

Documentație laborator tema 2

Simulator de cozi

Tehnici fundamentale de programare

Student: Dunca Denisa Mihaela

Grupa: 30227

1. **Obiectivul temei**

Cerința temei este de a crea și implementa o aplicație pentru simularea unui sistem de cozi și distribuirea unui set de clienți generat aleatoriu către aceste cozi având ca și scop principal minimizarea timpului de așteptare.

Tema de laborator “Simulator de cozi” are ca și obiectiv principal obținerea unui mod prin care clienții nu mai sunt nevoiți să aștepte mult la coadă, iar aplicația respectă paradigmele de programare orientată pe obiect folosind limbajul de programare Java.

Obiectivele secundare ale proiectului:

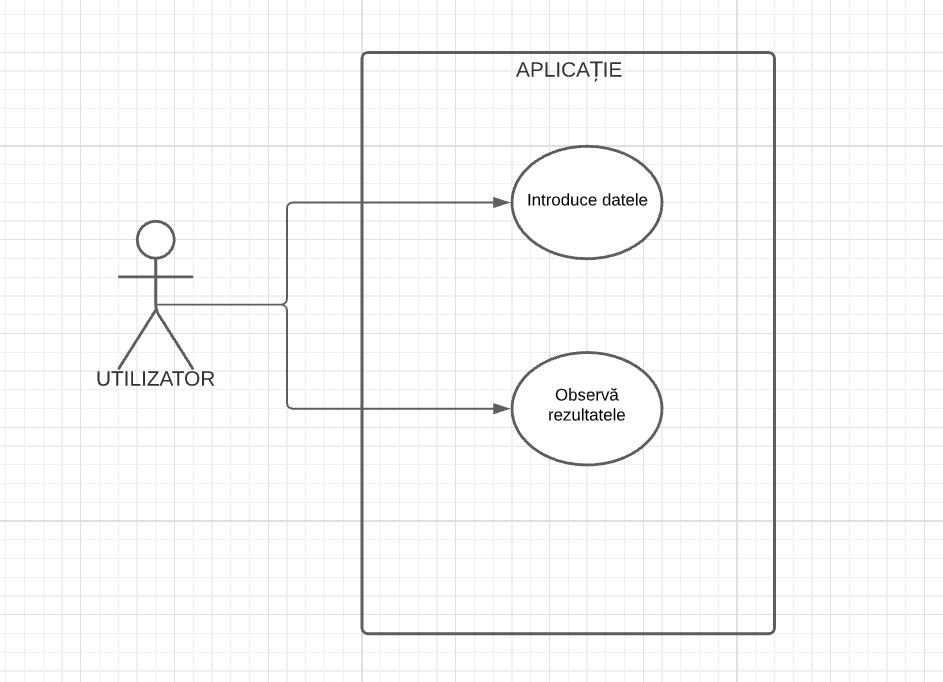
* Analizarea problemei prin înțelegerea corectă a cerinței și modelarea de scenarii și cazuri de utilizare.
* Proiectarea unei soluții care să se plieze pe toate cazurile pe care le introduce utilizatorul.
* Implementarea soluției alese prin scrierea codului implicit și a interfeței utilizator.
* Observarea rezultatelor finale și concluziile.

1. **Analiza problemei**

Analiza problemei a constat prin înțelegerea pe deplin a cerinței și a informațiilor pe care dorim să le primim la generarea unui rezultat final. S-a creat o schemă inițială care trebuie să prezinte în mare ceea ce se urmărește prin acest proiect. Pe de o parte, este necesară cunoaștereamodului prin care functioneaza o coada, faptul că merge pe principuil prin sosit, primul servit, este necesară si și cunoașterea tuturor elementelor pe care utilizatorul aplicației trebuie sa le introduca de la tastatura și anume: numarul de clienți, numărul de cozi, timpul simulării, timpul minim si timul maxim de sosire la coadă și de așteptare.

Conceptul de coadă este adesea întâlnit in viata noastra de zi cu zi fiind structure ce content date de aceasi tip si care functioneaza dupa principal First In First Out. In cazul aplicatiei de față cozile sunt locurile unde sunt repartizați cleinți de exemplu, ca și clienții care stau la casa la un magazine pentru a le fi scanate produsele. Scopul nostrum principal este sa ii facem sa astepte cat mai putin la coadă penrtu a minimiza din timpul risipit asteptand. Ca si date de intrare sunt date introduce de catre utilizatorul care doareste sa creeze o simmulare cu client random pentru a vedea cum functionează programul.

