Documentație tema 2

Simularea sistemului de așteptare bazat pe coada

Sigoiu George

Grupa 30224

Cuprins

[1. Obiectivul temei 3](#_Toc68877884)

[a) Obiectivul principal 3](#_Toc68877885)

[b) Obiective secundare 3](#_Toc68877886)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 4](#_Toc68877887)

[a) Cerințe funcționale 4](#_Toc68877888)

[b) Descriere cazuri de utilizare 4](#_Toc68877889)

[3. Proiectare 5](#_Toc68877890)

[a) Diagrama UML 5](#_Toc68877891)

[b) Structuri de date 5](#_Toc68877892)

[c) Algoritmi 6](#_Toc68877893)

[d) Interfața utilizator 6](#_Toc68877894)

[4. Implementare 7](#_Toc68877895)

[a) Clasele ConcreteStartegyQueue și ConcreteStrategyTime 7](#_Toc68877896)

[b) Clasa Strategy 7](#_Toc68877897)

[c) Clasa Server 7](#_Toc68877898)

[d) Clasa Task 7](#_Toc68877899)

[e) Clasa SelectionPolicy 8](#_Toc68877900)

[f) Clasa Scheduler 8](#_Toc68877901)

[g) Clasa SimulationManager 8](#_Toc68877902)

[h) Interfața grafică (clasa SimulationFrame) 8](#_Toc68877903)

[5. Rezultate 10](#_Toc68877904)

[6. Concluzii 14](#_Toc68877905)

[7. Bibliografie 15](#_Toc68877906)

