

**FUNDAMENTAL PROGRAMMING TECHNIQUES**

**ASSIGNMENT 2**

**QUEUES SIMULATOR**

**Student: Suciu Ioana Magdalena**

**Grupa: 30228**

Contents

[1. Obiectivul temei 2](#_Toc69350540)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 2](#_Toc69350541)

[3. Proiectare 3](#_Toc69350542)

[*3.1.* *Decizii de proiectare* 3](#_Toc69350543)

[*3.2.* *Diagrama UML* 4](#_Toc69350544)

[*3.3.* *Interfața grafică (GUI)* 6](#_Toc69350545)

[4. Implementare 6](#_Toc69350546)

[4.1. Clasa Task 6](#_Toc69350547)

[4.2. Clasa Server 6](#_Toc69350548)

[4.3. Clasa Scheduler 7](#_Toc69350549)

[4.4. Clasa SimulationManager 7](#_Toc69350550)

[4.5. Clasa Controller 8](#_Toc69350551)

[4.6. Clasele ConcreyeStrategyQueue si ConcreteStrategyTime 9](#_Toc69350552)

[4.7. Clasa View 9](#_Toc69350553)

[5. Rezultate 9](#_Toc69350554)

[........................... 9](#_Toc69350555)

[Peek Hour: 21 9](#_Toc69350556)

[Avreage wait: 1.625 9](#_Toc69350557)

[Avreage service: 8.3 9](#_Toc69350558)

[........................... 9](#_Toc69350559)

[6. Concluzii 10](#_Toc69350560)

[7. Bibliografie 11](#_Toc69350561)

[1. Lesson: Concurrency (The Java™ Tutorials > Essential Classes) (oracle.com) 11](#_Toc69350562)

[2. java.util.Timer.schedule() Method - Tutorialspoint 11](#_Toc69350563)

[3. Java Thread Pool Example using Executors and ThreadPoolExecutor (javacodegeeks.com) 11](#_Toc69350564)

# **Obiectivul temei**

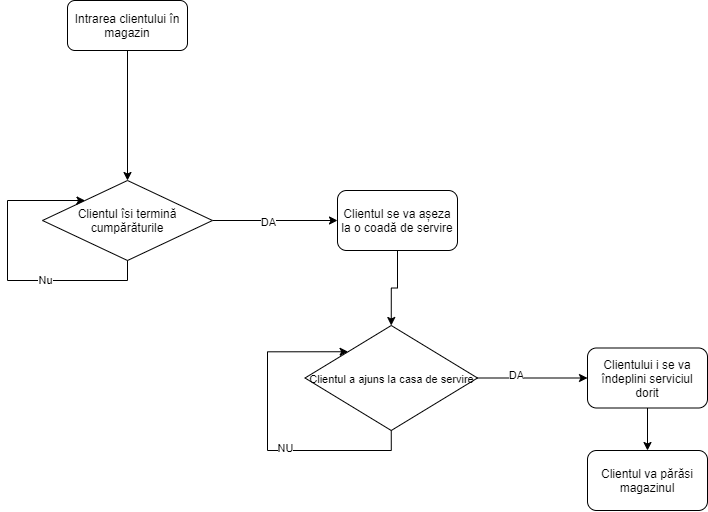
Obiectivul principal al acestei teme este reprezentat de proiectarea și implementarea unei aplicații de simulare care vizează analiza unui sistem bazat pe cozi, astfel în cadrul acestui sistem se determină si minimizează timpul de așteptare al clienților care vor urma a fi distribuiți unor servicii. În ceea ce privește obiectivele secundare, acestea sunt:

* Generarea unei serii de clienți care dispun de un ID (unic pentru fiecare client), timpul de așteptare înainte de a fi alocați unei cozi (tarrival) și timpul de servire (tservice), aceasta va avea loc la începutul simulării și se va baza pe input-ul furnizat de către utilizator.
* Generarea unui număr de cozi furnizat de utilizator
* Realizarea unei interfețe grafice care să faciliteze introducerea datelor
* Asignarea într-o manieră favorabilă a fiecărui client la o coadă

