**PROIECT “QUEUING BASED SYSTEMS”**

FILIP LIVIU PAUL

**Conținut:**

1. Obiectivul temei
2. Analiza problemei, modelare, scenarii, cazuri de utilizare
3. Proiectare
4. Implementare
5. Rezultate
6. Concluzii
7. Bibliografie

**Capitolul I. *Obiectivul temei***

Obiectivul temei “Queuing Based Systems” îl reprezintă proiectarea și implementarea unei aplicații care simulează un sistem de tip client – server și urmărește minimizarea timpului de așteptare pentru clienți.

Îndeplinirea obiectivului propus necesită completarea unor obiective secundare și pași intermediari, elemente pe care le vom sintetiza în următorul tabel:

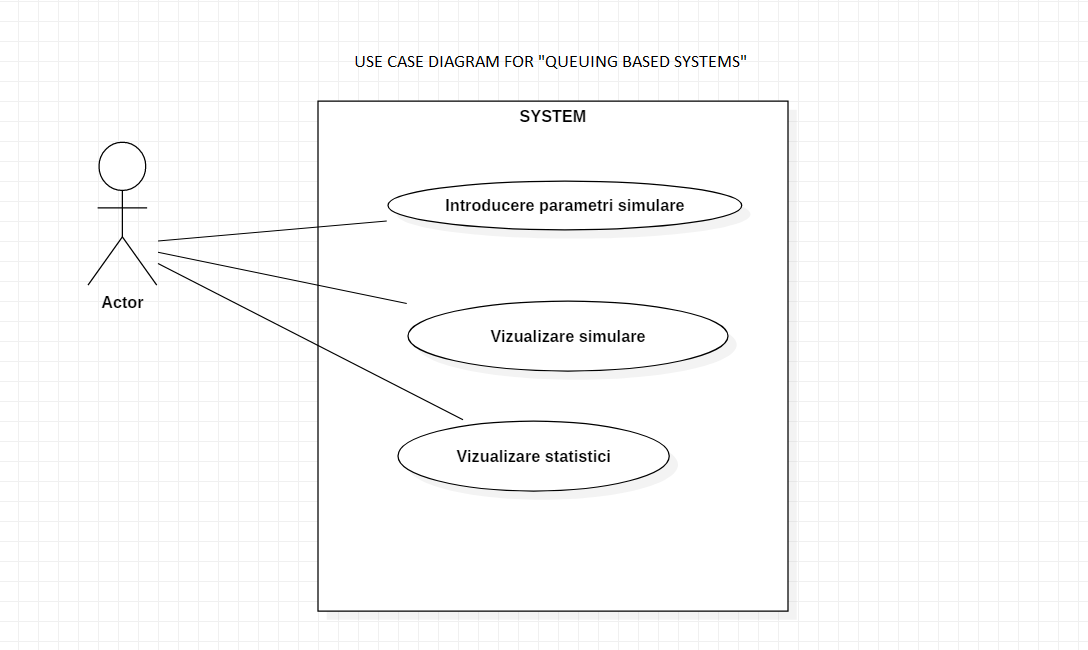
|  |  |
| --- | --- |
| Obiectiv secundar | Descriere |
| Înțelegerea conceptelor identificate în enunțul problemei și transpunerea lor în elemente software care pot manipula astfel de date. | * Se prezintă noțiunile de bază despre execuția concurentă și aplicabiliatea proceselor concurente în dezvoltarea aplicațiilor. * Aceste noțiuni vor fi descrise pe larg în capitolul II. |
| Analizarea problemei, modelarea conceptuală a elementelor ierarhice care constituiesc soluția problemei | * Se vor analiza cerințele problemei, se vor explica acțiunile care trebuie implementate. * Se va explica modelarea soluției pe baza cazurilor de utilizare și a ipotezei problemei. * Aceste sarcini vor fi tratate în capitolul II. |
| Identificarea scenariilor și cazurilor de utilizare a aplicației și rezolvarea eventualelor probleme care pot apărea în urma interacțiunii dintre utilizator și aplicație | * Se vor prezenta cazurile de utilizare sub forma de diagrame și descrieri de use-case. * Descrierile use-case-urilor se vor face sub forma unei liste conținând pașii execușiei pentru fiecare use-case * Se vor identifica eventualele nereguli ce pot apărea la interacțiunea utilizator – program și se va oferi tehnica abordată pentru soluționarea problemei. * Descrierea acestor pași se va face in capitolul II. |
| Alegerea elementelor optime de proiectare corespunzătoare paradigmei de programare orientată pe obiecte  Construirea unei interfețe utilizator prin care interacțiunea dintre sistem și user să fie ușurată, să conțină toate funcționalitățile dorite și să ofere rezultate care pot fi refolosite ulterior | * Se vor expune deciziile de proiectare luate pentru optimizarea aplicației în domeniile timp și memorie. * Se va prezenta proiectarea aplicației în conformitate cu paradigma de programare orientată pe obiecte. * Se prezintă diagramele UML de clase și de pachete pentru proiect. * Se evidențiază structurile de date folosite, algoritmii utilizați și interfețele adăugate. * Se evidențiază structura ierarhică a proiectului, acesta fiind împărțit în pachete, clase și subclase. Relațiile dintre acestea vor fi de asemnea vizualizate cu ajutorul diagramelor UML. * Se va descrie modul de proiectare a interfeței utilizator pentru ca această să satisfacă cerințele utilizatorului, să aibă un aspect “prietenos” și ofere posibilitatea utilizatorului să folosească funcțiile implementate. * Aceste idei vor fi pe larg prezentate în capitolul III. |
| Evidențierea și descrierea etapelor de implementare a problemei. | * Se va prezenta implementarea claselor cu câmpurile și metodele cele mai semnificative. * Se vor evidenția elementele de implementare a unor algoritmi specifici. * Se va descrie modul de implementare a interfeței grafice. * Aceste descrieri vor avea loc în capitolul IV. |
| Interpretarea și evaluarea rezultatelor | * Se vor construi date de intrare, cu ajutorul cărora se vor efectua operațiile implementate în aplicație * Se vor construi atât date valide, cât și invalide pentru evidențierea funcționării aplicației. * Aceste procese se vor prezenta în capitolul V. |
| Concluzii cu privire la temă și dezvoltări ulterioare | * Se vor trage concluzii cu privire la tema abordată, aplicabilitatea ei și eventuale dezvoltări ulterioare. * Acest subiect va fi tratat în capitolul VI. |