# Obiectivul temei

## Obiectivul principal

Principalul scop al proiectului se rezumă la a crea o aplicație prin prisma căreia să se simuleze funcționarea unui sistem bazat pe așteptatul la coadă a clienților. În cele din urmă, prin analizarea acestuia se va putea determina și minimiza timpul de așteptare al fiecărui client în parte. Cu toate acestea, obiectivul primordial al cozii rămâne acela de a plasa "un client" într-o anumită listă de așteptare, chiar înainte de a primi respectivul “serviciu”.

## Obiective secundare

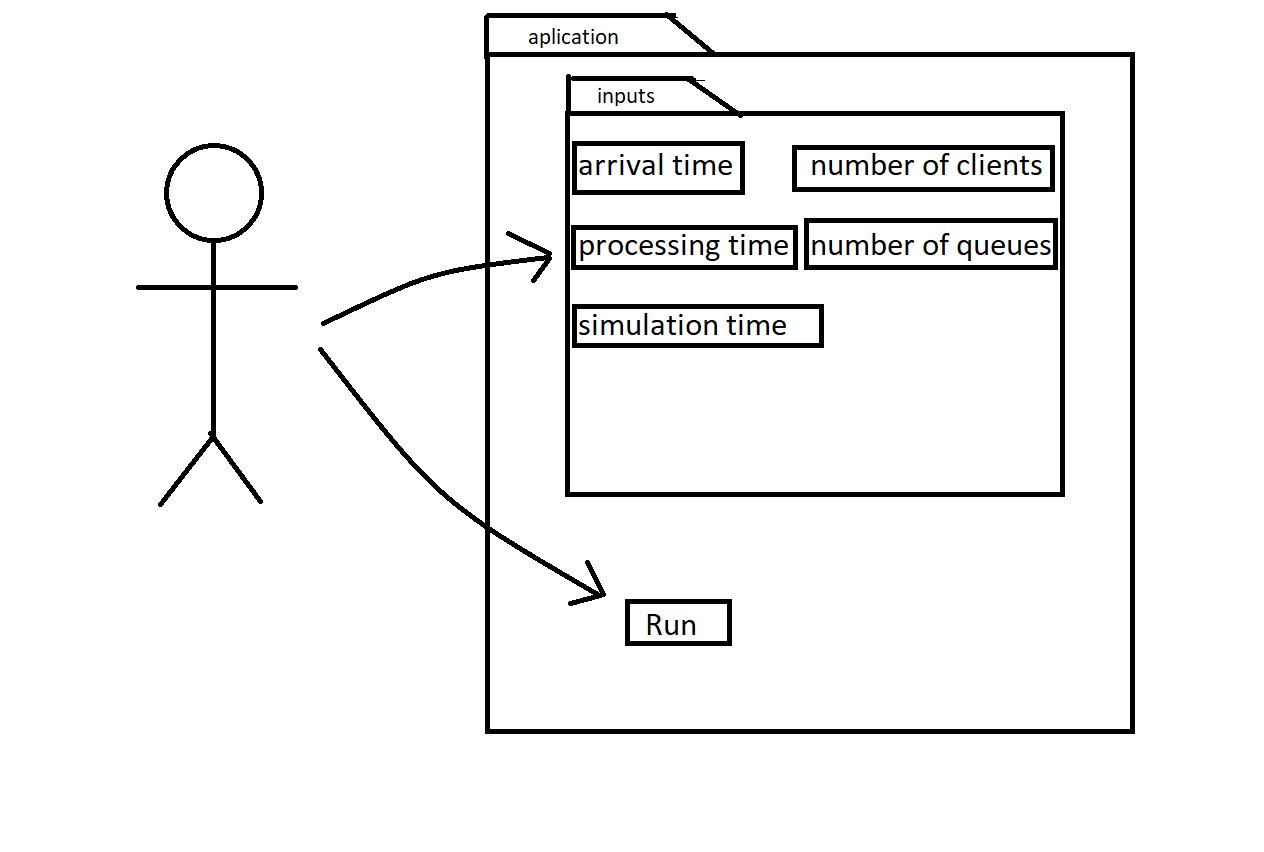
|  |  |  |
| --- | --- | --- |
| Obiectiv secundar | Descriere | Capitol |
| Dezvoltarea de use-case-uri și scenarii | Descrierea motului de utilizare a aplicației | [2](#_Analiza_problemei,_modelare,) |
| Alegerea structurilor de date | Descrierea structurilor de date folosite | [3](#_Structuri_de_date) |
| Crearea interfetei grafice | Descrierea elementelor din interfata grafica | [3](#_Interfața_utilizator) |
| Interfața grafică | Descrirea creerii interfetei grafice | [4](#_Interfața_grafică_(clasa) |

