Tema 2

Queue Simulation

Disciplina: Tehnici de programare

Mariciuc Andrei-Alexandru

Cuprins

[Obiectivul temei 2](#_Toc68698166)

[Analiza problemei 3](#_Toc68698167)

[Modelare 4](#_Toc68698168)

[Scenarii 5](#_Toc68698169)

[Proiectare 6](#_Toc68698170)

[Diagrama UML 7](#_Toc68698171)

[Structuri de date 8](#_Toc68698172)

[Algoritmi 9](#_Toc68698173)

[Implementare 10](#_Toc68698174)

[Concluzii 13](#_Toc68698175)

[Biliografie 14](#_Toc68698176)

# Obiectivul temei

Intriga de la care începe proiectul este dorința de a analiza timpul pe care îl petrec oamenii la coada unor ghișee bancare, complexe comerciale, parcări sau spectacole artistice. Astfel, dorim să dezvoltăm un sistem software capabil de a realiza o simulare cât mai reală a unor situații de așteptare la o coadă generică pe care hai s-o numim server. De asemenea, ne dorim ca modul de interacțiune cu sistemul să fie unul cât mai intuitiv și firesc, de aceea, dorim implementarea unei interfețe grafice, capabilă să preia informații despre datele necesare simulării (timpul simulării, intervalul timpilor de procesare, intervalul timpilor de ieșire, numarul de taskuri și numărul de servere) și să lanseze simularea arătând evoluția cozilor în timp real. Alte obiective secundare pot fi reprezentate de următoarele:

* Analiza atentă a nevoilor unui presupus utilizator și modelarea unei soluții care să rezolve problema data *[cap. Analiza Problemei.\*];*
* Proiectarea unei ierarhii de clase eficientă, care să faciliteze întreținerea aplicației și care să faciliteze înțelegerea codului de alți programatori *[cap. Proiectare]*;
* Dezvoltarea unor metode algorimice sigure de a selecta stategia de selectarea unui server de către taskuri*[cap. Proiectare.Algortimi]*;
* Design-ul unei interfețe cu utilizatorul folosind API-ul Swing *[cap. Implementare]*;
* Implementarea unei clase capabile să salveze fișierul .log care va furniza informații despre fiecare secundă a simulării*[cap. Proiectare.Algortimi]*;
* Afișarea grafică cât mai intuitivă folosind desenarea pe ecran*[cap. Implementare]*;

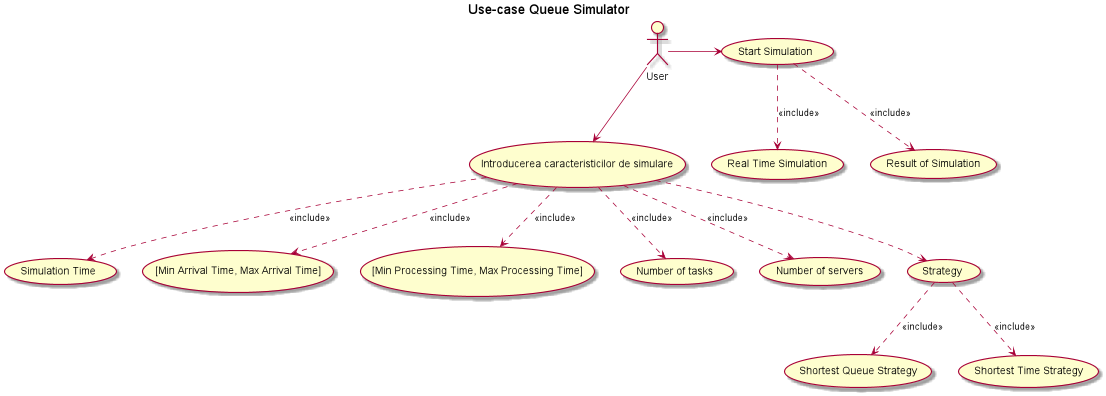
# Analiza problemei

Putem începe analiza prin identificarea unor cerințe funcționale fundamentale (“trebuie”) urmate de cele opționale („ar fi drăguț”):

