DOCUMENTATION

ASSIGNMENT 2

STUDENT NAME: Muntian Mariana Ionela
GROUP: 30423

CONTENTS

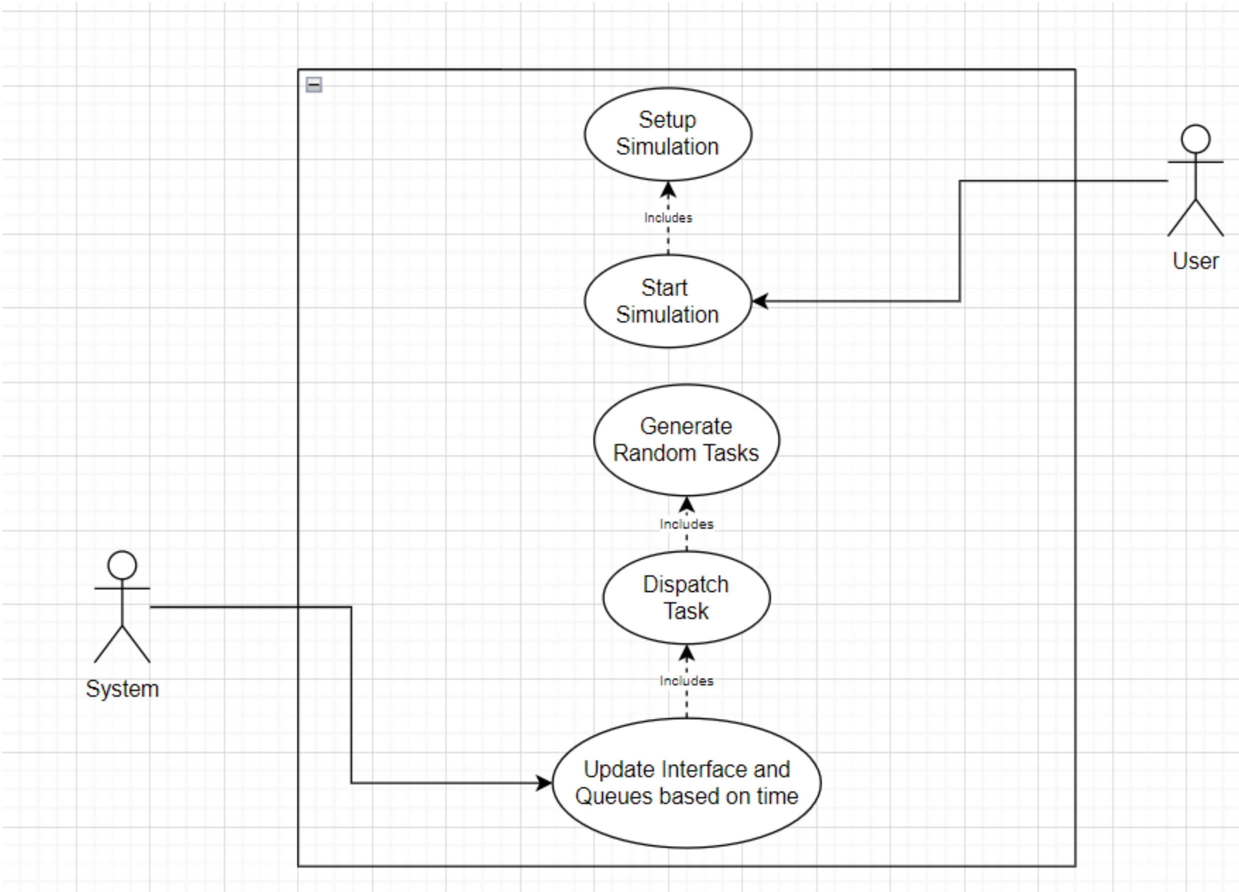
1. Assignment Objective

The main objective of this assignment is to 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*
2) *computing the average waiting time, average service time and peak hour .*

Sub-objective	***Description***	***Section***
1. Analyze the problem and identify requirements	*The problem is analyzed from the point of view of the use cases and the functional requirements.*	*2. Problem Analysis, Modeling, Scenarios, Use Cases*
2. Design the simulation application	*UML packages, UML diagram, the data structures and the algorithms that are used are presented.*	*3. Design*
3. Implement the simulation application	*The implementation of the interface and the structure of each class are described.*	*4. Implementation*
4. Test the simulation application	*A .txt file is creating, which displays the log of events of 3 tests cases.*	*5.Results*

2. Problem Analysis, Modeling, Scenarios, Use Cases



Reference: [Lecture_2_UML_Diagrams.pdf]