# Analiza problemei, modelare, scenarii, cazuri de utilizare

## Cerințe funcționale

Aplicația ar trebui sa simuleze o serie alcătuită dintr-un număr prestabilit de clienți (N) care ajung la prestație, fiind distribuiți într-un număr prestabilit de cozi ( Q ) de așteptare, fiind serviți pe rând, cate un client, si la final sunt scoși din coadă. Toți clienții sunt generați aleator la începutul simulării, fiind caracterizați de 3 parametrii: ID ( un număr intre 1 si N ), arrivalTime ( timpul generat când clientul respectiv este gata de a intra în coadă; de exemplu, timpul când clientul termină cumpărăturile și se așează apoi la o casă de marcat ), processingTime ( timpul necesar pentru a fi servit; de exemplu, timpul necesar ca casierul să scaneze toate produsele și clientul să efectueze plata ). Aplicația urmărește timpul total petrecut de fiecare client în coadă și calculează timpul mediu de așteptare. Fiecare client este adăugat în coada cu timpul de așteptare minim când arrivalTime este egal cu timpul de simulare.

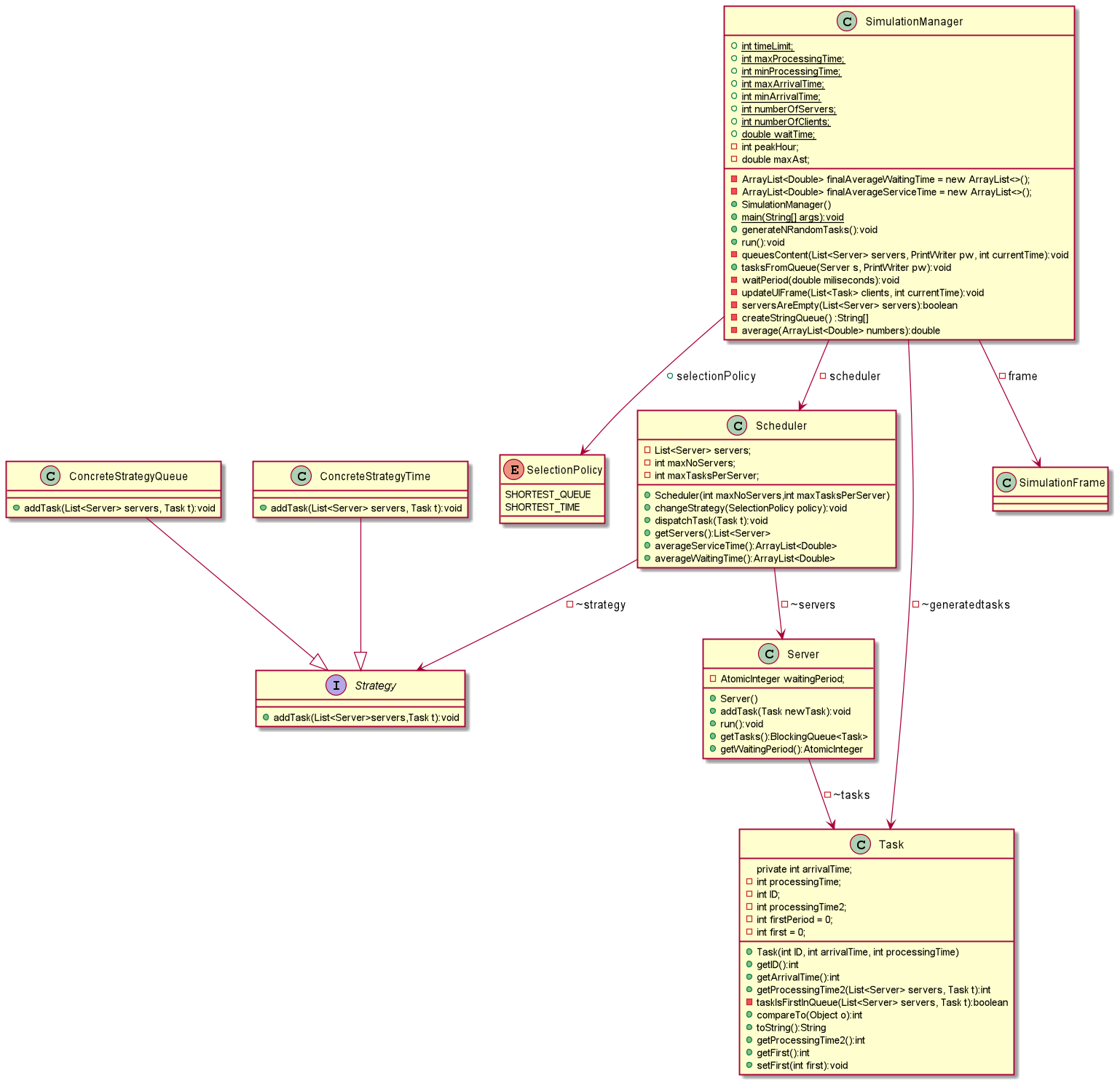
## Descriere cazuri de utilizare



Utilizatorul introduce datele de simulare în cod iar apoi rulează programul. Se pornește interfața utilizator, iar in zona de afișare a datelor de intrare ( [zona 4](#_Interfața_utilizator) ) vor apărea datele care au fost introduse de utilizator pentru simulare. Simularea se va încheia când va trece timpul maxim de simulare sau când nu vor mai exista clienți ( pentru că au fost scoși din cozi ).

# Proiectare

## Diagrama UML



## Structuri de date

Ca și structuri de date, mi-am definit următoarele clase: Task – care reprezintă clientul, Server – reprezintă coada de așteptare și Scheduler – reprezintă „componenta programului” care se ocupa de distribuirea clienților în cozi de așteptare.

