

...

Server

- tasks: BlockingQueue<Task>
- waitingPeriod: AtomicInteger

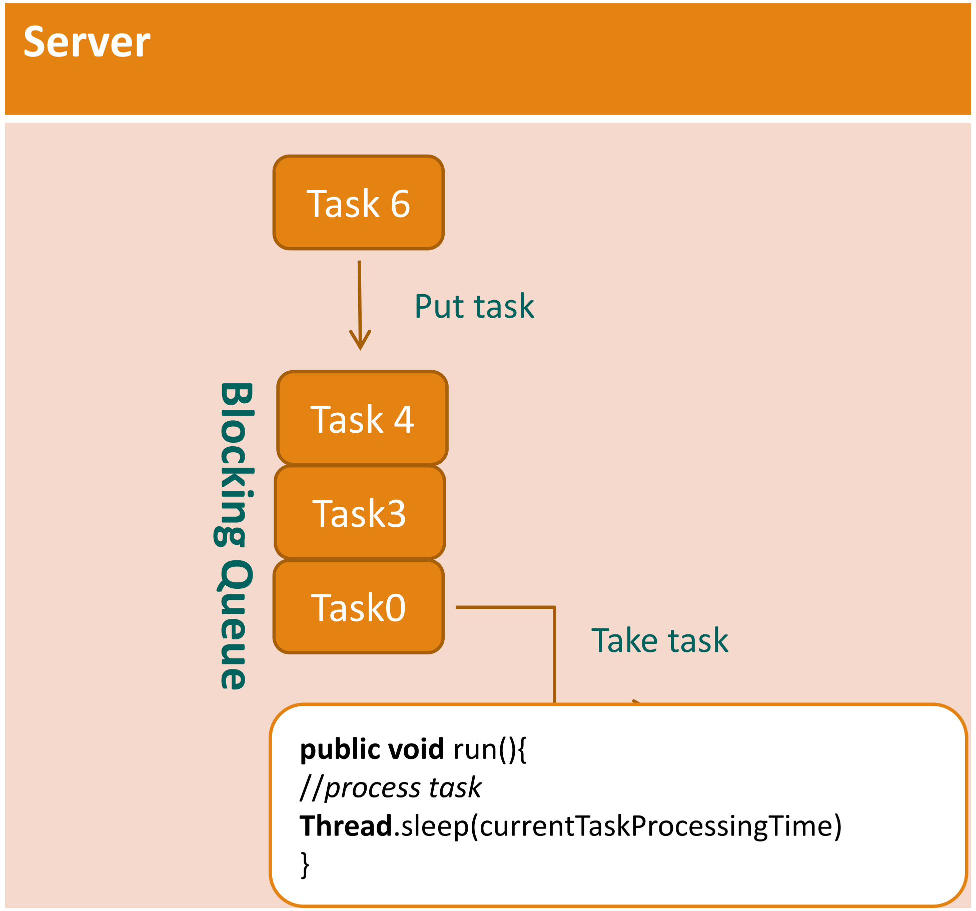
+ addTask(newTask: Task):void

...

What other classes, attributes, methods are necessary?

