DOCUMENTATIE

TEMA 2

Aplicatie pentru simularea cozilor de asteptare folosind thread-uri si mecanisme de sincronizare

NUME STUDENT: Sand Elena - Andreea

GRUPA: 30223

# CUPRINS

[1. Obiectivul temei 3](file:///C:\Users\Utilizator\Desktop\PT2021-2022_Documentation_Template.doc#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](file:///C:\Users\Utilizator\Desktop\PT2021-2022_Documentation_Template.doc#_Toc95297886)

[3. Proiectare 3](file:///C:\Users\Utilizator\Desktop\PT2021-2022_Documentation_Template.doc#_Toc95297887)

[4. Implementare 3](file:///C:\Users\Utilizator\Desktop\PT2021-2022_Documentation_Template.doc#_Toc95297888)

[5. Rezultate 3](file:///C:\Users\Utilizator\Desktop\PT2021-2022_Documentation_Template.doc#_Toc95297889)

[6. Concluzii 3](file:///C:\Users\Utilizator\Desktop\PT2021-2022_Documentation_Template.doc#_Toc95297890)

[7. Bibliografie 3](file:///C:\Users\Utilizator\Desktop\PT2021-2022_Documentation_Template.doc#_Toc95297891)

# 1.Obiectivul temei

Obiectivul acestei teme este de a implementa o aplicatie care simuleaza in timp real distributia a N clienti in Q cozi de asteptare astfel incat sa se obtina un timp de asteptare cat mai mic pentru fiecare client.

Pentru atingerea obiectivului principal, este necesara indeplinirea mai multor obiective secundare: aplicatia de gestionare a cozilor ar trebui sa simuleze ( prin definirea unui timp de simulare Tsimulation) o serie de N clienti care sosesc pentru servire, sunt distribuiti in una din cele Q cozi, asteapta, sunt serviti si parasesc coada.

# 2.Analiza problemei, modelare, scenarii, cazuri de utilizare

**2.1 Analiza problemei**

Principalul motiv pentru care aceasta aplicatie trebuie sa fie implementata in Java este ca pentru ca trebuie respectate cele patru strategii de baza ale programarii orientate pe obiect : incapsularea , abstractia , mostenirea si polimorfismul.

Cozile sunt utilizate in mod obisnuit pentru a modela demeniile lumii reale. Obiectivul principal al unei cozi este de a oferi un loc pentru ca un “client” sa astepe inainte de a primi un “serviciu”. Management-ul sisetemelor bazate pe cozi este interest de minimizarea timpului in care “clentii” asteapta sa fie serviti.

**Cerintele functionale ale acestui proiect sunt:**

* Simulatorul trebuie sa prelucreze în mod corect datele introduse de utilizator in interfara grafica.
* Simulatorul ar trebui sa genereze intr-un mod random datele celor N clienti care urmeaza sa fie introdusi în cozi.
* Simulatorul ar trebui să facă posibila vizualizarea unei reprezentari a cozilor actualizata în fiecare moment care sa corespunda evolutiei fiecarei cozi, într-o forma cât mai clara și mai ordonată.
* Simulatorul ar trebui sa faca afisarea intr un fisier .txt a informatiilor legate de evolutia cozilor in fiecare moment (timpul, clientii care au parasit coada, clientii care au fost adaugati intr-o coada, precum si datele despre acesti clienti – ID, timp de procesare a comenzii, si momentul in care a sosit -, si timpul de asteptare pentru fiecare coada dupa orice eveniment ).
* Simulatorul ar trebui să fie ușor de utilizat de către orice fel de utilizatori.

**Cerintele non-functionale ale acsetui proiect sunt:**

* Realizarea unei interfete grafice prietenoasa ( cat mai usor de utilizat ) – Interfata are rolul de a realiza interacțiunea dintre utilizator si program, prin prelucrarea datelor introduse de catre utilizator, dar și de a prezenta evolutia cozilor în fiecare moment de timp din intervalul ales.
* Calcularea si afisarea unor statistici rezultate in urma simularii (timp mediu de asteptare si timpul mediu de procesare a comenzilor).
* Generarea datelor esentiale despre fiecare client , ID-ul fiind un numar din intervalul [0, numar de clienti], atribuit in ordinea crescatoare a generarii, timpul de procesare si timpul de sosire sunt doua numere alese in mod aleator din intervalele alese de catre utilizator inaintea simularii.
* Distribuirea clientilor in una din cele Q cozi , utilizand una din cele doua strategii de selectare a cozii potrivite : timpul minim de asteptare sau lungimea ca mai mica a cozii.
* Actualizarea dupa fiecare eveniment a timpului de asteptare corespunzator cozii in care a avut loc adaugand sau scazand timpul de procesare a comenzi clientului care a fost adaugat in coada, respectiv a celui care a parasit coada.

