**DOCUMENTATIE**

TEMA 2

**Barnea Matei**

**Grupa: 30225**

**Cuprins**

1.Obiectivul

2.Analiza problemei si scenarii de utilizare

3.Proiectare

4.Implementare

5.Rezultat

6.Concluzii

7.Bibliografie

1.Obiectivul

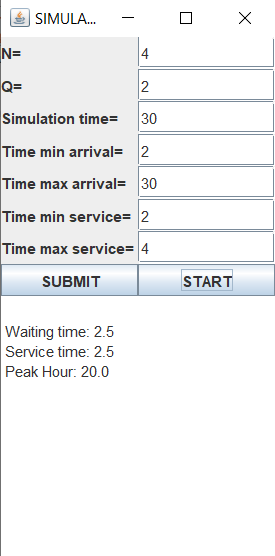
Obiectivul acestei teme este de a proiecta si de a implementa, un program care proceseaza operatiile cu cozi si cu threaduri si siguranta threadurilor. Scopul acestei teme este de a ne familiariza cu programarea paralela, deoarece pe orice calculator, fie ca are un processor cu un singur nucleu, sau cu mai multe, programele vor creea mai multe threaduri care se vor executa in paralel. In aceasta tema se simuleaza o serie de clienti care stau la mai multe case de marcat la un magazin, ei organizandu-se in cozi, fiecare in parte fiind servit cand ii vine randul, ceea ce ne duce cu gandul la faptul ca acestui client trebuie sa ii stim timplul cand soseste si timpul cat e nevoit sa stea la casa de marcat. Acesti parametrii fiind generati aleator pentru fiecare persoana in parte, exact ca si in viata reala. Am construit si o interfata grafica acestei aplicatii care ii permite utilizatorului sa introguca parametrii necesari simularii, ei fiind intervalul pentru timpii de procesare, intervalul pentru timpii de sosire dar si numarul de clienti, numarul de case de marcat, adica numarul de cozi la care se pot aseza clientii, si u in ultimul rand durata simularii. Tot in obiectivul lucrarii vreau sa precizez faptul ca in Java programarea concurenta se invarte in jurul threadurilor care implementeaza o asa zisa functie de run.

2.Analiza problemei si scenarii de utilizare

Pentru inceput trebuie sa facem comparatia intre un thread si un process. Un thread este un process mai simplu, dar in acelasi timp un thread va fi intodeauna dependent de un process, tot un thread va impartii intodeauna aceleasi resurse ceea ce va duce la probleme de comunicare; intre timp un proces contine propriul lui spatiu de memorie, iar cele mai multe implementari din masina viirtuala ruleaza pe un singur proces.

De cele mai multe ori cozile sunt utilizate pentru a putea modela domenii din lumea reala, in acest caz scopul principal al unei cozi este de a-i oferi un loc unui asa zis client care asteapta sa fie servit la un asa zis magazin. Se alege o implementare cu cozi deoarece si noi cand mergem la un magazin ne asezam la o coada pentru a putea fi serviti. Asfel fiecare coada are clienti care trebuie procesati. Fiecare client poate alege la ce coada vrea sa se aseze, modul de ascociere depinzand de problema propusa.

Proiectul meu are ca rol simularea unei cozi la care se aseaza serii de clienti, in functie de alegere, adica un client poate alege sa se aseze la coada cu cele mai putine persoane sau la coada unde timpul de asteptare este cel mai scurt. Pentru a putea simula acat mai usor am ales sa implementez si o interfata grafica care sa il ajute pe utilizator sa poata urmarii mai usor evolutia clientilor care asteapta la rand. Interfata are doua butoane, ea implicit este setata pe primul test prezentat temei, daca dorim sa modificam ceva va trebui mai intai sa dam submit si mai apoi poate incepe simularea cu datele dorite. Trebuie sa admit faptul ca atunci cand am creeat interfata nu am luat in cosiderare faptul ca utlizatorul vrea sa schimbe modul in care vrea sa aseze clientii la coada, asa ca daca chiar vrea sa faca acest lucru trebuie sa o faca efectiv din cod ceea ce nu este foarte incantator, in rest interfetei grafice nu ii mai trebuie nimic. Puteti vedea mai jos o poze cu ea din timpul simularii.