Un mod prin care se poate scoate în evidență această structură este generarea unei Diagrame USE CASE.



În diagrama de mai sus este prezentată structura logică a proiectului. Sistemul este aplicația în sine care distribuie clienți în cozi în funcție de datele introduse , user-ul reprezintă actorul, cel care introduce datele, cel care dorește un răspuns. Introducerea datelor este dependentă de actor, precum și apăsarea pe butonul de start și ieșirea din program. După ce au fost introduse datele și aplicația este pornită, se vor genera pe ecran distribuirea in timp a clienților pe cozi.

Un scenariu de utilizare ar fi:

* User-ul introduce datele de la tastatură

Ex: Number of clients: 5

Number of queues: 2

Simulation interval: 30

Minimum arrival time: 4

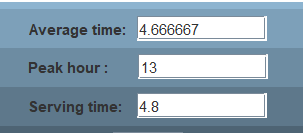
Maximum arrival time: 15

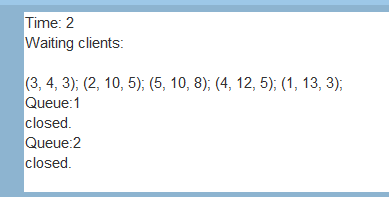
Minimum service time: 3

Maximum service time: 15

* Apasă pe butonul de start
* Pe ecran se genereaza rezultatul in timp si la final average time, peak hour și serving time.

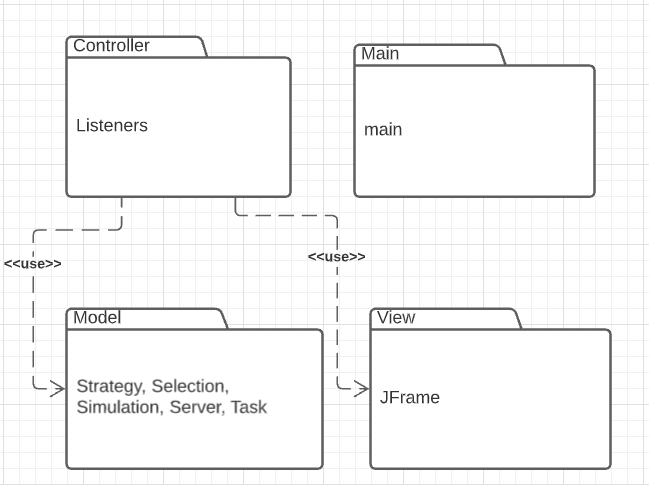
Ex:



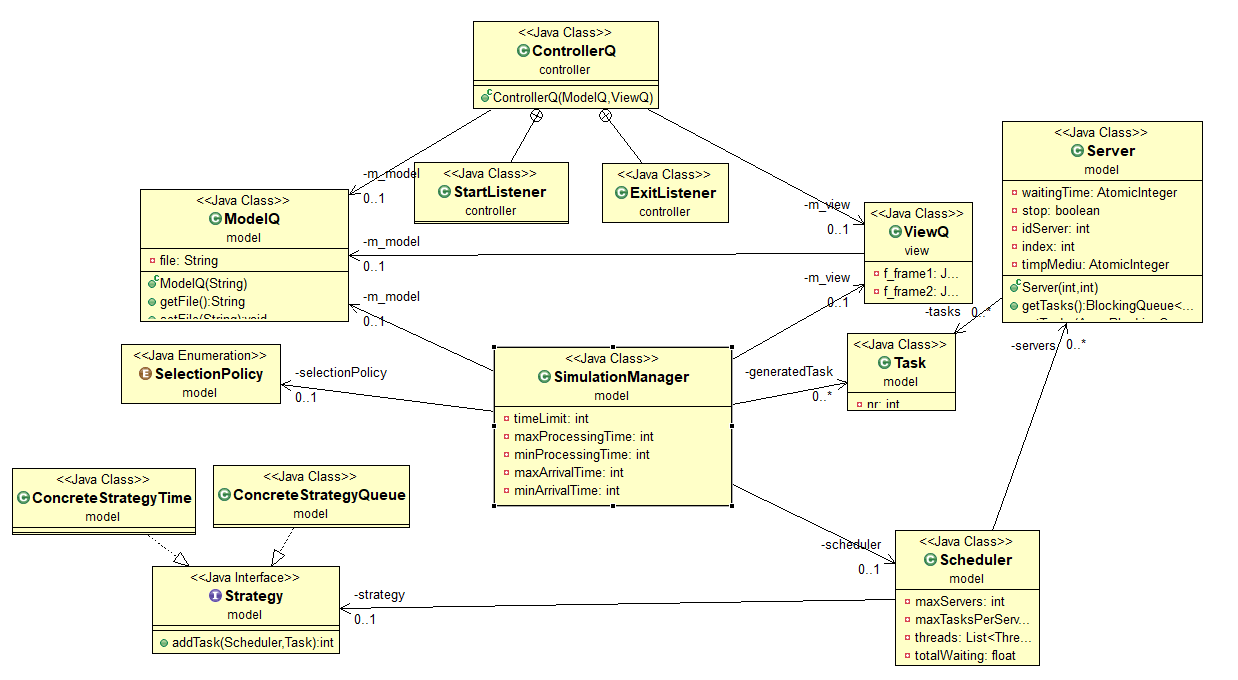


* User-ul iese din aplicație

1. **Proiectare**

Trecând la partea practică de rezolvare a problemei, și anume proiectarea în limbaj OOP, au fost urmăriți mai mulți pași, primul fiind crearea unei Diagrame de pachete urmărind modelul MVC (Model-View-Controller).

Controllerul face legătura dintre clasele din Model și View prin intermediul butoanelor apăsate. În model se află clasele pentru Strategy, Selection, Simulation, Server, Task și tot ce înseamnă funcționalitatea programului, iar în View partea ce va fi disponibilă utilizatorului spre folosire și afișarea rezultatului final.

Intrând într-o analiză mai amănunțită a proiectului, am generat și diagrama UML. Aici se pot vedea în detaliu legăturile dintre clase, metodele folosite, precum și structura logică a întregului proiect.

Este respectată în principal structura MVC a proiectului. În View am declarat toate obiectele de care am nevoie folosind clasa JFrame, în Controller am creat un obiect de tipul Model, respectiv View și am creat listener-e pentru cele două butoane. Modelul este alcătuit din mai multe clase, clasa Task, destinată clienților unde am definit un constructor ce are un ID, un timp în care acesta ajunge și un timp de procesare (cat timp stă la coadă), tot aici am și clasa de toString pentru a converti datele despre clienți în toString.

Clasa Server are ca structură de date folosită, ArrayBlockingQueue de clienți, tot aici am pus și toate metodele ce lucrează cu cozi, precum addTask, queueWaitingTime, run , toString etc. In clasa Scheduler calculez averageWaitingTime, dispatchTask, iar in clasa SimulationManager generez task uri random, calculez serving time si metoda run. De asemenea, in model sunt apar si clasele legate de strategia pe care o adoptăm în proiect care poate fi in functie de timp sau de servere, eu m-am folosit în mare de strategia de timp, dar am adăugat si celelalte clase si interfata deoarece apareau in prezentarea facuta pentru poriect.

**Implementare**

**Pachetul model**

**Clasa Task**

Clasa task reprezintă clasa client, acesta primește ca si atribute principale un ID, numit nr in proiect, un arrivalTime, timpul cand ajunge la coadă și un porcessingTime timpul cat sta la coadă. Aceste atribute le-am pus într-un constructor. Pe langa gettere si settere am creat si metoda toString și compareTo.

**Clasa Server**

Clasa server reprezintă clasa destinată cozilor acre implementează interfața Runnable, pentru ca pentru fiecare server va exista un thread separat, serverle procesând paralel clienții si primeste în primul rand un ArrayBlockingQueue de taskuri (clienti) colecție concurentă și thread-safe, un waitingTime de tipul AtomicInteger un ID si un index. Pe langa gettere, settere si constructorul principal, am implementat si metoda addTask care imi adaugă un task in coada, metodata queueWaitingTime care o folosesc pentru calcularea timpului mediu. Deoarece nu tot timpul sunt clienți în cozi, serverele verifică de fiecare data dacă au, iar dacă nu au se suspendă un timp procesul. Dacă în acel timp se adaugă clienți, atunci începe procesarea( cat suntem pe true in while). Procesarea suspendă timpul de procesare al clintului si apoi îl scoate din coadă.

**Clasa Scheduler**

Clasa scheduler se ocupă de distribuția clientilor la cozi si comunică cu toate serverele. Clasa se ocupă de crearea si oprirea cozilor, precum si modificarea timpului de procesare. Tot aici am si metoda de calcul al timpului mediu si o metoda toString. Practic, această clasă programează fiecare client sa intre la timpii potriviți în cozile care sunt disponibile.