**2.2 Modelare:**

Pentru realizarea acestui proiect este necesara cunoasterea conceptelor OOP si stapanirea unor cunostiinte despre thread-uri si sincronizare. Management-ul sistemelor bazate pe cozi incearca sa minimizeze timpul de asteptare pentru ca o sarcina sa fie procesata. O modalitate de a minimiza aceasta perioada este de a adauga mai multe cozi (fiecare coada avand un processor asociat), dar aceasta abordare creste costul producatorului).

**2.3 Scenarii, cazuri de utilizare**

Simularea modului de desfasurare a evenimentelor

**Actor principal**: utilizatorul

**Scenariul principal de succes:**  
1. Utilizatorul introduce numarul de clienti care trebuie ditribuit in cozile de asteptare.  
2. Utilizatorul alege care este numarul de cozi in care sa fie distribuiti clientii.

3. Utilizatorul selecteaza interalul de timp in care este ales timpul de sosire al clientilor, introducand valorile pentru timpul minim pentru venire a clientilor si timpul maxim pentru sosirea lor in text field-urile corespunzatoare.

4. Utilizatorul introduce in interfata grafica o valoare pentru timpul minim de procesare a unei comenzi.

5. Utilizatorul alege care este valoarea timpului maxim de procesarea a comenzilor si il introduce in locul corespunzator.

6. Utilizatorul introduce valoarea momentului de timp pana la care are loc simularea in interfata grafica.

9. Simulatorul genereaza numarul selectat de clienti, prin atribuirea unor valori unice pentru ID si a unor valori aleatoare pentru timpul de sosire al clientului, dar si pentru cel de procesare a comenzii acestuia, generate din intervalul ales.

10. Simulatorul afiseaza in interfata grafica o rerezentare a cozilor de asteptare actualizata in fiecare moment de timp.

11. Simulatorul scrie in fisierul a.txt o descriere cat mai scurta si organizat, dar si clara a evenimentelor corespunzatoare evolutiei fiecarei cozi.

**Secvență alternativă**: date de intrare introduse gresit  
1. Utilizatorul introduce texte care nu pot sa fie convertite in formatul aferent (din string in integer):

- Sirurile de caracere contin si alte caractere care nu sunt cifre;

- Sirurile de caracere contin numere negative;

- Datele de intrare lipsesc;

2. Este aruncata o exceptie de tipul IOException, simularea este intrerupta, iar pe interfata grafica nu este afisata nicio vizualizare a cozilor.  
3. Scenariul revine la etapa de introducere a datelor pana cand datele de intrare sunt valide.

# 3.Proiectare

Aplicatia contine 9 clase : Task, Server, Scheduler, SimulationManager , SimulationFrame , SimulationFrame2 , ConcreteStrategyQueue , ConcreteStrategyTime , Main.

Clasele Task si Server sunt folosite in principal pentru pastrarea datelor intr-o forma potrivita si pentru prelucrarea acsetora, ele fiind usor de accesat. strategiile sunt descrise in clasele ConcreteStrategyQueue si ConcreteStrategyTime, prima interfata ( cea in care utilizatorul introduce datele dorite ) este implementata in clasa SimulationFrame, iar a doua interfata ( cea care afiseaza in timp real evolutia cozilor ) este implementata in SimulationFrame2. Main-ul aplicatiei se afla in clasa Main.