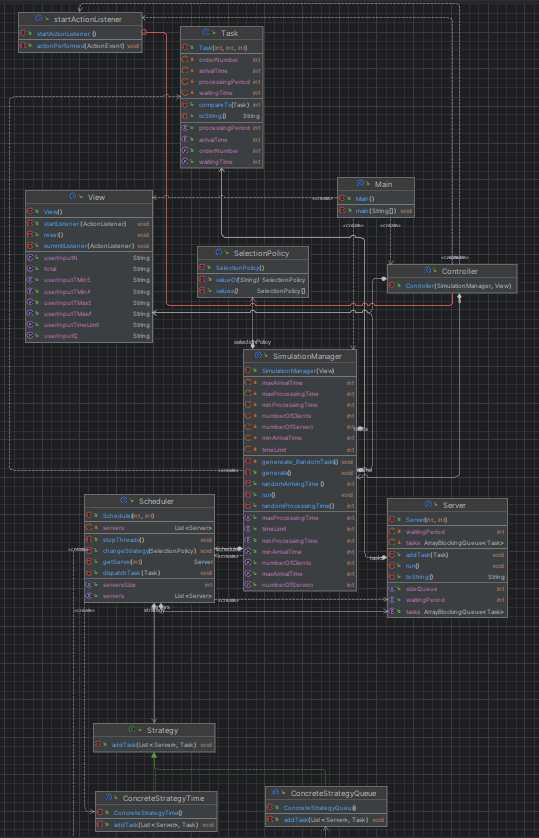


3.Proiectare

Unified Modeling Language (prescurtat UML) este un limbaj standard pentru descrierea de [modele](https://ro.wikipedia.org/w/index.php?title=Modelare_orientat%C4%83_pe_obiect&action=edit&redlink=1) și [specificații](https://ro.wikipedia.org/w/index.php?title=Specifica%C8%9Bie&action=edit&redlink=1) pentru [software](https://ro.wikipedia.org/wiki/Software). Limbajul a fost creat de către consorțiul [Object Management Group](https://ro.wikipedia.org/w/index.php?title=Object_Management_Group&action=edit&redlink=1) (OMG) care a mai produs printre altele și standardul de schimb de mesaje intre sisteme [CORBA](https://ro.wikipedia.org/w/index.php?title=CORBA&action=edit&redlink=1). UML a fost la bază dezvoltat pentru reprezentarea complexității programelor orientate pe obiect, al căror fundament este structurarea programelor pe clase, și instanțele acestora (numite și obiecte). Cu toate acestea, datorită eficienței și clarității în reprezentarea unor elemente abstracte, UML este utilizat dincolo de domeniul IT. Așa se face că există aplicații ale UML-ului pentru management de proiecte, pentru business Process Design etc.

Diagram

Description automatically generated



Pentru a gasi o solutie cat mai buna, am folosit patru pachet cu urmatoarele nume: controller, model, view, strategy in care am in total unsprezece clase si am ales sa folosesc patternul model-view-controller. In model am incapsulat date si am implementat functionalitati, in controller am implementat cate o subclasa care care implementiaza action listener, mai exact ascultatorii pentru butoare care asteapta ca noi sa apasam pe unul dintre ele. In view am implementat asa cum s-ar spune, asa zisa legatura dintre utilizator si calculator. Acest pattern sta la baza majoritara a aplicatiilor java cu interfata grafica. De remarcat ca am folosit pachetul java.SWING pentru implementarea interfetei.

La ce revine analiza problemei? Este foarte simplu, aceasta tema are un singur scop acela fiind de a ne invata sa lucram cu programarea paralela in Java, deoarece sa recunoastem ca in ziua de azi orice program ruleaza pe mai multe fire de executie si noi ca programatori trebuie sa ne adaptam mereu noilor schimbari.

\

4.Implementare

Am decis sa implementez clasele cat mai intuitiv posibil, cu un cod cat mai lizibil si cat mai usor de inteles deoarece in viitor va fi nevoie ca o ehipa intreaga sa inteleaga codul pentru a putea lucra toti la acel proiect. O sa descriu fiecare clasa pe rand:

**Clasa task** care are patru atribute de tip intreg cu nume sugestiv, un numar de ordine el fiin numarul dat unui anumit clinent in momentul in care el ajunge in coada, un dimp de sosire care reprezint momentul in care clientul ajunge la finalul cozi, un timp de asteptare care reprezinta timpul pe care o persoana trebuie sa astepte la rand, si un timp de procesare care reprezinta timpul de care este nevoie pentru a reusi sa servim clientul cu marfa dorita. Tot aici avem un constructor care initializeaza acesti timpi, dar si gettere si settere pentru acesti timpi. Am mai impementa si o suprascriere a metodei toString pentru a putea afisa cat mai citet o anumita persoana, pe care o folosesc atat in interfata graficat cat si pentru scrierea in fisier. Pentru a putea ordona clientii am implementat interfata Comparable ca sa pot sorta coada cat mai usor in aplicatia viitoare.

