

**FACULTATEA DE AUTOMATICA SI CALCULATOARE**

**DEPARTAMENTUL CALCULATOARE**

**DISCIPLINA TEHNICI DE PROGRAMARE**

**Documentatie Tema 2**

**Aplicatie pentru simularea cozilor**

Pavel Madalina Adriana

Grupa 30224

**CUPRINS**

**1. OBIECTIVUL TEMEI ……………………………………………………………………..3**

**2. ANALIZA PROBLEMEI, MODELARE, SCENARII, CAZURI DE UTILIZARE …...4**

**3. PROIECTARE (DECIZII DE PROIECTARE, DIAGRAME UML, STRUCTURI DE DATE, PROIECTARE CLASE, INTERFETE, RELATII, PACKAGES, ALGORITMI, INTERFATA UTILIZATOR) …………………………..……………………………………5**

**4. IMPLEMENTARE …………………………………………………………………………9**

**5. REZULTATE …………………………………….………………………………………..11**

**6. CONCLUZII .……………………………………………………………………………....13**

**7. BIBLIOGRAFIE …………………………………………………………………………..13**

**1.OBIECTIVUL TEMEI**

Enuntul temei este urmatorul :

Proiectati si implementati o aplicatie de simulare care vizeaza analiza sistemelor bazate pe cozi pentru determinarea si minimizarea timpului de asteptare al clientilor.

Obiectivul principal al unei cozi este de a oferi un loc unui client pentru a astepta inainte de a fi servit. Gestionarea sistemelor bazate pe cozi urmareste minimizarea timpului pe care clientul il petrece asteptand la coada sa fie servit. O modalitate de a minimiza timpul de asteptare este de a adauga mai multe cozi. Aplicatia trebuie sa simuleze o serie de clienti care sosesc pentru un serviciu, se aseaza la o coada asteptand sa fie serviti, iar dupa ce un client este servit, acesta paraseste coada.

Cerintele impuse sunt : aplicatia trebuie sa includa o interfata grafica care permite vizualizarea evolutiei in timp real a cozilor respective, de asemenea datele de intrare trebuie sa fie introduse tot prin intermediul interfetei; aplicatia trebuie sa contina mai multe cozi, iar pentru lucrul in paralel se vor utiliza firele de executie (thread-uri); evolutia fiecare cozi trebuie monitorizata prin intermediul jurnalizarii.

Obiectivele secundare sunt urmatoarele :

* Dezvoltarea de use-case-uri si scenarii (Capitolul 2)
* Alegerea structurilor de date (Capitolul 3)
* Impartirea pe clase, utilizarea interfetelor, evidentierea relatiilor (Capitolul 3)
* Dezvoltarea algoritmilor utilizati (Capitolul 4)
* Testarea solutiei (Capitolul 5)

**2.ANALIZA PROBLEMEI, MODELARE, SCENARII, CAZURI DE UTILIZARE**

Inainte de a porni simularea, aplicatia trebuie sa cunoasca cati clienti vor aparea in total. Acestia nu vor sosi deodata si nici la momente regulate de timp, ci fiecare client va sosi la un moment de timp stabilit aleator. Fiecare client are in plus un id pentru a putea fi identificat, un timp necesar servirii acestuia (si acest timp de procesare este stabilit aleator) si momentul de timp in care va parasi coada (acest timp este calculat in functie de ceilalti timpi). In momentul in care clientul ajunge in cadrul actiunii, el este nevoit sa aleaga coada la care trebuie sa astepte pana va fi servit. In cazul in care exista o singura coada, clientul se aseaza la aceasta. In cazul in care exista mai multe cozi, clientul trebuie sa decida la ce coada sa se aseze. Astfel, clientul va urmari mereu sa se aseze la coada la care ar astepta cel mai putin timp pana ar ajunge sa fie servit. In cazul in care toate cozile sunt pline, va astepta pana se va elibera o coada. In momentul in care ajunge sa fie servit, clientul asteapta timpul necesar procesarii, dupa care paraseste coada. Limitele pentru timpii de sosire si procesare, precum si timpul in care sa se efectueze simularea si numarul total de clienti si de cozi, pot fi setate prin intermediul unei interfete grafice puse la dispozitie de aplicatie, interfata in care se vor putea vizualiza si etapele simularii.