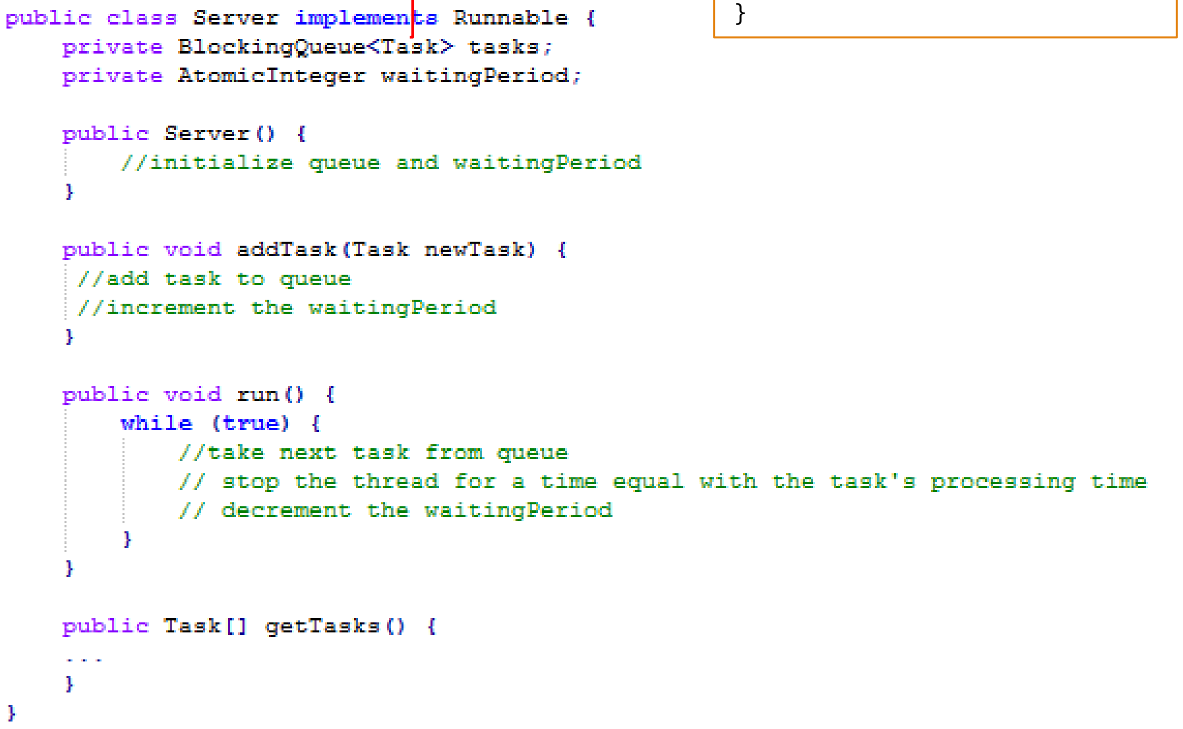
Implementation

- **Server**



```
public void run(){
//process task
Thread.sleep(currentTaskProcessingTime)
}
```

