**DOCUMENTAȚIE**

**QUEUES MANAGEMENT APPLICATION USING THREADS AND SYNCHRONIZATION MECHANISMS**

**TEMA 2**

NUME STUDENT: Filip Raul

GRUPA: 30225

# CUPRINS

￼

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](#_Toc95297886)

3. Proiectare 5

[4.](#_Toc95297888) Implementare 8

5. Rezultate 14

[6.](#_Toc95297890) Concluzii 16

[7.](#_Toc95297891) Bibliografie 16

# Obiectivul temei

Obiectivul acestei teme este implementarea și proiectarea unei aplicatii bazata pe un sistem de Q cozi care simuleaza venirea, pentru servicii, a unei serii de N clienti care asteapta in coada, ajung sa li se ofere serviciile, iar dupa indeplinirea serviciilor, pleaca de la coada. Trebuie analizat din punct de vedere statistic timpul de asteptare mediu, timpul de servire mediu si ora de varf ( cand sunt cei mai multi clienti care asteapta in toate cele Q cozi ).

Pentru realizarea obiectivului principal, trebuie să trecem printr-o serie de obiective secundare, după cum urmează:

* Analiza problemei, modelare, scenarii, cazuri de utilizare. În acest capitol se vor prezenta cerințele funcționale și cazurile de utilizare, precum și diagrama USE-CASE. (capitolul 2).
* Proiectarea. În acest capitol se va descrie diagrama UML de clase, structurile de date folosite (capitolul 3).
* Implementarea. În acest capitol se vor descrie metodele precum și atributele din fiecare clasă (capitolul 4).

# Analiza problemei, modelare, scenarii, cazuri de utilizare

In viata de zi cu zi, exista domenii in care pentru a te prezenta pentru prestarea unui serviciu, este nevoie de o programare. Desi aceste programari sunt menite sa scuteasca de aglomeratie numeroase centre si institutii, depinde foarte mult de cererea clientului, pentru ca daca acesta doreste prestarea mai multor servicii, coada la care este asezat clientul nostru este dorita a fi evitata. Astfel am creat un simulator care genereaza in mod aleator clienti care au un timp de sosire, un timp de procesare si un ID. Exista doua posibilitati prin care sa se filtreze aglomeratia generata:

1. SHORTEST TIME

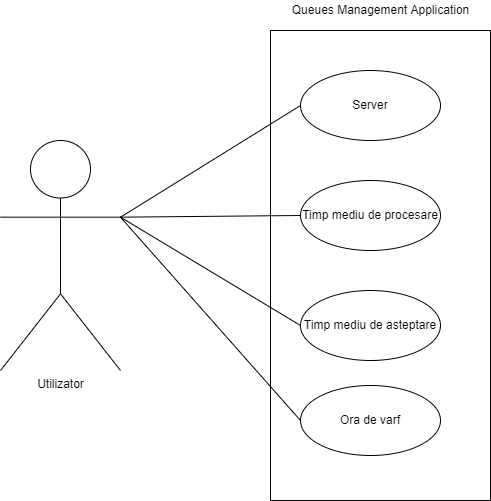
Prin aceasta metoda, in functie de timpul total de asteptare la o coada, este selectata coada cu timpul de asteptare cel mai mic, iar clientul este asezat acolo.

1. SHORTEST QUEUE

Prin aceasta metoda, in functie de numarul de persoane care astepata la o coada, este selectata coada cu cele mai putine persoane, iar clientul este asezat acolo.

O diagramă USE-CASE este o reprezentare grafică a posibilelor interacțiuni ale utilizatorului cu un sistem. O astfel de diagramă prezintă relația dintre utilizator și diversele posibilități de utilizare (reprezentate prin elipse sau cercuri). Utilizatorii sunt reprezentați prin omuleți. Aceste diagrame sunt înțelese mai ușor decât diagramele de clasă.

Exemplu USE-CASE:



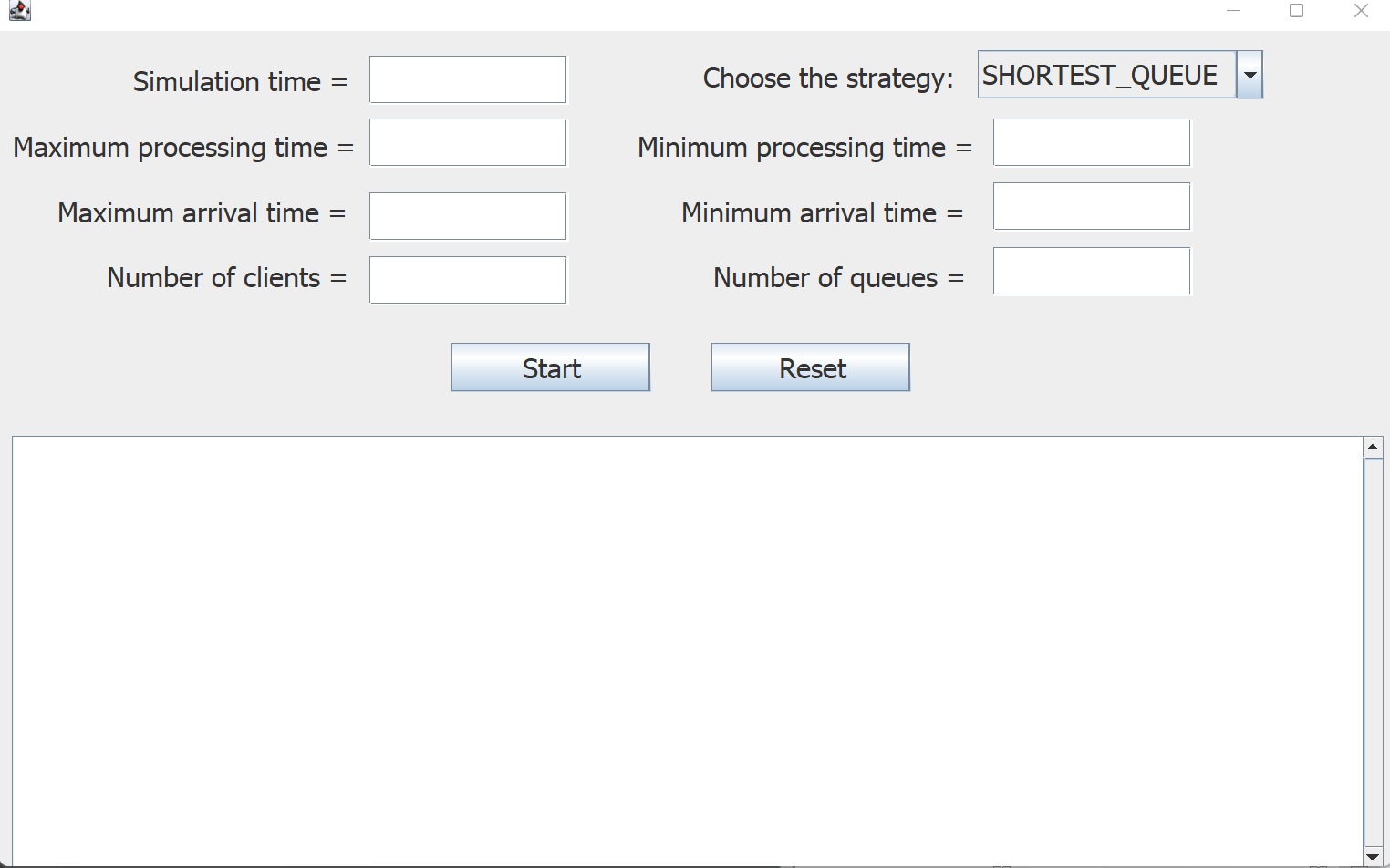
1. Utilizatorul intoduce simulation time-ul, maximum processing time-ul, minimum processing time-ul, maximum arrival time-ul, minimum arrival time-ul, number of clients, number of queues si se selecteaza tipul de strategie dorit: SHORTEST\_TIME sau SHORTEST\_QUEUE.
2. Utilizatorul apasă butonul de START care porneste executia simularii.
3. Clientii intra in cozi.
4. Se calculeaza timpul mediu de procesare, timpul mediu de asteptare si ora de varf.
5. Utilizatorul primește rezultatul in timp real, astfel se poate observa cum la inceput cozile se vor umple treptat, iar la final acestea se vor goli dupa ce toti clientii din lista de asteptare au fost inserati si dupa ce va fi finalizata prestarea de servicii din toate cozile.
6. Daca se doreste resetarea simularii, utilizatorul apasa butonul de RESET, iar toate datele introduse vor fi sterse, inclusiv istoricul cozilor. Pentru ca informatia sa nu se piarda definitiv, utilizatorul are un fisier numit „projectFile.txt” unde se pot vedea informatiile ce au fost afisate in interfata grafica.

# Proiectare

Structurile de date folosite sunt: ArrayList<>, BlockingQueue<>(), AtomicInteger.

Am implementat metoda compareTo (Object o) a interfeței Comparable, precum și metoda actionPerformed(ActionEvent e) a interfeței ActionListener.

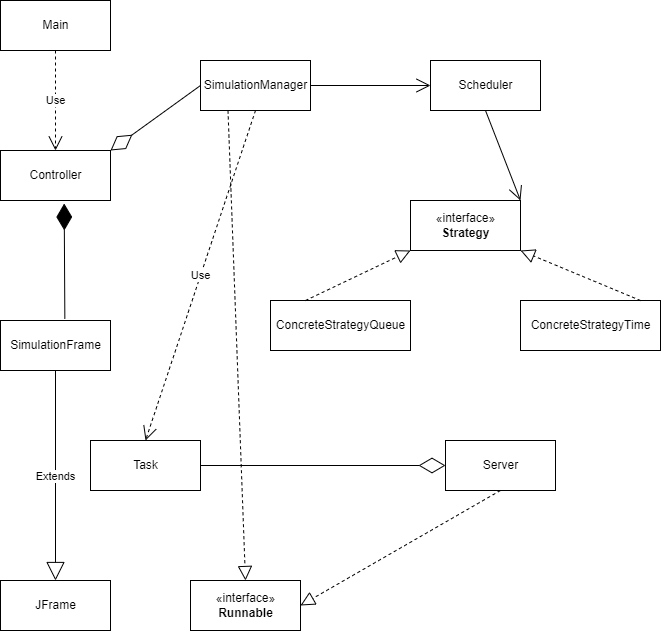
În momentul rulării aplicației, pe ecran va apărea interfața grafică cu cele doua butoane de START si de RESET, precum si field-urile care trebuie completate de catre utilizator.



Primul text field reprezintă câmpul text corespunzător introducerii timpului de simulare. Al doilea si al treilea text field reprezintă câmpul text corespunzător introducerii timpului maxim de procesare si al timpului minim de procesare pentru fiecare client. Al patrulea si al cincilea text field reprezinta câmpul text corespunzător introducerii timpului maxim de sosire si timpului minim de sosire in coada pentru fiecare client. In al saselea text field reprezinta câmpul text corespunzător introducerii numarului de clienti pentru simularea curenta. Al saptelea text field reprezinta câmpul text corespunzător introducerii numarului de cozi pentru simularea curenta. In dreptul fiecarui text field-uri avem câte un JLabel pentru a-i indica utilizatorului ce trebuie să introducă. Cele 2 butoane servesc pentru inceperea simularii si pentru resetarea datelor din interfata. Rezultatele vor fi afișate in timp real in JtextArea-ul din partea de jos a interfetei. Exista un scroll de tipul JscrollPane pentru cand JtextArea „se umple” si dorim sa vedem informatiile despre cozi din ultimii timpi de simulare.

Prin intermediul interfeței grafice se realizează legătura cu utilizatorul. Interfața grafică/ GUI (Graphical User Interface) este o interfață cu utilizatorul bazată pe un sistem de afișaj ce utilizează elemente grafice. Pentru a putea prezenta toate informațiile și acțiunile disponibile, un GUI oferă pictograme și indicatori vizuali, în contrast cu interfețele bazate pe text, care oferă doar nume de comenzi (care trebuie tastate) sau navigația text.

Diagrama UML este formată din10 clase si 2 interfete:



Clasele Server si SimulationManager implementeaza interfata Runnable. Clasele ConcreteStrategyTime si ConcreteStrategyQueue implementeaza interfata Strategy. Clasa SimulationFrame mosteneste clasa JFrame. Se poate observa ca intre clasele Task si Server exista o relatie de agregare deoarece un Server are un BlockingQueue<Task>.