**3. PROIECTARE (DECIZII DE PROIECTARE, DIAGRAME UML, STRUCTURI DE DATE, PROIECTARE CLASE, INTERFETE, RELATII, PACKAGES, ALGORITMI, INTERFATA UTILIZATOR)**

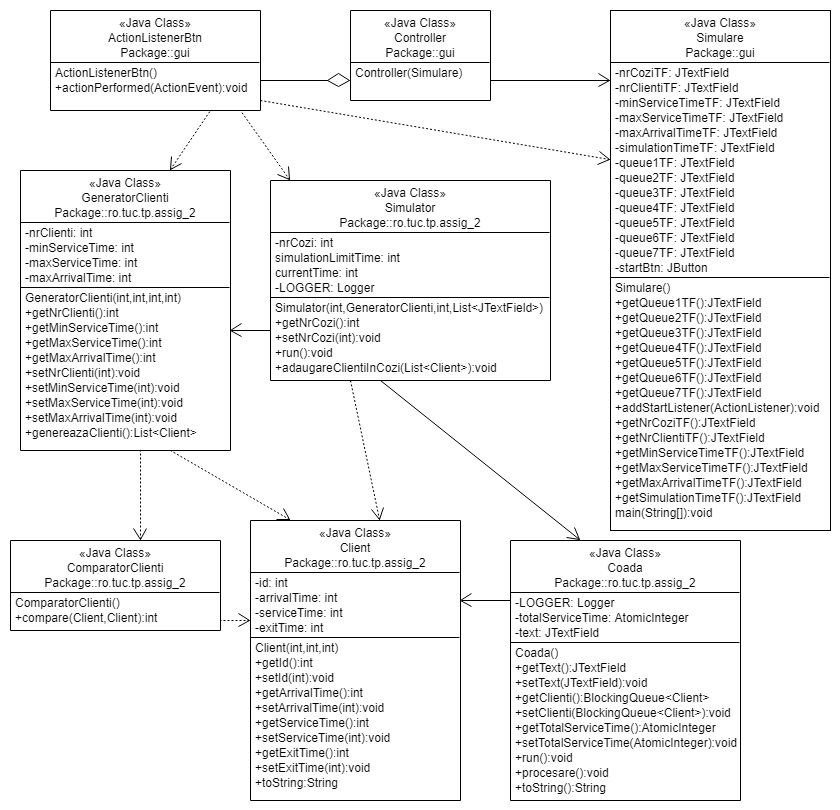
Pentru implementarea acestei teme am ales sa folosesc thread-uri (fire de executie) pentru a gestiona activitatea fiecarei cozi, in vederea realizarii unei simulari cat mai apropiate de realitate, in care cozile dintr-un magazin servesc clientii in paralel si independent unele de altele, in cadrul aceluiasi magazin. Simulatorul, cel care se ocupa de distribuirea clientilor in cozi si de managementul timpului corespunzator simularii, este de asemenea definit cu ajutorul unui thread. Thread-urile prezinta acest avantaj de a permite o executie oarecum concurenta in cadrul aceluiasi proces.

Clasele utilizate in realizarea acestei aplicatii sunt :

