

**QUEUES SIMULATOR**

Student: Zilai Denis

Grupa: 30238

**CUPRINS**

1. Obiectivul temei
2. Analiza problemei, modelare, scenarii, cazuri de utilizare
3. Proiectare
4. Implementare
5. Rezultate
6. Concluzii
7. Bibliografie

**1.Obiectivul temei**

Obiectivul temei curente este acela de a proiecta si implementa o aplicatie de cozi, care simuleaza desfasurarea actiunii dintr-un magazin, cu scopul de a realiza impartirea clientilor la cozile de asteptare pentru a fi serviti. Pentru aceasta aplicatie s-au utilizat 11 clase si o interfata pentru a se realiza cerinta acestei teme.

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

Cerinta acestei aplicatii este de a simula comportamentul unor clienti dintr-un supermarket, alaturi de descrierea strategiei de selectionare a cozii de asteptare.

Pentru a indeplini acest obiect s-a descris interfata Strategy care implementeaza metoda addTask, 2 clase care descriu strategiile de actiune ale clientilor, o clasa care implementeaza o strategie in care clientii aleg coada cu cei mai putini oameni in asteptare, iar o alta in care clientii aleg coada care are timpul cel mai mic de asteptare. O alta clasa necesara pentru realizarea acestui proiect este clasa Task care simuleaza clientii si care incorporeaza timpul de sosire, procesare si ID-ul clientilor.

De asemenea clasa Server reprezinta coada propriu zisa la care se va sta in asteptare si este una dintre clasele a caror obiecte ii sunt asociate threaduri pentru a rula concurent.

O alta clasa care are un thread asociat si ruleaza concurent este clasa SimulationManager care este centrul acestei aplicatii .Aceasta clasa implementeaza timer-ul principal al aplicatiei in metoda ei de run si face impartirea clientilor afisarea in GUI, calculul timpilor de analiza si multe altele printre care instantiaza si un obiect de tip Scheduler. Clasa Scheduler se ocupa de stocarea unui array de tip Servere si implementeaza metoda addTask care apeleaza obiectul strategy ca sa adauge clientii in cozi prin intermediul strategiilor implementate anterior.

De asemenea avem si 2 clase de interfata,care sunt responsabile cu introducerea datelor experimentului si afisarea in timp real a rezultatului.

Clasele View, View2 si Controller implementeaza partea tehnica de start a simularii in care gestionarea interactiunea cu primul GUI in care utilizatorul introduce datele si valideaza datele introduse.

Exemplu use case:

1. Utilizatorul introduce datele necesare (numar de cozi, numar de clienti, timp de simulare, timp de sosire maxim, timp de sosire minim, timp de procesare maxim si timp de procesare minim);
2. Clientii intra in cozi;
3. Dupa timpul de procesare respective fiecarui client, acesta iese din coada;
4. Se calculeaza timpul de procesare mediu;
5. Se calculeaza ora de varf;
6. Se calculeaza timpul de asteptare;

Diagrama Use-Case:

Diagram

Description automatically generated

**Proiectare**

In scopul proiectarii OOP a acestui proiect s-au folosit threadurile in Java si s-a preferat implementarea interfetei Runnable in detrimentul extinderii clasei Thread deoarece oferea o mai mare libertate de exprimare.

Multithreading în Java se refera la un program care executa simultan mai multe threaduri. Ele partajeaza o zona de memorie comuna. Multithreading in Java se refera la un program care executa simultan mai multe threaduri. Fiecare fir de lucru isi incepe executia la o locatie bine cunoscuta, predefinita.

Fiecare fir de lucru isi executa codul începand de la locatia de start, intr-o secventa ordonata, predefinita (pentru un set de date de intrare dat).

Fiecare fir de lucru isi executa codul independent de celelalte fire de lucru din program. Firele de lucru par a avea un anumit grad de simultaneitate în execuţie. Firele de lucru au acces la diferite tipuri de date. Un fir de lucru (thread) in Java incepe prin crearea unei instanțe a clasei java.lang.Thread.