**Capitolul II. *Analiza problemei, modelare, scenarii, cazuri de utilizare***

Cozile sunt deseori utilizate pentru modelarea aplicațiilor din lumea reală. Principalul obiectiv al unei cozi este să gestioneze locul unui “client” înainte ca acesta să primească un “serviciu”. Management-ul sistemelor bazate pe cozi urmărește minimizarea cantității de timp de așteptare pentru clienți înainte și eficientizarea proceselor prin adăugarea mai multor servere (dacă este posibil) pentru distribuirea clienților către toate serverele disponibile.

Aplicația implementată va simula un număr de clienți care vor fi distribuiți la un număr de case după o strategie care urmărește timpul minim de așteptare pentru client, în funcție de timpul de venire al clientului și aceștia vor primi un serviciu pe o perioadă de timp cunoscută, după care vor părăsi coada. Timpul de părăsire al unei cozi depinde de gradul de ocupare al unei cozi, de timpul de venire și de timpul de procesare.

Rolul actorului este luat de către utilizatorul aplicației, acesta poate fi un student, un profesor sau orice altă persoană care dorește să efectueze o simulare a acestui proces cu clienți și servere. În cele ce urmează, vom descrie fiecare use case sub forma unei liste conținând pașii execuției fiecărui use case.



1. Rularea aplicației pentru simulare

Use case: Introducere parametri simulare + vizualizare simulare + vizualizare statistici

Primary actor: Client

Main succes scenario:

* Clientul introduce parametrii pentru simulare în interfața graficp
* Sistemul verifică dacă datele introduse sunt corecte
* În caz afirmativ, se va afișa o fereastră în care se simulează evoluția în timp a cozilor după generarea clienților și asignarea la o coadă
* În același timp sunt afișate date despre generarea clienților și despre parcursul lor în timp
* La finalul simulării se va afișa o fereastrp cu statistici despre simulare.

Alternative sequences:

* Nu sunt introduși toți parametrii sau aceștia sunt invalizi pentru începerea simulării. În acest caz, se va afișa o fereastră de eroare.

**Capitolul III. *Proiectare***

În acest capitol, vom prezenta pe larg deciziile de proiectare luate în concordanță cu paradigma programării orientate pe obiecte, vom expune diagramele UML de clase, structura ierarhică a proiectului și strucuturile de date folosite.

Aplicația este structurată în pachete, cele principale fiind pachetele View, Strategies și Simulation în care sunt implementate atât structurile de date în care stocăm informația despre clienți, respectiv cozi, despre generearea și repartizarea acestora în simulare cât și în care avem implementată interfața grafică.

Diagrama UML pentru pachete :

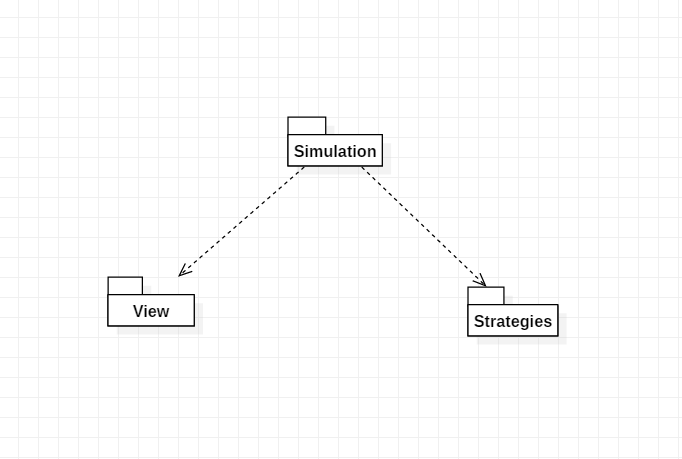


Diagrama UML pentru clasa Simulation:

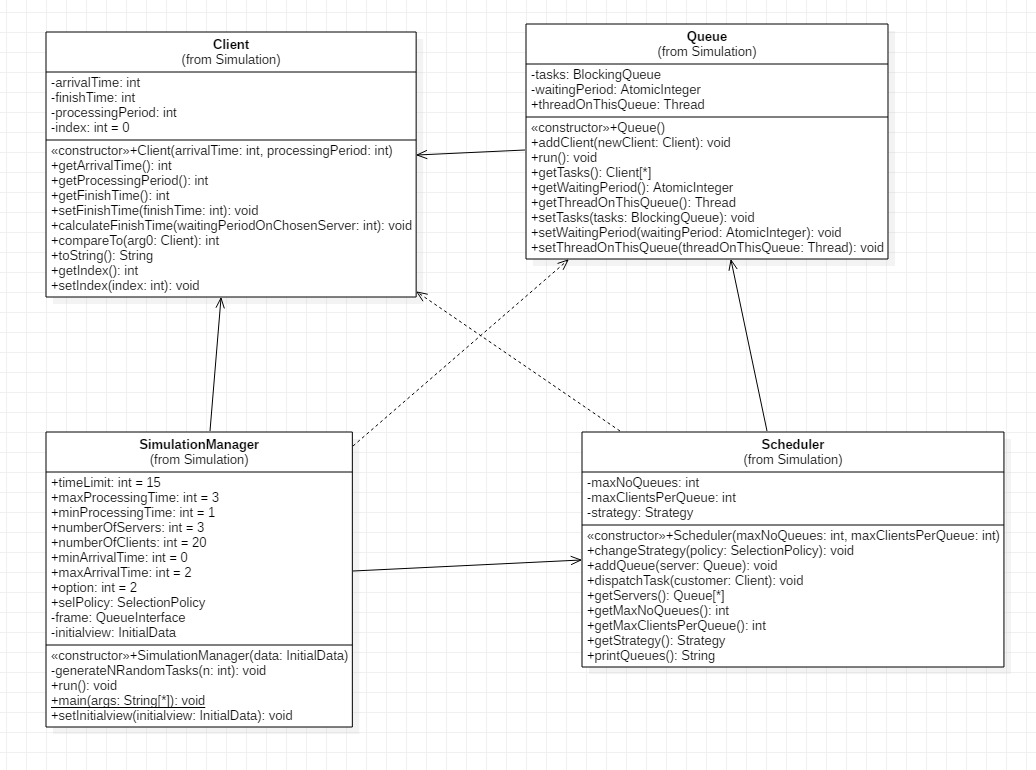


Diagrama UML pentru clasa View:

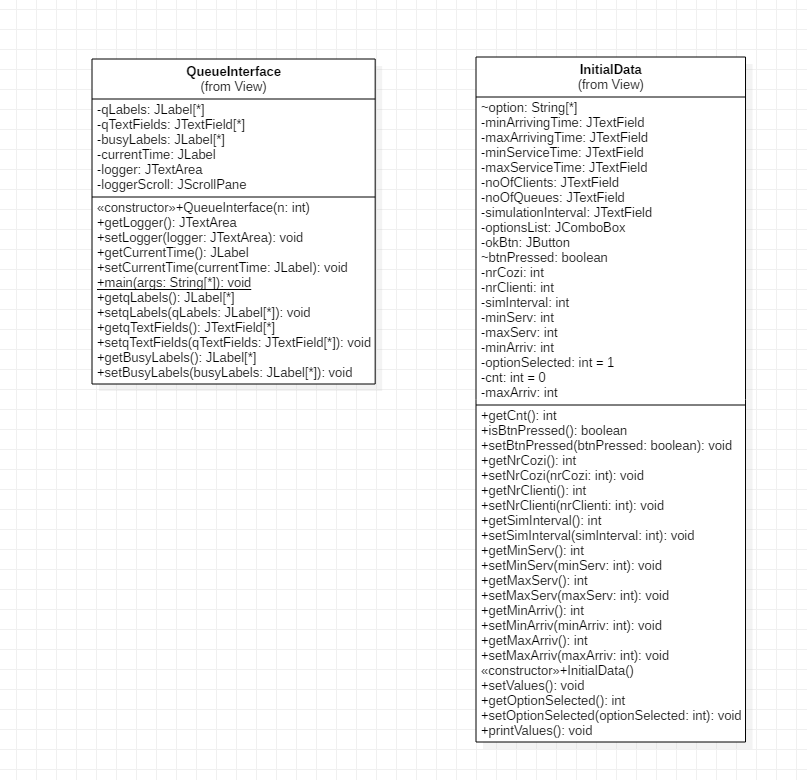


Diagrama UML pentru clasa Strategies:

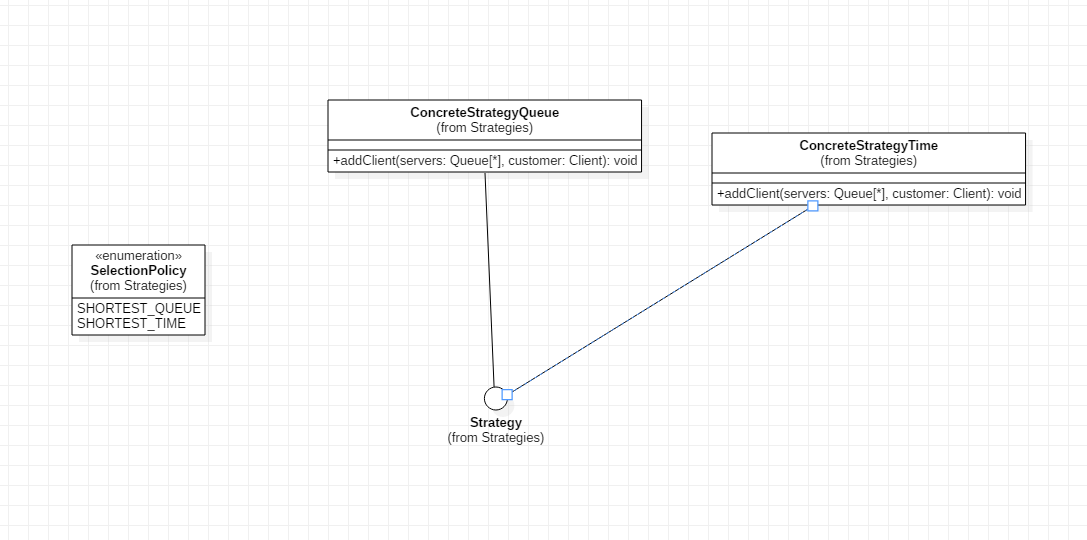
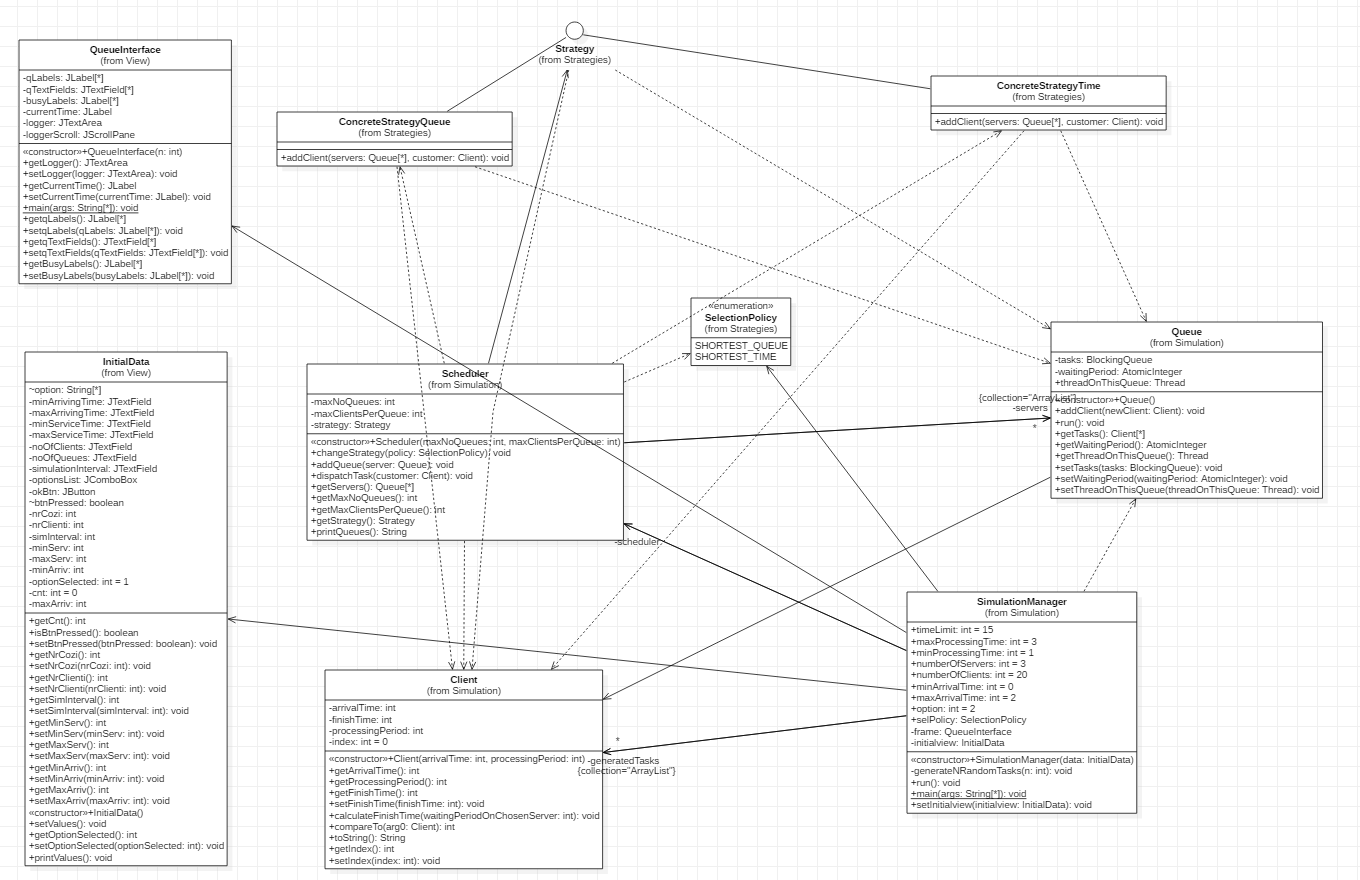


Diagrama UML pentru întreg proiectul:

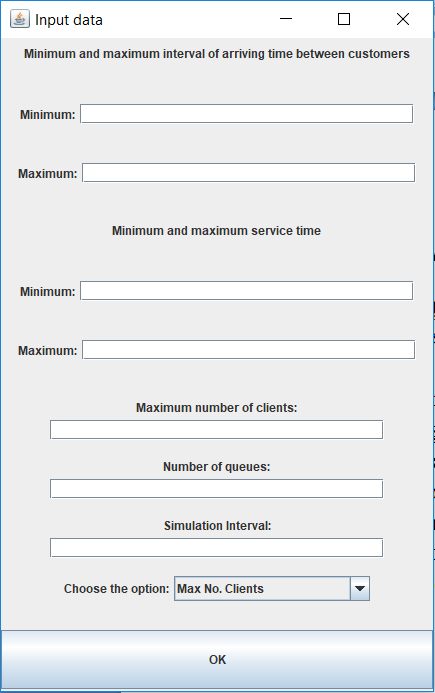


**Capitolul IV.  *Implementare***

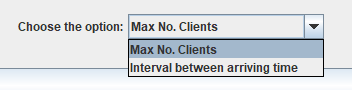
Întrucât aplicația vine cu o interfață grafică care să ușureze interacțiunea utilizatorului cu sistemul, am structurat clasele în mai multe pachete: pachetul **View** care conține clasele **InitialData** (în care se vor introduce datele pentru începerea simulării) și **QueueInterface** (va fi frame-ul unde se poate vizualiza simularea cozilor și a clienților).

Clasa **InitialData** conține Jlabel-uri și JtextField-uri pentru introducerea datelor, un ComboBox pentru selectarea opțiunii pentru simulare și un buton pentru a începe simularea. Pentru un buton am adăugat un ActionListener, iar clasa are atribute și pentru valorile introduse în câmpuri.

Constructorul clasei configurează frame-ul pentru a arăta în următorul mod:



Astfel, cu ajutorul acestei interfețe grafice, vom putea seta intervalul de venire între clienți, intervalul de procesare pentru un client , numărul maxim de clienți ce pot fi generați, numărul de cozi și intervalul de simulare. Totodată, se poate alege ca simularea să ruleze fără să țină cont de intervalul de timp între venirile clienților, această opțiune se alege prin intermediul ComboBox-ului:



Atributele clasei sunt :

String[] option = {"Max No. Clients", "Interval between arriving time"};

**private** JTextField minArrivingTime = **new** JTextField(30);

**private** JTextField maxArrivingTime = **new** JTextField(30);

**private** JTextField minServiceTime = **new** JTextField(30);

**private** JTextField maxServiceTime = **new** JTextField(30);

**private** JTextField noOfClients = **new** JTextField(30);

**private** JTextField noOfQueues = **new** JTextField(30);

**private** JTextField simulationInterval = **new** JTextField(30);

**private** JComboBox<String> optionsList = **new** JComboBox<String>(option);

**private** JButton okBtn = **new** JButton("OK");

**boolean** btnPressed;

**private** **int** nrCozi;

**private** **int** nrClienti;

**private** **int** simInterval;

**private** **int** minServ;

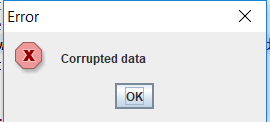
**private** **int** maxServ;

**private** **int** minArriv;

**private** **int** optionSelected = 1;

**private** **int** cnt = 0;

Atributul btnPressed este setat dacă butonul de OK a fost apăsat, în cadrul metodei setValue(), care update-ază atributele întregi pentru valorile adăugate prin metode mutatoare. Apoi, btnPressed este din nou setat la false pentru ulterioare modificări. În cazul în care datele introduse sunt corupte, se va afișa o fereastră de eroare, astfel:



Clasa **QueueInterface** construiește interfața grafică pentru afișarea simulării cozilor și clienților, împreună cu o porțiune de text care joacă rol de “logger” al evenimentelor. Atributele clasei sunt șiruri de Jlabel-uri, JtextArea-uri pentru afișarea fiecărei cozi. Atributele acestei clase sunt:

**private** JLabel[] qLabels;

**private** JTextField[] qTextFields;

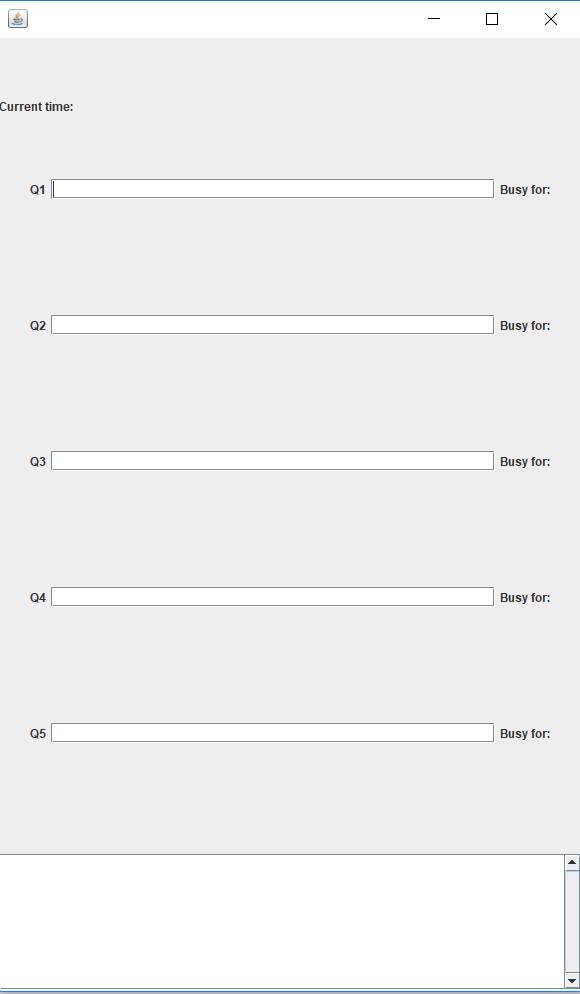
**private** JLabel[] busyLabels;

**private** JLabel currentTime;

**private** JTextArea logger = **new** JTextArea();

**private** JScrollPane loggerScroll = **new** JScrollPane(logger);

Constructorul acestui frame primește ca parametru un întreg n, care reprezintă numărul de cozi din interfață. De exemplu, pentru n = 5, interfața va fi următoarea:



Valorile atributelor clasei vor fi modificate pe durata simulării cu valorile aferente.