**Clasa SimulationManager**

Se poate spune ca această clasă este inima întregului proces si implementeaza interfata Runnable. Aici am facut metoda care imi genereaza aleator clienți, apoi ii ordonează după arrivalTime pentru a putea fi pus corect in cozi. Aici am o metodă pe care o folosesc pentru calcularea servingTime-ului si metoda run. Fiecare client este trimis spre o coadă in functie de cum alege schedulerul dar doar atunci cand arrival time-ul este egal cu timpul simularii. În cazul in care sunt mai multi clienți cu același arrival time, acestia sunt distribuiți in cozi diferite.

Tot aici am calculat si peak hour si am facut o functie de afisare pentru a putea sa pun toate datele returnate în textarea-ul di ViewQ. Pe lângă afișsarea distribuției clientilor in cozi și timpul simulării am afișat pe ecran și clineții care așteaptă să fie distirbuiți în cozi și datele calculate oentru average time, peak hour si serving time.

**Clasa ModelQ**

În această clasă am pus două metode care am considerat că nu au se potrivesc neapărat cu niciuna din clasele deja făcute. Aici apar metoda pentru calculul peak hour-ului și metoda care imi afisază in fisier folosindu-mă de BufferWriter.

**Pachetul view**

In clasa ViewQ am implementat, folosind JFrame, fațada aplicației. Mai întâi am declarat toate obiectele de care am avut nevoie, apoi am facut constructorul clasei unde, folosindu-mă de multe panel-uri am adăugat toate obiectele pe interfața aplicației. Pe lângă gettere si settere am facut si listenere pentru cele două butoane implementate respectiv pentru Strat si Exit. Pentru a afișa rezultatul pe ecran am folosit JTextField-uri si un JTextArea.

****Spre deosebire de afisarea in fereastra de comandă sau de afișarea din fișiere, afișarea de pe interfață, pe lângă timp, si cozi, apare in plus si linia cu clientii care asteaptă să fie repartzați in cozi, fiecare la timpul cerut.

Asa arată interfața cu utilizatorul:

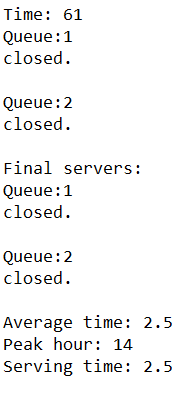
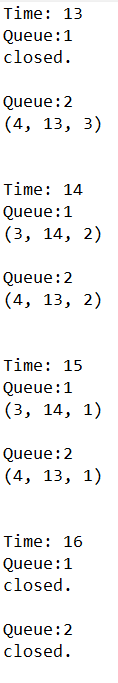
**Pachetul controller**

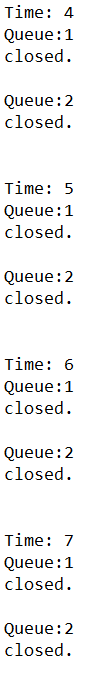
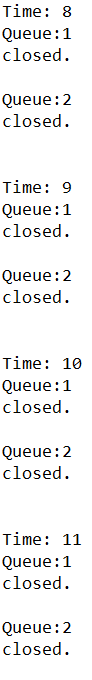
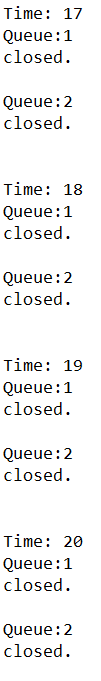
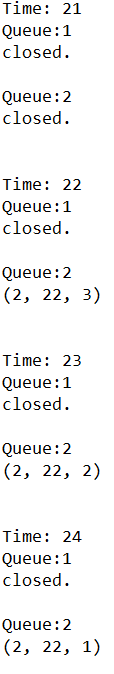
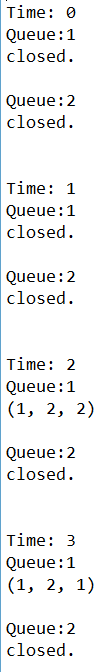
In Controller fac legătura dintre Model si View, aici am un constructor care primește un obiect model si un obiect view si care are doua button listerner , unul pentru start care primește un simulationManager cu toti parametrii de care este nevoie, imi creează threaduri pe baza simularii si imi porneste simularea. Am adugat si un PrintStream pentru a putea să pun rezultatele care ar trebui să fie afișate in consolă, intr-un fisier numit ”output.txt”.