**Clasa server** care are patru atribute, un numar de ordine a cozilor; de el avem nevoie pentru ca fiecare coada va rula pe un thread diferit, un BlockingQueue care este un vector de taskuri, am implementat asfel ca sa am o masura de ssiguranta pe threaduri. Timpul de asteptare de este crescut de fiecare data cand adaugam un task in aceasta coada, cu timpul de procesare a clientului respentiv. Timpul de asteptare al serverului este momentul curent al serverului. Nu am fost foarte explicit, dar pentru a remeddia acest lucru putem spune ca timpul de asteptare al serverului este ca si un ceas global. Cea mai importanta metoda de aici este run() unde luam persoana de la inceputul cozii si ii tot scadem timpul de procesare, scazand in acelasi timp si timpul de asteptare. Daca persoana care se afla la casa are timpul de procesare egal cu zero inseamna ca am terminat cu ea si trebuie scoasa din coada. Am mai suprascris metoda toSting() tot pentru a putea afisa in interfata si in fisier.

**Clasa scheduler** este clasa in care am implementa modul in care clientii sunt asezti in coada, exact ca si in viata reala unde orice persoana se poate aseza la coada care doreste. Sa recunoastem ca atunci cand ne asejam la coada avem de ales dupa doua criterii, unde sunt cele mai putine persoana sau unde trebuie sa asteptam cel mai putin timp; exact asa am implementat si eu dupa aceasta strategie. Am dat nume sugestive acestora shortest\_queu si shortest\_time pe care lea am pus intr-o enumeratie. Tot aici am creat si o functie care opreste threadurile ca sa se poata oprii mainul dupa executarea simularii.

**Clasa ConcreteStrategyQueue** este clasa unde am implementat asezarea clientilor in coada, acolo unde deja exista clienti, viitorul client se va aseza la coada unde sunt cele mai putine pesoane. Am reusit sa fac acest lucru cautand minimul din fiecare coada. Compar de fiecare data minimul cu cate persoane sunt in coada respectiva iar de fiecare data cand minimulm este actualizat in instanta aux pun coada unde vreau sa introduc noul client urmand ca mai apoi dupa terminarea structurii repetitive sa introduc noul client in coada. Aceasta clasa implementeaza interfata Strategy unde exista o singura metoda neimplementata, aceea fiind addTask care are ca parametrii toate cozile si noul client pe care vrem sa il adaugam.

**Clasa ConcreteStrategyTime** este clasa unde am implementat asezarea clientilor in coada, acolo unde deja exista clienti, viitorul client se va aseza la coada unde trebuie sa astepte cel mai putin timp. Am reusit sa fac acest lucru cautand minimul timpilor de asteptare din fiecare coada. Compar de fiecare data minimul cu minimul timpului de asteptare la coada respectiva iar de fiecare data cand minimulm este actualizat in instanta aux pun coada unde vreau sa introduc noul client urmand ca mai apoi dupa terminarea structurii repetitive sa introduc noul client in coada. Aceasta clasa implementeaza interfata Strategy, de altfel ca si clasa de mai sus, unde exista o singura metoda neimplementata, aceea fiind addTask care are ca parametrii toate cozile si noul client pe care vrem sa il adaugam.

**Clasa view** reprezinta implementarea interfetei grafice facute cu ajutorul pachetului java.swing, ea fiind foarte usor de implementat, de aceea am si ales-o dar si pentru ca deja o stiam folosi de semestrul trecut de la cursul de poo.

In aceasta clasa am am sapte text fielduri unde utilizatorulpoare introduce parametrii simulari, ei fiin numarul de clienti, numarul de cozi la care clientii se pot aseza, un timp general al simularii, adica timpul in secunde care reprezinta pentru cat timp se executa simularea. Tot aici mai sunt doua capete de interval pentru timpul de sosire si tot doua capete de interval pentru timpul de procesare. Din aceste intervale voi genera random cate un timp se sosire si cate un timp de procesare pentru fecare viitor client care se va aseza in coada. Tot in aceasta clasa am un JtextPane unde o sa afisez in timp real evolutia cozii, am ales JtextPane deoarece in el se afiseaza text mai frumos si se redimensioneaza in timp ce marim interfasa, daca se doreste acest lucru. . In continuare voi discuta despre cele doua butoane implementate care si ele la randul lor au nume sugestiv fiecare buton facand o operatie de submit care va transmite parametrii simularii si butonul de start care va porni simularea.

