**Facultatea de Automatica și Calculatoare**

**Calculatoare și Tehnologia Informației**



**Tehnici de Programare**

**Documentație**

**Tema 2:**

**SIMULARE COZI**

**Profesor îndrumator: Realizat de:**

**Cristina Pop Câmpean Casiana Ștefana**

**Grupa: 30228**

**Cuprins**

**1. Obiectivul temei**

**2. Analiza problemei, asumptii, modelare, scenarii, cazuri de utilizare, erori**

**3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase,**

**interfete, relatii, packages, algoritmi, interfata utilizator, modul de tratare a erorilor)**

**4. Implementare**

**5. Testare**

**6. Rezultate**

**7. Concluzii**

**8. Bibliografie**

**1. Obiectivul temei**

Proiectați și implementați o aplicație de simulare pentru a analiza sistemele bazate pe cozi pentru a determina și minimiza timpul de așteptare al cliențiilor la coada.

**Obiective secundare:**

Studierea și învățarea proiectării aplicațiilor și a interfețelor grafice cu fire de execuție.

**2. Analiza problemei, asumptii, modelare, scenarii, cazuri de utilizare, erori**

Cozile sunt des folosite pentru a modela domenii reale din viața de zi cu zi. Obiecivul princlipal al unei cozi este să asigure un loc de așteptare pentru un client înainte să fie servit. Scopul aplicației este acela de a minimiza timpul de așteptare la coadă al unui client. O variantă de a reduce acest timp este aceea de a mări numarul cozilor, dar această soluție ar fi mai costistoare. Varianta implementată in această aplicație este aceea de a calcula timpul de așteptare al fiecarei cozi și de a trimite clientul sosit la coada cu timp de așteptare minim.

Aplicația ar trebui să simuleze cum mai mulți clienți ajung pentru a fi serviți, cum aceștia sunt puși la coada cu timp de așteptare minim, cum sunt serviți și în final, cum părpăsesc coada. Această aplicație trebuie să rețină cât timp stau clientii la coadă și să afișeze timpul mediu de așteptare. Pentru a calcula timpul de așteptare este nevoie de timpul de sosire și de timpul de servire și de plecare. Timpul de așteptare si de sevire depinde de fiecare client în parte. Timpul de plecare și de așteptare depinde de numarul de cozi, de numarul de clienți al fiecărei cozi și de timpul de servire al clienților.

Pentru această simulare, trebuie implementată o interfață grafică unde trebuie introduse urmatoarele:

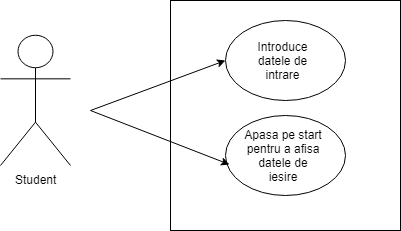
- minimul si maximul intervalului de sosire

- minimul si maximul intervalului intervalului de servire

- numărul cozilor

- timpul de simulare

După ce se termină simularea, trebuie să afișăm timul mediu de servire, timpul mediu de așteptare, evoluția cozilor, și timpul când a fost cel mai aglomerat.



Această aplicație este ușor de utilizat, trebuie doar să introducem datele de intrare corect și să apasăm pe start ca să ptem urmari simularea.

Use case: simulare cozi

Actor: studetul

Asumpții: date introduse corect

Scenariu succes:

1 . Studentul deschide interfața grafica

2 . Studentul introduce corect date de intrare, mai exact introduce timpii in secunde