# **Analiza problemei, modelare, scenarii, cazuri de utilizare**

. Cozile sunt utilizate preponderent pentru modelarea domeniilor din lumea reală, în cazul acestei teme sunt utilizate pentru a realiza o distribuție a „clienților” care așteaptă primirea unui „serviciu” astfel încât perioada de așteptare să fie cât mai scurtă. Acest sistem de cozi se poate interpreta ca fiind un ansamblu de perechi de forma coadă de servire – client care așteaptă realizarea unui serviciu, însă realizarea unei astfel de potrivire este modelată pe baza unor reguli: fiecărei coadă îi sunt asignați o serie de clienți, dar clientul are particularitatea de a-și „alege” coada la care va aștepta, această „alegere” fiind determinată de strategia după care este modelată repartiția.

Aspectul funcțional al aceste aplicații este generarea unei serii de clienți care, mai apoi, vor fi repartizați unei cozi de așteptare. Cu alte cuvinte, generarea fiecărui client corespunde sosirii acestuia în „magazin”. Într-un anumit moment al simulării clientul decide că a terminat cumpărăturile și ia un loc la una dintre cozile de servire, atunci când este primul client asociat acestei cozi i se va realiza serviciul dorit, acest fiind caracterizat de o altă perioada temporală. Datorită faptului că fiecare client dorește ca perioada de așteptare în coada de servire să fie unul cât mai scurt este necesară monitorizarea evoluției fiecărei cozi atât din punct de vedere al clienților aflați în așteptare, cât și al timpului total de servire per coadă. Întregul proces prin care va trece un client este ilustrat în Figură 1..