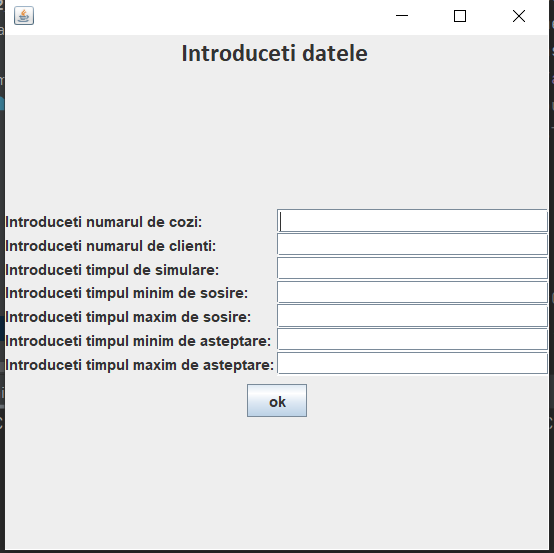
Pentru crearea interfetei grafice am folosit arhitectura Swing care isi are radacinile in arhitectura model – view – controller (MVC). Aceasta arhitectura cere ca aplicatia vizuala sa fie divizata in trei parti separate:

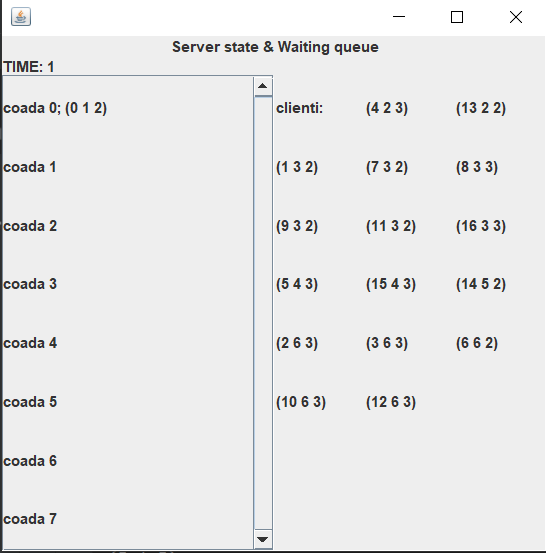
* Un model – reprezinta intern datele aplicatiei :Task, Server, Scheduler
* O vizualizare (view) – reprezentarea vizuala a datelor respective : SimulationFrame, SimulationFrame2
* Un controlor (controller) – care preia intrarea de la utilizator si o transpune in schimbari in model : SimulationManager

In acest proiect, crearea interfetei grafice se face in clasele SimulationFrame si SimulationFrame2, iar interactiunea dintre utilizator si interfata este descrisa in clasa SimulationManager.

Interfata contine sapte campuri pentru text (JtextField) care permit introducerea datelor necesare simulari. Interfata mai contine 1 buton “ok” care trebuie apasat de utilizator dupa introducerea datelor si care porneste simulare in timp real