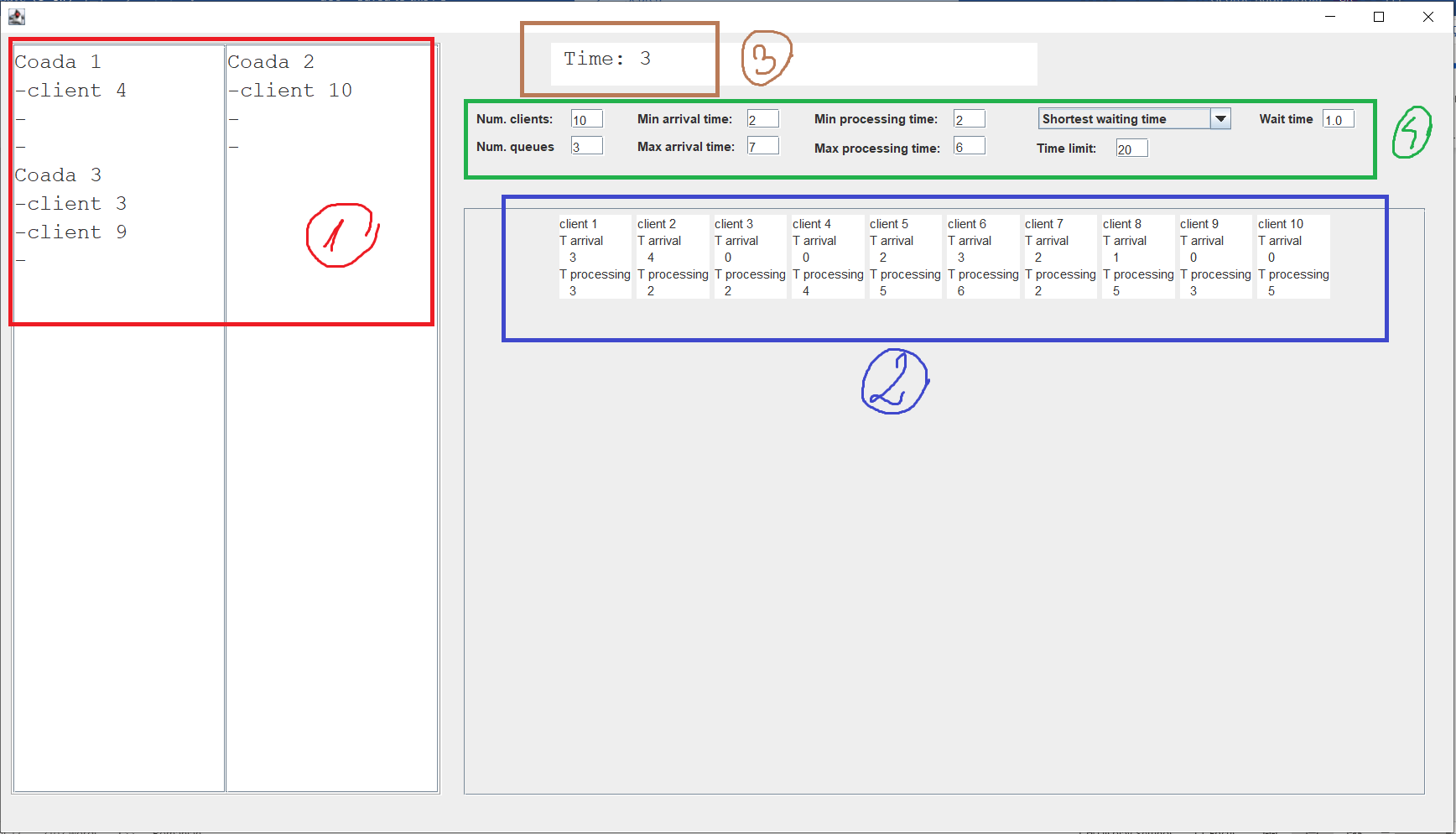
Clasa Task conține un ID, timpul de sosire la coadă și timpul cât durează servirea, atribute de tip int. Clasa Server conține o listă sincronizată de task-uri ( clienți ) și un integer sincronizat care reprezintă timpul înn care toți clienții vor părăsii coada. Clasa Scheduler conține o listă de cozi în care va distribui clienții în funcție de timpul de terminare al fiecărei cozi.

## Algoritmi

Am folosit algoritmul de sort din Collections pentru a sorta un arraylist, ce conține clienți, sortare efectuată în funcție de timpul de adăugare în coadă a fiecărui client, pentru a gestiona mai usor clienții care trebuie introduși într-o coadă.

## Interfața utilizator

Interfața utilizator este alcătuită din 4 zone principale, delimitate în figura de mai jos.



Zona 1: este alcătuită din 2 JTextArea, introduse in JScrollPane fiecare pentru a avea posibilitatea de scroll în caz de teste cu un număr mare de clienți sau un număr mare de cozi. În coloana din stânga vor apărea cozile cu numărul de ordine impar, împreună cu clienții distribuiți la acel moment de timp, iar în coloana din dreapta vor apărea cozile cu numărul de ordine par.

Zona 2: este alcatuită dintr-un JScrollPane, pentru posibilitatea de a da scroll în cazul de un număr mare de clienți, iar pentru fiecare client va apărea un JTextArea în care se vor afișa informațiile despre clientul respectiv. Informațiile sunt actualizate în fiecare moment de timp. În câmpul „T arrival” se indică timpul care mai trebuie așteptat de client pentru a fi introdus în coadă, iar în câmpul „T processing” se indică timpul de servire al fiecărui client (acest timp va fi decrementat in fiecare secunda daca clientul se află primul la coada respectivă ).