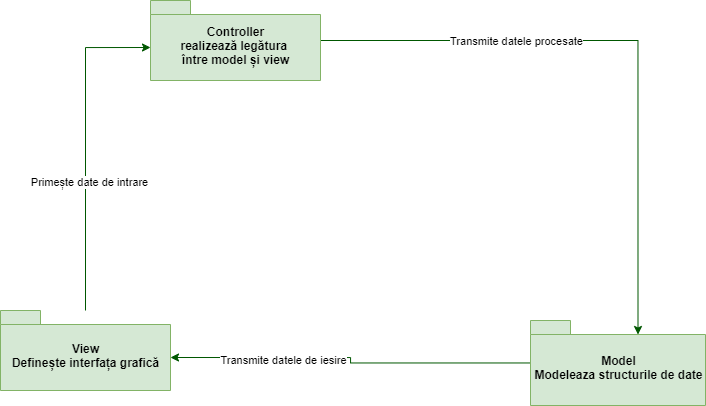


Figură 1 Use-chart

# **Proiectare**

## *Decizii de proiectare*

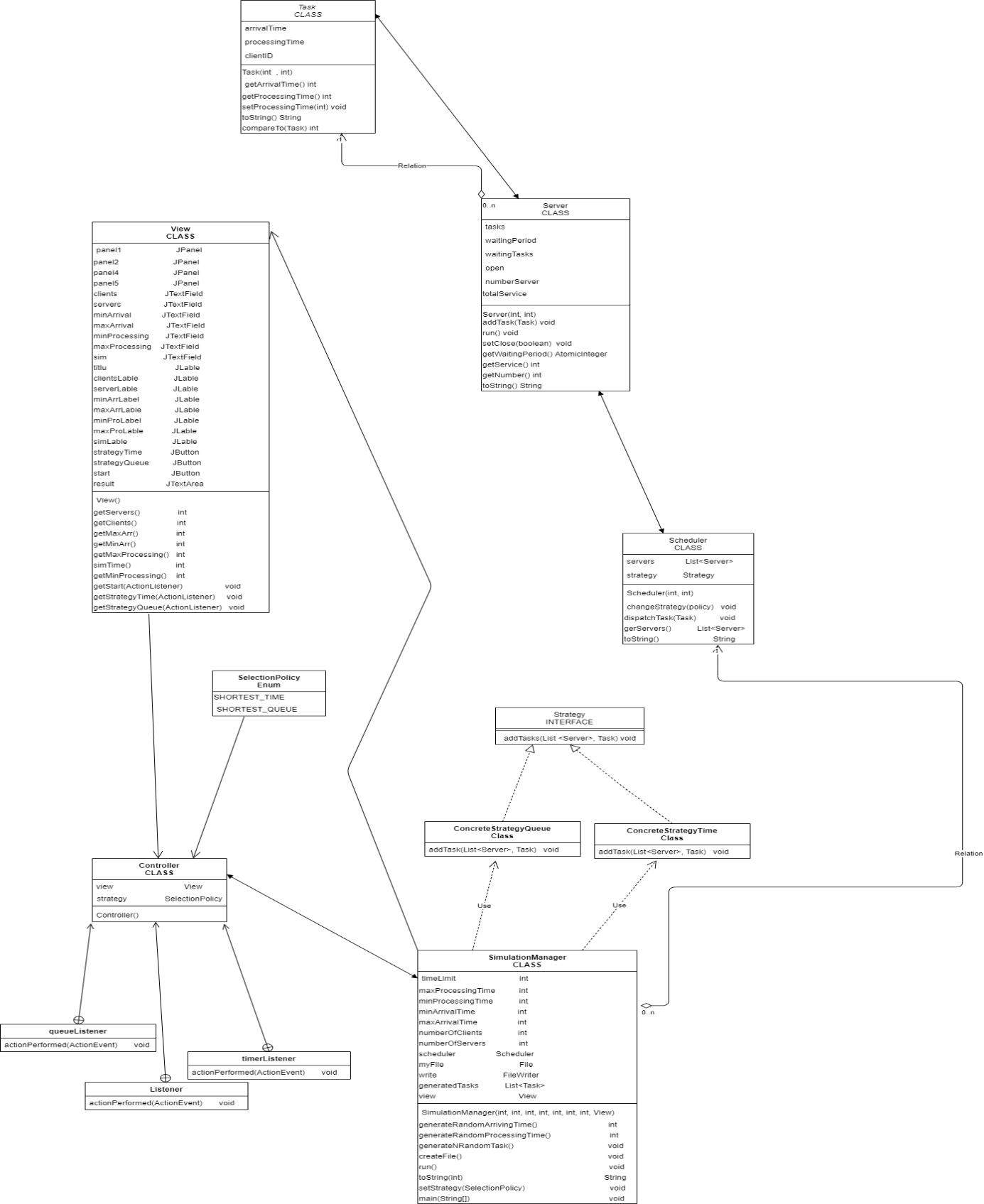
Pentru a realiza o abstractizare a problemei, am utilizat o proiectare orientă pe obiecte, fiecărei componente corespunzându-i o clasa. Cu toate acestea pentru a modulariza întreaga aplicație, am utilizat un model arhitectural de tipul Model, View, Controller, astfel fiecare pachet va avea o anumită funcționalitate asociată. După cum se poate observa în diagrama de pachete prezentată la Figură 2., între cele 3 pachete se realizează o corespondență, fiecare pachet depinzând de celălalt. În cadrul pachetului View este definită interfața grafică ce va prelua datele de input și i le va transmite Controllerului, care, la rândul său, va procesa aceste informații si le va transpune pachetului Model, acesta având sarcinile de a modifica structurile de date utilizate pentru ca aplicația să funcționeze conform dorințelor utilizatorului si de a transmite datele de ieșire pachetului View pentru afișare.



Figură 2 Diagrama de pachete

## *Diagrama UML*

UML(Unified Modeling Language) este notația internațională standard pentru analiza și proiectarea orientată pe obiecte. În diagrama prezentată la Figură 3., sunt reprezentate relațiile create între clase, printre aceste relații se numără și relațiile de asociere (agregare, compoziție), relațiile de dependență, relațiile de generalizare și relațiile de realizare (implementare).