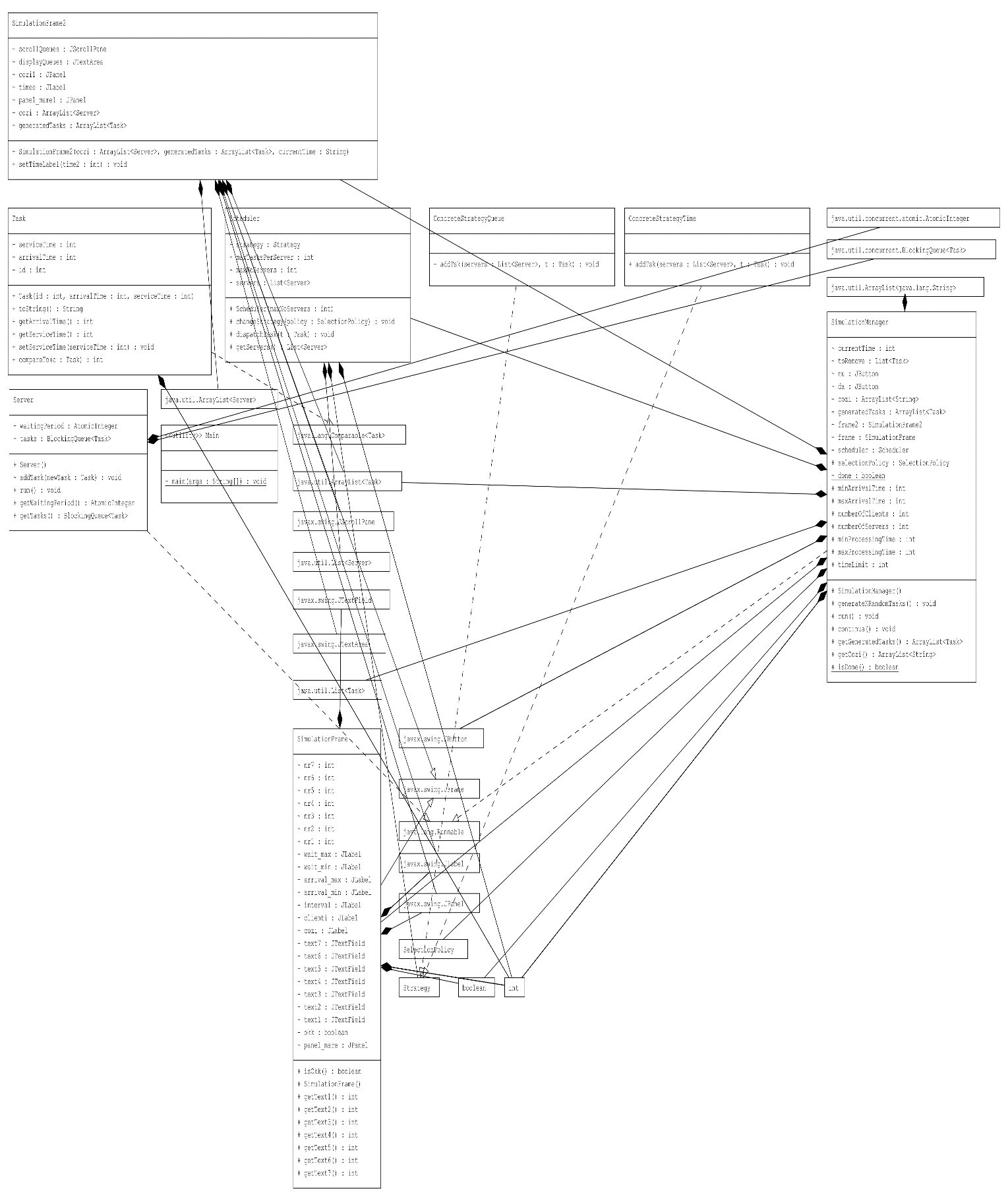
Interfata 1: Interfata 2:





.

Diagrama de clase:



# 4.Implementare

**Clasa Task** contine 3 campuri : id-ul de tip private int, arrivalTime de tip private int si serviceTime de tip private int. Este implementat cun constructor public Task(int id, int arrivalTime, int serviceTime) care initializeza cele trei campuri. Metoda toString implementat in aceasta clasa este folosita pentru afisarea datelor in interfata, in fisier sau in consola (pentru verificare). De asemenea, este implementata metoda public int compareTo(Task c) folosita pentru ordonarea crescatoarea a clientilor in functie de arrivalTime pentru a putea fi distribuiti in aceasta ordine intr-o coada. In aceasta clasa mai sunt implementate si metodele de setter si getter pentru atributele de care avem nevoie sau care trebuie modificate ( serviceTime).

**Clasa Server** care implementeaza interfata Runnable contine doua campuri private : BlockingQueue<Task> tasks in care sunt memorati clientii introdusi in coada prin metoda addTask( Task newTask), si AtomicInteger waitingPeriod in care este memorat timpul de asteptare pentru fiecare coada, acesta fiind modificat tot in metoda addTask, adunandu-se cu serviceTime-ul specific clientului introdus. Metoda run() implementata in acesta clasa parcurge lista de client si pentru fiecare client simuleaza timpul de asteptare la o coada, decrementand serviceTime-ul la fiecare perioada de timp din simulare.

**Clasa Scheduler** contine trei campuri : List < Server > servers in care este memorata lista de cozi, int maxNoServers care contine numarul maxim de cozi si Strategy strategy care memoreaza numele strategiei dorite. In constructor, se creaza maxNoServers cozi si sunt adaugate in servers, iar pentru fiecare coada se creeaza un thread. In metoda changeStrategy ( SelectionPolicy policy ) este aleasa strategia dorita: Shortest\_Queue sau Shortest\_Time. Metoda dispatchTask ( Task t ) apeleaza functia din clasa Strategy care distribuie clientul din parametru la o coada (depinzand de strategia aleasa)

**Clasa ConcreteStrategyTime** contine metoda addTsk ( List < Server > servers, Task t ) care parcurge lista de servere ( cozi ) cu ajutorul unui for si memoreaza in variabila Server q\_optim coada cu timpul de asteptare ( waitingPeriod ) minim, apoi adauga clientul t la acea coada.