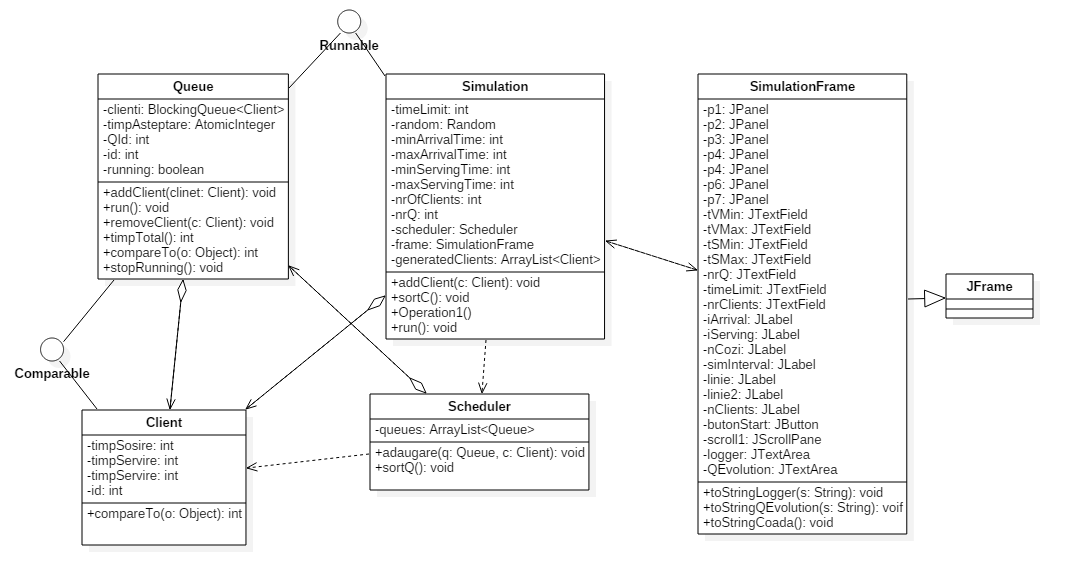
3 . Studentul apasa pe butonul de start si urmareste simularea in timp real a introducerii clientiilor in coada , si a iesirii acestora din coada

**3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator, modul de tratare a erorilor)**

**Decizii de proiectare**

Pentru a implementa un proiect de simulare al cozilor cu fire de execuție, am ales să implementez mai întai clasele Client și Queues. Fiecare coadă va conține o listă de clienți. Dupa care am implementat clasa Scheduler care va conține lista de cozi, la final am proiectat clasa Simulation unde am realizat simularea propriu-zisă a proiectlui cu thread-uri, mai apoi am realizat clasa SimulationFrame unde este proiectată interfața grafică.

**Diagrame UML**



**Structuri de date**

În acest proiect am folosit mai multe structuri de date, printre care ArrayList și BlockingQueue.

Acestea au fost folosite în clasa Queue, deoarece fiecare coadă va avea un BlockingQueue de Clienți deoarece cu ajutorul lor putem folosi cel mai bine firele de execuție, acestea fiind thread-safe, în clasa Scheduler, aceasta va conține un ArrayList alcatuit din toate cozile ultilizate la simulare, și în ultimul rând am folosit în clasa Simulation pentru că în această clasă generam un anumit număr de clienți care se vor afla in ArrayList-ul de clienți.

**Proiectare clase**

Am ales să proiectez 5 clase:Client, Queue, Scheduler, Simulation, SimulationFrame. Fiecare clasă este compusă din toate elementele necesare realizării simularii cerute. Pentru a simula venirea și servirea clienților la o mai multe cozi am proiectat clasele enumerate mai sus. Clasa client are atributele timpSosire, timpAsteptare cu care vom putea realiza simularea corectă, clasa Queue conține lista de Clienți, clasa Scheduler are lista de cozi, astfel vom putea trimite la fiecare coadă un client din lista aflata in clasa Simulation.

**Interfete**

Clasa Client și Queue implementează interfața Comparable, iar pe lângă aceasta, clasa Queue și Simmulation implemeteaza interfața Runnable.

**public** **class** Client **implements** Comparable<Object>

**public** **class** Queue **implements** Runnable, Comparable<Object>

**public** **class** Simulation **implements** Runnable

Interfețele Comparable au fost folosite pentru a sorta Clienții din lista generatedClients din clasa Simulation după timpul lor se sosire, pentru a fi introduși mai ușor in coadă la momentul venirii lor. Sortarea cozilor se realizează după timpul de așteptare, deoarece mereu va trebui să trimitem clientul sosit la coada cu timp de așteptare minim.

@Override

**public** **int** compareTo(Object o) {

**return** Integer.compare(**this**.timpSosire, ((Client)o).timpSosire);

}

@Override

**public** **int** compareTo(Object o) {

**return** Integer.compare(**this**.getTimpAsteptare().get(), ((Queue)o).getTimpAsteptare().get());

}

**Interfata utilizator**

Interfața utlizator este alcatuită din mai multe date de intrare care vor fi introduse de la tastatură: timpul minim si maxim al timpului de sosire și de servire al fiecarui client, numarul cozilor, numarul clienților și timpul de simulare. Toti timpii sunt reprezentati in secunde. La ieșire se va afișa evoluția cozilor, un jurnal al intrarii clientilor intr-o anumita coada si a timului de asteptare inainte de a fi serviti și la final timpul mediu de servire și de așteptare și timpul cel mai aglomerat. Toți timpii sunt calculați în secunde. Pentru a se afișa datele de ieșire, trebuie apasat butonul de Start. Acesta va porni simularea, astfel se vor porni thread-urile, pe ecran vor fi afișate date pe durata timpului de simulare.