Metodele din clasa Thread pentru manipularea firelor de executie sunt, de exemplu:

•start()

•yield()

•sleep()

•run()

Actiunea firului de executie incepe la invocarea metodei run(). La apelul metodei run() se creaza o noua stiva de apel pentru threadul executat. In Java, fiecare thread are propria stiva de apeluri.

Definirea si instantierea unui thread poate fi facuta in unul din cele doua feluri:

• Extinderea clasei java.lang.Thread

• Implementarea interfetei Runnable

Singurul motiv pentru care are sens sa se extinda clasa Thread este cazul cand se doreste realizarea unei versiuni mai specializate a clasei Thread.

In restul cazurilor (majoritatea cazurilor) cand se doreste doar sa se specifice ce anume trebuie sa execute threadul, se defineste o clasa care implementeaza interfata Runnable.

Limitarea extinderii clasei Thread o reprezinta faptul ca clasa care a extins clasa Thread nu va mai putea extinde alta deoarece in Java nu este posibila mostenirea multipla.

Implementarea interfetei Runnable ofera flexibilitatea de a extinde orice alta clasa, pastrandu-si proprietatea de a putea fi executata într-un fir de lucru separat.

Un obiect de extinde interfata Runnable este trimis ca argument unui obiect de tip Thread nou creat pentru ca Threadul sa stie a cui metoda run o va executa cand este pornit.

Structurile de date folosite au fost in principal array urile generale si concurente implementate de Java cum ar fi ArrayList ArrayBlockinQueue etc.

Interfata implementata este Strategy,ea este responsabila cu metoda addTask.

In interiorul unui thread lucrurile se petrec intr-o ordine predictibila, dar actiunile threadurilor multiple pot fi amestecate intr-o ordine neprevazuta.

Daca acelasi cod se ruleaza de mai multe ori, sau pe masini diferite, rezultatul poate sa fie diferit.

Diagrama UML de pachete:

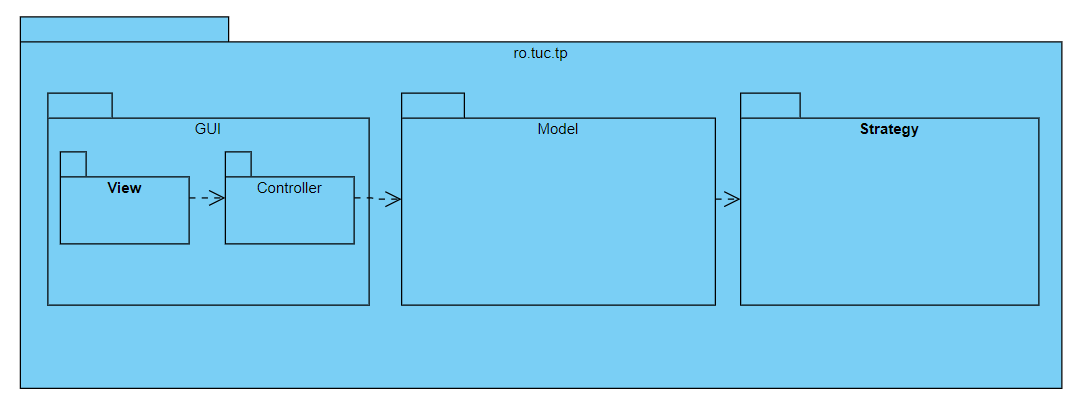
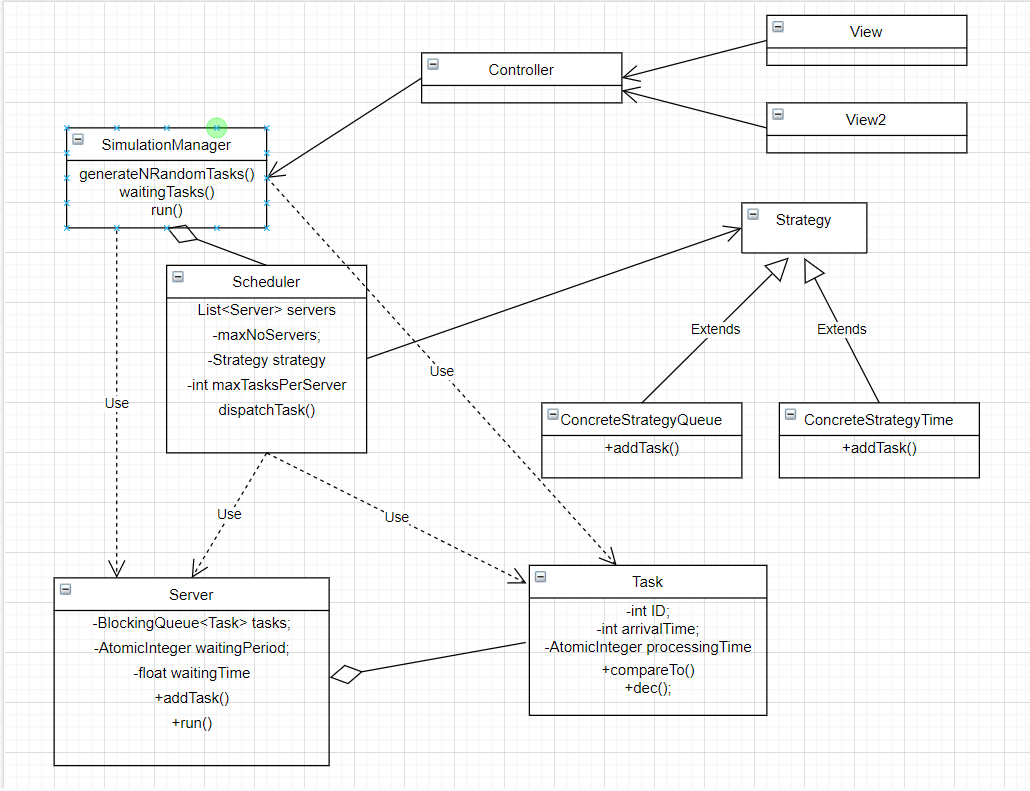