* Aplicația trebuie sa lase utilizatorul sa insereze caracteristicile de simulare: *timpul de simulare, [timpul minim de procesare, timpul maxim de procesare], [timpul minim de început, timpul maxim de început], numărul de task-uri;*
* Simulatorul trebuie să permită afișarea unor obiecte geometrice desenate care să se sincronizeze cu evoluția reală în timp a cozilor, dar și prin intermediul unui log of events genrat într-un fișier text;
* Simulatorul ar trebui să creeze un thread pentru a simula întreaga funcționalitate a unui set de cozi de așteptare;
* Simulatorul trebuie să genereze random un set de date de intrare (taskuri și servere) în urma cărora să se execute simularea;
* Simulatorul ar trebui să genereze Q threaduri pentru simularea fiecărei cozi, cu funționalitate identică;
* Simulatorul trebuie să furnizeze niște concluzii despre simulare la încheierea acesteia cum ar fi: media timpului așteptat (average waiting time), media timpului de servire așteptat (average service time), timpul la care au fost cele mai multe taskuri care să aștepte la coadă (peak hour);
* Ar fi bine dacă, putem revenii printr-un sistem *undo* la datele introduse anterior;
* Ar fi drăguț, dacă aș putea implementa două moduri de afișare, unul text și unul grafic;
* Ar fi tare dacă, putem introduce datele de intrare și prin intermediului unor fișiere;

## Modelare

Analiza bazată pe cerințele funcționale de mai sus conduce spre o reprezentare grafică a interacțiunii unui presupus utilizator cu sistemul descrisă în *figura 1*.



**Figura 1** *Use-case calculator de polinoame*

Odată pornită aplicația, utilizatorul intră în contact direct cu toate funcționalitățile principale ale simulatorului. El are posibilitatea de a lansa o fereastră cu simularea prorpiu-zisă, dar trebuie să introducă mai întâi caracteristicile de simulare: *Simulation Time, [Min Process Time, Max Process Time], [Min Service Time, Max Service Time], Number of tasks, Number of servers, Strategy (Shortest Time Strategy, Shortest Queue Strategy)*. După ce simularea este încheiată, sunt afișare atât în fișierul .log cât și în interfață cu utilizatorul rezultatele simulării.

## Scenarii

În continuare se vor prezenta două scenarii de utilizare, unul care va trata cazul fericit, iar altul, cazul mai puțin fericit de interacțiune a utilizatorului cu sistemul.

* Scenariul fericit - caracteristicile de simulare au fost corect introduse:

1. Utilizatorul introduce prin intermediul interfeței grafice, caracteristicile de simulare în mod corect! (inputul este reprezentat prin numere și *max arrival time + max processing time <= simulation time*);
2. Utilizatorul selectează una din cele două strategi;
3. Simulatorul efectuează calculul și afișează în interfața grafică evoluția cozilor și a task-urile în timp real;
4. Se termină simularea, istoria evoluției este salvată în fișierul text și rezultatele corecte ale simulării apar în interfața grafică;

* Scenariul mai puțin fericit - input greșit/efectuarea unor operații imposibile:

1. Utilizatorul introduce alte caractere decât numere în field-urile de text asociate caracteristicilor de simulare sau *max arrival time + max processing time > simulation time*. Alt caz special nefericit, este atunci când se încearcă simularea unui pachet de date care nu vor termina de a fi servite în timpul alocat simulării, acest caz va fi semnalat la sfârșitul simulării care va indica dacă s-a terminat cu bine sau nu execuția
2. Pentru cazurile descrise mai sus, interfața grafică va arunca o fereastră pop-up care va descrie problema întâmpinată (ex: “nu ai introdus un număr întreg la simulation time”, ”remaining unprocessed status: true” etc), cu posibilitatea de a revenii în aplicație din starea în care a fost lasată printr-o singură apăsare de buton: ”OK!”.

