Universitatea Tehnică Cluj-Napoca

Documentaţie

Tema 2

Aplicatie de simulare a unui sistem de cozi

Mureşan Ioana Diana

CTI-Romana

Gr.30227

Cuprins

1.Cerinte Functionale................................................... 3

2.Obiective ...................................................................3

2.1.Obiectiv Principal.................................................3

2.2.Obiective Secundare............................................3

3.Analiza Problemei ......................................................3

4.Proiectare ..................................................................5

4.1.Structuri de date..................................................5

4.2.Diagrama de clase................................................5

5.Implementare.............................................................7

6.Rezultate ..................................................................13

7.Concluzii si Dezvoltari Ulterioare..............................14

8.Bibliografie ..............................................................14

**1.Cerinte Functionale**

Proiectarea unei aplicatii de simulare care analizeaza sistemul de cozi pentru determinarea si minimizarea timpului de asteptare al clientilor.

**2.Obiective**

**2.1. Obiectiv principal**

Obtinerea unei aplicatii care simuleaza un sistem de cozi si arata in timp real procesarea clientilor.

Obiectivele obligatorii pentru acceptarea temei sunt: respectarea paradigmelor OOP, aplicatia organizata pe pachete, clase de maxim 300 de linii(exceptie clasele de UI), metode de maxim 30 de linii(exceptie case cu switch-uri multe) si Java naming conventions.

Cerintelele pentru nota 5 sunt: implementarea interfetei grafice care afiseaza in timp real evolutia cozilor, redactarea unei documentatii, implementarea unei componente care genereaza aleator clienti, cel putin un fir de executie si log-ul de evenimente, iar mai departe se vor implementa si alte operatii pentru o nota mai buna.

**2.2. Obiective secundare**

Pentru atingerea obiectivului final se parcurg urmatoarele etape:

-alegerea structurilor de date

-impartirea pe clase

-dezvoltarea algoritmilor

-implementarea solutiei

-testare

**3.Analiza Problemei**

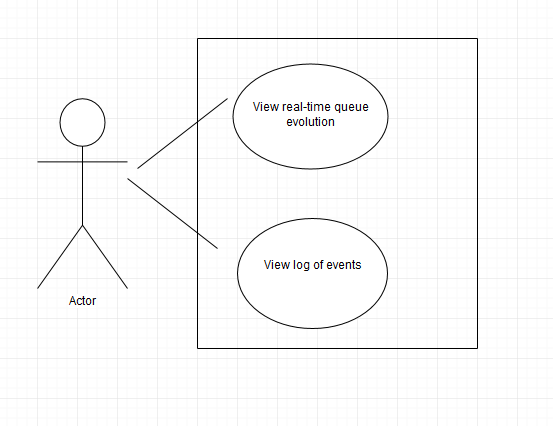
In prima faza, de analiza a cerintei ne vom raporta la conceptele programarii orientate pe obiecte care au la baza ideea unificarii datelor cu modalitatile de prelucrarea a acestora si manevreaza enititati reprezentate sub forma de obiecte.Programarea Obiectuala ofera posibilitati de modelarea a obiectelor, a proprietatilor si a realtiilor dintre ele, dar si posibilitatea de a descompune o problema in componentele sale.

Vom studia atent enuntul problemei pentru a putea stabili care sunt obiectele cu care vom lucra mai departe, cum vor interactiona unele cu altele si ce metode pot fi implementate .Astfel putem spune deja ca vor exista clasele Task si Server.

Datorita faptului ca exista mai multe servere care trebuie procesate simultan trebuie folosit conceptul de Thread. Conceptul de **thread** (fir de execuție) definește cea mai mică unitate de procesare ce poate fi programată spre execuție de către [sistemul de operare](https://ro.wikipedia.org/wiki/Sistem_de_operare). Este folosit în programare pentru a eficientiza execuția programelor, executând porțiuni distincte de cod [în paralel](https://ro.wikipedia.org/wiki/Calcul_paralel) în interiorul aceluiași [proces](https://ro.wikipedia.org/w/index.php?title=Proces_(sisteme_de_operare)&action=edit&redlink=1). Câteodata însă, aceste portiuni de cod care constituie corpul threadurilor, nu sunt complet independente și în anumite momente ale execuției, se poate întampla ca un thread să trebuiască să aștepte execuția unor instructiuni din alt thread, pentru a putea continua execuția propriilor instrucțiuni. Această tehnică, prin care un thread asteaptă execuția altor threaduri înainte de a continua propria execuție, se numește sincronizarea threadurilor.