Diagrama UML de clase:



**Implementare**

Pentru acest proiect s-au implementat 10 clase, o enumerare si o interfata.

La acest proiect am implementat 12 clase, o enumerare si o interfata.

1. Interfata Strategy

Definiţie: o interfaţă este o colecţie de constante și definiţii

de metode, colecţie care are un nume

 O interfaţă nu este o clasă, ci un set de cerinţe pentru

clasele care doresc să se conformeze interfeţei

 Interfeţele sunt folositoare pentru

 Reţinerea asemănărilor între clase ne-înrudite fără a forţa o relaţie

de clasă

 Declararea de metode pe care una sau mai multe clase ar trebui să

le implementeze

 Dezvăluirea interfeţei de programare a unui obiect fără a-i dezvălui

clasa

 Modelarea moştenirii multiple, care permite ca o clasă să aibă mai

mult de o superclasă

Definirea unei interfeţe are două componente:

declaraţia interfeţei şi corpul interfeţei

 Declaraţia interfeţei defineşte diferitele atribute ale

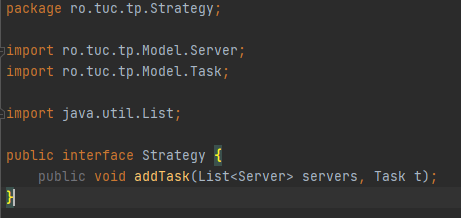
interfeţei, cum sunt numele şi dacă ea extinde alte