Putem observa că în diagramă se aplică regulile de vizibilitate pentru atribute și pentru metodele din clase. Fiecare modificator de accees este reprezentat printr-un simbol:

* Public (+): vizibil oriunde în proiect.
* Private (-): vizibil doar în clasa din care apaține.
* Protected (#): vizibil în clasele moștenite și în pachet.
* Package (~): vizibil în pachet.

Model-view-controller (MVC) este un model arhitectural folosit în ingineria software. Cu ajutorul controller-ului, al model-ului sau al view-ului putem manipula următoarele elemente: datele.

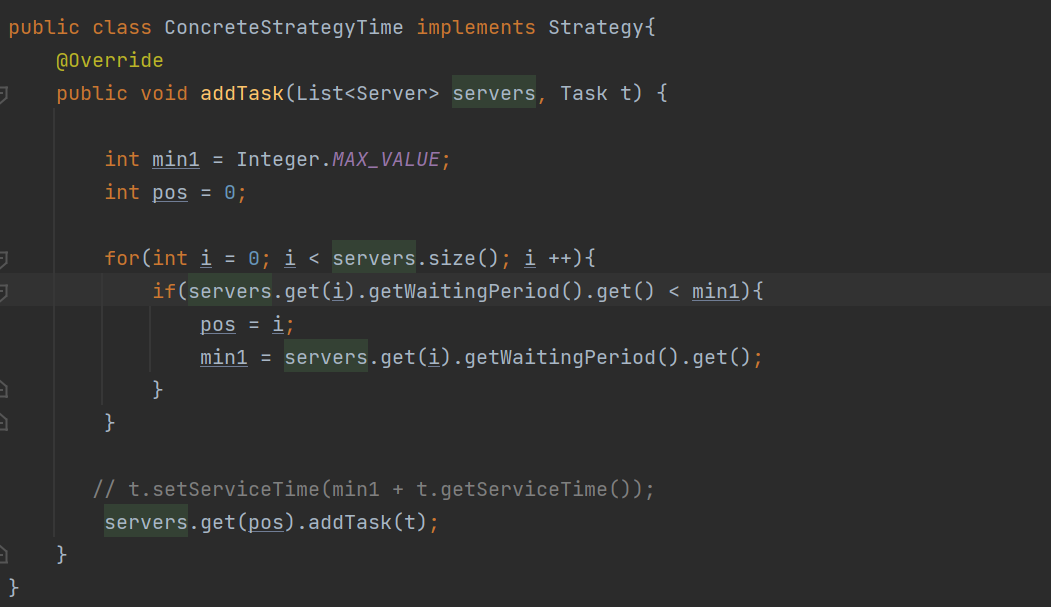
* Controller-ul este elementul cu care putem controla accesul la aplicația noastră. Pot fi fișiere, script-uri sau programe, în general orice tip de informație permisă de interfață.
* SimulationManager-ul manipulează operațiunile logice și de utilizare de informație trimisă dinainte de către rangul său superior) pentru a rezulta de o formă ușor de înțeles.
* View-ului îi corespunde reprezentarea grafică, sau mai bine zis, exprimarea ultimei forme a datelor: interfața grafică ce interacționează cu utilizatorul final. Rolul său este de a evidenția informația obținută până ce ea ajunge la controlor.



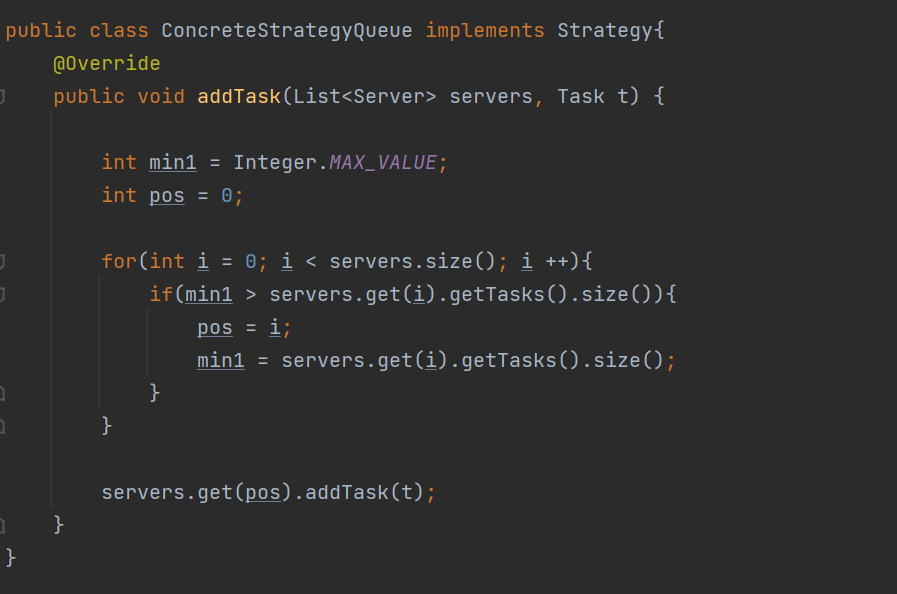
1. **Implementare**

În acest proiect am implementat 9 clase.

1. Clasa ConcreteStrategyTime implementeaza interfata Strategy si implicit metoda addTask ( List<Server>, Task ). In aceasta metoda cautam coada cu cel mai mic waitingPeriod in care vom adauga Task-ul, adica clientul.