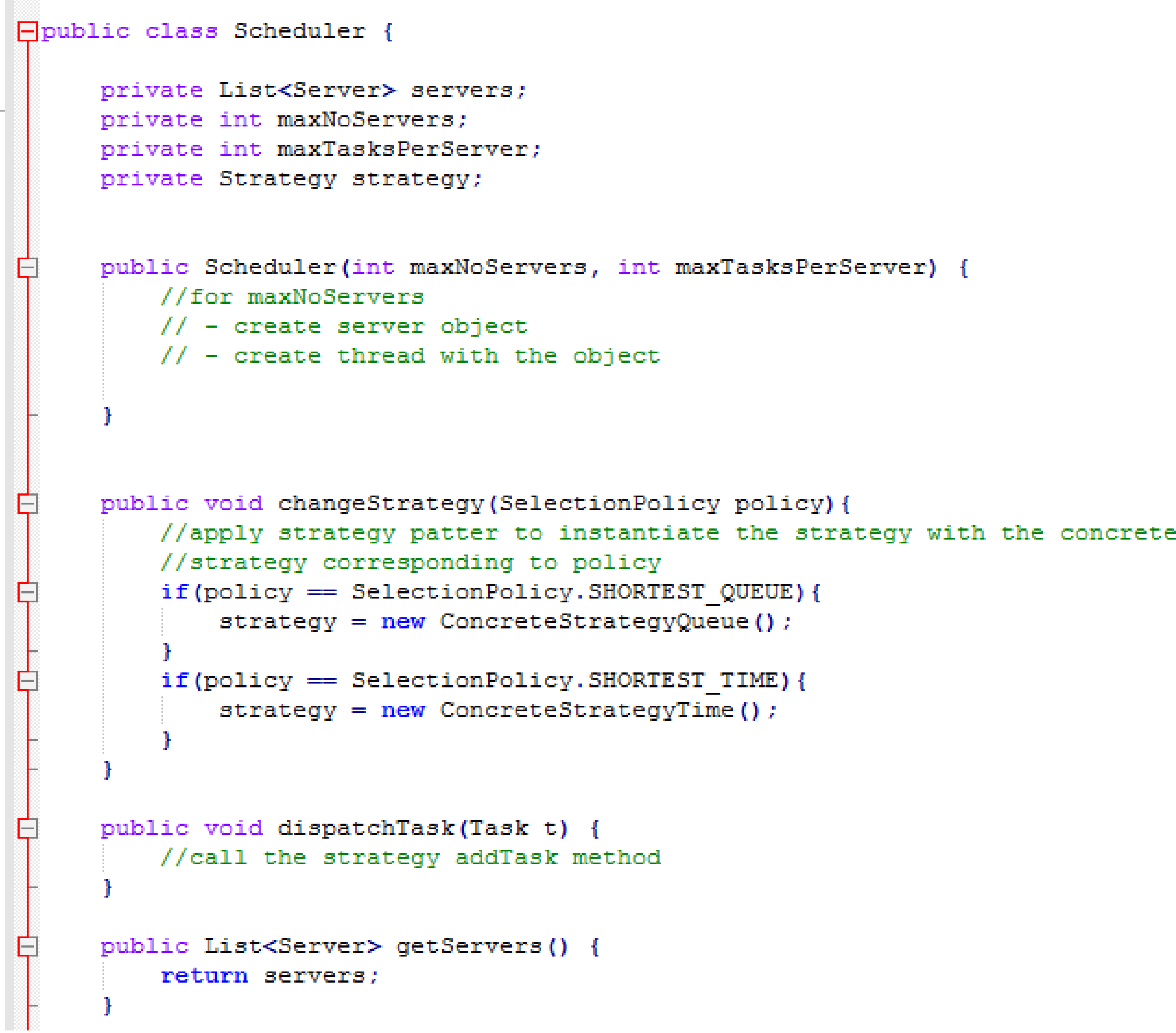
```
public class Task{
   private int arrivalTime;
   private int serviceTime;
   …
}
```



```
public class Server implements Runnable {
    private BlockingQueue<Task> tasks;
    private AtomicInteger waitingPeriod;

    public Server() {
        //initialize queue and waitingPeriod
    }

    public void addTask(Task newTask) {
     //add task to queue
     //increment the waitingPeriod
    }

    public void run() {
        while (true) {
            //take next task from queue
            // stop the thread for a time equal with the task's processing time
            // decrement the waitingPeriod
        }
    }

    public Task[] getTasks() {
    ...
    }
}
```

Implementation

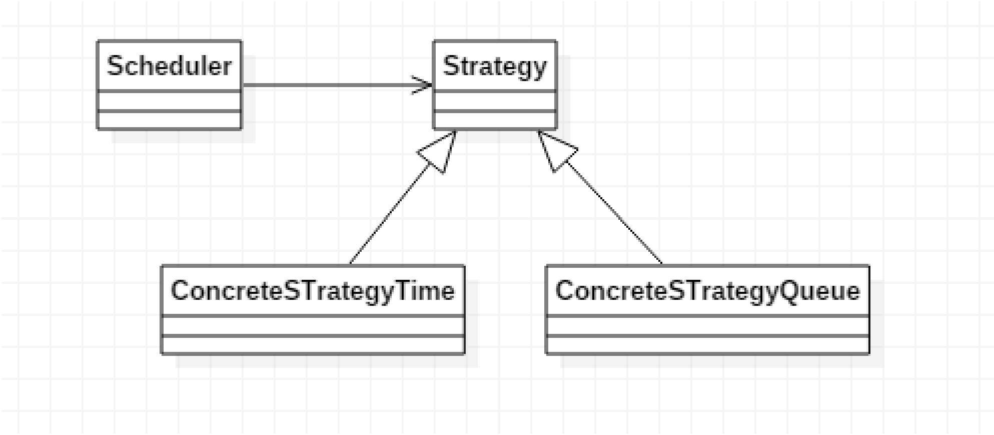
- **Scheduler**
    - Sends tasks to Servers according to the established strategy