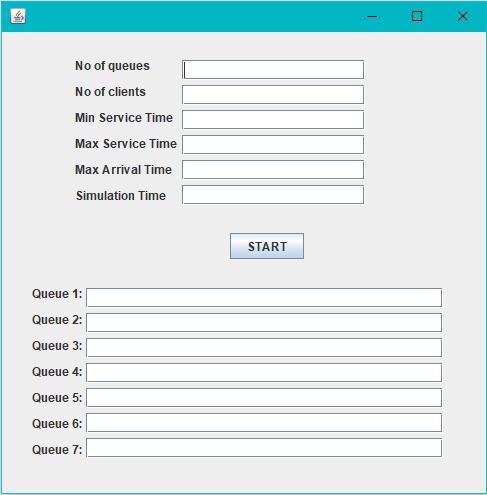
1. **Clasa Client** - contine 4 variabile instanta, care retin datele de interes in cadrul acestei aplicatii despre un client: id-ul acestuia (de tip int) si timpii necesari simularii : arrivalTime, serviceTime si exitTime (de tip int). Pe langa constructor, clasa contine si settere si gettere, pentru a modifica si accesa aceste variabile din exterior, si metoda toString, utilizata pentru afisarea datelor despre client.
2. **Clasa Coada**  - contine 4 variabile instanta : lista de clienti (de tipul BlockingQueue<Client>, un obiect de tipul Logger, timpul total pe coada (de tipul AtomicInteger) si un obiect de tipul JTextField (face corespondenta cu GUI pentru afisare). Pentru stocarea clientilor din coada am ales structura de date BlockingQueue deoarece aceasta asigura sincronizarea, problema ce apare foarte des in momentul rularii mai multor thread-uri. In plus, aceasta interfata pune la dispozitie metoda take(), care are marele avantaj ca va pune thread-ul in asteptare daca nu va avea niciun element disponibil, pana cand se vor adauga elemente in lista. Obiectul de tip Logger este utilizat pentru a afisa evolutia fiecarui client, care consta in momentul in care acesta ajunge sa fie servit la casa si momentul in care paraseste coada. Variabila responsabila cu stocarea timpului total de procesare al cozii este de tip AtomicInteger pentru a proteja aceasta resursa in momentul in care 2 thread-uri ar dori sa o modifice in acelasi timp, asigurand totodata sincronizarea thread-urilor, operatiile executate pe aceasta variabila fiind atomice. Clasa implementeaza interfata Runnable si descrie corpul metodei run(), din care este apelata metoda procesare(), cu rolul de a servi un client si de a-l elimina din coada dupa ce acesta a fost procesat. Pe langa constructor, clasa contine si metode de get si set pentru variabilele instanta.
3. **Clasa ComparatorClienti**  - nu are variabile instanta. Aceasta clasa implementeaza interfata Comparator<Client> si descrie corpul metodei compare, prin intermediul careia se realizeaza compararea clientilor in functie de arrivalTime.
4. **Clasa GeneratorClienti**  - are 4 variabile instanta, toate de tipul int, care au rolul de a stoca numarul total de clienti care vor aparea pe parcursul simularii, timpul minim de procesare al unui client, timpul maxim de procesare al unui client si momentul de timp maxim pana la care pot aparea clienti in cadrul simularii. Pe langa constructor, metode de get si set pentru variabilele instanta, clasa contine si o metoda numita genereazaClienti(), care returneaza o lista de clienti.
5. **Clasa Simulator**  - are 6 variabile instanta : nrCozi (de tipul int), care retine cate cozi se vor deschide pentru simulare, cozi (de tipul List<Coada>) care va contine cele nrCozi cozi, un obiect de tipul GeneratorClienti responsabil cu generarea clientilor necesari simularii, 2 variabile statice (de tip int) simulationLimitTime si currentTime responsabile cu contorizarea secundelor simularii si impunerea unei limite de timp la care sa se opreasca simularea, dar si un obiect de tipul Logger pentru afisarea timpului simularii si al momentului de timp la care se aseaza un client la coada, toate aceste afisari avand ca scop prezentarea evolutiei simularii pas cu pas. Clasa implementeaza interfata Runnable, fiind responsabila cu managementul cozilor si introducerea in timp a clientilor in cozi. Respectand contractul impus de aceasta interfata, clasa implementeaza metoda run(), care in interiorul ei va apela metoda adaugareClientiInCozi(List<Client> listaClienti), responsabila cu distribuirea clientilor in cozile deschise.
6. **Clasa Simulare**  - este responsabila cu view-ul pentru interfata grafica a aplicatiei; are variabile instanta : cate un JTextField pentru : numarul de cozi, numarul total de clienti, timpul minim de procesare al unui client, timpul maxim de procesare al unui client, momentul de timp la care poate aparea ultimul client si limita de timp pana la care sa se simuleze evolutiile cozilor, toate acestea reprezentand datele de intrare ale aplicatiei, ce trebuie introduse de utilizator, si cate un JTextField pentru fiecare coada; un obiect de tipul JButton pentru butonul de start, care se va apasa abia dupa ce toate datele de intrare au fost introduse in mod corect. Pe langa constructorul in care se creeaza fereastra aplicatiei, aceasta clasa mai contine si gettere si settere pentru variabilele instanta, dar si o metoda care adauga un actionListener butonului de start. Metoda main a acestei clase lanseaza in executie aplicatia.
7. **Clasa Controller**  - are ca variabila instanta un obiect de tipul Simulare; aceasta clasa are rolul de a controla butonul de start al interfetei. Clasa contine o clasa interna numita **ActionListenerBtn**, care implementeaza interfata ActionListener, descriind corpul metodei actionPerformed(ActionEvent e), contract impus de interfata.