În limbajul Java, threadurile sunt un element esențial pentru execuția paralelă, întrucât o aplicație ce rulează în mașina virtuală Java reprezintă un singur proces. Clasa java.lang.Thread poate fi utilizată pentru implementarea unui fir de execuție separat. Acțiunile ce se execută pe thread se pot defini în metoda run() a acestei clase (sau în cea a unui obiect ce implementează java.lang.Runnable și care este pasat ca argument constructorului clasei java.lang.Thread). Firul de execuție se pornește prin apelul metodei start() a acestuia.

**Diagrama use case**

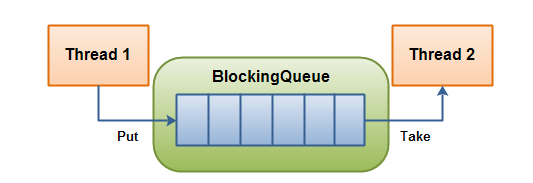


Utilizatorul va introduce numarul de task-uri, numarul de servere, intervalul de simulare, timpul maxim si minim de procesare a unui task.

**4.Proiectare**

**4.1.Structuri de date**

Pentru evitarea implementarii unor metode sincronizate care adauga sau sterg un element dintr-o coada am ales sa folosesc o structura de tip BlockingQueue pentru reprezentarea cozilor.



Thread-ul care produce si insereaza obiecte noi intr-o coada va continua sa faca acest lucru pana cand coada atinge numarul maxim de elemente pe care il poate contine, si atunci thread-ul se blocheaza pana cand thread-ul care sterge elemente din coada va elimina un obiect.

Daca thread-ul care sterge elemente din coada incearca sa stearga cand coada este goala, acesta va fi blocat si va astepta pana cand celalalt thread va insera un nou obiect in coada.

Am folosit aceasta structura pentru atributul tasks din clasa Servers:

**private** BlockingQueue<Task> tasks;

**4.2.Diagrama de Clasa**

Diagramele de clase sunt folosite pentru a specifica structura statica a sistemului, adica ce clase exista in sistem si care este legatura dintre ele.

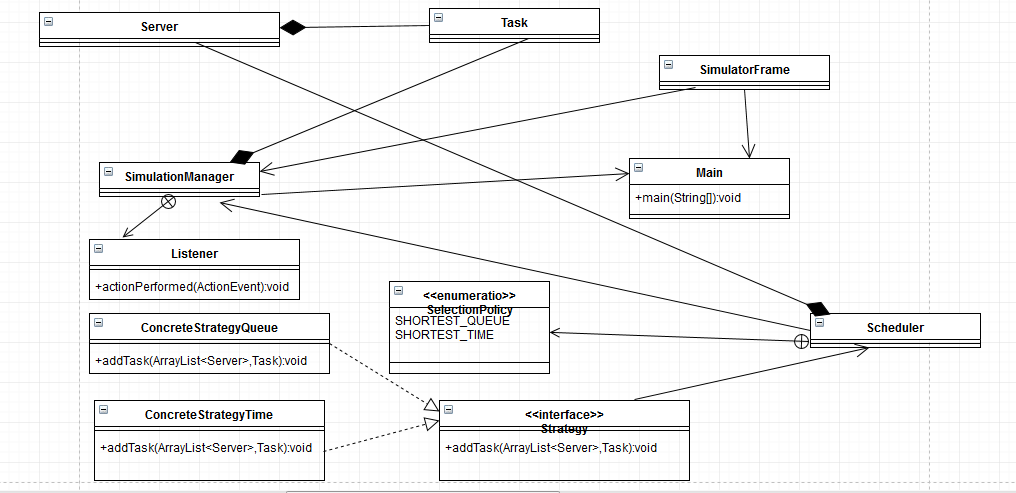
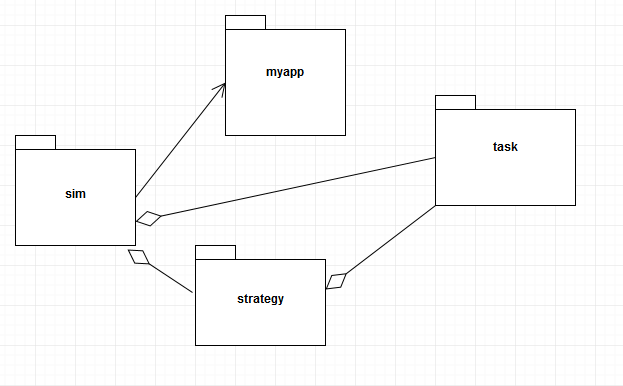


