Tema 2

Simulator de Queues-Servere (cu Clienti - Taskuri)

UTCN, Orsan Tudor, Grupa 30225

1. Obiectivul temei:

Obiectivul principal al acestei teme este de a realiza servere pe care sa introducem taskuri. Aceste servere sunt threadurile ce lucreaza in paralel, care primesc taskuri la momente de timp stabilite. Importan este faptul ca ele lucreaza in paralel, deci pot sa primeasca taskuri diferite la acelasi moment de timp.

Obiective principale : Random client generator, Multi-Thread approach, 1 thread per queue, sa folosim structuri de date potrivite pentru sincronizarea datelor, sa facem un log intr-un fisier text, in care sa scriem informatii despre clienti;

Alte obiective ce pot fi considerate principale, insa le voi considera secundare: Interfata, prin care sa se observe evolutia in timp a serverelor, adica adaugarea de taskuri la servere la fiecare timp, sa afisam average waiting time, service time si peak hour la finalul fisierului text sau pe interfata (am ales in fisierul text), sa rulam cu unele date de intrare date si sa punem acele loguri (cu cele 3 teste) in documentatie.

1. Analiza problemei / cazurile de utilizare:

Acest mecanism de gestionare a threadurilor are anumite cazuri de utilizare, si aceste cazuri pot fi descrise cu ajutorul use-caseurilor.

Cazurile de functionare ale programului sunt acestea:

- In primul rand, ar trebui sa poata fi inserate datele de intrare, adica numarul de servere, numarul de clienti, numarul minim de arrival time, numarul maxim de arrival time, numarul minim de processing time, numarul maxim de processing time si numarul maxim de timp al executarii (daca se ajunge pana acolo);

- Dupa introducerea tuturor acestor date de intrare, putem apasa pe butonul de „Simulate”, si asa ne ducem pe un alt frame (view), unde putem vedea momentele de executie ale gestionarii de threaduri;