Cazuri de utilizare

* Simulatorul poate fi folosit pentru simularea “statului la coadă”, în vederea obținerii unei concluzii despre modul în care trebuie organizate serverele, fie ca acestea reprezintă complexe comerciale, ghișee, transport în comun, sistem medical;
* Pentru simularea sistemului medical la preluarea pacienților cu cazuri urgente, pentru asta trebuie implementată o altă strategie, în care să se țină cont la alegerea unui task de gradul gravității stării de sănătate a pacienților;

# Proiectare

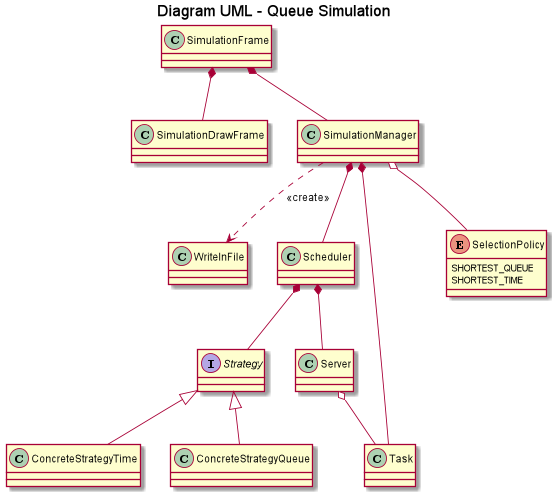
Pentru proiectarea unei soluții aleg să folosesc modelul MVC (Model View Controller). Mai întâi trebuie să proiectăm *modelul,* așadar avem nevoie clase care să poate modela cuvintele cheie enumerate până acum (simulare, coada, desenare ș.a). Definim “cel care așteaptă la coadă” ca fiind un *Task,* iar “cel la care se așteaptă” ca fiind un *Server.* Se observă că taskurile vor face parte din alcătuirea unui server la un anumit moment în timp, asta conduce spre concluzia că clasa *Task* va fi asociată prin agregare clasei *Server*. Serverele ar trebui să lucreze în paralel, de aceea vom folosi o tehnică bazată pe fire de execuție (thread-uri), pentru asta fie clasa *Server* trebuie să extindă clasa *Thread* din java, fie să implementeze interfața *Runnable,* vom alege să implementăm a doua variantă întrucât nu trebuie să moștenim mai mult decât metoda funcționalitatea clasei *Thread*, și anume faptul că implementăm metoda *run().* Faptul că avem strategii de alocare a taskurilor la serverele potrivite, înseamană că ar fi de ajutor să implementăm două clase corespunzătoare celor două strategii (Shortest Queue și Shortest Time). Hai să le denumim *ConcreteStrategyQueue* și *ConcreteStrategyTime*, iar pentru a evidenția funcționalitatea comună a celor două clase, acestea vor implemnta o interfață numită *Strategy.* Acum că avem taskurile, serverele și strategiile modelate, nu rămâne decât să avem o clasă care să managereze sistemul creat, astfel, avem nevoie de un planificator care să țină cont de strategie, să lanseze aplicația și să împartă taskurile la servere. Numim *Scheduler* clasa care modelează planificarea simulării, și mai mult observăm că exită o relație de agregare intre intefața care ține cont de strategie și planificator iar pe altă parte, o relație de compoziție între servere și planificator. Pentru scrierea în fișier am ales să declar de asemenea o clasă, și anume *WriteInFile.*

După ce datele caracteristice simulării sunt preluate din interfața grafică avem nevoie de o clasă care să preia aceste date, să genereze random un set de taskuri și să dea startul simulării. Numim această clasă *SimulationManager*, care va fi compus din planificator, dintr-o enumerație care corespunde cu cele două strategi (*SelectionPolicy*), și dintr-o listă de taskuri. Clasa de simulare va fi nevoită de asemenea să implementeze interfața *Runnable* deoarece ea corespunde unui thread mama din care se vor da naștere mai multor threaduri.

Partea de view este combinată cu cea de controller întrucât controlul se bazează pe un singur buton și anume cel care dă start la simulare. Totuși, această parte este despărțită în două clase, una care să controleze datele de intrare sub forma unui frame (SimulationFrame), și alta care să afișeze pașii simulării sub formă grafică (SimulationDrawFrame).