**4. Implementare**

Proiectul conține 5 clase :

* Client
* Queue
* Scheduler
* Simulation
* SimulationFrame

**Clasa Client**

Clasa client este alcătuită din atributele timpSosire, timpAsteptare și idClient. Această clasă conține metodele de set și get pentru fiecare atribut, necesare pentru realizarea corectă a simularii cu fire de executie. Pe langă acestea este metoda suprascrisă compareTo, care este utilă pentru a sorta Clienții din lista aflată in Simulation. În clasa Simulation noi generam random un anumit număr de Clienți, pe care îi vom sorta după timpul lor de sosire, pentru că vom avea nevoie să accesăm mai ușor fiecare client care are timpul de venire egal cu timpul curent în simulare.

@Override

**public** **int** compareTo(Object o) {

**return** Integer.compare(**this**.timpSosire, ((Client)o).timpSosire);

}

**Clasa Queue**

Clasa Queue conține Qid, running, un BlockingQueue<Client>clienti, și un AtomicInteger timpAsteptare. În aceasta sunt implementate metodele addClient și removeClient care vor adăuga sau șterge un client in coadă, settere si gettere. Metoda suprascrisă compareTo este va fi folosită pentru a sorta cozile după timpul lor de așteptare, pentru că este necesar să trimitem fiecare client la coada cu timpul de așteptare cel mai mic.

@Override

**public** **int** compareTo(Object o) {

**return** Integer.compare(**this**.getTimpAsteptare().get(), ((Queue)o).getTimpAsteptare().get());

}

Clasa Queue implementează Runnable, astfel metoda suprascrisă run are rolul de a face threadul să aștepte un timp egal cu timpul de servire al clientului, după care acest client sa fie scos din BlockingQueue. Metoda stopRunning va seta atributul boolean running pe false. Acest atribut este folosit în metoda run, atata timp cat running este true, thread-ul va funcțtiona. Metoda stopRunning va fi folosită la finalul simularii.

**Clasa Scheduler**

Clasa Scheduler conține un ArrayList de cozi. Constructorul acestei clase va adauga un anumit număr de cozi in listă și va porni fiecare coadă.

**for**(Queue q:queues) {

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

t.start();

}

De asemenea, această clasă mai are implementate medoele adaugare, care adaugă o coada in lista, și sortQ, care va sorta lista de cozi după timpul de așteptare.

**Clasa Simulation**

Acesta clasă este cea mai importantă, deoarece aici realizăm efectiv simularea cerută. Această clasă conține atributele timeLimit, minArrivalTime, maxArrivalTime, minServingTime, maxServingTime, nrOfClients, nrQ care reprezinta datele de intrare necesare proiectului. Pe langă acestea avem un scheduler de tip Scheduler, frame de tip SimulationFrame, o listă de clienți generati aleator și random de tip Random.

Metoda de generare a clienților aleator este generateRandomClient. Aceasta va genera un anumit număr dat de clienți, setând pentru fiecare în mod aleator timpul de sosire și de servire.

int timpSosire = random.nextInt(maxArrivalTime - minArrivalTime) + minArrivalTime;

int timpServire = random.nextInt(maxServingTime - minServingTime) + minServingTime;

Clasa implementează Runnable, astfel metoda suprascrisă run va reprezenta simularea propriu-zisă a programului. Aceasta se va executa atâta timp cât timpul curentul nu este egal cu timpul de simulare. Toți timpii sunt în secunde. Astfel, dacă timpul de sosire al unui client este egal cu timpul curent, atunci acest client va fii introdus în coada cu timul de așteptare minim, după care vom afișa în interfață detaliile despre sosirea si timpul de așteptare al clientului,după care vom incrementa timpul curent, vom scadea timpul de așteptare al fiecarei cozi cu o secundă, vom afișa evoluția cozilor și vom pune thread-ul să aștepte o secundă. La finalul simularii, după ce toți clienții au fost scoși din coadă se va afișatimpul mediu de simulare și de servire si timpul cand a fost cel mai plin.