Clasele prezentate la numerele 1.-5. constituie Modelul problemei, clasa Simulare are rolul de a desena fereatra aplicatiei (imaginea acesteia), iar clasa Controller are rolul de a face legatura dintre Model si View. Astfel, aplicatia prezinta o implementare de tip MVC, implementare ce protejeaza informatiile cu privire la modul de implementare al operatiilor.

Clasele prezentate la numerele 1.-5. se regasesc in pachetul ro.tuc.pt.assig\_2, iar clasele prezentate la numerele 6.-.8 fac parte din pachetul gui. Diagrama UML este :



Interfata utilizator este urmatoarea :



**4. IMPLEMENTARE**

Metodele implementate in fiecare clasa sunt :

1. Clasa Client

* String toString() – returneaza un String reprezentand id-ul clientului, de forma : “id:” + id.

1. Clasa Coada

* void run() – metoda declarata in interfata Runnable, interfata ce este implementata de aceasta clasa. In cadrul acestei metode este definit comportamentul unui thread, in cazul nostru, al cozii. In cadrul acestei metode, cat timp thread-ul este in “alive”, se apeleaza metoda void procesare().
* synchronized void procesare() throws InterruptedException – metoda care descrie modul in care coada proceseaza clientul : coada ia clientul din capul cozii, pune thread-ul in starea “sleep” cat timp trebuie procesat acest client, moment in care decrementeaza timpul total al cozii. Dupa ce procesarea clientului s-a terminat ( thread-ul a stat in starea “sleep” serviceTime-ul clientului ), coada il elimina, fiind mai apoi pregatita sa extraga urmatorul client. Metoda arunca exceptia InterruptedException datorita apelului metodei take() in cadrul acesteia, care extrage cate un client din capul cozii. Aceasta metoda este sincronizata deoarece procesarea clientului de catre un thread (o coada) nu trebuie sa fie intrerupta de alt thread.
* String toString() – metoda are rolul de a afisa lista clientilor, folosindu-se de metoda toString() din clasa Client.

1. Clasa ComparatorClienti

* int compare(Client o1, Client o2) – metoda stabileste modul in care se compara doi clienti in functie de arrivalTime-ul lor, pentru a se putea face sortarea acestora. Metoda este declarata in interfata Comparator<Client> si este implementata de aceasta clasa deoarece Comparator<Client> implementeaza interfata Comparator<Client>.

1. Clasa GeneratorClienti

* List<Client> genereazaClienti() – metoda creeaza o lista de clienti, din care Simulatorul va extrage cate unul si il va adauga intr-una din cozi. Valorile de arrivalTime si serviceTime sunt alese random pentru o simulare cat mai apropiata de realitate, iar pentru a asigura unicitatea id-ului, acesta se va da in ordine fiecarui client. Inainte de a returna lista de clienti generati random, metoda ii va sorta in functie de arrivalTime, pentru a usura munca Simulatorului, care va trebui doar sa extraga cate un client din aceasta lista.

1. Clasa Simulator

* void run() – metoda declarata in interfata Runnable, interfata ce este implementata de aceasta clasa. In cadrul acestei metode este definit comportamentul unui thread, in cazul nostru, simulatorul. In cadrul acestei metode, se creeaza thread-urile pentru cozi, apoi se apeleaza pentru fiecare metoda start(). In continuare, se va instantia o lista de clienti, care va stoca clientii generati de generatorul de clienti, iar cat timp simularea nu se va termina (deoarece impunem o limita de timp), se vor introduce clienti in cozi, prin apelul metodei adaugareClientiInCozi(List<Client> listaClienti), dupa care thread-ul este pus pe “sleep”.
* void adaugareClientiInCozi(List<Client> listaClienti) – metoda va verifica totalServiceTime-ul de pe fiecare coada si va adauga clientul in acea coada care are timpul minim, deoarece presupunem ca fiecare client se gandeste sa se aseze la coada la care ar trebui sa astepte cel mai putin timp. Metoda se ocupa si cu actualizarea valorilor de timp care se modifica in urma acestei inserari pe coada aleasa.