Use Case 0: Setup Application	
Brief Description	*This use case involves configuring the parameters for the simulation, including the number of queues, number of clients, maximum and minimum arrival times, maximum and minimum service times, and simulation interval.*
Parent Scenario	-
Actor(s)	*The User*
Priority	*High*
Trigger	*Open the application*
Pre-conditions	-
Post-conditions	*Start Simulation*
Basic Flow	*Step 1: Open the simulation application.* *Step 2: Configure the simulation parameters, including the number of queues, number of clients, maximum and minimum arrival times, maximum and minimum service times, and simulation interval.* *Step 3: Start the simulation.*
Alternate Flow(s)	*Step 1: If necessary, adjust additional settings or parameters.* *Step 2: Review the configured parameters.* *Step 3: Start the simulation.*
Exception Flow(s)	*Step 1: If an error occurs during the setup process, handle the error accordingly, changing the parameters accordingly.* *Step 2: Review the configured parameters.*

	Step 3: Start the simulation.

Use Case 1: Start Simulation	
Brief Description	*The user initiates the simulation process with the configured parameters.*
Parent Scenario	*Setup Application*
Actor(s)	*The User*
Priority	*High*
Trigger	*User clicks on the "Start Simulation" button.*
Pre-conditions	*Simulation application is open and configured.*
Post-conditions	*Simulation is running.*
Basic Flow	*Step 1: Click on the "Start Simulation" button.* *Step 2: Simulation manager generates random tasks based on the configured parameters.* *Step 3: Simulation manager dispatches tasks to servers based on the selected strategy.* *Step 4: Simulation manager updates the interface with task information.* *Step 5: Simulation continues until the time limit is reached.* *Step 6: Simulation manager computes average waiting time, average service time, and peak hour.*
Alternate Flow(s)	*-*
Exception Flow(s)	*Exception Flow 1: If an error occurs during the simulation process, handle the error accordingly.*

Use Case 2: Generate Random Tasks	
Brief Description	*Generates random tasks based on the configured parameters.*
Parent Scenario	*-*
Actor(s)	*System*
Priority	*High*
Trigger	*User clicks on the "Start Simulation" button.*
Pre-conditions	*Simulation application is open and configured.*
Post-conditions	*Random tasks are generated based on specified parameters.*
Basic Flow	*Step 1: System initiates the generation of random tasks.* *Step 2: System uses the parameters configured in the SimulationManager to generate random tasks.* *Step 3: System updates the interface with the generated task information.*
Alternate Flow(s)	*-*
Exception Flow(s)	*-*

Use Case 3: Dispatch Tasks	
Brief Description	*Dispatches tasks to servers based on the specified selection policy.*
Parent Scenario	*Start Simulation*
Actor(s)	*System*
Priority	*High*
Trigger	*User clicks on the "Start Simulation" button.*
Pre-conditions	*Random tasks are generated based on specified parameters.*
Post-conditions	*Tasks are dispatched to servers according to the selection policy.*
Basic Flow	*Step 1: System initiates the dispatching of tasks.* *Step 2: System uses the selection policy specified in the SimulationManager*

	to dispatch tasks to servers. *Step 3: System updates the interface to reflect the current state of the servers and tasks.*
Alternate Flow(s)	-
Exception Flow(s)	-

Use Case 4: Update Interface and Queues based on time	
Brief Description	*Updates the interface and server queues based on the passage of time during the simulation.*
Parent Scenario	*Dispatch Tasks*
Actor(s)	*System*
Priority	*High*
Trigger	*Simulation time advances.*
Pre-conditions	*Tasks are dispatched to servers according to the selection policy.*
Post-conditions	*Interface and server queues are updated to reflect the current state of the simulation.*
Basic Flow	*Step 1: System updates the simulation time.* *Step 2: System updates the interface to reflect the current time.* *Step 3: System updates the server queues based on task dispatching and execution.*
Alternate Flow(s)	-
Exception Flow(s)	-

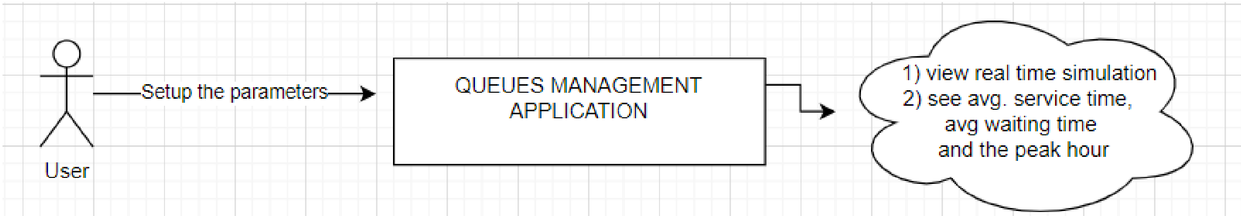
Functional requirements:

- *The simulation application should allow users to setup the simulation*
- *The application should validate user inputs for simulation parameters to ensure they are within acceptable ranges and formats.*
- *The simulation application should allow users to start the simulation*
- *The simulation application should display the real-time queues Evolution*