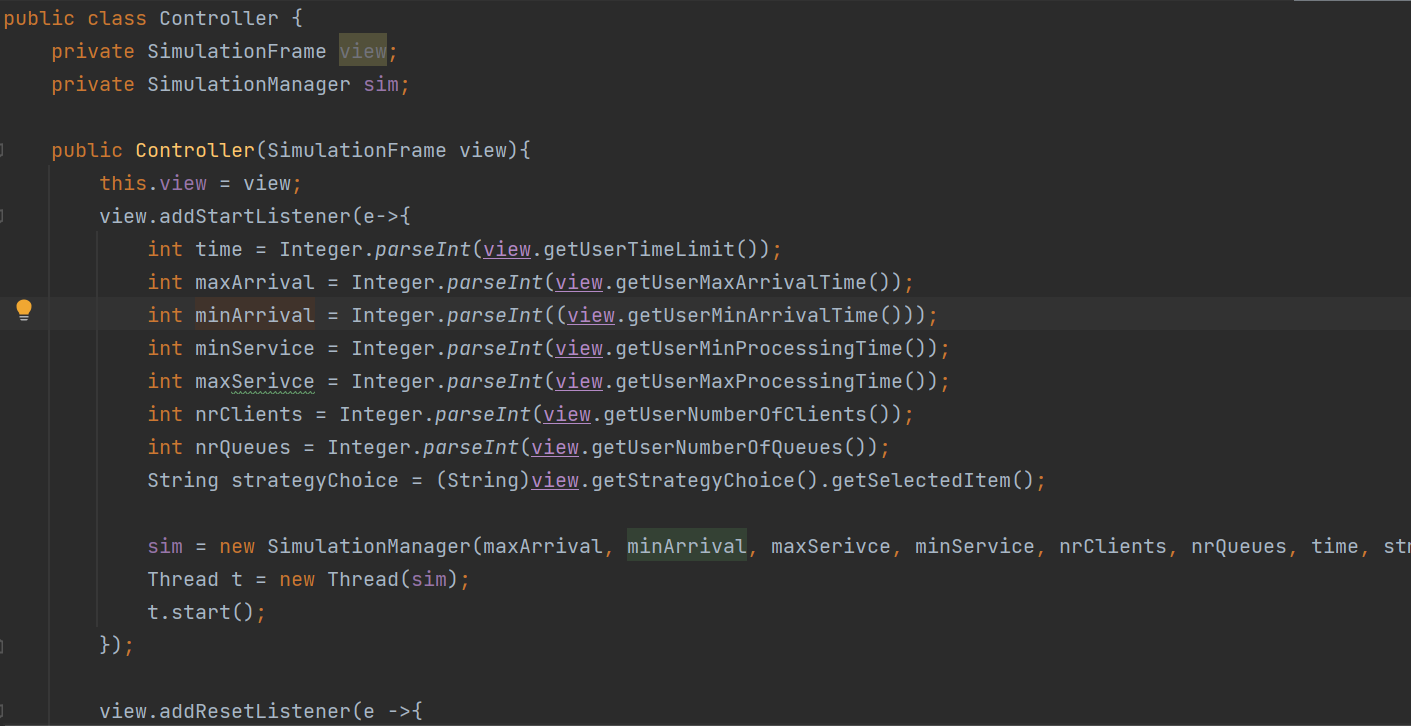
1. Clasa ConcreteStrategyQueue implementeaza interfata Strategy si implicit metoda addTask ( List<Server>, Task ). In aceasta metoda cautam coada cu cei mai putini clienti in care vom adauga Task-ul, adica clientul.



1. Clasa Controller este cea în care am făcut legătura între clasa SimulationManager și clasa View. Aici am implementat și comportamentul interfeței grafice (de exemplu: ce se întâmplă atunci când se apasă un buton sau când se introduce un text de către utilizator).

Această clasă este formată din două atribute: view de tip SimulationFrame și sim de tip SimulationManager. Pentru fiecare buton din view am adaugat un Listener.

Atunci cand butonul START este apasat, se citesc string-urile introduse de catre utilizator si se convertesc in int-uri. Se creeaza un obiect de tipul SimulationManager cu atributele corespunzatoare.

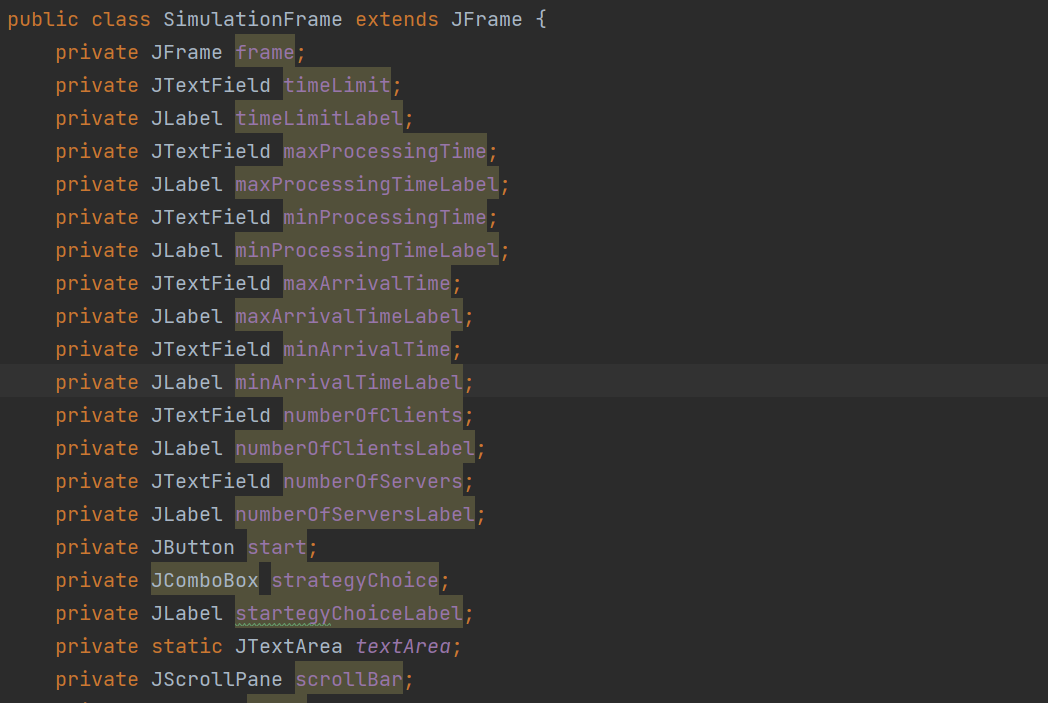


1. Clasa SimulationFrame este cea responsabilă cu crearea și aspectul interfeței grafice.

Această clasă este formată din 21 de atribute: frame, timeLimit, timeLimitLabel, maxProcessingTime, maxProcessingTimeLabel, minProcessinfTime, minProcessingTimeLabel, maxArrivalTime, maxArrivalTimeLabel, minArrivalTime, minArrivalTimeLabel, numberOfClients, numberOfClientsLabel, numberOfServers, numberOfServersLabel, start, reset, strategyChoice, strategyChoiceLabel, scrollBar, textArea .Această clasă moștenește clasa Jframe.