## Diagrama UML

În cele ce urmează este prezentată diagrama UML, care pornește de la ideile din prima secțiune a acestui capitol. Detalierea metodelor și atributelor se va face în *Proiectare.Proiectare clase.*

**

Structuri de date

O structură de date foarte importantă este lista înlănțuită, care este folosită foarte frecvent, atât în generarea taskurilor cât și în comunicarea între thread-uri. Pentru a fi safe din punct de vedere al firelor de execuție, am folosit *CopyOnWriteArrayList<>()*, care e o clasă care se comportă la fel ca *ArrayList<>()* dar care conferă siguranță la scrierea în memorie în diferite thread-uri datorită faptului că în spate se lucrează cu copii de conținut, și nu cu referințe.

O altă structură de date care a fost folosită, este reprezentată de firele de execuție. Prin intermediul clasei *Thread*, am putut lansa în execuție mai multe fire, care vor reprezenta de fapt serverele la care așteaptă taskurile să fie rezolvate.

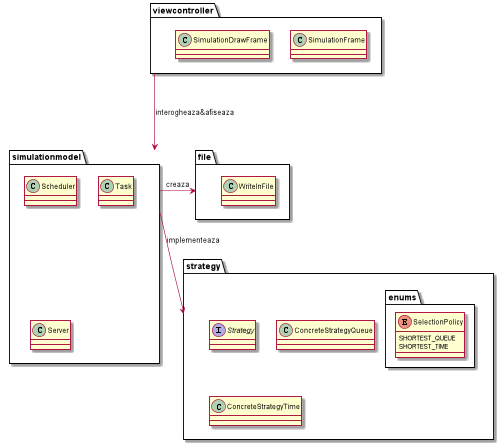
Pentru ca am avut nevoie de siguranța la scrierea în memorie a datelor, comunicarea dintre servere trebuia să fie securizată, de aceea am folosit *AtomicInteger*, care permite folosirea unui intreg în comunicarea între taskuri diferite.

Pentru coada de taskuri ale unui server, care așteaptă spre a fi procesate, am folosit *BlockingQueue<>,* o coadă specială care sincronizează munca cu thread-uri diferite.

Pentru updatarea cozilor din UI am folosit *SwingWorker<,>,* din motivele menționate și mai sus, adică pentru sincronizare threadurilor.

Packages

Organizarea pachetelor a fost insipirată din modelul MVC, așa cum se poate observa în diagrama de mai jos:



## Algoritmi

În continuare descriu principalii algoritmi folosiți în proiect.

**Algoritm pentru determinarea average time waiting**. Acesta a fost calculat după următoarea formulă: ) . Suma timpilor de așteptare a fost făcută în fiecare sever separat, iar interogarea rezultatelor a fost făcută în thread-ul de start, care e lansat din clasa *SimulationManager.* Odată cu terminarea simulării, toate informațiile sunt culese de la fiecare server și se adună pentru a forma media mediilor de așteptare din fiecare server.

**Algoritm pentru determinarea average service time.** Acesta a fost calculat după următoarea formulă: ) . Abordarea este similară cu cea făcută la algoritmul precedent, singura diferență constând în faptul că la calcul se punea la socoteală și timpul de servire efectiv dar și cel de așteptare a taskului la coadă.

**Algoritm pentru determinarea peak hour.** Acest algoritm se poate interpreta ca o funcție de maxim între numărul de taskuri din cadrul serverelor la orice timp din simulare. Complexitatea lui vine de fapt din faptul că trebuie să avem grijă la comunicarea între thread-uri. Practic, comunicarea a fost efectuată între thread-ul care susținea simularea și toate thread-urile care susțineau serverele. Pentru a nu avea probleme de inconsistență a memoriei, am ales să folosesc structuri de date deja sincronizate în locul cuvântului cheie synchronized.

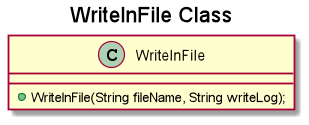
**Algoritm pentru afișarea log-ului sub formă text.** La fiecare pas al simulării, am conturat datele simulării, interogând serverele și am construit un string care în urmă simulării va fi scris în fișierul .log prin intermediul clasei *WriteInFile.*

**Algoritm pentru afișarea în UI.** În partea stângă sunt afișare sub formă text taskurile generate și sunt despărțite de o linie verticală. Cantitatea de ecran rămasă este împărțită la numărul de servere, care sunt reprezentate atât sub formă text cât și sub forma unui dreptunghi. De fiecare dată când un task este la coda unui server, acesta este reprezentat sub forma unui dreptunghi chiar sub serverului de care aparține. El este colorat fie în verde (adică, este procesat), fie într-o culoare de nuanță portocalie (așteaptă să fie procesat).