Diagrama de pachete

****

**5.Implementare**

In pachetul task am implementat clasele Task si Server.

**Clasa Task**

In aceasta clasa avem urmatoarele atrbute: **private** **int** arrivalTime;

**private** **int** processingTime;

**private** **int** waitingTime;

**private** **int** finishingTime;

**private** **int** id;

Clasa implementeaza interfata Comparable pentru a putea suprascrie metoda int compareTo astfel incat sa putem sorta mai tarziu lista de Task-uri in ordine crescatoare dupa timpul de sosire.Am implementat de asemenea gettere si settere, metoda displayTask() pe care am folosit-o sa afisez datele in log-ul de evenimente si metoda setFinishingTime() care seteaza timpul de finalizare ca o suma intre timpul de sosire, timpul de asteptare si timpul de procesare.

**Clasa Server**

Clasa Server implementeaza Runnable pentru ca mai tarziu sa putem crea obiecte Server de tip Runnable ce sunt folosite la crearea thread-urilor.Am definit 3 atribute:

**private** BlockingQueue<Task> tasks;

**private** AtomicInteger waitingTime;

**private** **int** id;

Motivul pentru care am folosit o structura de tip BlockingQueue pentru a reprezenta cozile este cel mentionat mai sus, si anume atunci cand coada isi atinge capacitatea sa nu se mai poata insera elemente pana cand nu se sterge unul sau atunci cand devine goala sa nu se mai poata sterge elemente pana cand nu se introduce alt element.Variabila waitingTime retine timpul de asteptare al server-ului.

Metoda **public void addTask(Task n)** va seta timpul de asteptare al task-ului cu timpul de asteptare curent al server-ului, apoi va calcula timpul de finalizare si va adauga task-ul la server.Totodata timpul de asteptare al server-ului se va incrementa cu timpul de procesare al task-ului adaugat.

Metoda **run():**cat timp server-ul contine task-uri se va pune thread-ul pe sleep un timp de secunde egal cu timpul de procesare al task-ului aflat la inceputul cozii dupa care va fi scos din coada.Timpul de asteptare al server-ului va fi decrementat cu timpul de procesare al task-ului procesat.

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

{

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

{

**while** (tasks.isEmpty()!=**true**)

{

Task t=tasks.element();

**if** (t!=**null**)

{

**try** {

Thread.*sleep*(1000\*t.getprocessingTime());

}**catch** (InterruptedException e) {

e.printStackTrace();

}

tasks.remove();

**this**.setWaitingTime(**this**.waitingTime.get()-t.getprocessingTime());

//timpul de asteptare al server-ului se decrementeaza cu timpul de procesare al task-ului finalizat

}

}

}

}

Metoda **toString()** este folosita la afisarea in timp real a evolutiei cozilor.

In pachetul Strategy am implementat o interfata Strategy cu metoda addTask(ArrayList<Server> servers,Task t) care va fi implementata in functie de strategia aleasa.