Pe lângă cele 21 atribute, mai avem și un constructor și 18 metode după cum urmează:

* Metodele getTimeLimit(), getMaxProcessingTime(), getMinProcessingTime(), getMaxArrivalTime(), getMinArrivalTime(), getNumberOfServers(), getNumberOfClients() care nu primesc niciun parametru și care returnează atributele clasei SimulationFrame de tipul JTextField.
* Metodele getUserTimeLimit(), getUserMaxProcessingTime(), getUserMinProcessingTime(), getUserMaxArrivalTime(), getUserMinArrivalTime(), getUserNumberOfServers(), getUserNumberOfClients() care nu primesc niciun parametru si care returneaza string-urile introduse de catre utilizator in interfata grafica.
* Metoda addStartListener ( ActionListener ) care adaugă atributului start de tipul JButton un ascultător care ulterior redă utilizatorului comportamentul specific inceperii simularii.
* Metoda addResetListener ( ActionListener ) care adaugă atributului reset de tipul JButton un ascultător care ulterior redă utilizatorului comportamentul specific resetarii interfetei, a stergerii datelor ce au fost introduse de catre utilizator si a rezultatelor generate.
* Metoda getStrategyChoice() care nu primeste niciun parametru si care returneaza atributul strategyChoice de tipul JComboBox al clasei SimulationFrame.
* Metoda getTextArea() care nu primeste niciun parametru si care returneaza atributul textArea de tipul JTextArea al clasei SimulationFrame.



1. Clasa Task este cea care reprezinta clientul nostru ce trebuie adaugat in coada. Aceasta clasa implementeaza interfata Comparable, deci implicit implementeaza metoda compareTo ( Object ). Aceasta clasa are 4 atribute: arrivalTime, serviceTime, ID, nr ( toate de tipul int ). Variabila statica nr este incrementata in constructorul Task-ului si este asignata ID-ului. Constructorul are 2 parametrii care sunt generati random in clasa SimulationManager, unde se si creaza obiecte de tipul Task.

Am implementat5 metode în această clasă, după cum urmează:

* Metoda toString() pe care am suprascris-o pentru a realiza o afișare cât mai prietenoasă a task-ului.
* Metoda getArrivalTime() care nu primeste niciun parametru si care returneaza atributul arrivalTime al clasei Task.
* Metoda getServiceTime() care nu primeste niciun parametru si care returneaza atributul serviceTime al clasei Task.
* Metoda setServiceTime ( int ) care primeste un parametru de tipul int si care da update atributului serviceTime al clasei Task.
* Metoda compareTo ( Object ) care primeste un parametru de tipul Object si pe care o folosim ca sa sortam crescator dupa arrivalTime lista de Task-uri generate random.

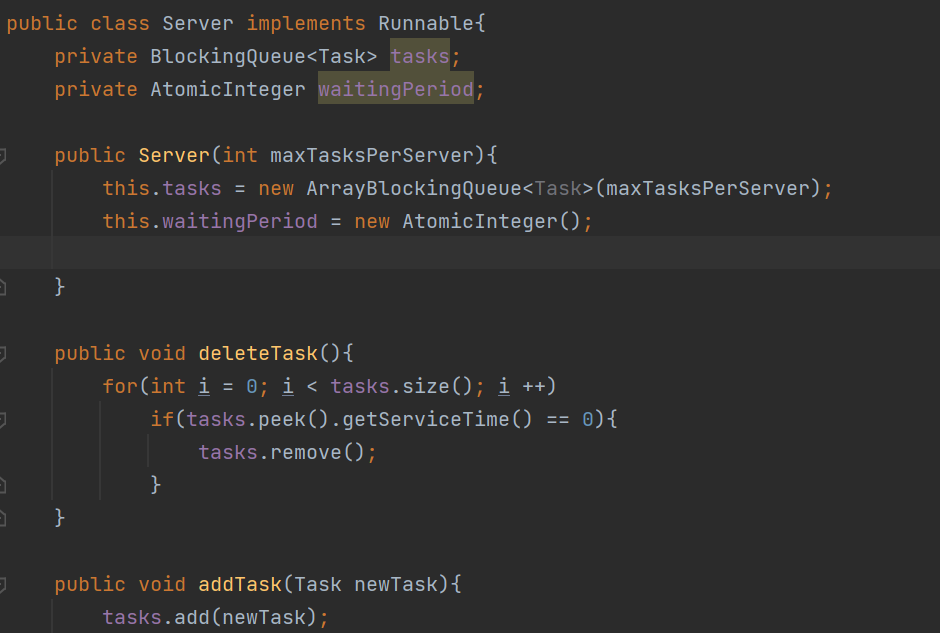


1. Clasa Server este cea care reprezinta coada la care trebuie sa ii adaugam clienti. Aceasta clasa implementeaza interfata Runnable, deci implicit si metoda run(). Aceasta clasa contine 2 atribute: tasks de tipul BlockingQueue<Task> si waitingPeriod de tipul AtomicInteger pe care le folosim in mediul concurent, fara sa mai folosim Synchronized ().

Folosim metodele peek(), add (Task), isEmpty(), remove() ale clasei BlockingQueue. Folosim metoda addAndGet ( int ) a clasei AtomicInteger.