interfeţe

 Corpul interfeţei conţine declaraţiile de constante şi de

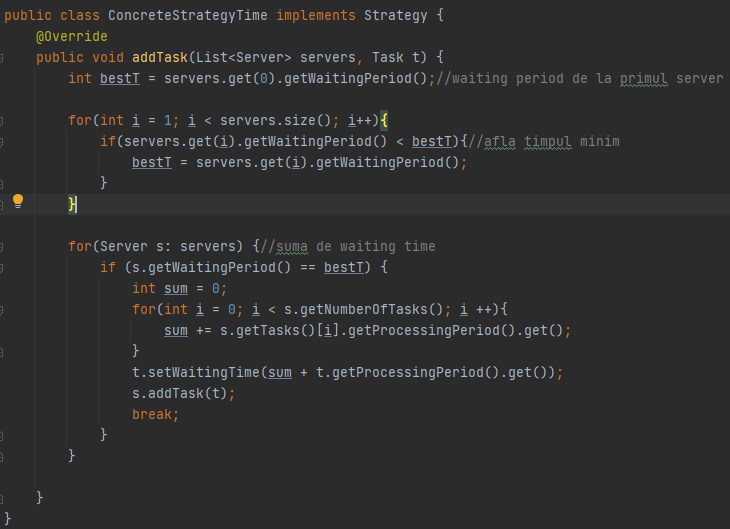
metode pentru interfaţa respectivă

Aceasta interfata este implementata de urmatoarele doua clase: ConcreteStrategyTime si ConcreteStrategyQueue. Contine o singura metoda numita addTask ().



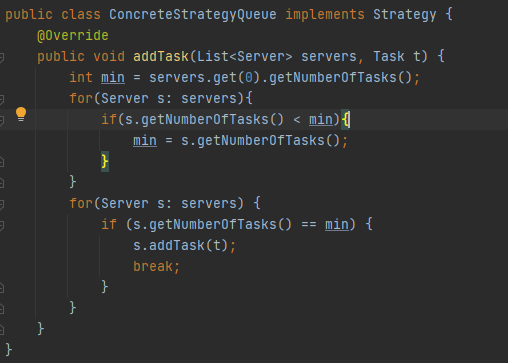
1. Clasa ConcreteStrategyTime

Aceasta clasa implementeaza interfata Strategy, deci in mod clar si metoda addTask() care adauga clienti la cea mai scurta coada.



1. Clasa ConcreteStrategyQueue

Aceasta clasa implementeaza interfata Strategy, deci in mod clar si metoda addTask() care adauga clienti la coada cu cel mai scurt timp de asteptare.



1. Enumerarea SelectionPolicy

O enumerare, sau “enum,” este pur şi simplu un set de constante pentru reprezentarea diferitelor valori.

O enumerare conține un set de constante întregi

reprezentate prin identificatori

• Permite folosirea unor nume sugestive pentru valori

numerice

• Constantele sunt asemănătoare constantelor simbolice și

au valori setate automat

• Valorile încep implicit de la 0 și sunt incrementate cu 1

• Se pot seta valori explicite prin asignare cu operatorul =

• Numele constantelor trebuie să fie unice

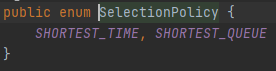
• Variabilele de tip enumerare își pot asuma doar una din

valorile constante din set

• Nu se poate garanta că reprezentarea pe tipul întreg a unei

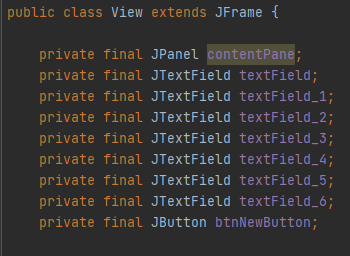
variabile de tipul enumerare poate fi folosită pentru a stoca alt

întreg



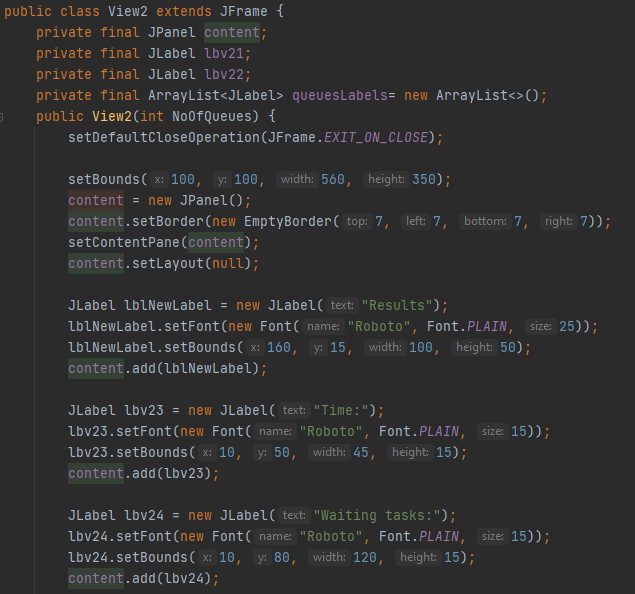
1. Clasa View este clasa care se ocupa de crearea si design-ul primei