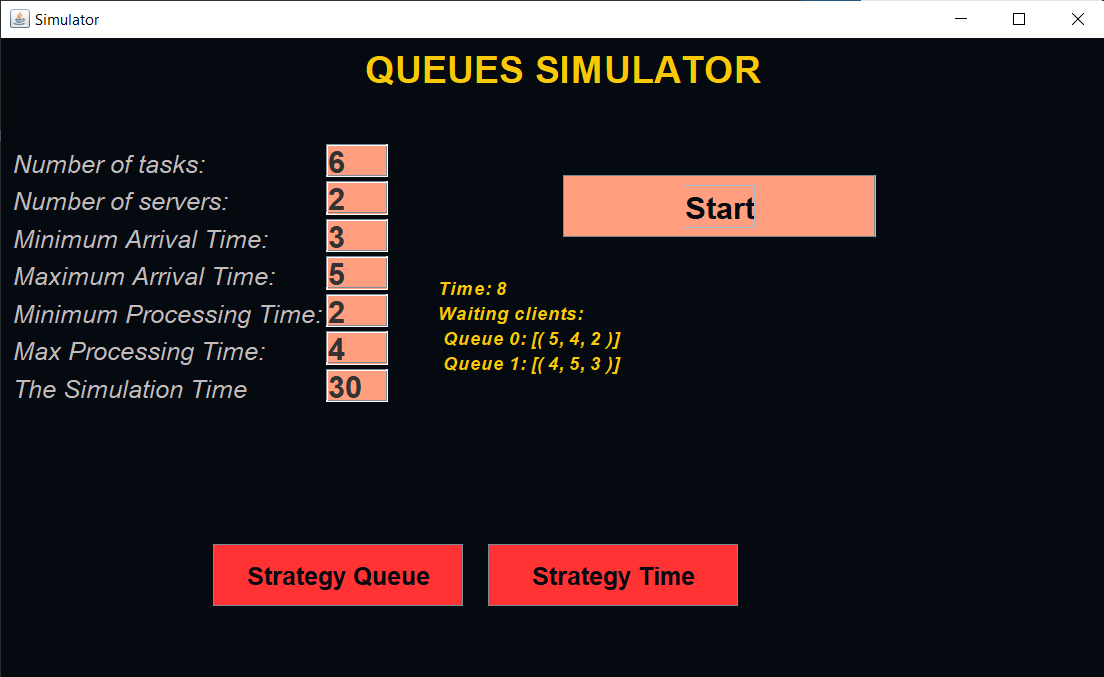
Figură 3 Diagrama UML

Descrierea claselor:

* Clasa Task este reprezentarea unui client care are ca si caracteristici un ID, un arrivalTime (momentul simulării în care va fi alocat unei cozi) și un serviceTime (durata de îndeplinire a serviciului dorit atunci când va fi primul în coadă), este de
* Clasa Server reprezintă coada propriu zisă la care se vor aloca taskurile
* Clasa Scheduler are rolul de a aloca un număr de servere și care va avea sarcina de a asigna fiecare task unui server în funcție de strategia dorită pentru optimizarea timpului de așteptare
* Clasa View definește interfața grafică unde se va introduce inputul și va fi prezentat outputul, această interfață este prezentată în Figură 4.
* Clasa Controller va realiza comunicarea între partea de back-end (clasele care sunt in pachetul Model) și partea de front-end (clasa View) prin transmiterea de date și apelarea de alte metode.
* Clasele queueListener, timeListener și Listener sunt subclase ale clasei Controller și care la apăsarea de butoane din interfață vor apela diferite metode din pachetul Model
* Clasa SimulationManager reprezintă centrul de control al aplicației, în această clasă setându-se toate regulile după care se va baza execuția.
* Interfața Strategy este implemetată de clasele ConcreteStrateyQueue și ConceteStrateyTime care vor stabili în ce mod vor fi alocate taskurile pe servere la momentul necesar
* Enumerația SelectionPolicy reprezintă tipurile de strategii disponibile