Am implementat 5 metode in aceasta clasa, dupa cum urmeaza:

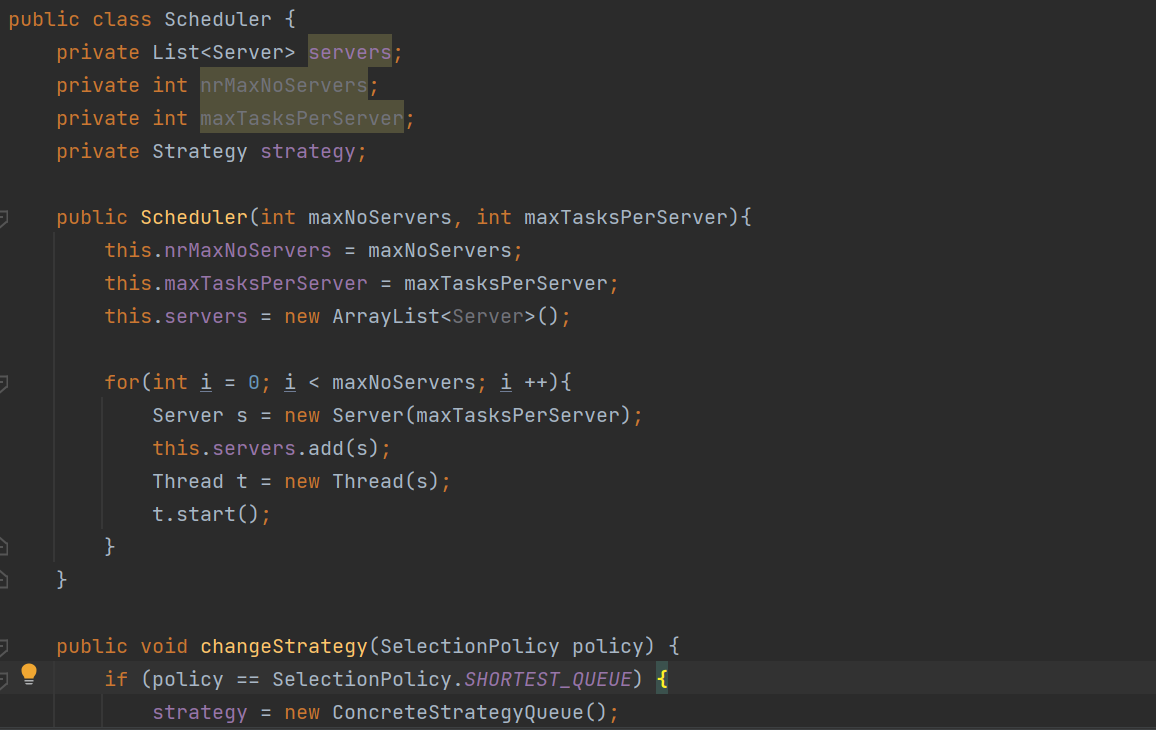
* Metoda deleteTask() care nu primeste niciun parametru si care sterge din tasks peek-ul daca acesta are serviceTime-ul egal cu 0.
* Metoda addTask ( Task ) care primeste un parametru de tipul Task, adauga in BlockingQueue-ul tasks acel task primit ca parametru si adauga atomic la waitingPeriod valoarea serviceTime-ului task-ului trimis ca parametru.
* Metoda run() care cat timp tasks nu este goala, ia peek-ul din coada si daca acesta nu este null, adoarme Thread-ul curent si actualizeaza waitingPeriod-ul cu –serviceTime-ul peek-ului. Daca tasks este goala, iesim din while-ul curent.
* Metoda getTasks() care nu primeste niciun parametru si care returneaza atributul tasks de tipul BlockingQueue al clasei Server.
* Metoda getWaitingPeriod() care nu primeste niciun parametru si care returneza atributul de tipul AtomicInteger al clasei Server.



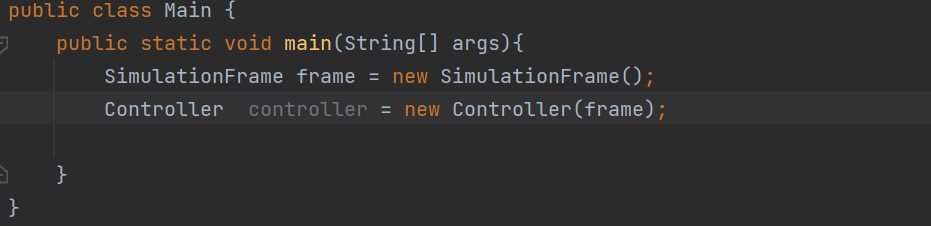
1. Clasa Scheduler care contine 4 atribute: servers de tipul list<Server>, maxNoServers, maxTasksPerServer ( ambele de tipul int ), strategy de tipul Strategy. In constructorul acestei clase ne cream cate un Thread pentru fiecare server, precum si serverele. Folosim metoda start() pentru thread-urile create.

Am implementat 3 metode in acesta clasa:

* Metoda changeStrategy ( SelectionPolicy ) care primeste ca parametru un obiect de tipul SelectionPolicy si verifica care strategie a fost aleasa de catre utilizator: SHORTEST\_TIME sau SHORTEST\_QUEUE.
* Metoda dispatchTask ( Task ) care primeste un parametru de tipul Task si care apeleaza metoda addTask ( List<Server>, Task ) pentru adaugarea in fucntie de strategia aleasa a clientilor in cozi.
* Metoda getServers() care nu primeste niciun parametru si care returneza atributul servers al clasei Scheduler.