Zona 3: este un counter care afișează timpul curent al simulării.

Zona 4: este zona în care se afișează datele cu care s-a pornit simularea, câti clienți sunt inițial, câte cozi au fost create, intervalele de timp folosite pentru generarea aleatoare a timpului de sosire și a timpului de procesare, limita de timp și strategia de adăugare în coadă (clientul este adăugat în coadă cu cei mai puțini clienți sau în coada cu cel mai puțin timp de așteptare pentru a fi servit).

# Implementare

## Clasele ConcreteStartegyQueue și ConcreteStrategyTime

Aceste 2 clase implementeaza interfata Strategy si contin o singura metoda, o metoda de adaugare a unui client intr-o coada, avand ca parametrii lista de cozi si clientul respectiv.

ConcreteStrategyQueue introduce clientul in coada cu cel mai mic numar de clienti existenti in coada respectiva, iar ConcreteStrateguTime introduce clientul in coada cu cel mai scurt timp de asteptare pentru a fi servit.

Metode:

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

## Clasa Strategy

Aceasta clasa este o interfata si conține o metoda care trebuie implementata de clasele ConcreteStrategyTime si ConcreteStrategyQueue.

Metode:

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

## Clasa Server

Reprezinta o coada, contine o lista de task-uri si un integer ce reprezinta timpul de asteptare pentru ca coada sa ramana fara clienti. Clasa Server implementeaza interfata Runnable pentru ca fiecare coada va fi reprezentata printr-un thread.

Metode:

* public void addTask( Task t ) – adauga un client in coada, este diferita de metoda din Strategy.
* public void run( ) – descrie functionarea cozii, metoda in care, pentru fiecare client, se apeleaza sleep un timp egal cu timpul de servire al primului client, apoi clientul este eliminat din coada si este preluat urmatorul.

## Clasa Task

Reprezinta un client caracterizat de ID-ul unic, timpul de sosire la coada si timpul de servire cand ajunge primul la caoda. Clasa mai contine 3 atribute de tip int, folosite in unele metode pentru a determina timpul de asteptare si de sosire care trebuie afisate in interfata, sunt atribute pentru conditii suplimentare. Contine metode de tip getter si setter.

Metode:

* private boolean taskIsFirstInQueue( List <Server> servers, Task t ) – returneaza true daca clientul respectiv este primul in coada.
* private boolean getProcessingTime2( List <Server> servers, Task t ) – folosit pentru a face update in fiecare secunda la interfata utilizator.

## Clasa SelectionPolicy

Este o clasa de tip enum, care contine 2 elemente SHORTEST\_TIME si SHORTEST\_QUEUE.

## Clasa Scheduler

Reprezinta „dispecer-ul” care se ocupa cu organizarea si distribuirea clientilor in cozi. Contine ca atribute o lista de cozi, numarul de cozi, numarul maxim de clienti si strategia in functie de care se introduc clientii in cozi. Constructorul acestei clase creeaza cozile (numarul de cozi dat ca argument in constructor) si creeaza fiecare coada apoi creeaza fiecare thread corespunzator fiecarei cozi.

Metode:

* public void changeStrategy( SelectionPolicy policy ) – seteaza strategia in functie de care se vor adauga clientii in coada.
* public void dispatchTask( Task t ) – seteaza strategia in functie de care se vor adauga clientii in coada
* private ArrayList<Double> averageServiceTime( ) – returneaza o lista cu timpul medie de servire pentru fiecare coada.
* private ArrayList<Double> averageWaitingTime( ) – returneaza o lista cu timpul medie de astepatare pentru fiecare coada.

## Clasa SimulationManager

Este clasa care rulează întreaga simulare de gestiune a cozilor. Conține ca atribute toate datele necesare simulării:

* timeLimit – timpul maxim al simulării,
* maxProcessingTime și minProcessingTime – reprezină intervalul din care se va genera pentru fiecare client timpul de servire când ajunge primul la coadă,
* maxArrivalTime și minArrivalTime – reprezintă intervalul din care se va genera pentru fiecare client timpul de sosire la coadă,
* numberOfServers – numărul de cozi,
* numberOfClients – numărul de clienți,
* selectionPolicy – [strategia](#_Clasa_Strategy) de adăugare în coadă,
* [scheduler](#_Clasa_Scheduler) – dispecerul, distribuie clienții cozilor,
* frame – interfața grafică,
* generatedTasks – listă de clienți generați.