## *Interfața grafică (GUI)*

Interfața grafică realizează permite utilizatorului să interacționeze cu sistemul de simulare prin pictogramele grafice. Interfața creată în cadrul acestei teme este compusă dintr-un frame care conține atât etichete destinate facilitării introducerii datelor necesare, dar și zone de TextField în care utilizatorul aplicației va adăuga datele care vor deservii ca reguli ale simulării. În urma introducerii inputului necesar persoana care dorește realizarea simulării va trebui să selecteze strategie pe care se va baza expedierea clienților către cozile de servire prin apăsarea butonului aferent. Din momentul în care este apăsat butonul de start, aplicația se va lansa în execuție, iar în interfață vor apărea în mod succesiv datele pentru fiecare moment al simulării, date care vor persista pe ecran timp de o secundă, iar mai apoi zona de afișare va fi rescrisă cu datele de ieșire din următoarea secundă.



Figură 4. GUI

# **Implementare**

### Clasa Task

Această clasă are scopul de a reprezenta clientul prin caracteristicile sale (ID, arrivalTime, serviceTime). În ceea ce privește metodele regăsite în această clasă: getArrivalTime are scopul de a returna arrivalTimeul taskului și va fi apelată pentru stabilirea momentului în care taskul va fi afiliat unui server, getProcessingTime() va returna serviceTimeul, iar setProcessingTime() va seta noul serviceTime atunci când taskul se va afla pe prima poziție a cozii și timpul de servire va începe să scadă, toString() va returna caracteristicile taskului sub forma unui String. Este de remarcat faptul că această clasa extinde Comparable<Task> si drept urmare implementează metoda compareTo care va stabili ordinea dintre 2 taskuri in funcție de arrivalTime.

### Clasa Server

Această clasă va implementa interfața Runnable datorită motivului că pentru a realiza un sistem cât mai optim de simulare este necesar ca fiecărei cozi, server, să îi fie alocat un fir de execuție propriu. Se întâlnesc ca atribute o structură de tipul BlokinQueue și un AtomicInteger, aceste tipuri de date au fost alese din motive de thread safety. La fel ca și taskurile, fiecare server alocat are ca indicator un număr ce îi constituie unicitatea. Metode implementate: addTask (adaugă un task in coadă), setClose (setează coada ca fiind „deschisă”), getWaitingTasks (returnează numărul de taskuri care sunt în prezent în coadă), getWaitingPeriod (returnează waitingPeripdul prezent), getService (returnează timpul total de servire din pentru coadă), getNumber (returnează ID-ul serverului), toString (returnează su formă de șir de caractere caracteristicile cozii pentru afișare). Având în vedere că această clasă extinde Runneble a fost necesară implementarea metodei Run() care are modul de funcționare: cât timp pe server mai sunt taskuri ce așteaptă procesarea se va extrage primul task din coadă, timpul său de procesare va fi scăzut după fiecare secundă a simulării, iar când acesta ajunge la 0 respectivul task va fi eliminat din coadă, decrementându-se atât numărul de taskuri care așteaptă, cât și perioada de așteptare.



### Clasa Scheduler

În această clasă se vor inițializa atât serverele care sunt dispuse sub forma unei structuri de timp List, cât și threadurile care vor fi lansate în execuție prin threads.get(i).start.. În cadrul acestei clase se va fae stabilirea strategiei de alocare pe baza cerințelor utilizatorului prin metoda changeStrategy și se vor asigna taskurile unui server din listă pe baza strategiei alese prin metoda dispatchTask. Pe lângă aceste metode, au mai fost implementate și getServers() care returnează lista de servere create, dar si metoda toString() care are scopul de a returna sub forma unui String serverele din listă pentru afișare.