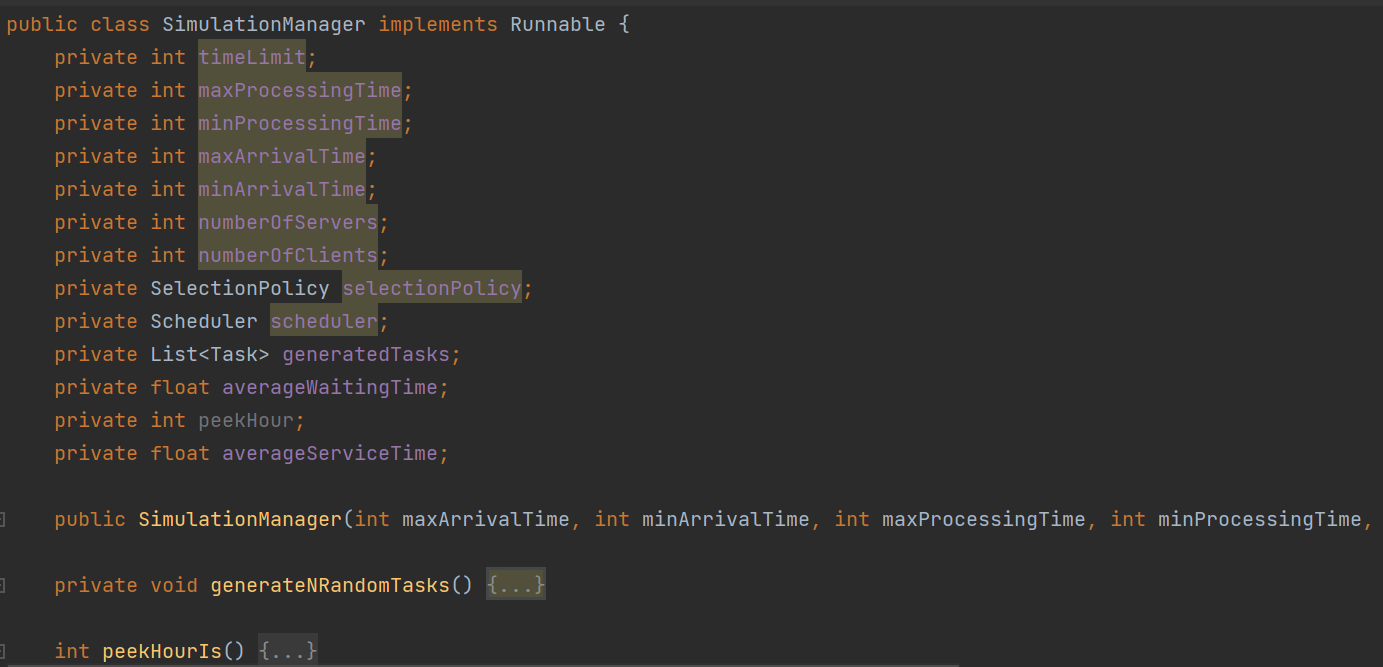
1. Clasa Main în care se află main-ul proiectului. Această clasă conține metoda main în care avem instanțiate următoarele obiecte: frame de tipul SimulationFrame si controller de tip Controller.



1. Clasa SimulationManager implementeaza interfata Runnable, deci implicit si metoda run(). Clasa contine 13 atribute: timeLimit, maxProcessingTime, minProcessingTime, maxArrivalTime, minArrivalTime, numberOfServers, numberOfClients, peekHour, averageServiceTime, averageWaitingTime ( de tipul int ), selectionPolicy de tipul SelectionPolicy, scheduler de tipul Scheduler, generatedTasks de tipul List<Task>.

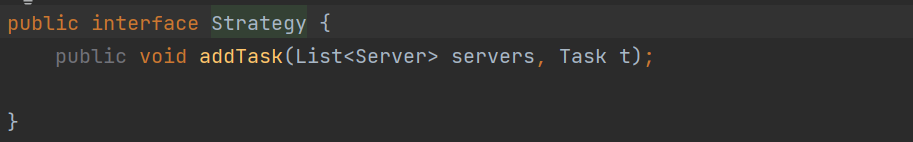
Am implementat 2 metode in acesta clasa:

* Metoda generateNRandomTasks() care genereaza o lista de tipul Task de dimensiune numberOfClients in care se adauga obiecte de tipul Task ale caror parametrii au fost generati random intre un range ce tine cont de datele introduse de utilizator in interfata grafica. Tot aici se sorteaza crescator lista generata. Am calculat averageWaitingTime si averageServiceTime.
* Metoda run() pe care am suprascris-o si care pune clientii in cozi si ii sterge din lista generatedTasks. Tot aici verificam daca peek-ul fiecarei cozi are serviceTime-ul egal cu 0, in caz pozitiv, stergem Task-ul din coada respectiva si in caz contrar il scade. Daca cumva lista de clienti generati este goala si cozile sunt si ele goale, este oprita simularea pentru optimizare. In aceasta metoda scriem in fisier rezultatele simularii si le afisam in interfata grafica.

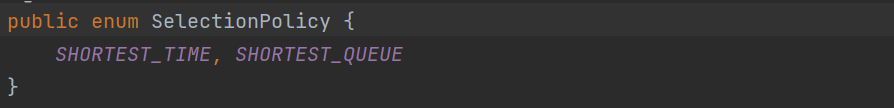


Pe langa cele 9 clase, am creat si o interfata si un enum.

1. Interfata Strategy care contine metoda addTask ( List<Server>, Task ) care este implementata de clasele ConcreteStrategyTime si ConcreteStrategyQueue.

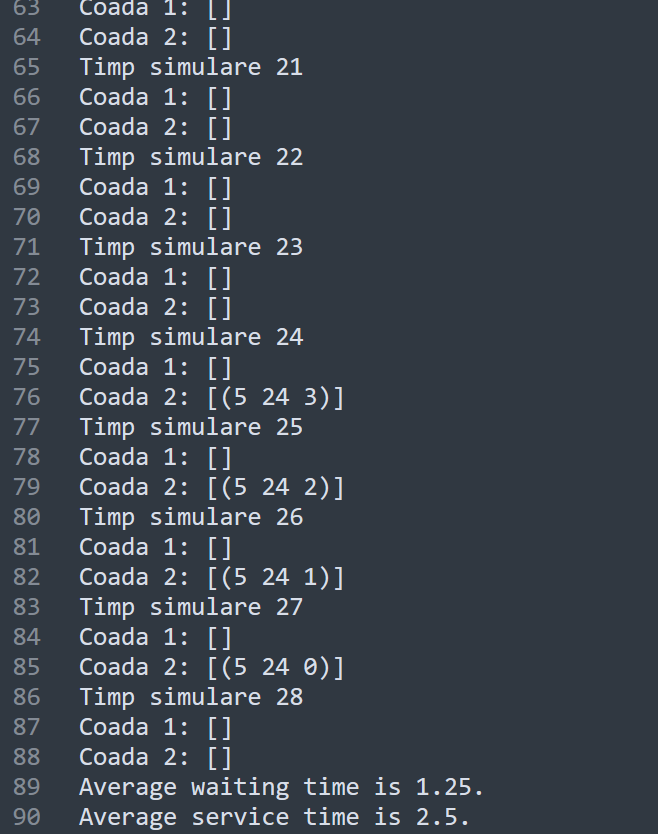


1. Enumerarea SelectionPolicy este un set de constante pentru reprezentarea diferitelor valori. Intr-un enum, numele constantelor trebuie sa fie unice, iar variabilele de tip enum isi pot asigna doar una din valorile constante din set.