Clasa mai conține 2 liste de numere zecimale, semnificând timpul mediu de așteptare, respectiv timpul mediu de servire, iar media acestor 2 liste va afișa la final timpul de așteptare mediu și timpul de servire mediu.

Metode:

* public void generateNRandomTasks( ) – generează un număr prestabilit de clienți cu valori random în ceea ce privește timpul de sosire și timpul de servire.
* public void run( ) – este metoda care descrie “viața” simulatorului, adică reprezintă funcția pe care trebuie să o îndeplinească clasa SimulationManager. În fiecare secundă a simulării trimite câte un client dispecerului (scheduler ) pentru a așeza într-o anumită coadă clientul care urmează să-și “plătescă achizițiile din magazin”, apoi generează informațiile corespunzătoare pentru a fi afiște în interfața grafică.
* private String[] createStringQueue( ),

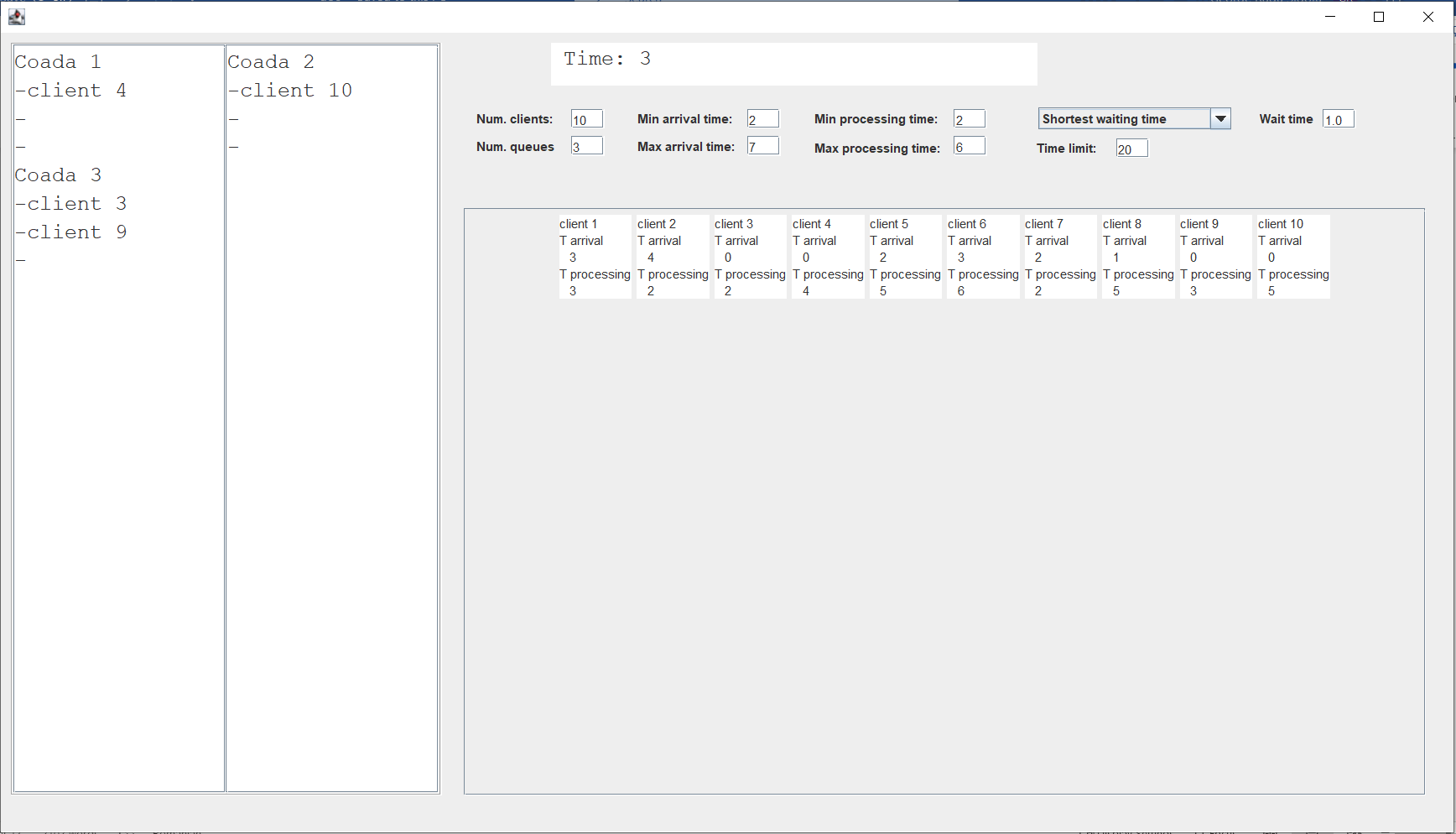
private void tasksFromQueue( Server s, PrintWrtire pw ),