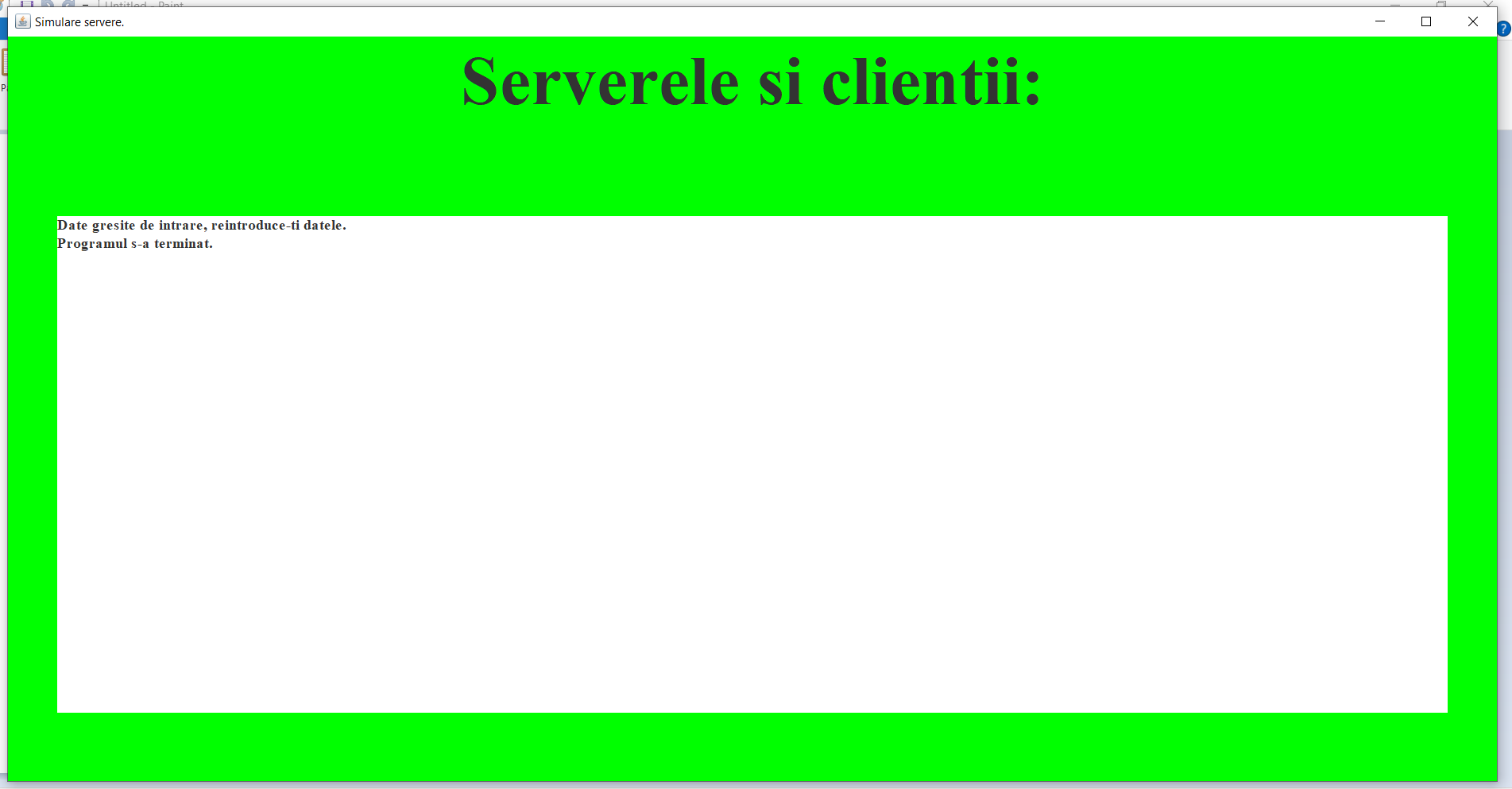
-Nu am un buton de back pentru a trimite utilizatorul inapoi la inceput, sa se poata introduce alte date de intrare, dupa ce se termina simularea, trebuie inchisa fereastra si introduse alte date de intrare din nou daca se doreste acest lucru;

-Principalul scop a fost sa evidentiez momentele de timp ale simularii (la fiecare secunda a simularii)

Cerintele non-functionale ale programului sunt:

-Ar trebui sa fie usor de folosit de catre utilizator, sa fie clar unde trebuie sa introduca date, cum functioneaza si unde se afiseaza rezultatul.

-In caz de erori, sa fie evidentiat faptul ca avem acea eroare, si sa cunoasca utilizatorul faptul ca trebuie sa schimbe valorile de intrare, timpii introdusi (a intrarilor adica).



-Sa poata fi reutilizabil, daca se doresc alte date de intrare, sa se poata face accesibila aceasta optiune cat mai usor.

Am un singur use case, voi evidentia flow-chartul acestui use case pe urmatoarele randuri:

Generare servere + clienti:

Primary actor : User;

Scenariul de succes principal:

* Se insereaza cele 7 text fielduri ca date de intrare in mod corect (se introduc date int), dupa care se asteapta simularea treptata pe momente de timp, pana la finalizarea timpului de rulare.
* Pe langa interfata, se genereaza si fisierul text in folderul programului, unde se poate observa executia detaliata a acelui scenariu, cu acele date de intrare;