interfete grafice. Aceasta este formata din 9 atribute: contentPane de tip JPanel, textField de tip JTextField, textField\_1 de tip JTextField, textField\_2 de tip JTextField, textField\_3 de tip JTextField, textField\_4 de tip JTextField, textField\_5 de tip JTextField, textField\_6 de tip JTextField, btnNewButton de tip JTextField si 8 metode (getteri si setteri): getStartButton() care returneaza un JButton, getNumarCozi() care returneaza un String, getNumarClienti() care returneaza un String, getTimpSimulare() care returneaza un String,getTimpProcesareMaxim() care returneaza un String, getTimpProcesareMinim() care returneaza un String, getTimpSosireMaxim() care returneaza un String, getTimpSosireMinim() care returneaza un String.



1. Clasa View2 este clasa care se ocupa de crearea si design-ul cele

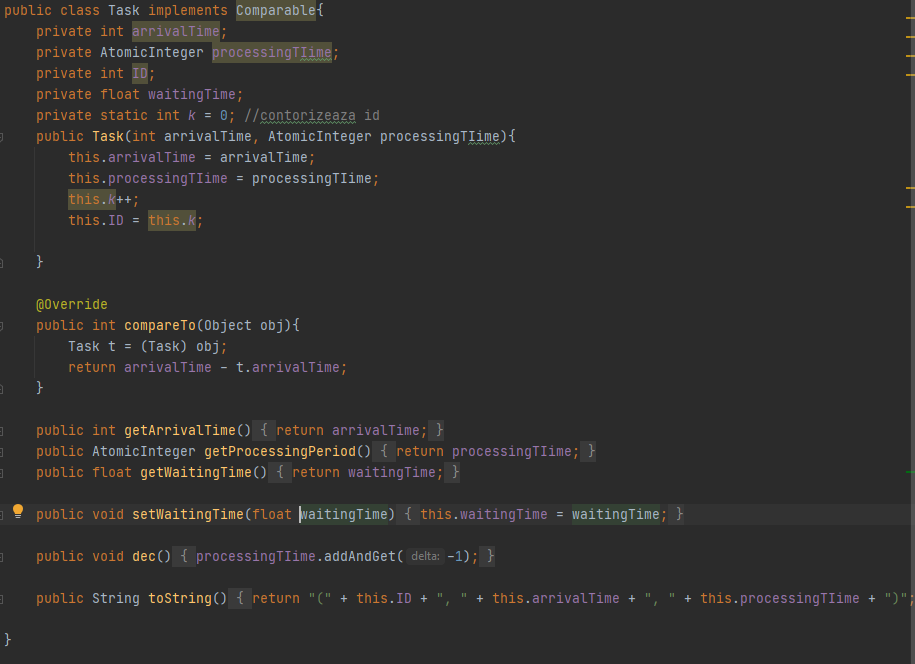
de-a doua interfete grafice. Aceasta este formata din 7 atribute: contentPane de tip JPanel, lblNewLabel\_3\_1 de tip JLabel, lblNewLabel\_3 de tip JLabel, queuesLabels de tip ArrayList<JLabel>, avg de tip JLabel, peekTime de tip JLabel si waitingTime de tip JLabel si 5 metode: setTimpSimulare este de tip void, primeste un parametru de tip String si seteaza textul etichetei lblNewLabel\_3 in Stringul primit, setClientiAsteptare este de tip void, primeste un parametru de tip String si seteaza textul etichetei lblNewLabel\_3\_1 in Stringul primit, getQueuesLabels este de tip ArrayList<JLabel>, nu primeste niciun parametru si returneaza lista de etichete pentru cozi, setClienti este de tip void si primeste doi parametri: unul de tip String si unul de tip int. Aceasta seteaza textul etichetei care contine clientii din fiecare coada, setTime care este de tip void si primeste ca parametrii de tip String.