Am implementat doar strategia care distribuie task-ul la server-ul cu numarul cel mai mic de task-uri in clasa ConcreteStrategyQueue.

**public** **void** addTask(ArrayList<Server> servers, Task t) {

//adaug la server-ul cu cele mai putine task-uri

**int** size=servers.get(0).getTasks().size();

**for** (Server s:servers) //caut minimul

{

**if** (size>s.getTasks().size())

size=s.getTasks().size();

}

**for** (**int** i=0;i<servers.size();i++) { //adaug la server

**if** (servers.get(i).getTasks().size()==size)

{

servers.get(i).addTask(t);

**break**;

}

}

}

**Clasa Scheduler**

Contine un atribut ArrayList<Server> servers, numarul maxim de servere si numarul maxim de tasks/server , un atribut de tip Strategy si o structura de tip:

**public** **enum** SelectionPolicy{

***SHORTEST\_QUEUE***,***SHORTEST\_TIME***;

}

Constructorul acestei clase va primi ca si parametri numarul de servere si numarul maxim de tasks/server si pentru fiecare server va crea un thread folosind un obiect Runnable de tip Server :

**public** Scheduler(**int** maxNoServers,**int** maxTasksperServer)

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

{ Server s=**new** Server(i);

servers.add(s);

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

thread.start();

}

}

Metoda **void changeStrategy(SelectioPolicy policy)** va alege strategia de distribuire a task-urilor la servere si va crea fie un obiect de tipul ConcreteStrategyQueue fie un obiect ConcreteStrategyTime in variabila strategy.

In functie de obiectul creat metoda **void dispatchTask(Task t)** va apela metoda addTask corescpunzatoare strategiei(implementata doar strategie bazata pe nr de task-uri) care va adauga task-ul la unul din serverele din lista servers.

In pachetul sim am implementat clasele SimulatorFrame si SimulationManager.

**Clasa SimulationManager**

Contine componenta **generateRandomTasks()** care va genera n clienti cu timpul de sosire si timpul de procesare astfel:

-timpul de procesare:intre minim si maxim introduse de utilizator

-timpul de sosire :intre 1 si intervalul de simulare

Se va sorta lista de task-uri crescator dupa timpul de sosire

**public** **void** generateRandomTasks()

{

Random rand=**new** Random();

**for** (**int** i=0;i<clientsN;i++)

{

**int** n=0;

n= rand.nextInt((maxProcTime - minProcTime) + 1) + minProcTime;

**int** processingTime=n;

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

n = rand.nextInt((timeLimit - 1) + 1) + 1;

**int** arrivalTime=n;

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

Task t=**new** Task(i+1,arrivalTime,processingTime);

System.***out***.println(t.toString());

**this**.generatedTasks.add(t);

}

Collections.*sort*(generatedTasks);

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

}

Metoda **displayFields()** va afisa in text field-ul corespunzator fiecarui server lista de task-uri la un moment dat al evolutiei.

In metoda **run()** se va parcurge intervalul de simulare ales de utilizator si se vor retine in lista sametime task-urile care au acelasi timp de sosire, iar apoi pentru fiecare task se va apela metoda dispatchTask care il va distribui la un anumit server, se vor afisa informatiile in log-ul de evenimente si se va pune thread-ul pe sleep pentru o secunda.Totodata am folosit metoda displayRemoved care va afisa task-urile eliminate din servere la un moment dat.Dupa scurgerea intervalului de simulare voi afisa cozile pana cand aceste devin goale.