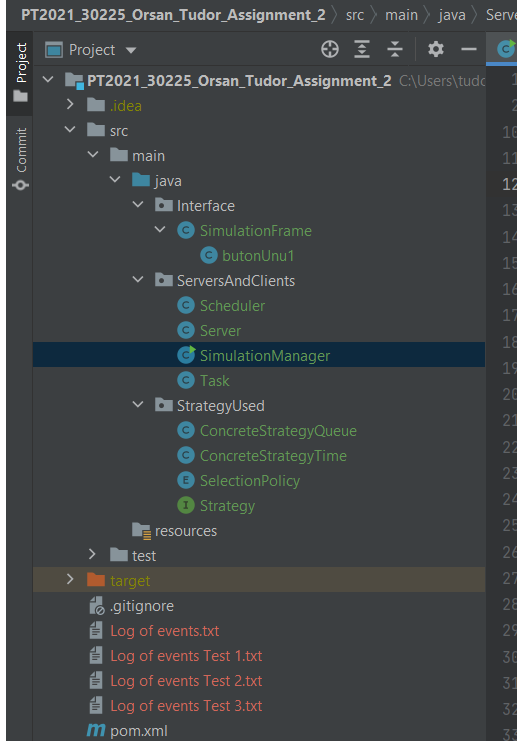
Scena alternativa de functionare:

* Daca se introduc date gresite de intrare, cum ar fi litere, etc... ce nu pot fi convertite in int. Pentru acesetea, exista un mesaj de eroare pe cea de a 2-a interfata, si se doreste reintroducerea altor date de intrare.
* O alta observatie importanta este faptul ca pentru ca programul sa functioneze, este necesar sa nu se lase spatiile goale, cele 7 de intrare, caci altfel nu va exista mesaj de eroare, ci doar un text gol, pentru ca se asteptau alte date de intrare.

1. Proiectarea problemei:

Pentru proiectarea problemei, am sa intru in mai multe aspecte:

Primul aspect: Pachete



Avem nivele diferite de functionare, si anume: pentru interfata, am o clasa (Nu am facut cu metoda MVC), unde creez interfata, dupa care o dau ca parametru functiei principale, SimulationManager. In „ServersAndClients”, am partea de logica axata pe legatura intre servere, clienti, si cum actioneaza threadurile intre ele. Cu ajutorul lui scheduler si SimulationManager (clase preluate din materialul ajutator), aranjez taskurile in functie de servere dupa diferite metode, evidentiate mai jos.

In „StrategyUsed”, ma leg de strategia folosita pentru a imparti acele taskuri pe servere, deci ma leg de una din 2 strategii: Shortest Time si Shortest Queue. Cu interfata strategy si enumerarea SelectionPolicy, aleg una din aceste 2 strategii si o aplic pe toate impartirile de taskuri in servere.

Acestea au fost cele 3 pachete folosite.

Al doilea aspect: Fieldurile (Campurile)

Fieldurile sunt private, in majoritatea claselor, pentru a putea exista o incapsulare mai puternica a codului. Pentru toate fieldurile importante si folosite si altundeva decat in clasa proprie, am creat setteri si getteri.

Al treilea aspect: Interfata Grafica:

Pentru interfata grafica, nu am ales un model anume precum MVC,

ci am folosit doar un singur fisier in care am o clasa ce contine atat partea de view, cat si de controler si de model. M-am axat la acest proiect pe afisarea clientilor la servere, nu pe o aranjare eficienta a interfetei, dar tot a iesit precum doream. In special, am folosit „TextArea”, unde pot sa folosesc spatiu de afisare ca un fisier text, si astfel pot sa scriu aproape la fel de usor ca intr-un fisier text. Puteam alege o aordare mai interesanta, dar am ales simplitate, pentru a putea urmari partea mai importanta a temei.

Al patrulea aspect: Threadurile:

Ce este inedit la aceasta tema este folosirea threadurilor, adica implementarea interfetei Runnable. Cu acestea, am putut sa execut „mini-programe” in paralel, si anume serverele, care primesc acele taskuri, clienti. Cu implementarea runnable, pot sa dau start la threaduri in paralel, si asa ele, daca sunt sincronizate, merg una langa cealalta dupa dorintele programatorului. Cu sleep am reusit sa le sincronizez, nu am folosit alte metode precum sincronyze sau wait, ci a fost destul sa folosesc structurile cu Blocking si AtomicInteger pentru a sincroniza serverele intre ele. Mai multe voi vorbi mai jos, despre implementare.

