DOCUMENTATIE

TEMA *2*

NUME STUDENT: Floare George-Octavian

GRUPA: 30222

# CUPRINS

[1. Obiectivul temei 3](#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](#_Toc95297886)

[3. Proiectare 3](#_Toc95297887)

[4. Implementare 3](#_Toc95297888)

[5. Rezultate 3](#_Toc95297889)

[6. Concluzii 3](#_Toc95297890)

[7. Bibliografie 3](#_Toc95297891)

# Obiectivul temei

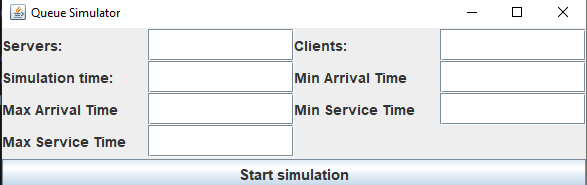
# Scopul principal al acestei teme este de a crea o aplicatie eficienta care sa simuleze procesul de gestionare a mai multor cozi de clienti, astfel incat clientii sa fie impartiti in coada care are timpul de asteptare minim. Acest lucru va minimiza timpul total de asteptare necesar pentru indeplinirea cerintelor si va imbunatati experienta utilizatorului. Pe langa obiectivul principal, mai exista si cateva obiective secundare care trebuie indeplinite. Unul dintre aceste obiective este verificarea input-ului introdus de utilizator pentru a se asigura o legatura unu la unu intre ceea ce este introdus in casetele text si ceea ce se afla in memoria aplicatiei. In plus, aplicatia trebuie sa utilizeze doua butoane pentru a verifica input-ul dat de utilizator in cadrul celor sapte casete de text si pentru a realiza simularea conform parametrilor introdusi. De asemenea, aplicatia trebuie sa genereze clientii aleatoriu, conform datelor introduse, si sa-i introduca in cozi in functie de timpul de sosire si de timpul de realizare a serviciilor pentru fiecare client. Evolutia simularii trebuie afisata in timp real, astfel incat utilizatorul sa poata urmari toti clientii si sa obtina date importante din simulare. Pentru a evita situatii in care programul se afla intr-o stare necunoscuta, toate erorile trebuie interceptate si afisate in pop-up-uri si coduri de eroare care pot fi depanate mai departe consultand un manual specific pentru erori. De asemenea, utilizatorul trebuie informat cu privire la corectitudinea input-ului introdus de la tastatura prin pop-up-uri.

# Pentru a simula paralelismul vietii reale, aplicatia trebuie sa foloseasca un thread separat pentru fiecare coada. Aceasta functionalitate va permite clientilor sa fie serviti simultan, ceea ce va reduce timpul de asteptare si va imbunatati eficienta aplicatiei. In concluzie, acest proiect va imbunatati experienta utilizatorului si va oferi o solutie eficienta si concisa pentru gestionarea mai multor cozi de clienti.

# Analiza problemei, modelare, scenarii, cazuri de utilizare

O problema comuna intalnita in gestionarea clientilor intr-o afacere este organizarea eficienta a timpului si resurselor disponibile. Pentru a remedia aceasta problema, exista o metoda simpla si eficienta, care consta in crearea unei liste pentru clientii sositi si introducerea lor in coada atunci cand timpul lor de sosire este egal cu timpul simularii. Aceasta lista este generata aleatoriu, pentru a asigura o distributie uniforma a clientilor la casele de servire. Fiecare client este apoi preluat din coada la momentul potrivit si servit, prin decrementarea timpului de servire si incrementarea timpului total de simulare. Pe masura ce timpul avanseaza, se recalculeaza timpul de asteptare pentru fiecare coada, pentru a putea introduce noi clienti si a le oferi o experienta cat mai placuta si eficienta. Odata ce timpul de servire al unui client se termina, acesta este scos din coada si se repeta procesul pentru urmatorul client din coada. Aceasta abordare permite o organizare mai buna a resurselor si o reducere a timpului de asteptare pentru clienti, ducand la o experienta mai placuta si satisfacatoare pentru toti cei implicati.

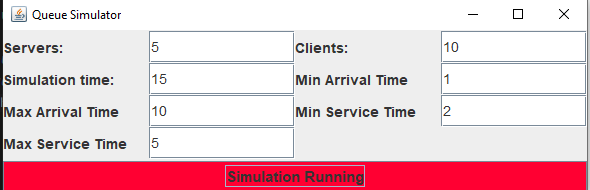
Interfata grafica are o singura fereastra in care se introduc datele simularii si se porneste simularea.