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

{

**int** currentTime=1;

ArrayList<Task> deleted=**new** ArrayList<Task>();

**while**(currentTime<=timeLimit)

{

String s="Time="+currentTime;

frame.addLog(s);

ArrayList<Task> sametime=**new** ArrayList<Task>();

displayFields();

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

**for**(Task t:**this**.generatedTasks)

{

**if** (t.getarrivalTime()==currentTime)

{

sametime.add(t);

}

}

**for** (Task tt:sametime)

{

**this**.scheduler.dispatchTask(tt);

frame.addLog(tt.displayTask());

deleted.add(tt);

**this**.generatedTasks.remove(tt);

}

displayFields();

displayRemoved(deleted,currentTime);

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

currentTime++;

}

**int** empty=0;

**while**(empty==0)

{

**int** nr=0;

**for** (**int** i = 0; i <scheduler.getServers().size(); i++)

{

**if** (!(scheduler.getServers().get(i).getServer().isEmpty()))

{

**try** {

displayFields();

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**else** nr++;

}

**if** (nr==scheduler.getServers().size())

empty=1;

}

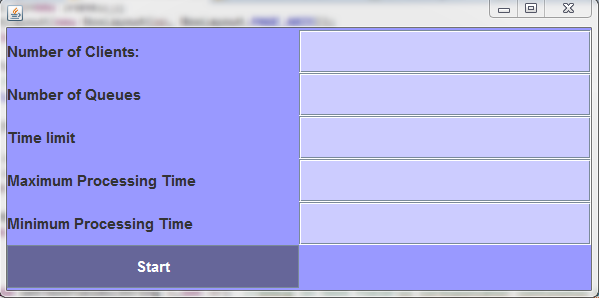
displayFields();

}

**Implementarea Interfetei Grafice**

Interfata grafica este descrisa in clasa SimulatorFrame.Initial, la rulare va aparea o fereastra in care utilizatorul trebuie sa introduca numarul de clienti, numarul de cozi, intervalul de simulare, timpul maxim de procesare si timpul minim de procesare pentru care se doreste afisarea simularii evolutiei cozilor.

Se presupune ca utilizatorul va introduce date corecte, reprezentate de numere intregi positive.



La apasarea butonului Start va incepe simularea si va aparea o noua fereastra deoarece am creat o metoda in care am construit fereastra din TextField-uri JLabels si log-ul de evenimente iar frame-ul se va face vizibil doar la apasarea butonului cand este apelata metoda:

**public** **void** openFrame(**int** s) //construiesc frame-ul cu serverele si log-ul de evenimente si il fac vizibil

{

JPanel p3=**new** JPanel();

p3.setLayout(**new** BoxLayout(p3,BoxLayout.***X\_AXIS***));

p3.setBackground(**new** Color(204, 204, 255));

p3.add(**this**.printLabels(s));

p3.add(**this**.printTextFields(s));

JPanel p4=**new** JPanel();

p4.setLayout(**new** BoxLayout(p4,BoxLayout.***Y\_AXIS***));

p4.add(p3);

p4.add(logpane);

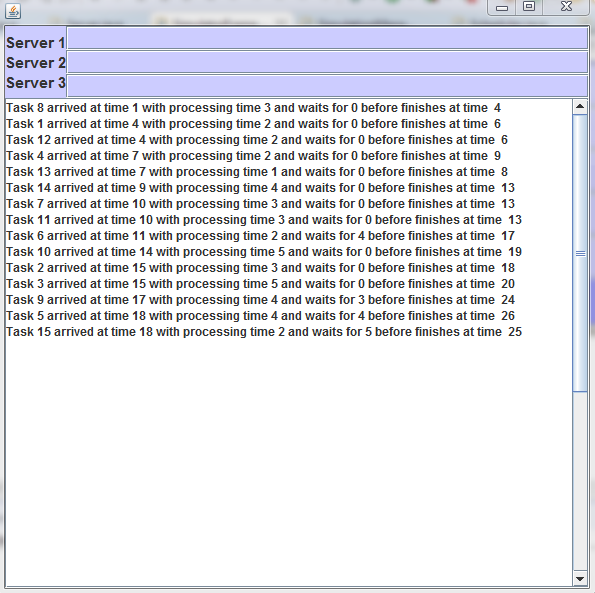
f2.setContentPane(p4);