Al cincilea aspect: Relatiile intre clase: (Nu diagrama UML, ci relatiile intre clase)

Exista relatii multe intre clase, pentru ca sunt foarte interconectate. Avem clasa Task, ce are o relatie de dependenta cu clasa Servers, deoarece sunt adaugate taskuri la lista de taskuri din servere. Dupa, in mod asemanator, serverele sunt adaugate la lista de servere din clasa Scheduler, iar in final, avem relatii diferite in SimulationManager. Aici creeam atat schedulerul, adica il instantiem cu „new”, cuvant cheie, cat si serverele, la care dam start. Nu doar atat, ci pana si taskurile le generam la intamplare, deci relatia aici este de compozitie. Clasele sunt facute in asa fel incat sa comunice cat mai usor una cu alta, si de asta avem relatii atat de interconectate intre ele.

1. Implementarea problemei:

Voi intra in detalii vorbind despre fiecare clasa in parte:

Clasa Task:

-Este clasa generica, folosita doar pentru a genera niste obiecte de pus in servere. In ea avem fieldurile de timp de arrival, timp de procesare si id-ul fiecarui client (pentru a ii diferentia), generate random in alta clasa. Pe langa setteri si getteri, avem implementare cu comparable pentru a putea sorta in functie de arrival time. Acel arrival time este timpul la care un task poate fi introdus sau nu in server. Timpul de procesare este cat de mult va sta sa fie procesat atunci cand este in varful cozii.

Clasa Server:

-Aceasta clasa este foarte importanta, ea reprezinta threadul programului (nu cel principal, ci cel pe care il gestionam). Avem lista ArrayBlockingQueue de taskuri, si waitingPeriod pentru fiecare server. Lista este de acest tip pentru a sincroniza cu usurinta serverele intre ele (fara folosire de lockuri, sincronize, etc...), iar AtomicInteger este folosit pentru a ne asigura ca datele nu se suprascriu de alte threaduri la momente nepotrivite. Pe langa constructor si getteri, avem o functie de adaugat task in lista de taskuri si avem functia necesara de implementat atunci cand folosim Runnable, si anume run: Cu aceasta metoda rulam threadurile, adica ele vor fi in stare activa, in viata, in timp ce ele fac run. Avem conditie de while(true), deoarece voi opri threadurile doar la sfarsitul threadului principal din SimulationManager, pana atunci ele putand sa primeasca oricand date noi de procesare (la fiecare secunda). Ce fac in acest run este sa dau sleep pentru 1 secunda, dupa care sa actualizez waitingPeriod la acel server, dupa care sa verific daca trebuie sau nu scos taskul ce si-a terminat timpul de serviciu, de procesare. Toate aceste procesari au loc in paralel, thread cu thread, si sunt gestionate in alta clasa, de un thread principal, care gestioneaza in ce mod sa fie impartite taskurile pe servere.

Clasa Scheduler:

-Aceasta clasa are un rol important, de stocare a serverelor. Aceasta clasa stie cat de multe servere avem si cat putem stoca (ca nr de clienti) pe fiecare server. Pe langa, mai avem si stabilirea strategiei cu care vom lucra. In constructor, initializam lista de servere, dar nu aici pornim threadurile (cu start). Aici putem alege strategia, in functie de ce vom vedea mai jos, si pe langa putem da dispatch la task, impreuna cu lista tuturor serverelor, iar din strategie vom sti exact unde este pus taskul, in acea lista de servere.