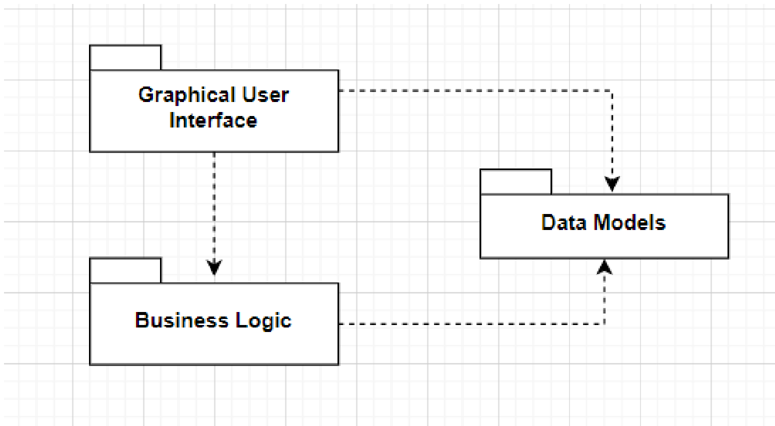
Non-Functional requirements:

- *The simulation application should be intuitive and easy to use by the user*
- *The system should be capable of handling a large number of tasks and servers efficiently.*
- *Response times for updating the interface and dispatching tasks should be minimal.*
- *The system should be scalable to accommodate future enhancements and modifications.*

3. Design

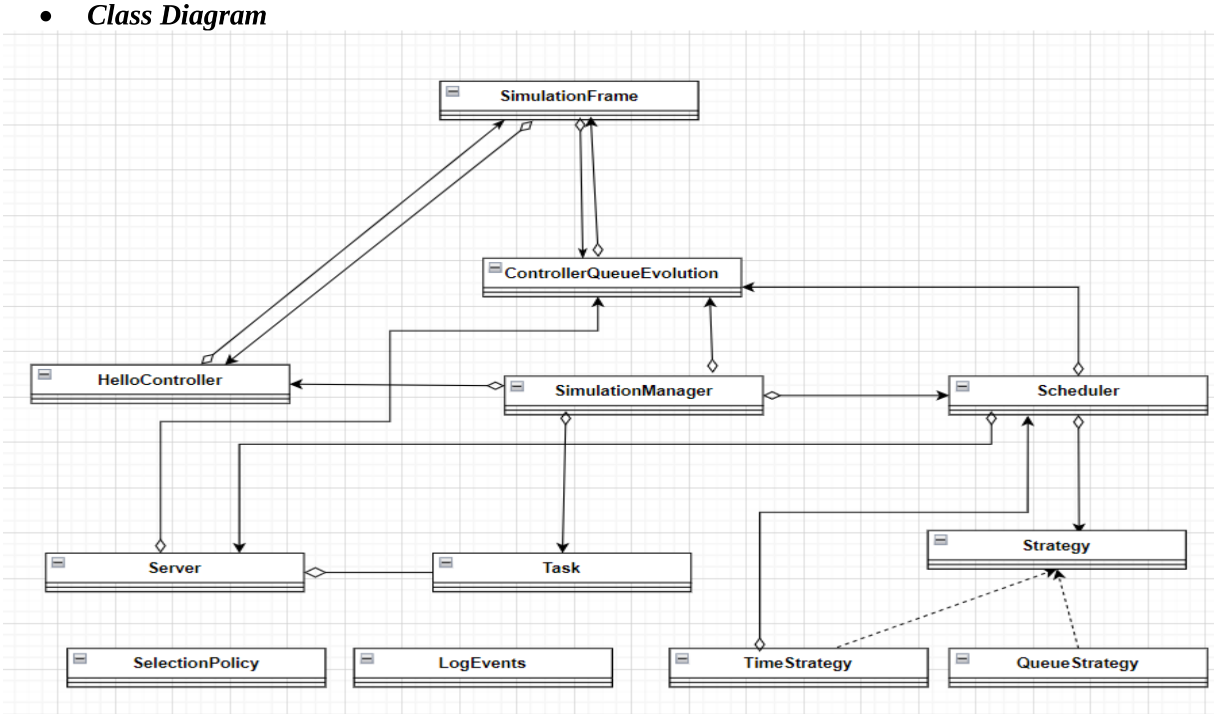


- ***UML packages***



Reference: https://dsrl.eu/courses/pt/materials/PT_2024_A1_S1.pdf

- ***Class Diagram***



- ***Used Data Structures***

1) ***Synchronized Methods***
   - *When a method is synchronized, it means that only one thread can execute that method at a time on a particular instance of the class. Other threads that try to execute the synchronized method on the same instance are blocked until the current thread releases the lock.*

```java
public synchronized void computePeakHour(int currentTime) {
    int nrPoepleInQueues = 0;
    for (Server s : this.serverList) {
        nrPoepleInQueues = nrPoepleInQueues + s.getTasksList().size();
    }

    if (nrPoepleInQueues > this.maxNrPeople) {
        setMaxNrPeople(nrPoepleInQueues);
        setPeakHour(currentTime);
    }
}
```