Pachetul **Strategies** conține două clase, **ConcreteStrategyQueue** și **ConcreteStrategyTime**, interfața **Strategy** și o enumerație **SelectionPolicy.**

Interfața Strategy conține definiția de metodă addClient, iar cele 2 clase din acest pachet implementează interfața Strategy, acestea reprezentând modul de adăugare al unui client într-o coadă, după principiul timpului cel mai scurt sau coada cea mai scurtă.

**public** **interface** Strategy {

**public** **void** addClient(ArrayList<Queue> servers, Client customer);

}

Astfel, clientul customer va fi adăugat într-o coadă din lista de cozi servers.

Pachetul **Simulation** conține clasele **SimulationManager, Schedule, Client și Queue.** Clasa Client este implementarea pentru un client. Atributele acestei clase sunt timpul de sosire, timpul de procesare și timpul de finalizare. O metodă importantă pe care am implementat-o în această clasă este calculateFinishTime, care primește un întreg care reprezintă perioada de așteptare a unei cozi, calculează și setează timpul de finalizare al clientului ca suma dintre timpul de sosire, timpul de procesare și timpul de așteptare la coadă.

**private** **int** arrivalTime;

**private** **int** finishTime;

**private** **int** processingPeriod;

**private** **int** index = 0;

**public** **void** calculateFinishTime(**int** waitingPeriodOnChosenServer) {

**int** finishTime = **this**.getProcessingPeriod() + **this**.getArrivalTime() + waitingPeriodOnChosenServer;

**this**.setFinishTime(finishTime);

}

Atributul index reprezintă un indice care se va atribui fiecărui client generat în simulare. Acesta ne ajută în urmărirea cu atenție a simulării.

Clasa **Queue** reprezintă implementarea cozii.

Atributele acestei clase sunt o listă de clienți de tip **BlockingQueue,** această abordare fiind avantajoasă în lucrul cu thread-uri. Următorul atribut este un AtomicInteger care reprezintă timpul de așteptare al cozii, de asemenea folosim tipul AtomicInteger pentru a ușura utilizarea thread-urilor pentru fiecare coadă. Ultimul atribut al clasei Queue este un Thread, ce reprezintă firul de lucru ce va fi pus în execuție odată cu crearea unei noi instanțe de coadă.

**private** BlockingQueue<Client> tasks;

**private** AtomicInteger waitingPeriod;

**public** Thread threadOnThisQueue;

**public** **void** addClient(Client newClient) {

tasks.add(newClient);

waitingPeriod.addAndGet(newClient.getProcessingPeriod());

}

Metoda addClient adaugă un client la coadă și în același timp si actualizează timpul de așteptare propriu.

**public** **void** run() {

// **TODO** Auto-generated method stub

**while** (**true**) {

Client nextClient = **this**.getTasks().peek();

**if** (nextClient != **null**) {

**try** {

threadOnThisQueue.*sleep*(nextClient.getProcessingPeriod() \* 1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

tasks.remove();

}

}

}

Cum clasa Queue implementează interfața Runnable, vom suprascrie metoda run astfel: vom scoate clientul care se află in capătul cozii și vom adormi thread-ul cozii pentru o perioadă egală în secunde cu timpul de procesare al clientului extras din coadă. După scurgerea acestei perioade, vom elimina clientul din lista de clienți.

Clasa **Scheduler** conține următoarele atribute:

**private** ArrayList<Queue> servers;

**private** **int** maxNoQueues;

**private** **int** maxClientsPerQueue;

**private** Strategy strategy;

Această clasă va implementa o listă de cozi și o strategie de repartizare a clienților la cozi.

**public** **void** changeStrategy(SelectionPolicy policy){

**if**(policy == SelectionPolicy.***SHORTEST\_QUEUE***){

strategy = **new** ConcreteStrategyQueue();

}

**if**(policy == SelectionPolicy.***SHORTEST\_TIME***){

strategy = **new** ConcreteStrategyTime();

}

}

Această metodă setează strategia de repartizare a clienților la cozi, utilizând cele două elemente din enumerația SelectionPolicy definită în pachetul Strategies.