Clasa SimulationManager:

-Aceasta este clasa unde se intampla majoritatea operatiilor, adica aici creem multe din obiecte si le folosim. Incepem cu fieldurile: am cele 7 intrari, cu nr de servere, clienti, min/max la timpi, si timpul de rulare, initializati cu 0. Dupa am un field cu intrefata, un field cu SelectionPolicy, un field pentru citirea in fisier, si niste fileduri pentru salvarea rezultatelor pentru average service si waiting time. Am lista de threaduri pornite, pentru a avea acces la ele in mod rapid. Pe langa, mai am acel scheduler despre care am vorbit mai sus, care va fi creeat aici, si lista de taskuri generate random. Toate acestea au roluri bine stabilite, si aceste roluri vor fi reflectate in cele ce urmeaza sa explic: In constructor, insantiez frameul in care lucrez, si fac astfel incat sa astepte intrarile, dupa care sa continue logica (pe intrati va pune ce gaseste pe interfata, si cu aceste intrari vom lucra). Aici instantiez de asemenea si schedulerul, lista de threaduri, si incep toate serverele, cu metoda start. (adaugandu-le totodata in lista de threaduri). Tot aici aleg si strategia, si in functie de acea strategie urmeaza sa le dau deploy in curand, nu aici. Aici generez si toate N taskurile random, folosind o metoda ce urmeaza in aceasta clasa. Nu voi intra in prea multe detalii despre cum functioneaza functiile de calcul la average servire, waiting time si peak hour, important este ca la service time calculez cat proceseaza fiecare client, si iau si cazurile in care sunt clienti ce nu au intrat in servere sau care nu au fost terminati de procesat la finalizarea programului. La fel si la waiting time, dar aici calculez cat se asteapta atunci cand se intra in coada, deci cat timp mai trebuie sa treaca pentru a incepe sa fie procesat, el fiind acum in capul cozii. La peak hour este cel mai simplu, doar trebuie sa testezi de fiecare data, dupa fiecare secunda, sau la cat dai sleep, daca este un nou numar max de taskuri la toate threadurile, si daca este il actualizam si afisam la finalul executiei.