**Clasa ConcreteStrategyQueue** contine metoda addTsk ( List < Server > servers, Task t ) care parcurge lista de servere ( cozi ) cu ajutorul unui for si memoreaza in variabila Server q\_optim coada cu nuamarul minim de clientii si adauga client t la acea coada.

**Clasa SimulationFrame** contine elementele din interfata grafica. Aceasta contine 4 JPanel-uri: panel\_mare, panel\_1, penel\_2 si panel\_3. Panel\_mare cotine celelalte 3 panel\_uri. Panel\_1 contine titlul “Introduceti datele”. Panel\_2 contine 7 JLabel-uri si 7 TextField-uri necesare pentru introducere datelor. Panel\_3 contine butonul “ok”. De asemenea, in aceasta clasa este implementata metoda addActionListener pentru butonul “ok” care memoreaza in variabile valorile introduse in interfata, acestea putand fi accesata cu metodele getter aferente.

**Clasa SimulationFrame2** contine elementele specifice interfetei afisate in timpul simularii. Acesata contine 4 JPanel-uri : panel\_mare1, panel1, cozi1 si client. Panel\_mare1 contine celelalte panel-uri. Panel1 contine titlul “Server state & Waiting queue”. Panel-urile Cozi1 si Clienti contin un JScrollPanel si un JTextArea folosite pentru afisarea cozilor, respective a clientilor. Aceasta clasa contine o lista de cozi, o lista de client si o variabila currentTime care sunt folosite ca si parametrii in constructor, in acest fel fiind posibila actualizarea interfetei la fiecare moment de timp din clasa SimulationManager.

**Clasa SimulationManager** implementeaza interfata Runnable. In constructor se verifica daca s-a apasat butonul “ok” din prima interfata. In caz afirmativ, datele din interfata se memoreaza in variabilele numberOfServers ( numarul cozilor ), numberOfClients ( numarul clientilor) , timeLimit (timpul de simulare), minArrivalTime (timpul minim la care aunge clientul, la care poate fi distribuit intr-o coada), maxArrivalTime (timpul maxim la care poate ajunge clientul) , minProcessingTime ( timpul minim care este necesar procesarii clinetului, cat timp minim trebuie sa stea in coada ), maxProcessingTime ( timpul maxim care este necesar procesarii clientului). In metoda generateNRandomTasks() sunt generati N clienti aleator, folosind datele specific din interfata.Aceste date pot fi afisate in interfata2 (sunt datele de la timpul 0) . In metoda run(), folosind variabile currentTime ( care initial e 0 ), parcurgem cu un while fiecare moment de timp care este mai mic decat timeLimit (adica pana nu se opreste simularea). Pentru fiecare currentTime parcurgem lista de client si verificam pentru care client arrivalTime == currentTime. Toti acei client sunt adaugati in coada potrivita apeland metoda dispatchTask(t) din clasa Scheduler si sunt adaugati intr-un arrayList<Task> toRemove, urmand sa fie stersi din lista de clienti. Dupa parcurgerea acestui for, se afisaeaza lista actuala de cozi si de clienti in interfata2 sau in fisier. Se incrementeaza variabila currentTime si se apeleaza Thread.sleep(4000) pentru a fi oprita simularea timp de 4 secunde, pentru a putea vedea schimbarile din interfata.