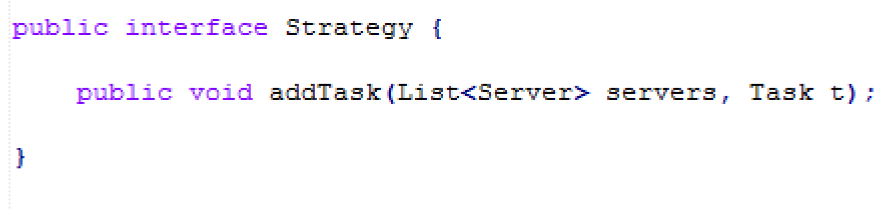


```java
public class Scheduler {

    private List<Server> servers;
    private int maxNoServers;
    private int maxTasksPerServer;
    private Strategy strategy;


    public Scheduler(int maxNoServers, int maxTasksPerServer) {
        //for maxNoServers
        // - create server object
        // - create thread with the object

    }


    public void changeStrategy(SelectionPolicy policy){
        //apply strategy patter to instantiate the strategy with the concrete
        //strategy corresponding to policy
        if(policy == SelectionPolicy.SHORTEST_QUEUE){
            strategy = new ConcreteStrategyQueue();
        }
        if(policy == SelectionPolicy.SHORTEST_TIME){
            strategy = new ConcreteStrategyTime();
        }
    }

    public void dispatchTask(Task t) {
        //call the strategy addTask method
    }

    public List<Server> getServers() {
        return servers;
    }
```

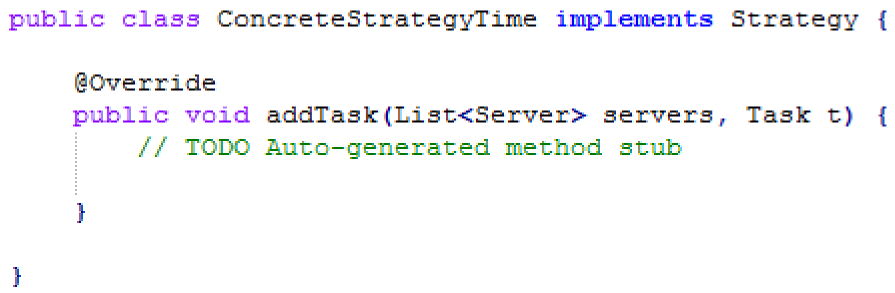
Implementation

- **Scheduler – Strategy Pattern**
  - Choose the policy to distribute clients

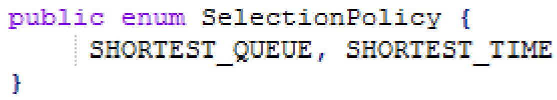




```
public interface Strategy {

    public void addTask(List<Server> servers, Task t);

}
```