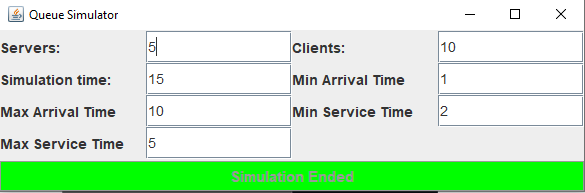
Interfata este formata din 7 Text Field-uri:

* “Servers” in care se introduce numarul de servere, echivalent cu numarul de cozi din simulare;
* “Clients” in care se introduce numarul de clienti;
* “Simulation time” in care se introduce timpul maxim al simularii pana la care trebuie sa ruleze;
* “Max Arrival Time” in care se introduce timpul maxim la care un client poate ajunge;
* “Min Arrival Time” in care se introduce timpul minim la care un client poate ajunge;
* “Max Service Time” in care se introduce timpul maxim de servire al unui client;
* “Min Service Time” in care se introduce timpul minim de servire al unui client;

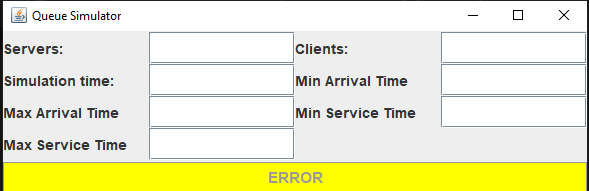
De asemenea, exista si un buton de pornire a simularii : “Start simulation”. In momentul in care este apasat acesta isi schimba background-ul in rosu, iar textul in “Simulation running”.



In momentul in care simularea se termina, butonul isi schimba iar culoarea in verde si textul in “Simulation ended”.



Daca in momentul in care simularea este pornita un input nu e corect se va afisa un mesaj de eroare.

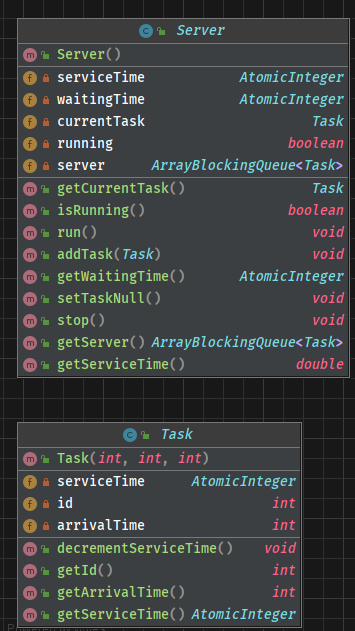


# Proiectare

Implementarea orientarii pe obiecte (**OOP**) a aplicatiei noastre este cruciala pentru asigurarea unei dezvoltari eficiente si usor de intretinut pe termen lung. O metoda populara si bine stabilita de a implementa OOP este prin utilizarea modelului Model-View-Controller (MVC), care separa aplicatia in trei componente distincte: Model, View si Controller. In ceea ce priveste structurarea datelor, utilizarea modelului MVC ne ofera avantajul de a putea separa datele din aplicatie in functie de rolul lor si de a le gestiona intr-un mod uniform. Astfel, putem separa datele referitoare la model intr-un mod clar si organizat, data fiind importanta lor critica pentru functionarea aplicatiei. De asemenea, prin separarea logicii de afisare, putem avea o viziune clara asupra inter-actiunii dintre aplicatie si utilizator, ceea ce conduce la o proiectare concisa si eficienta. Pe scurt, utilizarea modelului MVC pentru implementarea OOP a aplicatiei noastre reprezinta o solutie excelenta pentru o structurare clara si uniforma a datelor si a logicii, asigurand astfel o dezvoltare eficienta si usor de intretinut a aplicatiei pe termen lung.