Clasa **SimulationManager** implementează interfața Runnable și este direct responsabilă pentru procesul de simulare. Atributele acestei clase sunt

**public** **int** timeLimit = 15;

**public** **int** maxProcessingTime = 3;

**public** **int** minProcessingTime = 1;

**public** **int** numberOfServers = 3;

**public** **int** numberOfClients = 20;

**public** **int** minArrivalTime = 0;

**public** **int** maxArrivalTime = 2;

**public** **int** option = 2;

**public** SelectionPolicy selPolicy = SelectionPolicy.***SHORTEST\_TIME***;

**private** Scheduler scheduler;

**private** ArrayList<Client> generatedTasks;

**private** QueueInterface frame;

**private** InitialData initialview;

Atributele clasei reprezintă valorile parametrilor pe care le vom introduce în interfața InitialData. Câmpul Option reprezintă optiunea aleasă, 1 pentru generare fără a ține cont de intervalul de sosire a clienților și 2 altfel. Lista generatedTasks va fi populată prin apelul metodei generateNRandomTasks

Metoda generateNRandomTasks(int n) primește ca argument un întreg și va genera maxim n clienți, în funcție de opțiunea aleasă. În cazul în care se optează primul caz, la generarea clienților nu se ține cont de timpul de sosire precedent sau de intervalul de sosire între clienți. Se vor genera clienți doar pe baza intervalului timpului de procesare. În cazul opțiunii 2, se vor genera clienți și pe baza intervalului introdus în interfață.

După generarea tuturor clienților, se va ordona lista de clienți în ordinea crescătoare a timpilor de sosire, pentru a eficientiza simularea ulterioară. Codul pentru această metodă este următorul :

**private** **void** generateNRandomTasks(**int** n) {

generatedTasks = **new** ArrayList<Client>(n);

**int** j = 0;

**int** randomProcessingTime = -1;

**int** randomArrivalTime = -1;

**if** (option == 1) {

System.***out***.println("Se genereaza " + n + " clienti");

**for** (**int** i = 0; i < n; i++) {

**boolean** ok = **true**;

**while** (ok) {

randomProcessingTime = (**int**) (Math.*random*() \* 10);

**if** (j == 0)

randomArrivalTime = 0;

**else**

randomArrivalTime = (**int**) (Math.*random*() \* 10);

System.***out***.println("proces aleatoriu : " + randomProcessingTime);

System.***out***.println("timp aleatoriu: " + randomArrivalTime);

**if** (randomProcessingTime < minProcessingTime || maxProcessingTime < randomProcessingTime

|| randomArrivalTime + randomProcessingTime >= timeLimit)

ok = **true**;

**else** {

ok = **false**;

}

}

Client newClient = **new** Client(randomArrivalTime, randomProcessingTime);

generatedTasks.add(newClient);

System.***out***.println("S-au generat " + (j + 1) + " clienti");

j++;

}

} **else** **if** (option == 2) {

**int** prevArrivalTime = 0;

**boolean** ok = **true**;

**int** nr = (**int**) (Math.*random*() \* 100);

**if** (nr > n || nr == 0)

nr = n;

System.***out***.println("se vor genera maxim " + nr + " numere");

**while** (nr > 0) {

ok = **true**;

**while** (ok) {

System.***out***.println("prev: " + prevArrivalTime);

**if** (j == 0)

randomArrivalTime = 0;

**else**

randomArrivalTime = (**int**) (prevArrivalTime + Math.*random*() \* 10);

randomProcessingTime = (**int**) (Math.*random*() \* 10);

System.***out***.println("proces aleatoriu : " + randomProcessingTime);

System.***out***.println("timp aleatoriu: " + randomArrivalTime);

**if** (randomProcessingTime < minProcessingTime || maxProcessingTime < randomProcessingTime

|| randomArrivalTime - prevArrivalTime < minArrivalTime

|| randomArrivalTime - prevArrivalTime > maxArrivalTime) {

ok = **true**;

**if** (j == 0) {

**if** (randomProcessingTime < minProcessingTime || maxProcessingTime < randomProcessingTime)

ok = **true**;

**else** {

ok = **false**;

}

}

} **else** {

ok = **false**;

prevArrivalTime = randomArrivalTime;

}

System.***out***.println(ok);

}

System.***out***.println(randomArrivalTime + " si " + randomProcessingTime);

Client newClient = **new** Client(randomArrivalTime, randomProcessingTime);

generatedTasks.add(newClient);

System.***out***.println("S-au generat " + (j + 1) + " clienti");

j++;

nr--;

}

}

Collections.*sort*(generatedTasks);

}

Cum această clasă implementează interfața Runnable, este necesar să implementăm metoda run(), care va realiza modificările la fiecare moment de timp din cadrul simulării. Totodată, vom calcula anumite statistici în cadrul acestei metode, cum ar fi timpul de așteptare mediu, timpul de servire mediu și ora de vârf.

La fiecare moment de timp, vom repartiza clienții cu timpul de sosire egal cu timpul current la cozi după strategia abordată și vom afișa un mesaj în interfața grafică despre evenimentele petrecute. Totodată, la fiecare moment de timp, cozile vor fi actualizate în concordanță cu procesele aflate în desfășurare.

La finalul execuției, thread-urile pentru fiecare coadă vor fi oprite și se va afișa o fereastră cu statisticile pentru intervalul de simulare rulat.