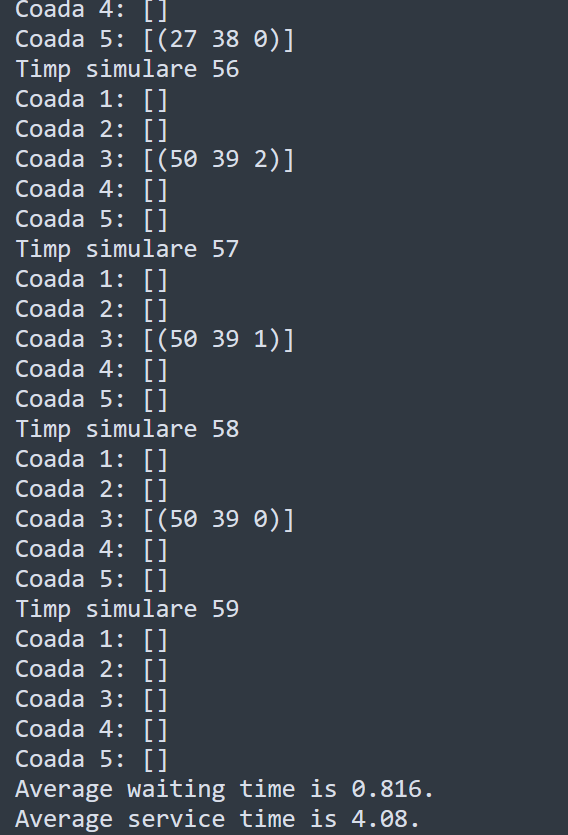


# Rezultate

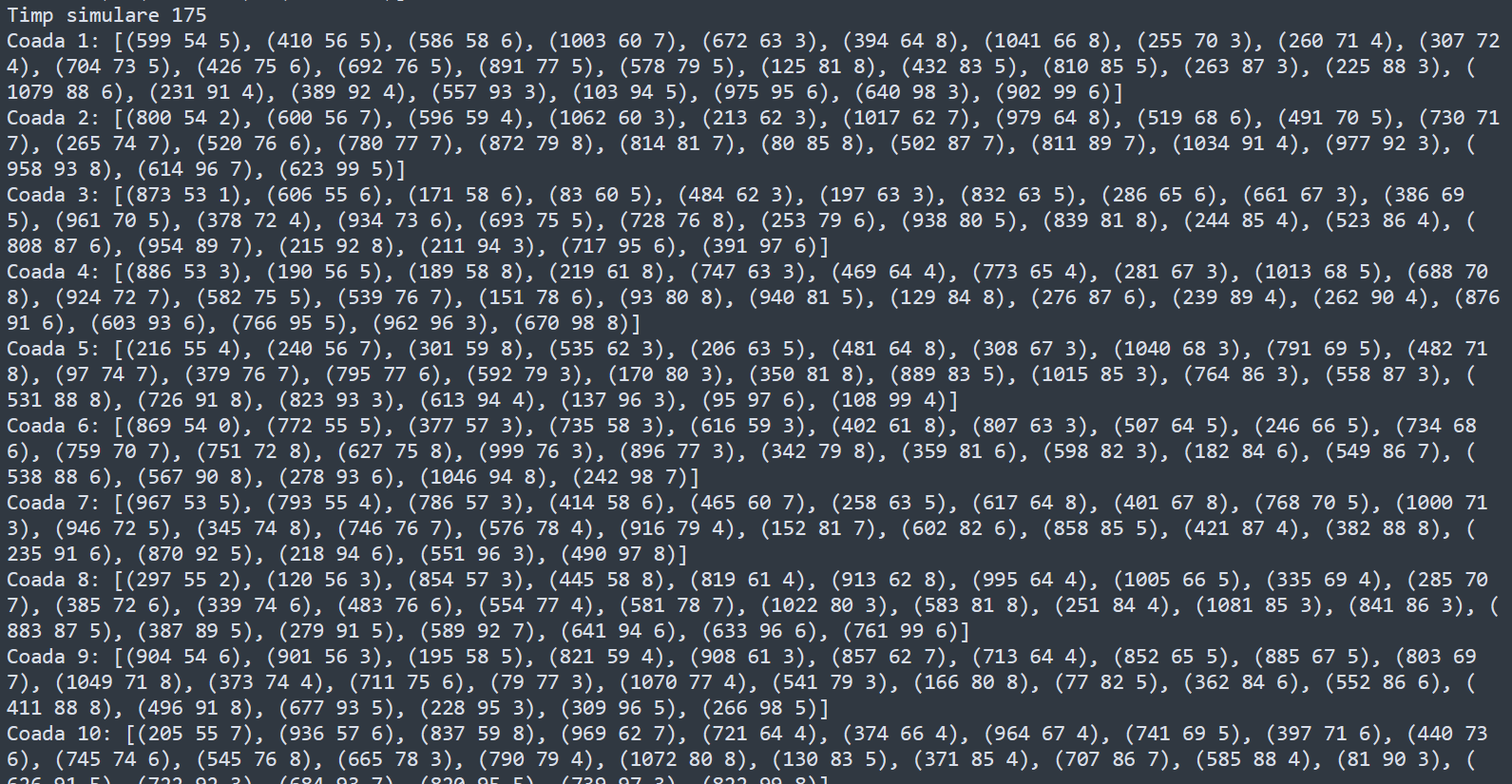
1. Testul 1



1. Testul 2



1. Testul 3



# Concluzii

În urma implementării acestei aplicații numite QUEUES MANAGEMENT APPLICATION USING THREADS AND SYNCHRONIZATION MECHANISMS am reușit să imi aprofundez cunostintele legate de Thread-uri si de gestionarea acestora .

Printre îmbunătățirile pe care le-aș aduce eu proiectului meu, ar fi adaugarea unei clase Service de unde sa se selecteze tipul de serviciu pe care clientii doresc sa-l primeasca. In functie de servicul ales, interfata sa fie si ea schimbata, iar o coada sa ii corespunda unui singur serviciu, dar un serviciu sa poata avea mai multe cozi in functie de cat de mare este cererea.

# Bibliografie

1. <https://stackoverflow.com/questions/10177183/add-scroll-into-text-area>
2. <https://www3.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI_2.html>
3. https://www.w3schools.com/java/java\_files\_create.asp