2) ***Atomicity***
   - *Atomic action cannot be interleaved => avoids thread interference*
   - *Reads and writes are atomic for reference variables and for most primitive variables(all types except long and double).*
   - *Reads and writes are atomic for all variables declared volatile (including long and double variables).*

```java
3 usages
private AtomicInteger waitingTimeServer;
```

3) ***Blocking Queues***
   - *BlockingQueues offer blocking operations, such as put() and take(), which block the calling thread until space becomes available in the queue (for put()) or until an element is available in the queue (for take()). This feature is crucial for efficient resource utilization and preventing busy-waiting.*

```java
private BlockingQueue<Task> tasksList;
```

```java
5 usages
private BlockingQueue<Server> serverList = new LinkedBlockingQueue<>();
```

- ***Defined Interfaces***

The Strategy interface represents a strategy for adding tasks to servers in a queues management application. It provides a contract for implementing different strategies for task allocation based on various criteria.

```java
public interface Strategy {
    2 usages   2 implementations   unknown
    public void addTask(List<Server> serverList, Task t) throws InterruptedException;
}
```

The TimeStrategy class implements the Strategy interface and represents a strategy based on minimizing waiting times in the queues. It selects the server with the minimum waiting time for task allocation.

The QueueStrategy class implements the Strategy interface and represents a strategy based on queue lengths. It selects the server with the smallest number of tasks in its queue for task allocation.

- ***Used Algorithms***

  The strategy for minimizing queue time is chosen by the user from the GUI interface.

a) *If the user selects the Shortest Queue Strategy, the task is directed to the queue with the fewest tasks.*

```
procedure addTask(serverList, t):
  min = serverList[0]  // Initialize min to the first server in the list
  for each server s in serverList:
    if size(s.tasksList) < size(min.tasksList):
      min = s  // Update min to the server with the smallest number of tasks
  minWaitingTime = min.waitingTimeServer + t.serviceTime
  t.waitingTime = minWaitingTime
  t.waitingTimeFinal = minWaitingTime
  min.addTask(t)
end procedure
```

b) *In the final version of the system, tasks are directed to the queue with the shortest waiting time. The waiting time of a queue is calculated as the cumulative sum of the service times of all tasks currently in the queue. This waiting time is decremented by one second for each second that elapses. For example, if a single client with a service time of 3 seconds arrives in the first second, the queue's waiting time is initially set to 3. However, by the second second, it decreases to 2. This mechanism ensures efficient task allocation and reflects real-time changes in queue status.*

```
addTask(serverList, t):
  minServer = serverList.peek()   // Get the first server in the list
  minWaitingTime = minServer.getWaitingTime() + t.getServiceTime()   // Calculate
waiting time for the first server

  // Iterate through each server in the list
  for each server s in serverList:
    waitingTime = s.getWaitingTime() + t.getServiceTime()   // Calculate waiting time
for the current server
    if waitingTime < minWaitingTime:
      minWaitingTime = waitingTime   // Update minimum waiting time
      minServer = s   // Update the server with the minimum waiting time

  t.setWaitingTime(minWaitingTime)   // Set waiting time for the task
  minServer.addTask(t)   // Add the task to the server with the minimum waiting time
```

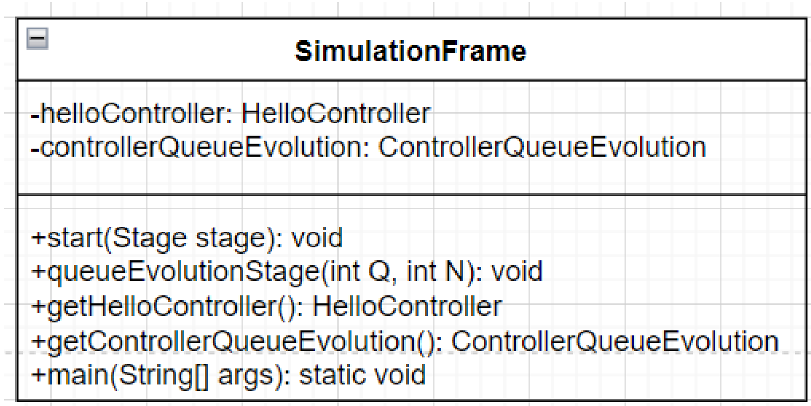
c) *If the user selects the Shortest Time Strategy, the task is directed to the queue with the shortest waiting time. If multiple queues have the same minimum waiting time, those queues are stored in another list, and the strategy is dynamically switched to the Shortest Queue Strategy. From this new list, the queue with the fewest tasks is selected.* ***(THIS WAS THE INITIAL VERSION, but it will be considered a future development).***

```
Procedure addTask(serverList, t):
  serverListMinimumWaitingTimes = Empty List

  // Step 1: Initialization
  minServer = serverList[0]
  minWaitingTime = minServer.waitingTimeServer + t.serviceTime

  // Step 2: Finding the Minimum Waiting Time Server
  for each server in serverList:
    waitingTime = server.waitingTimeServer + t.serviceTime

    if waitingTime < minWaitingTime:
      minWaitingTime = waitingTime
      minServer = server
      serverListMinimumWaitingTimes.clear()
      serverListMinimumWaitingTimes.add(server)
    else if waitingTime == minWaitingTime:
      serverListMinimumWaitingTimes.add(server)

  // Step 3: Updating Task Information
  t.waitingTime = minWaitingTime
  t.waitingTimeFinal = minWaitingTime

scheduler.setServerListMinimumWaitingTimes(serverListMinimumWaitingTimes)

  // Step 4: Task Allocation
  if size of serverListMinimumWaitingTimes is 1:
    minServer.addTask(t)
  else:
    scheduler.changeStrategy(SelectionPolicy.SHORTEST_QUEUE)
    scheduler.dispatchTask(t)
```