Tot in aceasta clasa am adaugat ascultatori pentru fiecare buton, dar implementarea propriu-zisa fiind in clasa controller. In aceasta clasa mai exista mai metode cu care luam stringurile introdus in text fielduri.

**Clasa controller** are mai multe subclase, cate o subclasa pentru fiecare buton, mai exact cate o clasa pentru pecare ascultator al fiecarui buton. Am ca si atribute un model si un view deoarece controllerul trebuie sa faca legatura intre model si view.

Pentru ascultatorul butonului de submit, cu ajutorul metodelor din view am loat fiecare string introdus in fielduri, le-am convertit in intregi si iam transmis clasei de simulare. Pentru celalalt asculator am avut grija sa generez pe prima linie simularea cu parametrii introdusi de utilizator; daca nu as fi facurt acest lucru simularea nu ar avea nici o relevanta; mai apui am resetat viewul dupa care am pornit simularea.

**Clasa SimulationManager** reprezinta propriu zis clasa de simulare. Aici am toti parametrii introdusi de utilizator, implicit ei initializandu-se cu valorile primului test propus in cerinta temei. Tot in aceasta clasa se calculeaza concluziile simularii acestea fiind un timp mediu de asteptare, un timp mediu e procesare si momentul in care in cozi exista cei mai multi clienti. Am avut grija sa scriu in fisier de fiecare data cand se executa o simulare ca sa se poata verifica ulterior comportamentul ei. Tot aici am implementat metodele de generare random a timpilor de procesare si de sosire cu ajutorul metodei din Math, random(). Aici mai exista si metoda de generare a taskurilor si de introducere a lor in cozi, avand grija ca la final sa sortez cozile cu ajutorul metodei compareTo() din clasa task.

Pentru metoda run() in functie de timpul curent la care sae afla simularea se proceseaza persoanele si se scriu in fisier datele. Pentru inceput am deschis fisierul iar mai apui am afisat persoanele care nu sea afla in nici o coada urmand aca mai apui sa afisez fiecare coada cu persoanele din componenta ei; urmand ca la sfarsit sa afisez concluziile simularii iar mai apui sa opresc threadurile si sa inchid fisierul deschis. Pot sa mentionez ca aici am metoda care afiseaza cozile in timp real. Puteti vedea mai jos antetul clase:

**Clasa Main** am instantiat un view care l-am transmis ca si parametru simularii si am instantiat si un controller dar si clasa de simulare. Am facut acest lucru pentru a respecta patternul model view controller, un pattern destul de utilizat in java deoarece face codul mult mai lizibil si mult mai usor de iterpretat in caz ca cineva vrea sa aduca modificari ulterioare; urmand ca la sfarsit sa fac vizibil viewul.

5.Rezultat

Pentru aceasta tema am avut de implementat trei teste diferite pentru a putea vedea diferite situatii de simulare si pentru a putea specula viitoarele rezultate. Asa ca trebuie sa mentionez ca au fost doua teste maai scurte care au mers destul de repede dar ultimul test a fost mai lung si a durat mai mult, rezultatele acestor teste fiin in mare parte o statistica. O sa pun mai jos datele de intrare si rezultatele acelor test.

6.Concluzii

Pot spune ca a fost prima data in viata mea cand am lucrat cu threaduri in java si chiar a fost destul de interesant. Am intelezi ca aceste cozi modeleaza un model din viata reala existent in mai multe domenii. Principalul obiectiv al cozilor a fost acela de a creea un loc pentru un asa zis client si de a astepta ca acel client sa fie servit, fiecare dintre ei fiind nevoiti sa astepte pana cel de dinaintea lui isi termina treaba pentru a putea fi si el la randul lui proccesat. Ca o concluzie am vazut ca daca ezista mult mai multe cozi cu atat timpul de asteptare si timpul de servire este mai scurt, exact ca si in viata reala. Acest program ar putea fi folosit pentru a specula un numar minim de case de marcat necesare unui magazin daca stim ca un anumit numari de clienti au avut tangente cu acel magazin.

7.Bibliografie

<https://ro.wikipedia.org/wiki/Model-view-controller>

<https://docs.oracle.com/javase/7/docs/api/javax/swing/JTextPane.html>

[Lesson: Concurrency (The Java™ Tutorials > Essential Classes) (oracle.com)](https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html)

<https://docs.oracle.com/javase/tutorial/essential/concurrency/sleep.html>

<https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html>