### Clasa SimulationManager

Din această clasă este realizat controlul întregii simulări. Atributele sale sunt preponderent datele de intrare furnizate de Controller: timpul de simulare (temeLimit), timpul maxim de servirte (maxProcessingTime), timpul minim de servire (minProcessingTime), timpul minim de alocare unui task la un server (minArrivaltime), timpul maxim de alocare al unui task la un server (maxArrivalTime), numărul de servere ce vor urma a fi create (numeberOfServers) și numărul de clienți care vor fi generați (numberofClients). Pe lângă aceste atribute se mai numără și un atribut de tipul Scheduler, va fi creat în constructorul de clasă, lista de taskuri care se vor genera în mod aleator, un atribut de tipul File și unul atribut de tipul FileWtiter pentru a scrie datele de ieșire într-un fișier care va fi creat, dar și un atribut de tipul View pentru a putea afișa datele de ieșire în interfață. Metoda de generare a taskurilor (generateNRandomTask()) are scopul de a crea un număr de taskuri aferent atributului numberOfClients într-un mod aleator, dar urmând anumite reguli: fiecare timp corespunzător unui task este necesar să fie inclus în intervalul corespunzător acestuia. Pentru o realizare cât mai optimă am creat metodele generateRandomArrivingTime() și generateRandomProcessingTime() care vor returna un număr aleator (se apelează Math.random) de tipul int care va fi calculat după formula:

((int)(Math.*random*()\*(maxArrivalTime-minArrivalTime+1))+minArrivalTime)

În urma generării tuturor taskurilor se va realiza o sortare bazată pe timpul la care fiecare task va intra într-o coadă, pentru a optimiza execuția programului, deoarece pentru fiecare încercare de alocare a unui task se va cunoaște faptul că primul task din listă este cel care va urma a fi primul alocat, nu mai este necesară parcurgerea listei la fiecare încercare. Metoda createFile() va crea un nou fișier cu numele „result.txt” unde vor fi stocate datele de ieșire, iar în caz de eroare se va arunca o excepție de tipul IOException. Dat fiind faptul că această clasă are funcția de control asupra simulării, este necesară și controlarea firelor de execuție corespunzătoare serverelor, din acest motiv este necesară implementarea interfeței Runnable și definirea metodei Run().

În această metodă se vor calcula atât ora de vârf (peek hour), cât și datele statistice referitoare la timpul mediu de servire și timpul mediu de așteptare. Înainte de a aloca taskurile unor cozi este necesară deschiderea fișierului creat pentru scriere. Cât timp simularea este încă în rulare ( currentTime este mai mic decât timeLimit) se va continua alocarea si procesarea taskurilor pe servere. Cât timp mai există takuri nealocate unui server se va compara timpul primului alocării primului task din listă la o coadă cu timpul curent al simulării, în cazul în care avem același timp taskul va fi expediat la un server și va fi scos din listă, urmând a fi testat noul prim task. Dacă în momentul curent nu vom mai avea taskuri care pot fi expediate pe servere se va calcula numărul de taskuri din servere pentru determinarea orei de vârf și se vor aduna timpii de așteptare curenți și timpii în care un task este scos din server pentru determinarea timpilor medii. Atunci când simularea va ajunge la final datele vor fi scrise în fișier, iar fișierul va fi închis.

### Clasa Controller

Această clasa va realiza corespondența dintre SimulationManager si View. În cadrul său există trei subclase care implementează interfața ActionListener, și deci implementează metoda actionPerformed. Fiecare astfel de subclasă corespunde unui buton aflat pe interfață, clasele queueListener și timeListener vor selecta tipul de strategie dorită, iar clasa Listener va corespunde butonului de start, la apăsarea căruia va începe execuția programului prin crearea unui obiect de tipul SimulationManager și crearea unui fir principal de execuție.

### Clasele ConcreyeStrategyQueue si ConcreteStrategyTime

Acestea vor implementa interfața Strategy prin definirea metodei addTask(), care va decide în funcție de fiecare tip de strategie pe ce server va urma a fi alocat taskul pentru a obține o minimizare a timpului de așteptare.