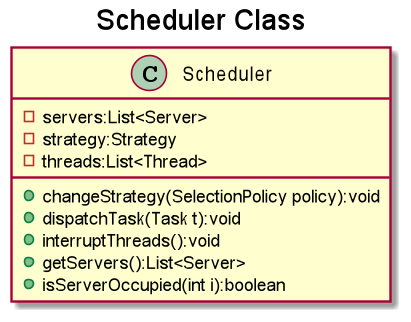
# Implementare

Se prezintă în continuare toate clasele cu o scurtă descriere.

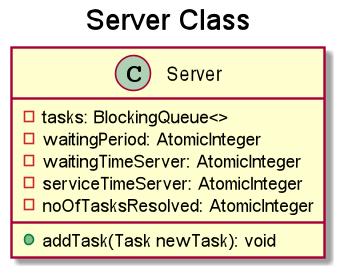
1. **class WriteInFile -** se ocupă cu deschiderea și scrierea în fișierul text. Totul se întâmplă în constructor care primește un string cu ce trebuie să scrie în fișierul text și calea către fișierul text.

****

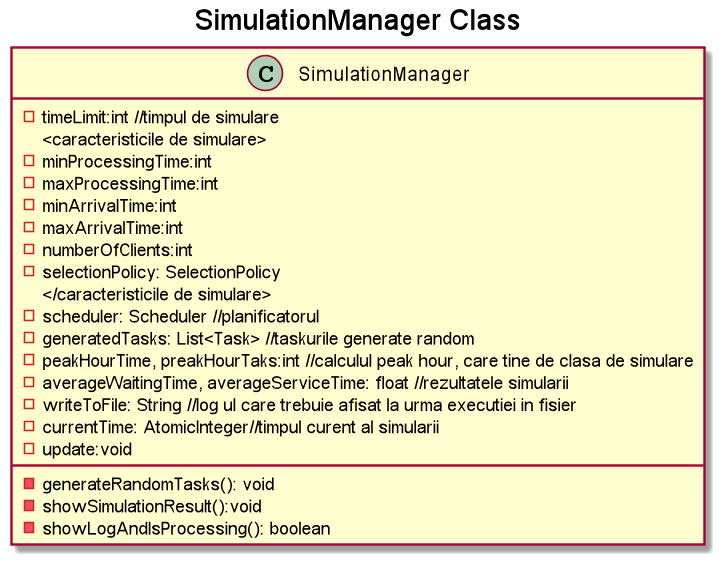
1. **class Scheduler -** această clasă asigură planificarea serverelor dar și împărțirea taskurilor în funcție de strategia aleasă. Metoda *changeStrategy(...)*, schimbă implementarea strategiei, în timp ce metoda *dispatchTask(...)*, adaugă taskul care trebuie procesat către servereul ales după implementarea interfeței *strategy.* Așa cum zice și numele metodei *interruptThreads()***,** aceasta intrerupe execuția serverelor. Iar *isServerOccupied(int i),* întoarce true, dacă serverul numărul i are la coadă măcar un task și false altfel.

****

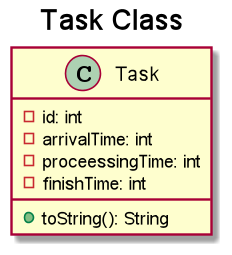
1. **class Server -** clasa care implementează interfața *Runnable* și care are o coda de taskuri și diferți parametrii definiți sugestiv care vor facilita comunicarea între threaduri și care vor ajuta la calcularea rezultatelor de la finalul simulării.