1. Clasa Simulare

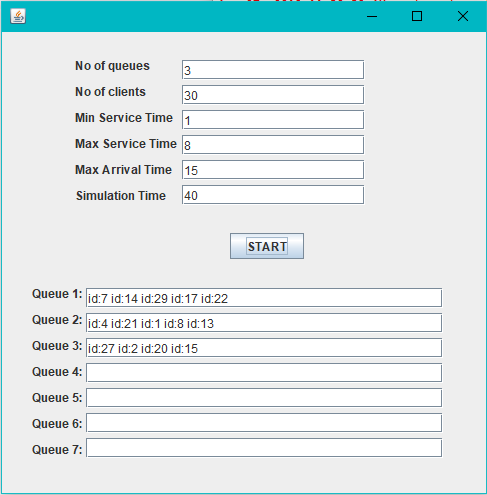
* static void main(String[] args) – metoda din care este lansata in executie aplicatia.

1. Clasa ActionListenerBtn

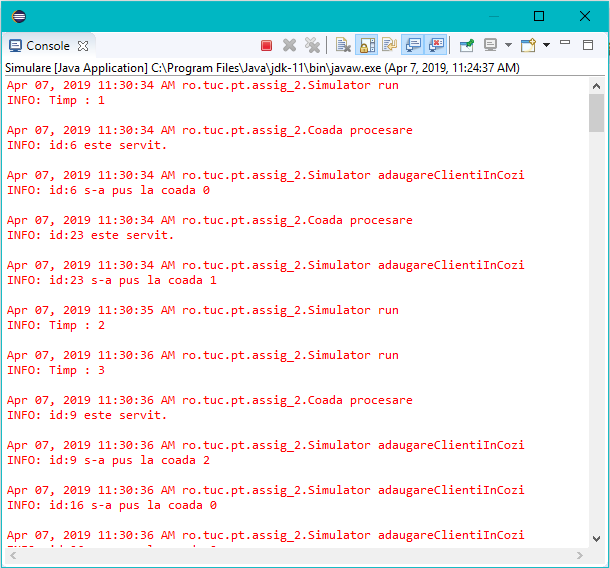
* void actionPerformed(ActionEvent e) – metoda extrage datele de intrare din textField-uri si instantiaza generatorul si thread-ul simulatorului, dupa care apeleaza metoda start() pentru acest thread.

**5. REZULTATE**

Din cauza datelor generate random si a comportamentului specific al thread-urilor, rezultatele nu pot fi identice de la o rulare la alta. Asadar, un posibil rezultat este ilustrat mai jos :



In plus, evolutia cozilor poate fi urmarita si prin vizualizarea mesajelor afisate in consola:



**6. CONCLUZII**

In concluzie, aceasta tema m-a ajutat sa imi imbunatatesc cunostiintele legate de conceptul de thread, modul in care functioneaza, utilitatea acestora si modul in care se pot sincroniza.

Printre posibilitatile de dezvoltare ulterioara se numara : ilustrarea simularii sa se reprezinte printr-o animatie, in care sa se poata observa modul in care clientii apar, se aseaza si asteapta la coada, iar dupa ce sunt procesati, pleaca; interfata sa afiseze si alte posibile date de interes, cum ar fi : timpul mediu de asteptare la o coada sau ora de varf, la care au fost cei mai multi clienti.

**7. BIBLIOGRAFIE**

* Materialele de curs si laborator de OOP
* <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>
* <https://docs.oracle.com/javase/7/docs/api/java/io/OutputStream.html>
* <https://www.codejava.net/java-se/swing/redirect-standard-output-streams-to-jtextarea?fbclid=IwAR0G0GRNS9IETgRFTWBcvH4TdN7zCHF43S1332Uieartd4XOJJBJr-DmauE>