**MODEL**: cuprinde doua clase, Server si Task. Aceste clase vor fi folosite pentru simulare.

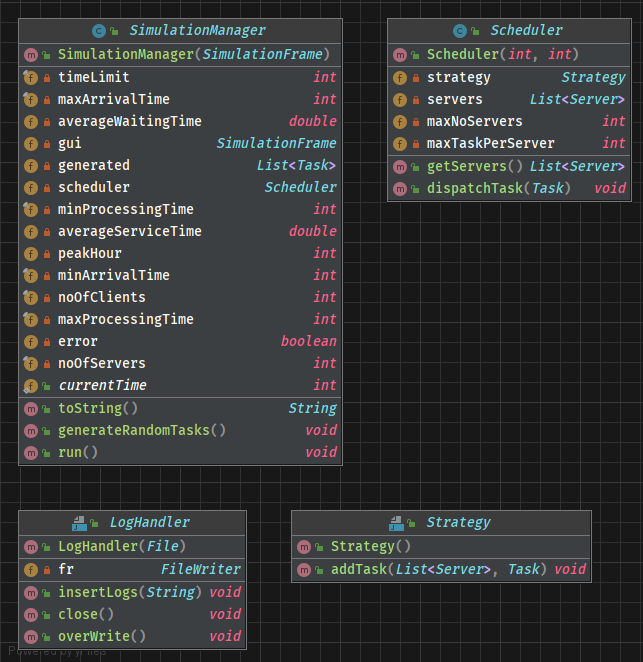
* + **Task** este clientul care va ajunge in coada. Aceasta clasa contine:
    - “**id**” : id-ul fiecarui client, de tip int;
    - “**arrivalTime**”: momentul la care clientul ajunge in coada;
    - “**serviceTime**”: timpul necesar rezolvarii task-ului
  + **Server** este coada care va fi folosita. Clasa contine:
    - “**serviceTime**”, “**waitingTime**”, “**currentTask**” ;
    - “**server**” care este un **ArrayBlockingQueue<Task**> care ajuta la sincronizarea dintre thread-uri;
    - “**running**” care ajuta la inchiderea thread-urilor dupa finalizarea operatiilor;



* **VIEW**: implementarea interfetei grafice a aplicatiei, cu un constructor care initializeaza dimensiunile ferestrei, text field-urile, label-urile si butonul, folosite pentru a obtine datele de intrare ale aplicatiei.



* Pachetul **Business** ( Controller) realizeaza conexiunea dintre model si view.
  + Clasa **LogHandler** se ocupa de realizarea log-urilor in care se poate vedea output-ul aplicatiei.
  + Clasa **Scheduler** instantiaza thread-urile pentru fiecare server si implementeaza metoda “**dispatchTask**” folosita pentru a adauga clienti noi in coada.
  + Clasa **Strategy** contine metoda “**addTask**” care se ocupa cu alegerea serverului adecvat in functie de timpul minim de asteptare, dupa care introduce un nou Task ( client ) in coada.
  + Clasa SimulationManager se ocupa de sincronizarea aplicatiei in functia “run”. Aceasta contine:
    - **currentTime** : timpul curent al simularii care este incrementat o data la o secunda
    - **maxProcessingTime** : timpul maxim de servire
    - **minProcessingTime** : timpul minim de servire
    - **maxArrivalTime** : timpul maxim la care poate ajunge un client
    - **minArrivalTime** : timpul minim la care poate ajunge un client
    - **timeLimit** : timpul maxim al simularii
    - **noOfServers** : numarul de thread-uri ce trebuie create
    - **noOfClients** : numarul de clienti care intra in cozi
    - **scheduler** : o instanta a clasei Scheduler
    - **generated** : lista de clienti generati aleatoriu
    - **error** : variabila boolean care verifica daca input-ul este corect; in caz contrar se afiseaza un mesaj de eroare in interfata grafica



# Implementare

**Diagrama UML** generala a proiectului:



In clasa **SimulationManager**:

* “**run**” este responsabila cu sustinerea simularii cat timp timpul curent (currentTime) este mai mic decat timpul maxim al simularii (timeLimit). Cat timp timpul este valid, se realizeaza scrierea in fisierul log.
* “**checkGeneratedClients**” verifica daca toti clientii generati au fost serviti. In cazul in care toti au fost serviti se incheie simularea, intrucat nu are rost sa fie pornita simularea cand nu mai sunt clienti care trebuiesc serviti.
* “**toString**” realizeaza o concatenare a datelor pentru a putea fi scrise in fisierul log.
* “**generateRandomTasks**” genereaza numberOfClients clienti, intr-un mod aleator, fiind folosita clasa Random din Java. Id-ul fiecarui client este un numar unic, de la 0 la numberOfClients – 1 iar arrivalTime si serviceTime corespunzatoare fiecarui task, sunt cuprinse intre minimul si maximul lor aferente (intervalele), introduse prin interfata grafica.
* “computeAverageServiceTime” calculeaza timpul total de servire pentru toata simularea care este ulterior impartita la numarul de clienti

In clasa **Server**:

* “**run**” ruleaza cat timp coada nu e goala sau cat timp task-ul curent nu e null si decrementeaza waitingTime si pune thread-ul pe sleep. In cazul in care o exceptie este aruncata, “running” se seteaza pe fals, moment in care thread-ul se inchide.
* “stop” seteaza atributul “running” pe fals
* “**addTask**” recalculeaza “waitingTime” adunand “serviceTime” si adauga un Task in coada

In clasa **Scheduler**:

* **Constructorul** initializeaza fiecare server si il porneste.
* “**dispatchTask**” apeleaza metoda “addTask” din clasa **Strategy**

In clasa **Strategy**:

* “**addTask**” se ocupa cu alegerea serverului adecvat in functie de timpul minim de asteptare, dupa care introduce un nou Task ( client ) in coada. Aceasta operatie se efectueaza la fiecare incrementare a timpului curent.