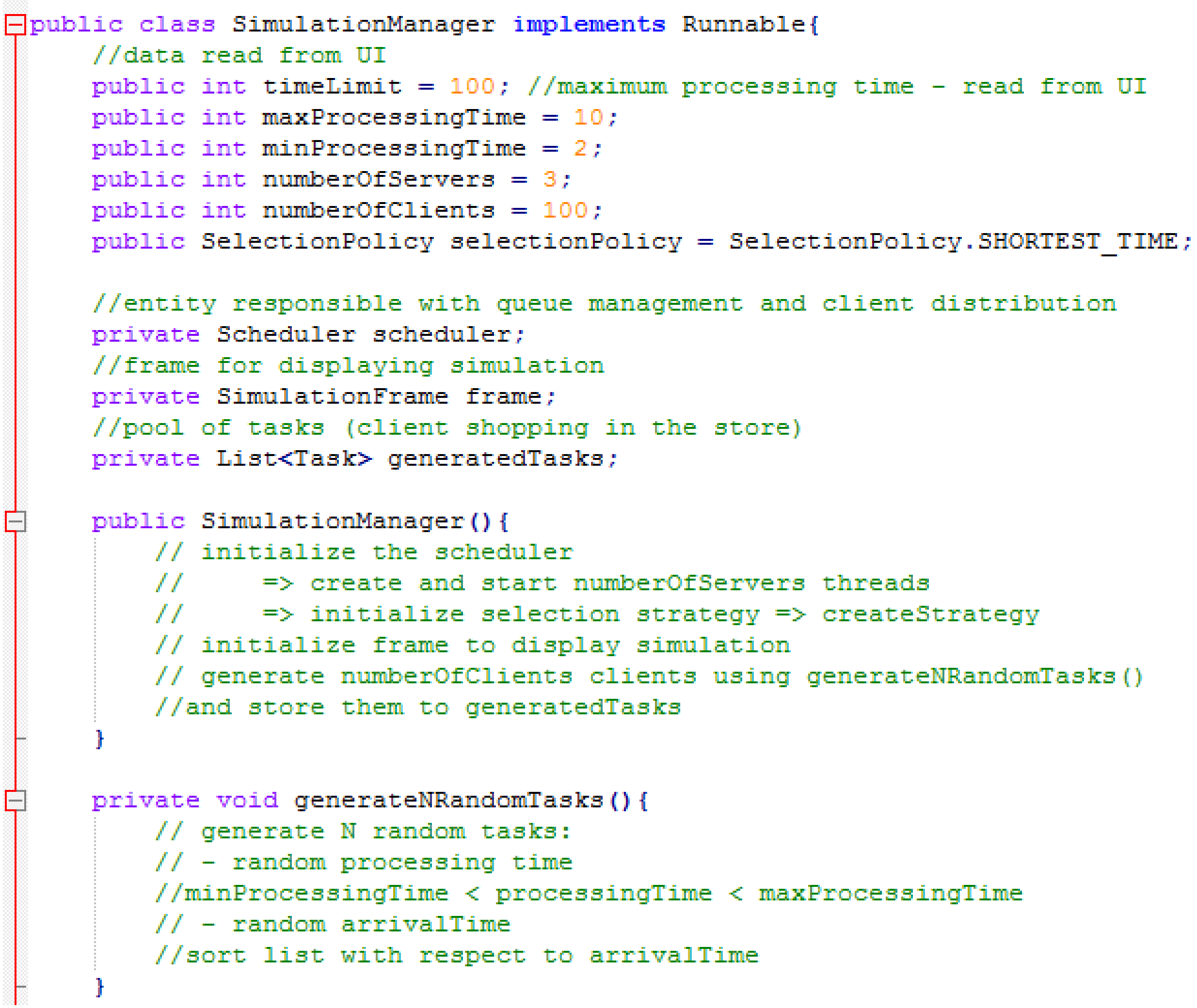
```
public class ConcreteStrategyTime implements Strategy {

    @Override
    public void addTask(List<Server> servers, Task t) {
        // TODO Auto-generated method stub

    }

}
```



```
public enum SelectionPolicy {
    SHORTEST_QUEUE, SHORTEST_TIME
}
```