De asemena, aplicația va fi rulată din această clasă, prin metoda main, astfel:

**public** **static** **void** main(String[] args) {

InitialData view = **new** InitialData();

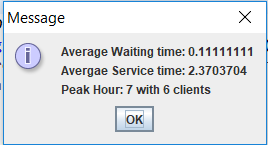
SimulationManager sim = **new** SimulationManager(view);

Thread t = **new** Thread(sim);

t.start();

}

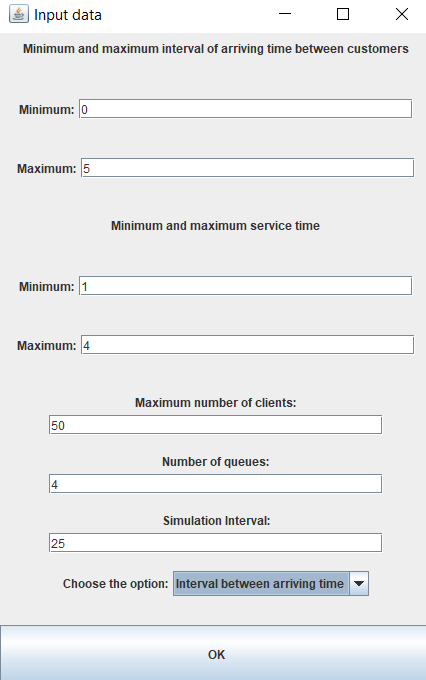
Fereastra cu statisticile simulării va arăta ca în exemplul următor:



**Capitolul V. *Rezultate***

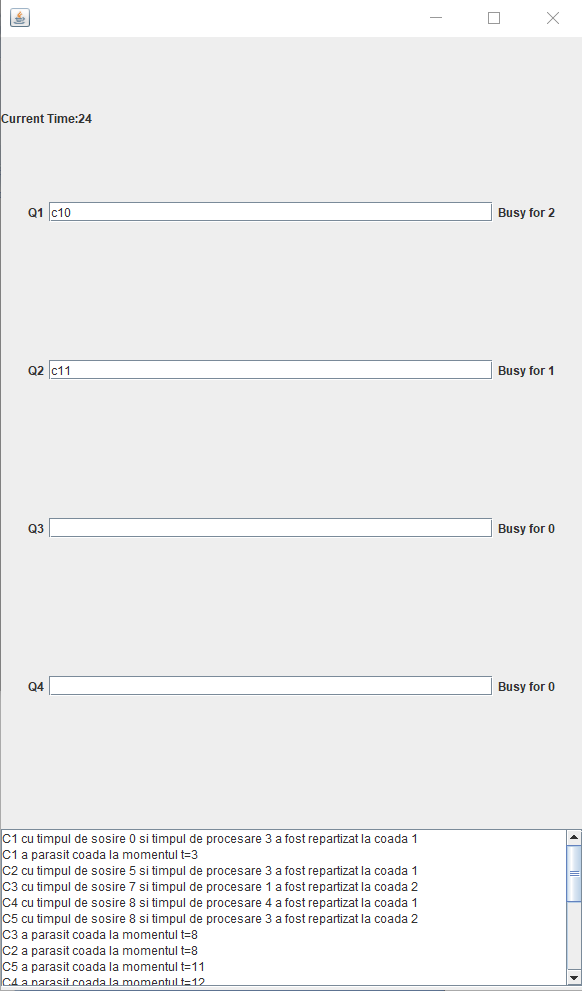
Întrucât în acest proiect clienții sunt generați aleator, în acest capitol vom prezenta un caz de utilizare a aplicației.

Vom introduce inițial următoarele valori de start:

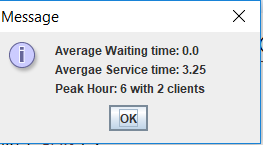


Așadar, vom opta pentru opțiunea ce generează clienți ce depinde de intervalul timpului de sosire între clienți (minim 0 secunde, maxim 5 secunde), cu constrângerea ca timpul de procesare va fi în intervalul [1,4]. Numărul maxim de clienți ce vor fi generați este 50, aceștia vor fi distribuiți la 4 cozi, iar timpul de simulare va fi de 25.

Urmărim evoluția cozilor pe parcursul trecerii timpului:



Și mesajul cu statistici de la finalul execuției simulării:



Astfel, putem evidenția diferite ipoteze de simulare, după anumite intervale pe care le considerăm importante și utile.

**Capitolul VI. *Concluzii***

Tema prezentată impune cunoașterea și aplicarea noțiunilor din paradigma programării orientate pe obiect. Așadar, rezolvarea cerinței permite aprofundarea și dezvoltarea capacităților de a scrie cod Java, identificarea cauzelor și tratarea problemelor ce pot apărea în procesul de implementare. Totodată, datorită implementării și a unei interfețe grafice, programatorul trebuie să conceapă o interfață inteligibilă prin care utilizatorul poate experimenta toate funcționalitățile puse în evidență.

Această temă este importantă deoarece utilizează concepte din programarea concurentă și anume lucrul cu firele de execuție ce permit programelor să execute mai multe secvențe de cod în același timp. Totodată, această problemă stă la baza modelării proceselor din viața reală și rezolvarea acesteia oferă un avantaj atât în evidențierea trăsăturilor programării concurente, cât și în abstractizarea cât mai eficientă a problemelor din viața reală pentru implementare software.

Aplicația curentă poate urma diferite etape de dezvoltare, atât pe parte grafică, cât și pe parte de eficientizare și algoritmică sau mapare pe probleme de gestiune din viața reală.

**Capitolul VII. *Bibliografie***

[1] <http://users.utcluj.ro/~crisb_pop/pt2018.html>

[2] <http://users.utcluj.ro/~igiosan/teaching_poo.html>

[3] <https://www.tutorialspoint.com/>