**Clasa SimulationFrame**

Această clasă realizează implementarea intefeței grafice, clasa exitinzând JFrame. Mai întâi am declarat toate componentele necesare pentru datelele de intrare și de ieșire ale proiectului:

-7 JPanel-uri, care reprezinta semnificatia fiecarui JtextField ;

-mai multe JTextField -uri unde se vor introduce timpul maxim si minim de servire si de sorire, numarul de cozi și de clienți și timpul de simulare ;

-Jbutton-ul de start care pornește simularea și care va afișa datele de ieșire ;

-2 JtextArea pe care vor fi afișate datele de ieșire ;

În constructorul clasei am inițializat toate componentele necesare, realizând design-ul interfeței. Aici am impementat pentru butonul de start un ActionListeler care va inițializa o nouă simulare cu datele de intrare introduse în interfață, afișând mai apoi datele de ieșire cerute. Datele introduse in simulare, mai exact timpii sunt reprezentati in secunde .

Simulations = **new**Simulation(simTime, minArrivalTime, maxArrivalTime, minServingTime, maxServingTime, nClienti, nrCozi) ;

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

t.start();

Pe lângă aceasta, am implementat și metodele toStringLogger, toStringQEvolution, și toStringCoada cu ajutorul cărora voi afișa datele cerute în JtextArea, metode aplicate in clasa Simulation. Aceste metode vor folosi metoda append :

logger.append( s );

**5. Testare**

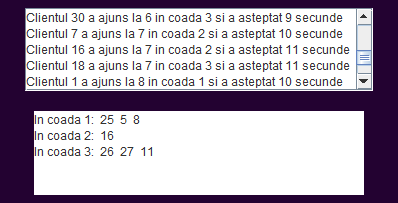
Testarea proiectului a fost realizată pentru fiecare clasă in parte în main. Mai întai am testat dacă se introduc clientii in cozi, după care am testat și alte metode, cum ar fi sortarea clienților și a cozilor, la final am realizat testarea in main-ul clasei simulation. Acolo am afișat fiecare client cum intra intr-o anumită coadă, am afișat tot în consolă timpii de asteptare si alte informatii necesare pentru verificare corectitudii algoritmilor.

După ce am corectat greșelile și după ce am verificat corectitudinea datelor afișate în consolă am realizat afișarea în interfață.

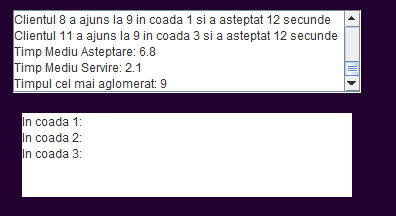
**6. Rezultate**

Datele rezultate în interfață sunt corecte, afișările se realizează în timp real, pe durata timpului de simulare, la finalul acesteia fiind observate și timpul medi de sosire si servire si secunda cea mai aglomerată.

În poză este capturat un anumit moment din simulare în care cozile sunt ocupate cu clienți.



Acestă poză este de la finalul simulării, unde se vede că în coadă nu mai este nici un client, fiind afișate timpul mediu de servire și de așteptare si timpul cel mai aglomerat.



**7. Concluzii**

În concluzie, din acest proiect am învățat pentru prima oară despre thread-uri și despre cum se folosesc acestea. De asemenea am realizat afișarea în timp real în interfață pentru,ceea ce a fost nou pentru mine. Am reușit să îmi imbunatatesc abilitatile de realizare a unei interfete, neavand experinta foarte vastă cu acestea.

Există mai multe dezvoltări ulterioare, cum ar fi trimiterea clientilor la coada cu cu numarul clienților minim, sau afișarea a mai multor detalii legate de simulare.

**8. Bibliografie**

1. <https://docs.oracle.com/javase/tutorial/essential/concurrency/>
2. <https://www.youtube.com/watch?v=YdlnEWC-7Wo>
3. <https://www.youtube.com/watch?v=X5Q-Mecu_64>
4. <https://www.journaldev.com/1034/java-blockingqueue-example>
5. <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html>
6. <http://inf.ucv.ro/documents/tudori/laborator8_53.pdf>
7. <https://blog.trigent.com/concurrency-in-java-swing/>
8. <https://docs.oracle.com/javase/tutorial/essential/concurrency/simple.html>