### Clasa View

În această clasă este implementată interfața grafică prin extinderea clasei JFrame. Sunt utilizate atribute de tipul JPanel, JLabel, JTextField (pentru introducerea inputului), JButton (pentru alegerea strategiilor și a pentru lansarea execuției) și JTextArea (pentru afișarea rezultatelor). Fiecare input introdus de utilizator va fi transmis clasei Controller printr-o metodă care va parsa atributul de tipul String la un int.

# **Rezultate**

|  |  |  |
| --- | --- | --- |
| *Test 1 V1 Strategy Queue* |  |  |
| N = 4  Q = 2  𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛𝑀𝐴𝑋= 60 seconds  [𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐼𝑁, 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐴𝑋] = [2, 30]  [𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐼𝑁,𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐴𝑋]= [2, 4] | | |

### ...........................

### Peek Hour: 21

### Avreage wait: 1.625

### Avreage service: 8.3

|  |  |  |
| --- | --- | --- |
| *Test 1 V2 Strategy Time* |  |  |
| N = 4  Q = 2  𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛𝑀𝐴𝑋= 60 seconds  [𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐼𝑁, 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐴𝑋] = [2, 30]  [𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐼𝑁,𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐴𝑋]= [2, 4] | | |

### ...........................

Peek Hour: 22

Avreage wait: 1.5916667

Avreage service: 6.31666660

|  |
| --- |
| *Test 2 V1 Strategy Queue* |
| N = 50  Q = 5  𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛𝑀𝐴𝑋= 60 seconds  [𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐼𝑁, 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐴𝑋] = [2, 40]  [𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐼𝑁,𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐴𝑋]= [1, 7] | |

Peek Hour: 6

Avreage wait: 8.783334

Avreage service: 2106.1833

*Test 2 V2 Strategy Time*

N = 50

Q = 5

𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛𝑀𝐴𝑋= 60 seconds

[𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐼𝑁, 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐴𝑋] = [2, 40]

[𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐼𝑁,𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐴𝑋]= [1, 7]

Peek Hour: 8

Avreage wait: 9.070001

Avreage service: 1673.2733

|  |
| --- |
| *Test 3 V1 Strategy Queue* |
| N = 1000  Q = 20  𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛𝑀𝐴𝑋= 200 seconds  [𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐼𝑁, 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐴𝑋] = [10, 100]  [𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐼𝑁,𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐴𝑋]= [3, 9]  Peek Hour: 99  Avreage wait: 142.42375  Avreage service: 7.9032968E7 | |

|  |
| --- |
| *Test 3 V2 Strategy Time* |
| N = 1000  Q = 20  𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛𝑀𝐴𝑋= 200 seconds  [𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐼𝑁, 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙𝑀𝐴𝑋] = [10, 100]  [𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐼𝑁,𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒𝑀𝐴𝑋]= [3, 9]  Peek Hour: 100  Avreage wait: 144.21  Avreage service: 9.2956416E7 | |

# **Concluzii**

După cum se poate observa în setadiul de testare al simulării, cu toate că pentru fiecare test s-au generat alte serii de clienți, însă după aceleași reguli, peek hour diferă cu o diferență de aproximativ 1, iar cazurile de avreage wait și avreage service au o variație relativ mică. Un alt aspect ce se poate observa este că, strategia Time este favorabilă în cazurile în care avem un număr relativ scăzut de clienți, iar strategia Queue este favorabilă în cazurile în care avem un număr mai ridicat de clienți.

# **Bibliografie**

### [Lesson: Concurrency (The Java™ Tutorials > Essential Classes) (oracle.com)](https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html)

### [java.util.Timer.schedule() Method - Tutorialspoint](https://www.tutorialspoint.com/java/util/timer_schedule_period.htm)

### [Java Thread Pool Example using Executors and ThreadPoolExecutor (javacodegeeks.com)](https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html)