private void queuesContent( List <Server> servers, PrintWriter pw, int currentTime)

* aceste metode generează informațiile ce trebuie afișate în logEvents.txt și în interfață.

## Clasa SimulationFrame

Clasa respectivă reprezintă interfața grafică.



Fereastra este construită din 2 panel-uri principale, așezate pe panel-ul frame-ului care are layout-ul AbsoluteLayout, ceea ce mi-a permis să așez cele 3 panel-uri în poziția dorită. Panelul 1, cel pe care au fost asezate 2 JTextArea ( partea din stânga ) în care se vor scrie informații referitoare la cozi. Panelul 2, cel pe care sunt așezate 10 JTextArea, a fost construit în așa fel încât să poae fi afșați câți mai mulți cienti, pentru a putea fi observată așezarea lor la coadă. Acest panel conține un alt panel în interiorul lui în care sunt introduse aceste JTextArea, iar la fiecare 12 clienți se creează un panel nou în acest panel principal și se adaugă următorii clienți în el. De exemplu, dacă la început avem 10 clienți, informațiile acestora vot fi afișate ca în imaginea de mai sus, dar daca avem 14 clienți, vor fi afișați 12 ca în imaginea de mai sus, iar sub acele JTextArea vor mai apărea în 2 JTextArea cu informațiile celor 2 clienți, iar dacă vor fi 30 de clienți, informațiile vor fi afișate pe 3 randuri. Fereastra de interfță grafică mai conține, dic câte se observă, 8 JLabel-uri și 1 JComboBox în care sun scrise variabilele pentru simulare, iar alături sunt 7 JTextField-uri în care sunt scrise valorile acestor variabile. În field-ul din partea de sus este afișat timpul curent.

# Rezultate

Pentru a testa buna funcționare a aplicației am pornit simularea cu date de intrare aleatoare pentru a observa dacă aplicația gestionează clienții și cozile într-un mod optim. Am realizat mai multe teste pentru a observa afișarea din fișier, dar și cea din interfață. Mai jos sunt afișate 2 teste, cu rezultatele lor.

Am testat pe date de intrare mici, de exemplu 4 clienți, 2 cozi, timp de așteptare din intervalul [2, 30], timp de așteptare din intervalul [2,4] și timp de simulare de 60 de secunde.

Rezultatul testului:

Text

Description automatically generated

A screen shot of a computer

Description automatically generated with medium confidence

Un alt test a fost făcut pe următoarele date de intrare, iar mai jos este rezultatul.

Text

Description automatically generatedText, letter

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

# Concluzii

Asemenea unei dezvoltări anterioare a acestui program, interfața grafică se poate modifica în așa fel, încât datele de intrare să poată fi citite încă din interfață, iar simularea să poată fi inițiată tot din interfață, odată cu apăsarea butonului de "START". Pe de altă parte, o altă metodă de dezvoltare ar fi aceea ca fiecare coadă să afișeze în cele din urmă timpul mediu de servire, din perspectiva reală a accepțiunii anterior menționate.

Realizând această temă am învățat să lucrez cu thread-uri, să înțeleg ce este un thread, la ce sunt bune, la ce pot fi folosite pentru a ușura treaba si, de asemenea, am învățat cum să le gestionez, cum să le descriu „treaba” pe care trebuie sa o facă.

# Bibliografie

Thread-uri

* [java.util.Timer.schedule() Method - Tutorialspoint](https://www.tutorialspoint.com/java/util/timer_schedule_period.htm)
* [Java Thread Pool Example using Executors and ThreadPoolExecutor (javacodegeeks.com)](https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html)
* <https://www.w3schools.com/java/java_threads.asp>
* <https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html>