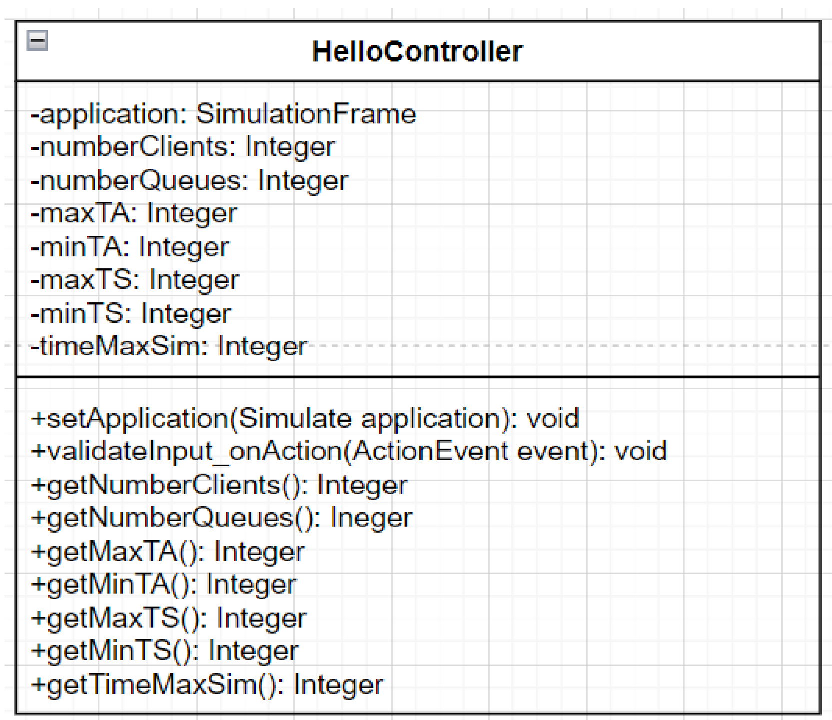
4 Implementation

I. GUI Package

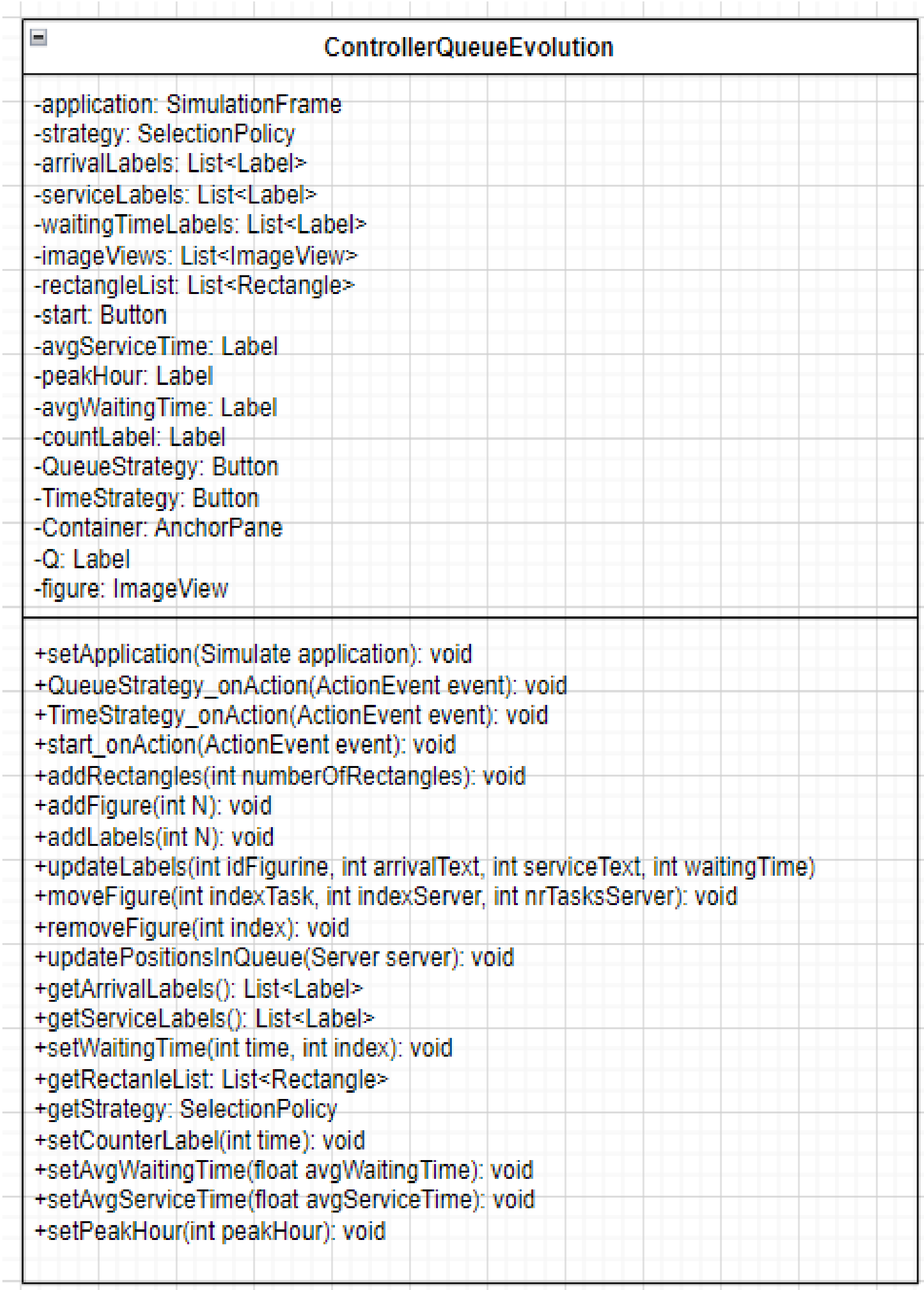
1) Simulation Frame Class



2) HelloController Class

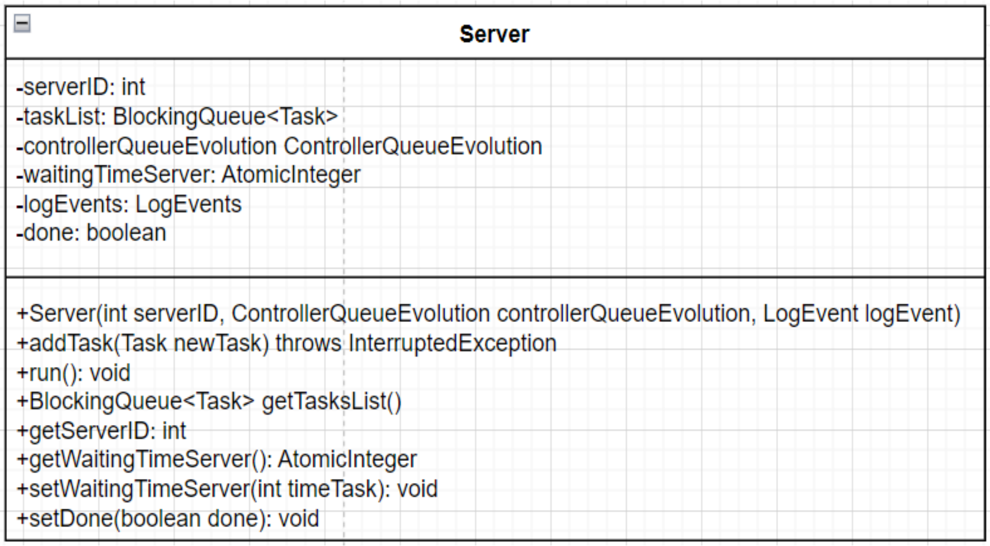


3) ControllerQueueEvolution Class

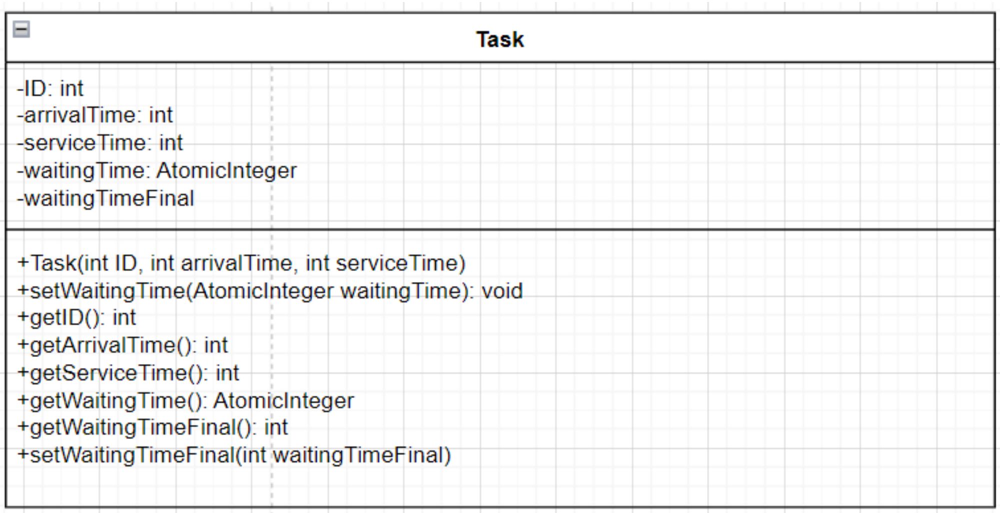


II. Model Package

1) Server



2) Task



III. BusinessLogic Package

1) SimulationManager Class

SimulationManager

-timeLimit: int
-minTimeService: int
-maxTimeService: int
-minTimeArrival: int
-maxTimeArrival: int
-nrQueues: int
-nrClients: int
-selectionPolicy: SelectionPolicy