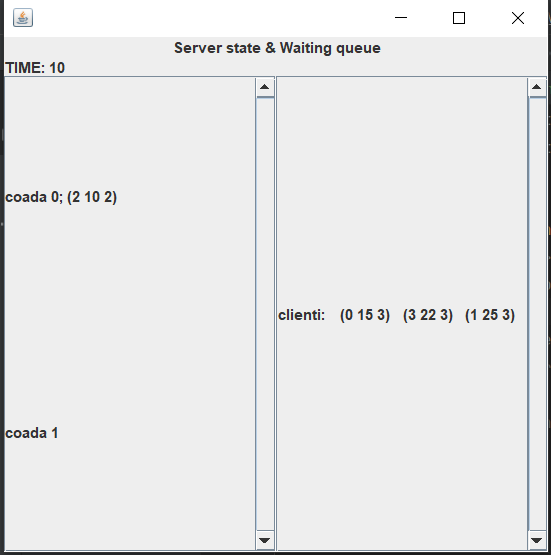
Dupa terminarea acestui while, se verifica daca au mai ramas client in lista de client ( timpul de simulare a fost prea scurt ). In caz afirmativ, se afiseaza o noua fereastra (frame3) care contine mesajul “Timpul simularii s-a terminat. Doriti sa continuati?” si doua butoane : “da” si “nu”. In cazul in care utilizatorul apasa butonul “da”, se continua simularea cat timp mai avem clienti in lista. In cazul in care utilizatorul apasa “nu”, se inchid toate cele trei ferestre si se termina simularea.

**Clasa Main** porneste simularea, folosind o instranta a clasei SimulationManager pentru care se apeleaza metoda run().

# 5.Rezultate

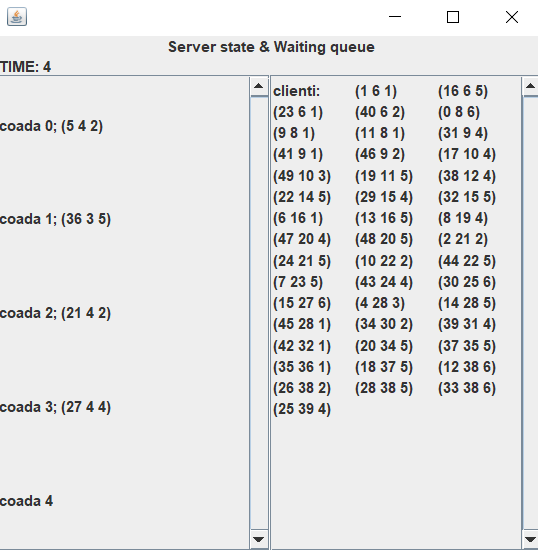
**Test 1:**

**N=4 Q=2 T(max simulation)=60 t(min arrival)=2 t(max arrival)=30 t(min service)=2 t(max service)=4**

****

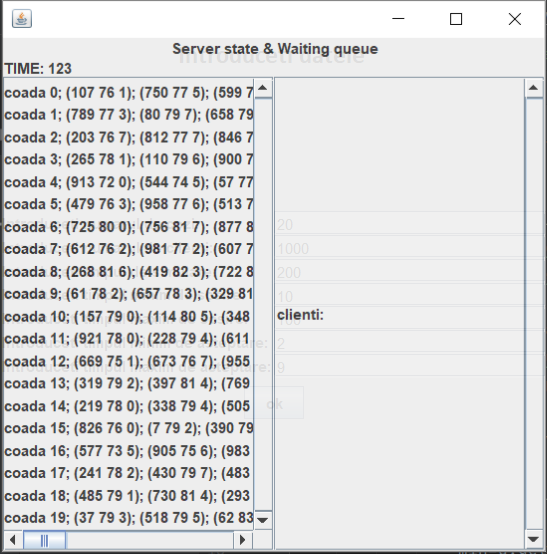
**Test 2:**

**N=50 Q=5 T(max simulation)=60 t(min arrival)=2 t(max arrival)=40 t(min service)=1 t(max service)=7**

****

**Test 3:**

**N=100 Q=20 T(max simulation)=200 t(min arrival)=10 t(max arrival)=100 t(min service)=3 t(max service)=9**

****

# 6.Concluzii

In analiza, proiectarea si implementarea acestei aplicatii, am recapitulate conceptele si paradigmele programarii orientate pe obiect si, totodata, am inteles mai bine functionarea thread-urilor. In plus, ascet proiect m-a ajutat sa exersez descompunerea unei probleme mari in subprobleme, pentru a scrie un cod mai bun sim ai eficient. De asemnenea, mi-am imbogatit cunostintele despre GUI.

# 7.Bibliografie

* http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
* http://www.tutorialspoint.com/java/util/timer\_schedule\_period.htm
* http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-andthreadpoolexecutor.html