****

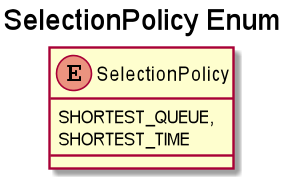
1. **class SimulationManager -** această clasă este cea care se va ocupa cu lansarea simulării și cu comunicarea cu interfața grafică. Pe lângă caracteristicile de simulare și rezultatele simulării, apar și alte varibile instanța care sunt descrise în figura de mai jos. Metoda *generatedRandomTask()* - generează random un set de taskuri în funcție de caracteristicile de simulare. Iar metoda *showSimulationLog()* - afișează rezultatele simulării, în timp ce metoda *showLogAndIsProcessing()* - are dublă responsabilitate, să afișeze evenimentele curente din servere și să întoarcă *true* dacă cel puțin un server are de lucru taskuri, altfel *false.*

****

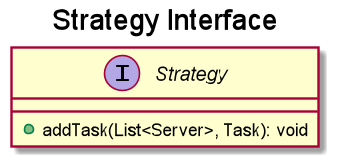
1. **class Task -** această clasă modeleză caracteristicile unui task printr-un id unic, un timp de inceput, unul de procesare și unul de terminare.



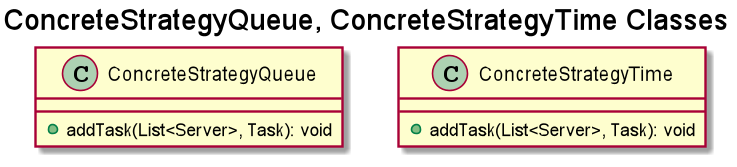
1. **enum SelectionPolicy -** este o enumerație care ajută la alegerea strategiilor de simulare, după cum sunt declarate în figura de mai jos.

****

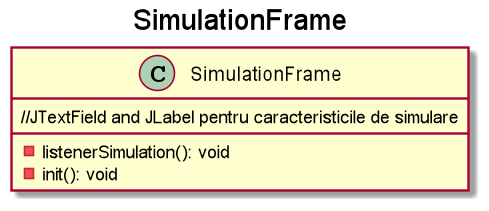
1. **interface Strategy -** reprezinta funcționalitatea pe care orice clasă care modelează strategia va trebui să o implementeze.

****

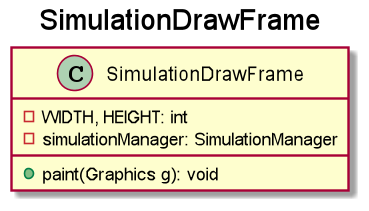
1. **class ConcreteStrategyQueue, ConcreteStrategyTime -** cele două clase care dau două implementări diferite pentru funcționalitatea de strategie definită mai sus.



1. **class SimulationFrame -** această clasă va intercepta de la utilizator caracteristicile de simulare și va lansa în execuție simularea.

****

1. **class SimulationDrawFrame -**clasă care afișează evoluția serverelor în timp real. Metoda *paint(Gpraphics g)* afișează sub forma grafică (dreptunghiuri și text), ce se petrece în clasa *SimulationManager.*

****

# Concluzii

Voi prezenta câteva din lucrurile pe care le-am câștigat prin intermediul temei:

* să lucrez cu firele de execuție (thread-uri);
* am aprofundat modelul MVC;
* să lucrez cu PlantUML editor, întrucât toate diagramele din documentație au fost procesate prin intermediul unui limbaj, prezent în anexele atașate cu documentația;
* am aprofundat colecții precum *BlockingQueue<>, AtomicInteger, CopyOnWriteArrayList<>, SwingWorker<,>*;
* am aprofundat modurile de comunicare interthread-uri;
* am învățat tehnici de a scrie cod cât mai condensat și thread safety;

Dezvoltări ulterioare:

* Posibilitatea de a afișa separat sub forma de text evoluția serverelor în timp;
* Implementarea unor noi strategii de distribuire a taskurilor, precum exemplu cu urgențele medicale;

# Biliografie

1. <https://www.baeldung.com/java-atomic-variables>
2. <https://plantuml-editor.kkeisuke.com/>
3. [Lesson: Concurrency (The Java™ Tutorials > Essential Classes) (oracle.com)](https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html)
4. [java.util.Timer.schedule() Method - Tutorialspoint](https://www.tutorialspoint.com/java/util/timer_schedule_period.htm)
5. <https://www.baeldung.com/java-blocking-queue>
6. <https://www.baeldung.com/java-copy-on-write-arraylist>
7. <https://www.javacodegeeks.com/2012/12/multi-threading-in-java-swing-with-swingworker.html>