In clasa **Task**:

* “decrementServiceTime” decrementeaza timpul de servire al unui client (task)

In clasa **LogHandler**:

* **Constructorul** primeste ca parametru un fisier si instantiaza un obiect de tip FileWriter.
* “**overwrite**” este un mecanism de securitate folosit pentru a asigura faptul ca fisierul este gol la inceputul unei simulari; acesta trebuie apelat la inceputul unei simulari
* “**insertLogs**” are rolul de a scrie in fisier un buffer dat ca parametru
* “**close**” trebuie apelat la finalul unei simulari si are rolul de inchide fisierul deschis in constructor

# Rezultate

In cadrul procesului de dezvoltare al oricarei aplicatii, testarea este una dintre cele mai importante etape. Aceasta asigura ca aplicatia va functiona corect si va fi capabila sa gestioneze situatii diverse, chiar si cele imprevizibile. In ceea ce priveste aplicatia noastra, partea de testare a fost realizata conform cerintelor si a implicat efectuarea a trei teste, fiecare test fiind descris intr-un fisier text: "test1.txt", "test2.txt" si "test3.txt". Pentru fiecare test, au fost pregatiti si introdusi parametri specifici de intrare pentru a verifica functionalitatea aplicatiei in situatii diferite si pentru a asigura ca aplicatia se comporta corespunzator. In cadrul fiecarui test, am verificat daca aplicatia raspunde conform asteptarilor si ca produce rezultatele asteptate. In cazul in care apar probleme, acestea sunt identificate, raportate si remediate inainte de a lansa aplicatia in mediul de productie. Testarea este un proces continuu si va fi efectuata pe intreaga durata de viata a aplicatiei, pentru a ne asigura ca aceasta va functiona corect si va satisface cerintele utilizatorilor.

**Test1**:

* Number of clients = 4;
* Number of queues = 2;
* Simulation interval = 60 s;
* [Minimum arrival time, Maximum arrival time] = [2, 30];
* [Minimum service time, Maximum service time] = [2, 4].

**Rezultat**:

* Average Waiting Time: 1.9090909090909092
* Average Service Time: 2.75
* Peak Hour: 4

**Test2**:

* Number of clients = 50;
* Number of queues = 5;
* Simulation interval = 60 s;
* [Minimum arrival time, Maximum arrival time] = [2, 40];
* [Minimum service time, Maximum service time] = [1, 7].

**Rezultat**:

* Average Waiting Time : 3.319018404907975
* Average Service Time: 3.26
* Peak Hour: 29

**Test3**:

* Number of clients = 1000;
* Number of queues = 20;
* Simulation interval = 200 s;
* [Minimum arrival time, Maximum arrival time] = [10, 100];
* [Minimum service time, Maximum service time] = [3, 9].

**Rezultat**:

* Average Waiting Time: 117.7293154
* Average Service Time: 5.516
* Peak Hour: 99

# Concluzii

Concluzia acestei documentatii este ca obiectivul acestei teme este de a crea o aplicatie care sa simuleze procesul de gestionare a mai multor cozi de clienti prin impartirea acestora la coada care are timpul de asteptare minim, pentru a minimiza timpul de asteptare total necesar indeplinirii cerintelor. Documentatia ofera o analiza a problemei, modele de scenarii si cazuri de utilizare, precum si o proiectare si implementare a solutiei propuse. Aplicatia utilizeaza modelul MVC si are o interfata grafica cu 7 casete de text si un buton pentru pornirea simularii. In plus, aplicatia ofera informatii cu privire la erori si ofera o verificare a input-ului utilizatorului. Rezultatele obținute sunt afisarea evolutiei simularii in timp real si obtinerea unor date importante din simulare. Prin urmare, aplicatia poate fi utilizata pentru a simula procesul de gestionare a mai multor cozi de clienti si pentru a evalua performanta sistemului in diferite scenarii de utilizare.

# Bibliografie

<https://www.digitalocean.com/community/tutorials/atomicinteger-java>

<https://www.javatpoint.com/java-filewriter-class>

<https://www.w3schools.com/java/java_threads.asp>

<https://www.vogella.com/tutorials/JavaConcurrency/article.html#:~:text=1.1.-,What%20is%20concurrency%3F,the%20interactivity%20of%20the%20program.>