Implementation

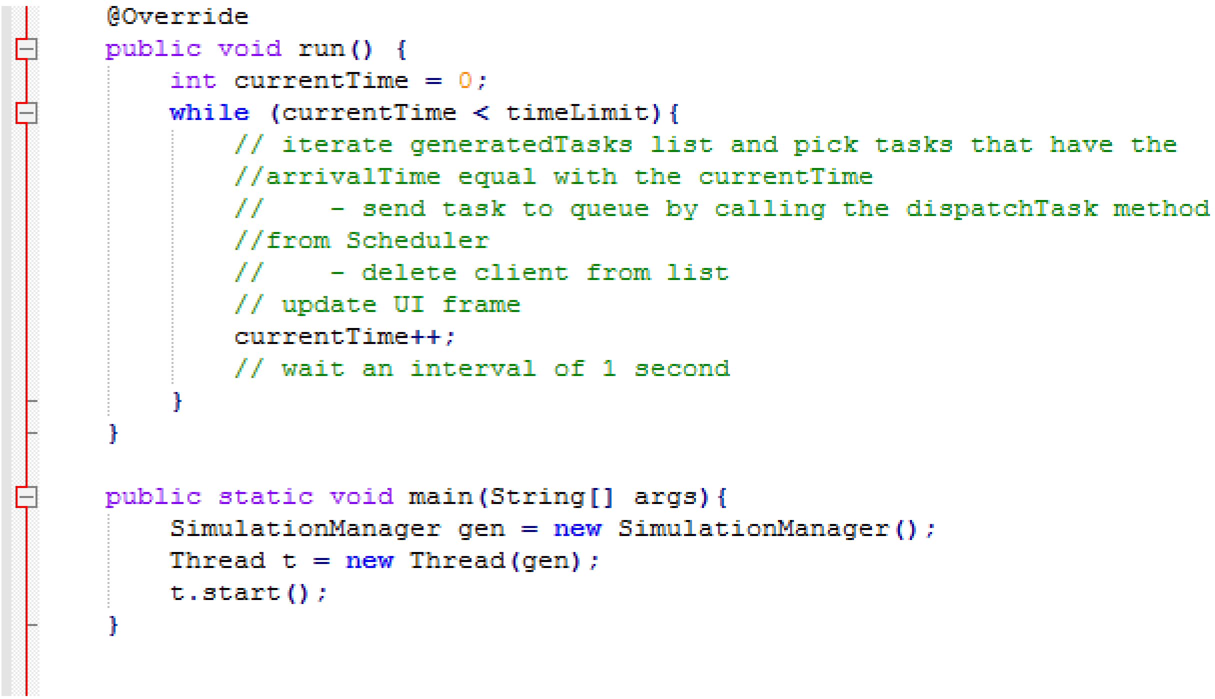
- **Simulation Manager**
  - Generates randomly the tasks with:
    - Arrival time
    - Service time
  - Contains simulation loop:
    - CurrentTime
    - Call scheduler to dispatch tasks
    - Update UI



```java
public class SimulationManager implements Runnable{
    //data read from UI
    public int timeLimit = 100; //maximum processing time - read from UI
    public int maxProcessingTime = 10;
    public int minProcessingTime = 2;
    public int numberOfServers = 3;
    public int numberOfClients = 100;
    public SelectionPolicy selectionPolicy = SelectionPolicy.SHORTEST_TIME;

    //entity responsible with queue management and client distribution
    private Scheduler scheduler;
    //frame for displaying simulation
    private SimulationFrame frame;
    //pool of tasks (client shopping in the store)
    private List<Task> generatedTasks;

    public SimulationManager(){
        // initialize the scheduler
        //      => create and start numberOfServers threads
        //      => initialize selection strategy => createStrategy
        // initialize frame to display simulation
        // generate numberOfClients clients using generateNRandomTasks()
        //and store them to generatedTasks
    }

    private void generateNRandomTasks(){
        // generate N random tasks:
        // - random processing time
        //minProcessingTime < processingTime < maxProcessingTime
        // - random arrivalTime
        //sort list with respect to arrivalTime
    }
```

Implementation

- **Simulation Manager**



```
@Override
public void run() {
    int currentTime = 0;
    while (currentTime < timeLimit){
        // iterate generatedTasks list and pick tasks that have the
        //arrivalTime equal with the currentTime
        //    - send task to queue by calling the dispatchTask method
        //from Scheduler
        //    - delete client from list
        // update UI frame
        currentTime++;
        // wait an interval of 1 second
    }
}

public static void main(String[] args){
    SimulationManager gen = new SimulationManager();
    Thread t = new Thread(gen);
    t.start();
}
```