**this**.f2.setVisible(**true**);

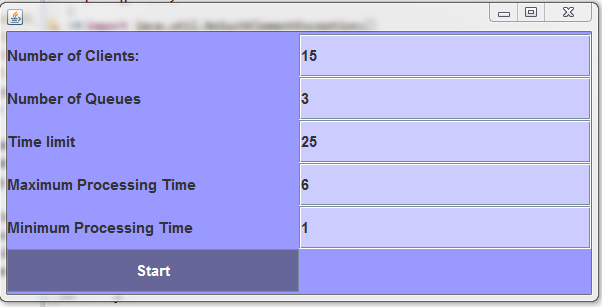
}

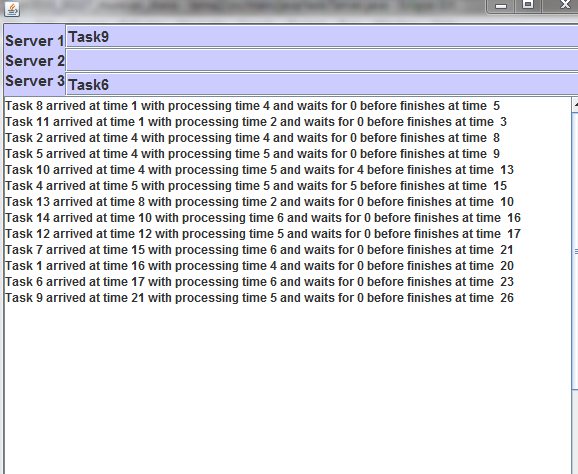
In noua fereastra se va afisa evolutia in timp real a cozilor pana cand acestea devin goale si log-ul de evenimente reprezentat de un JScrollPane in care se tine evidenta , se vor afisa timpul la care a sosit fiecare task, timpul de procesare, timpul de asteptare si timpul de finalizare.

Pentru afisarea fiecarei cozi am implementat o metoda care genereaza un numar de obiecte TextField si Jlabel egal cu numarul de cozi , iar fiecare coada se va afisa in TextField-ul corespunzator.



6.Rezultate



Pentru un numar de cozi egal cu 3 se vor afisa 3 TextField-uri in care se va vedea evoutia in timp a cozilor.Cei 15 clienti vor sosi pana la timpul 25 si vor avea timpul de procesare intre 1 si 6.Dupa scurgerea intervalului de 25 de secunde se va continua afisarea cozilor pana cand acesta vor fi goale.

7.Concluzii si Dezvoltari Ulterioare

Consider ca aceasta este o aplicatie cu care utilizatorul poate interactiona foarte usor, datorita simplitatii interfetei si consider ca modul de afisare in timp real a evolutiei cozilor se poate observa foarte usor.De asemenea, dezvoltarea acestei aplicatii m-a ajutat sa inteleg mai bine conceptul de thread cu care nu am avut ocazia sa ma intalnesc prea mult inainte,desi inca mai am de invatat.Modelul primit mi-a fost de foarte mare ajutor la realizarea acestei teme intrucat proiectul meu are aceeasi structura.

In ceea ce priveste dezvoltarea ulterioara, aceasta aplicatie de tip simulare a evolutiei cozilor poate fi extinsa prin implementarea strategiei de distribuire a task-urilor la server-ul cu cel mai mic timp de asteptare(pe care nu am apucat sa o implementez), si astfel se va permite alegerea din interfata a strategiei dorite de utilizator.Se mai pot aduce imbunatatiri pe partea de functionalitati prin afisarea unor statistici precum: media timpului de asteptare pentru fiecare server, momentul de timp la care existau cele mai multe task-uri etc.De asemenea pentru aceasta aplicatie eu am stabilit ca limita unui server este de 30 de task-uri, insa poate fi extinsa spre mai mult.

8.Bibliografie

* <http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html>
* <https://ro.wikipedia.org/wiki/Fir_de_execu%C8%9Bie>
* <http://users.utcluj.ro/~crisb_pop/pt2018.html>
* <https://docs.oracle.com/javase/tutorial/uiswing/components/scrollpane.html>
* <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html>
* <https://stackoverflow.com/questions/363681/how-do-i-generate-random-integers-within-a-specific-range-in-java>