-controllerQueueEvolution: ControllerQueueEvolution
-helloController: HelloController
-generatedTasks: List<Taks>
-scheduler: Scheduler
-logEvents: LogEvents

+SimulationManager(SimulationFrame simulationFrame)
+generateTasks(): void
+run(): void
+stopSim(): void
+computeAverageWaitingTime(): void
+computeAverageServiceTime(): void

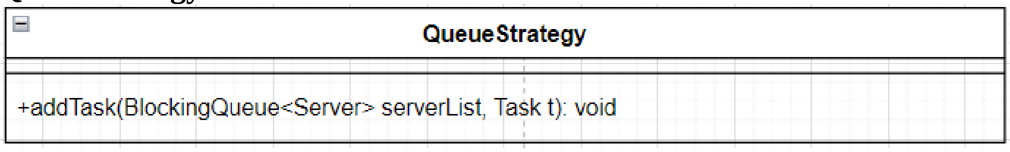
2) Scheduler Class

Scheduler

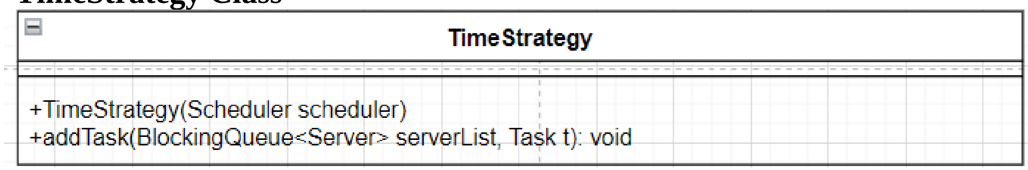
-serverList: BlockingQueue<Server>
-maxNoServers:int
-strategy: SelectionPolicy
-myStrategy: Strategy
-controllerQueueEvolution: ControllerQueueEvolution
-logEvents: LogEvents
-maxNrPoeple: int
-peakHour: int

+Scheduler(int maxNoServers, ControllerQueueEvolution controllerQueueEvolution, LogEvents logEvents)
+changeStrategy(SelectionPolicy selectionPolicy): void
+dispatchTask(Task t): void
+getServerList(): BlockingQueue<Server>
+getServerListMinimumWaitingTimes(): ConcurrentLinkedQueue<Server>
+setServerListMinimumWaitingTimes(ConcurrentLinkedQueue<Server> serverListMinimumWaitingTimes): void
+computePeakHour(int currentTime): void
+setPeakHour(int peakHour): void
+setMaxNrPeople(int maxNrPoeple): void
+getPeakHour(): int