1. Clasa Task

Aceasta este clasa care descrie fiecare client. Este formata din 5 atribute: arrivalTime de tip int, processingPeriod de tip int, ID de tip int, waitingTime de tip float, contor de tip int si 8 metode: setArrivalTime de tip void si care primeste un parametru de tip int (aceasta seteaza arriveTime in valoarea primita ca parametru), getArrivalTime de tip int (aceasta returneaza arriveTime), getProcessingPeriod de tip int (aceasta returneaza processingPeriod), setProcessingPeriod tip void (aceasta seteaza processingPeriod in valoarea primita ca parametru), decrement

de tip void (aceasta scade din processingPeriod 1), toString de tip String (aceasta returneaza clientii sub forma de string: (ID, arrivalTime, processingPeriod)), getWaitingTime de tip float (aceasta returneaza waiting time) si setWaitingTime de tip void (aceasta seteaza waitingTime in valoarea primita ca parametru).



1. Clasa Scheduler

Aceasta clasa contine 4 atribute: servers de tip List<Server>, maxNoServers de tip int, maxTasksPerServer de tip int si strategy de tip Strategy si 5 metode: changeStrategy este de tip void si stabileste pe care dintre cele doua strategii sa o folosim: pe cea in care avem coada mai goala sau pe cea in care avem timpul mai scurt, dispatchTask de tip void care primeste un parametru de tip Task si adauga un client in coada potrivita, getServers de tip ArrayList<Server> care returneaza arraylistul de cozi, getStrategy de tip Strategy care returneaza strategy si empty de tip boolean care returneaza false daca nici macar una din cozi nu este goala si true in caz contar. In constructorul acestei clase cream cate un thread pentru fiecare coada. Folosim thread-uri acolo unde trebuie să se întâmple mai multe lucruri simultan. Definirea unui thread prin implementarea interfeței java.lang.Runnable este o modalitate extrem de utilă atunci când clasa de tip Thread care se doreşte a fi implementată moşteneşte o altă clasă (Java nu permite moştenirea multiplă). Interfaţa Runnable descrie o singură metodă run().

Se urmează etapele:

- se creează o clasă care implementează interfaţa Runnable

- se implemetează metoda run() din interfaţă

- se instanţiază un obiect al clasei folosind new

- se creează un obiect din clasa Thread folosind un constructor care are ca parametru un obiect de tip Runnable (un obiect al clasei ce implementează interfaţa)

- se porneşte thread-ul creat la pasul anterior prin apelul metodei start()

AtomicInteger si BlockingQueue sunt tipuri de date pe care le putem folosi in mediul concurent, fara a mai folosi Synchronized (). AtomicInteger are urmatoarele metode:

* addAndGet()- Adăugă atomic valoarea dată la valoarea curentă și returnează nouă valoare *după* adăugare.
* getAndAdd() - Adăugă atomic valoarea dată la valoarea curentă și returnează valoarea veche.
* incrementAndGet()- Crește atomic valoarea curentă cu 1 și returnează o nouă valoare *după* increment. Este echivalent cu operația **++ i** .
* getAndIncrement()- Incrementează atomic valoarea curentă și returnează valoarea veche. Este echivalent cu operația **i ++** .
* decrementAndGet()- Scade atomic valoarea curentă cu 1 și returnează o nouă valoare *după* scădere. Este echivalent cu operația **i-** .
* getAndDecrement()- Scade atomic valoarea curentă și returnează valoarea veche. Este echivalent cu operația **- -i** .

BlockingQueue are urmatoarele metode:

* peek() returneaza primul element al BlockingQueue fără a-l elimina. Dacă BlockingQueue nu conține elemente, peek() va returna null.