Al doilea buton e butonul de exit care iese din program si inchide programul.

1. **Rezultate**

După rezolvarea problemelor din cod și ajungerea la proiectul final am testat proiectul pe cele 3 teste date în cerința proiectului. Toate testele au dat rezultate bune și pe fiecare in parte le am pus in cate un fișier output1.txt, output2.txt și output3.txt. Toate fișierele conțin outputul ce apare in textarea de pe interfață plus timpul mediu de procesare, ora de vârf si timpul de servire mediu. Exemplu dat din fișierul 1:

****

****

Din rezultatele generate se poate observa că fiecare client intră în coadă la timpul la care ajunge (arrival time) si stă atatia timpi cât este processing time. Spre exemplu primul client care intra, cel care id-ul 1, ajunge la timpul 2, astfel este pus in coada 1 la time = 2. Dupa aceea, din cauza ca trebuie sa fie servit timp de 2 timpi, la time = 3 se decrementează timpul de servire cu unu , iar la time = 4 clientul pleacă, iar coada este din nou goală, așteptând după următorii clienți. Average time reprezintă timpiul mediu de așteptare a clienților înainte de a fi serviți, peak hour reprezintă primul timp in care au fost cei mai mulți clienți in cozi, în cazul nostru timpul 14 este primul timp unde apar doi clienți per total în cozi, iar serving time este timpul mediu de servire, în cazul de fața s-a întâmplat ca acesta să coincidă cu timpul de la average time.

1. **Concluzii**

Per total, întregul proiect este destinat utilizării, și datorită testărilor și verificărilor făcut aș putea spune că ar trebui să funcționeze pentru orice caz pe care utilizatorul îl introduce. O dezvoltare ulterioară pe care aș putea să o adaug este modul în care este afișat rezultatul, pentru că în cazul în care sunt introduși foarte mulți clienți cum ar fi 100 deja interfața începe să nu mai încapă în ecran și astfel nu se pot observa corect rezultatele si mai ales nu mai poate fi accesat butonul de exit sau rezultatele de la timpul de așteptare, ora de vârf și timpul mediu de servire. Pe langa acestea, consider ca algoritmul de adăugare poate fi imbunătațit astfel incat in cazul in care avem un client care trebuie sa stea la coadă 20 de timpi iar dupa el vine un client care vrea sa stea doar 2 timpi, algoritmul sa intrerupa prelucraerea facută pentru primul client si sa il ia pe urmatorul pentru ca acesta sa nu astepte 22 de timp dupa primul client, apoi sa revina la clientul initial si tot asa. Un avantaj ar fi ca timpul mediu de asteptare ar fi mult mai mic. In plus s-ar putea aduga un mod prin care clientii care au ramas la coada chiar daca timpu de simulare este gata, acestia sa fie redistribuiti la prima pornire a simularii sau trimisi in alta simulare secundara penrtu a nu ramane ne serviti, sau se poate calcula inainte timpu de care este nevoie sa proceseze clientii ce urmeaza si sa il verifice daca se incadreaza in timpul ramas din simulare, astfel daca nu se incadreaza sa nu ii mai bage in coada.

Ce am reușit eu să deprind din acest proiect este modul prin care se utilizeaza threadurile prin implementarea interfeței Runnable si utilizarea metodei specifice Run. Am căutat cum aș putea sa pun rezultatele ce apar în fereastra de afisare intr-un file folosint PrintStream și am descoperit cum pot sa folosesc un textarea si cum sa pun informațiile generate pe parcurs în el folosind elemente din clasele Writer si BufferWriter.

Cu ajutorul prezentării ce însoțește cerințele proiectului, am reușit să aflu toate modalitățile de implementare a threadului astfel incat acestea să fie thread-safe folosind BlockingQueue si AtomicInteger.

1. **Bibliografie**
2. LucidChart pentru USE CASE Diagram

<https://lucid.app/lucidchart/>

<https://youtu.be/zid-MVo7M-E>

1. PrintStream

<https://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html>

1. UML Diagram

<https://www.jetbrains.com/help/idea/class-diagram.html>

1. BufferWriter

<https://docs.oracle.com/javase/7/docs/api/java/io/BufferedWriter.html>

1. ThreadSafe

<https://gowthamy.medium.com/concurrent-programming-fundamentals-thread-safety-6b44c026bd2a>