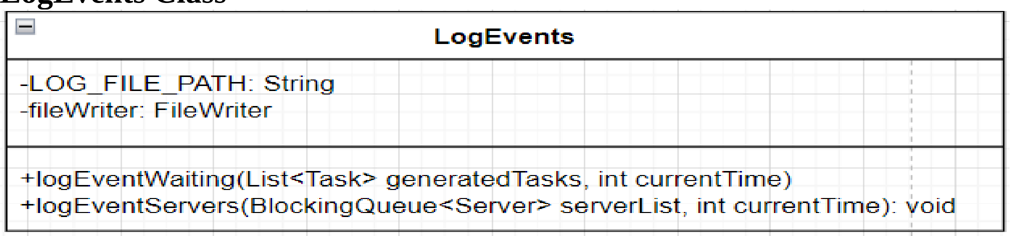
3) QueueStrategy Class



4) TimeStrategy Class



5) LogEvents Class



5. Results

- **Test 1**

  - N=4
  - Q=2
  - Tsim(max)=60 seconds
  - Tarr(min)=2, Tarr(max)=30 seconds
  - Tser(min)=2, Tser(max)=4 seconds

```
Time 0
Waiting Clients: (1 , 30 , 2) (3 , 8 , 2) (0 , 25 , 3) (2 , 11 , 4)
Queue 0: closed
Queue 1: closed

Time 1
Waiting Clients: (1 , 30 , 2) (3 , 8 , 2) (0 , 25 , 3) (2 , 11 , 4)
Queue 0: closed
Queue 1: closed

Time 2
Waiting Clients: (1 , 30 , 2) (3 , 8 , 2) (0 , 25 , 3) (2 , 11 , 4)
```

Queue 0: closed
Queue 1: closed

Time 3
Waiting Clients: (1 , 30 , 2) (3 , 8 , 2) (0 , 25 , 3) (2 , 11 , 4)
Queue 0: closed
Queue 1: closed

Time 4
Waiting Clients: (1 , 30 , 2) (3 , 8 , 2) (0 , 25 , 3) (2 , 11 , 4)
Queue 0: closed
Queue 1: closed

Time 5
Waiting Clients: (1 , 30 , 2) (3 , 8 , 2) (0 , 25 , 3) (2 , 11 , 4)
Queue 0: closed
Queue 1: closed

Time 6
Waiting Clients: (1 , 30 , 2) (3 , 8 , 2) (0 , 25 , 3) (2 , 11 , 4)
Queue 0: closed
Queue 1: closed

Time 7
Waiting Clients: (1 , 30 , 2) (3 , 8 , 2) (0 , 25 , 3) (2 , 11 , 4)
Queue 0: closed
Queue 1: closed

Time 8
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3) (2 , 11 , 4)
Queue 0: (3 , 8 , 2)
Queue 1: closed

Time 9
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3) (2 , 11 , 4)
Queue 0: (3 , 8 , 2)
Queue 1: closed

Time 10
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3) (2 , 11 , 4)
Queue 0: closed

Queue 1: closed

Time 11
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: (2 , 11 , 4)
Queue 1: closed

Time 12
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: (2 , 11 , 4)
Queue 1: closed

Time 13
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: (2 , 11 , 4)
Queue 1: closed

Time 14
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: (2 , 11 , 4)
Queue 1: closed

Time 15
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: closed
Queue 1: closed

Time 16
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: closed
Queue 1: closed

Time 17
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: closed
Queue 1: closed

Time 18
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: closed
Queue 1: closed

Time 19
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: closed
Queue 1: closed

Time 20
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: closed
Queue 1: closed

Time 21
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: closed
Queue 1: closed

Time 22
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: closed
Queue 1: closed

Time 23
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: closed
Queue 1: closed

Time 24
Waiting Clients: (1 , 30 , 2) (0 , 25 , 3)
Queue 0: closed
Queue 1: closed

Time 25
Waiting Clients: (1 , 30 , 2)
Queue 0: (0 , 25 , 3)
Queue 1: closed

Time 26
Waiting Clients: (1 , 30 , 2)
Queue 0: (0 , 25 , 3)
Queue 1: closed

```
Time 27
Waiting Clients:  (1 , 30 , 2)
Queue 0:  (0 , 25 , 3)
Queue 1: closed

Time 28
Waiting Clients:  (1 , 30 , 2)
Queue 0: closed
Queue 1: closed

Time 29
Waiting Clients:  (1 , 30 , 2)
Queue 0: closed
Queue 1: closed

Time 30
Waiting Clients:
Queue 0:  (1 , 30 , 2)
Queue 1: closed

Time 31
Waiting Clients:
Queue 0:  (1 , 30 , 2)
Queue 1: closed

Time 32
```

- **Test 2**
  - N=50
  - Q=5
  - Tsim(max)=60 seconds
  - Tarr(min)=2, Tarr(max)=40 seconds
  - Tser(min)=1, Tser(max)=7 seconds

  -see repository (it was too long)

- **Test 3**
  - N=1000 seconds

- Q=20
- Tsim(max)=200 seconds

- Tarr(min)=10, Tarr(max)=100 seconds
- Tser(min)=3, Tser(max)=9 seconds

-see repository (it was too long)

6. Conclusions

The project concludes that while threads can enhance efficiency in certain scenarios, their implementation demands considerable developer time and effort. Additionally, the presence of unpredictable errors further complicates the development process. This underscores the importance of carefully considering the trade-offs between performance benefits and development complexity when deciding whether to employ multithreading in a project.

7. Bibliography

1. https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
2. https://www.tutorialspoint.com/java/util/timer_schedule_period.htm
3. https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html