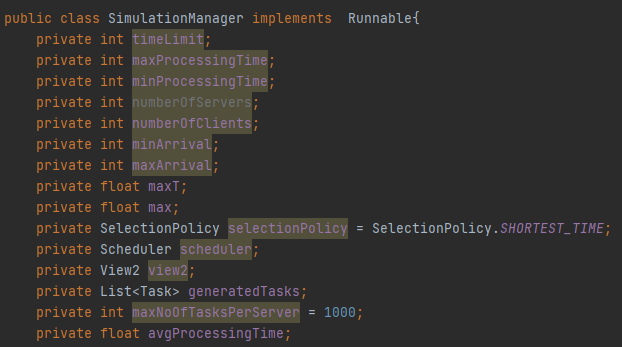
• poll() elimina primul element din BlockingQueue. Dacă BlockingQueue nu conține elemente, poll() va returna null.

Aceasta clasa are 3 atribute: tasks de tip BlockingQueue<Task>, waitingPeriod de tip AtomicInteger si sumWaitingTime de tip float si 6 metode: addTask() care adauga un client si creste waitingPeriod cozii cu processingPeriod ul clientului respectiv, run() care seteaza waitingPeriod ul fiecarei cozi, getTasks() de tip Task[] care returneaza un vector cu cozile, getNumberOfTasks() de tip int care returneaza numarul de client dintr-o coada, getWaitingPeriod() de tip int care returneaza waitingPeriod si getSumWaitingTime de tip float care returneaza sumWaitingTime.

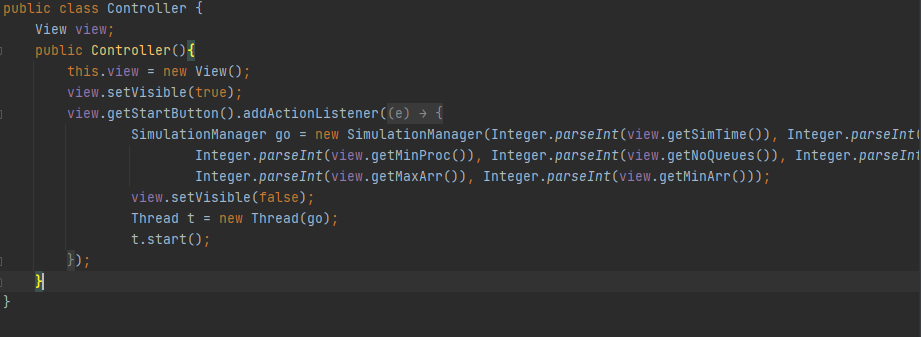


1. Clasa SimulationManager care de asemenea implementeaza interfata Runnable, respectiv metoda run().

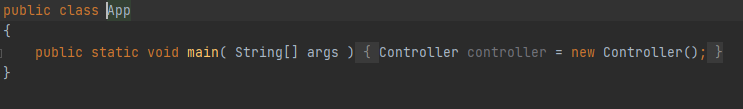
Aceasta clasa are 17 atribute: private int timeLimit, maxProcessingTime, minProcessingTime, numberOfServers, numberOfClients, minArrival, maxArrival, peekTime, waitingTime, max, output, selectionPolicy, scheduler, view2, generatedTasks, maxNoOfTasksPerServer, avgProcessingTime si 3 metode: generateNRandomTasks de tip void care genereaza numberOfClients clienti pe care ii initializeaza cu valori random (timpul de sosire de la maxArrival pana la minArrival si timpul de procesare de la minProcessingTime pana la maxProcessingTime) si ordoneaza clientii in ordine crescatoare in functie de timpul de sosire, run() care pune clientii in coada si ii scoate la momentul de timp potrivit si calculateAvg().



1. Clasa Controller care controleaza actiunea butonului “Start” de la prima interfata.



1. Clasa App



**Rezultate**

Rezultatele au fost cele asteptate, neintalnindu-se erori sau greseli in gestionarea cozilor, diverse teste asigurand functionalitatea proiectului.

**Concluzii**

Acest proiect m-a ajutat sa inteleg mai bine si sa aprofundez conceptele de OO psi Threads, reusind sa imi dezvolt capacitatile pe acest limbaj de programare Java.

**Bibliografie**

Cursuri si laboratoare prof. Ion Giosan

GeeksForGeeks

StackOverflow

Wikipedia