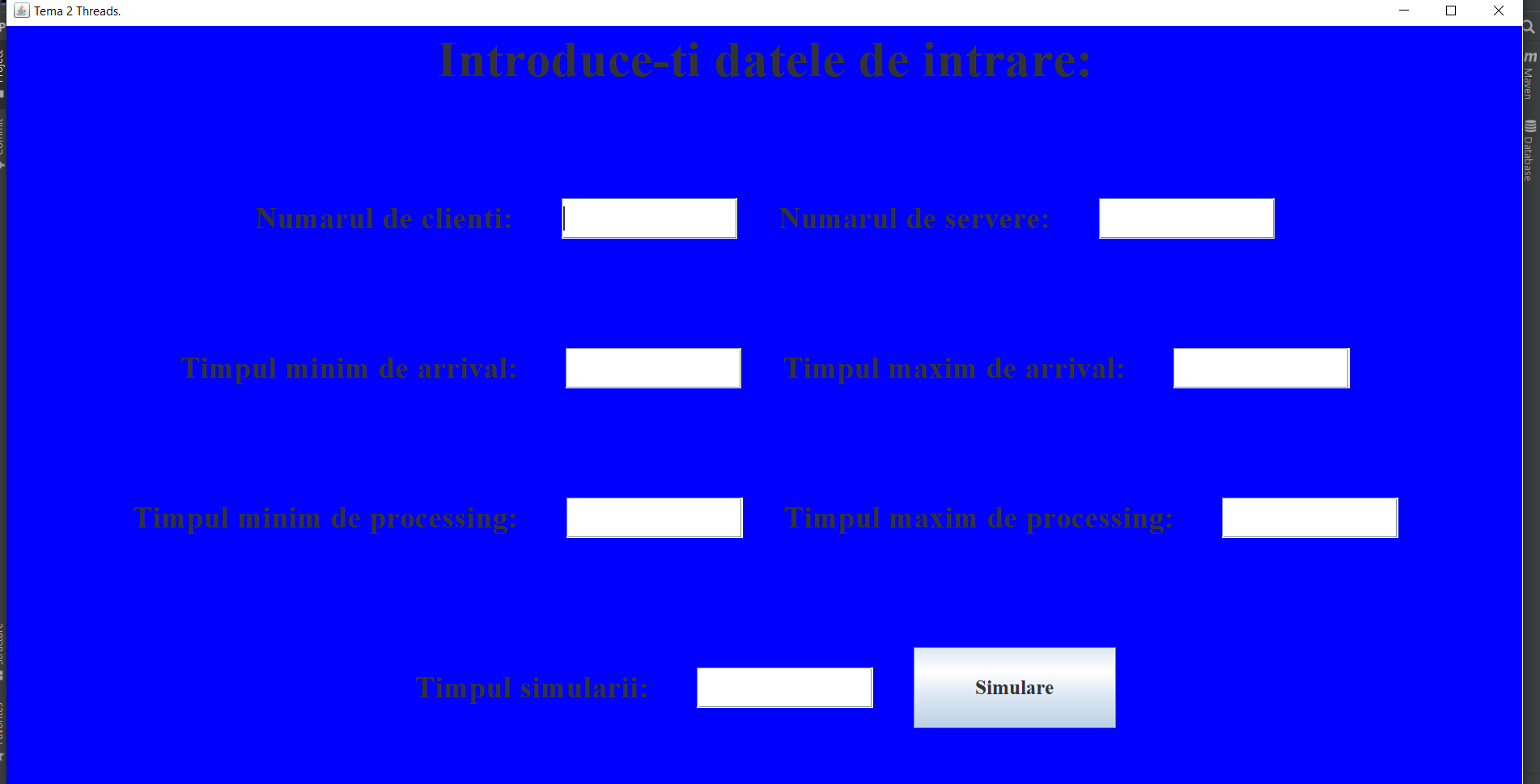
-Pentru generarea celor „N” taskuri, folosesc clasa rand, ce face logica sa functioneze pentru cazurile in care min<=max in ambele timpuri de la intrare, si sa dea eroare (exceptie) daca min>max. Dupa ce genereaza acesti timpi, impreuna cu id-ul, se introduce fiecare client in lista, si se lucreaza cu aceasta lista atunci cand introducem noi clienti in servere. Pe langa metoda run pentru threadul principal, mai avem si metodele de afisare a clientilor din lista generata random, cu toate caracteristicile lor, in ordinea arrival time-ului, si o metoda pentru afisarea serverelor la un moment de timp, cu toate taskurile ce le stocheaza (procesing time-ul actualizandu-se la fiecare secunda, impreuna cu waitingPeriod la servere). Cu aceste metode scriu atat in consola (doar pentru inceput, pentru verificari, desi am lasat si scrisul in consola), cat si in fisierul text si in gui. Scrierile sunt foarte asemanatoare, voi vorbi la sectiunea de rezultate unde stochez ce.

-Metoda importanta ramasa de discutat este run, care creeaza threadul, si daca nu se ajunge la una din cele 2 conditii de oprire, daca este lista generata goala sau timpul de rulare = cu timpul maxim de rulare, atunci se gasesc clienti ce au arrival time-ul egal cu timpul curent de rulare. Ei se extrag din coada, se adauga in servere cu ajutorul schedulerului, cu metoda dispatchTask, si dupa asteptam cu sleep executia threadurilor in paralel, si pe langa mai si calculam date ce vor avea rol la final, precum average timeurile sau procesing timeurile la fiecare task, etc... .Dupa ce se termina executia la threadurl principal, imediat dupa se termina executia la threadurile paralele, adica la servere, pentru a nu merge pana la infinit, nu au alta conditie de oprire. Tot ce a mai ramas este afisarea in consola, pe fisier si pe gui a rezultatelor si terminarea programului.

Clasa SimulationFrame:

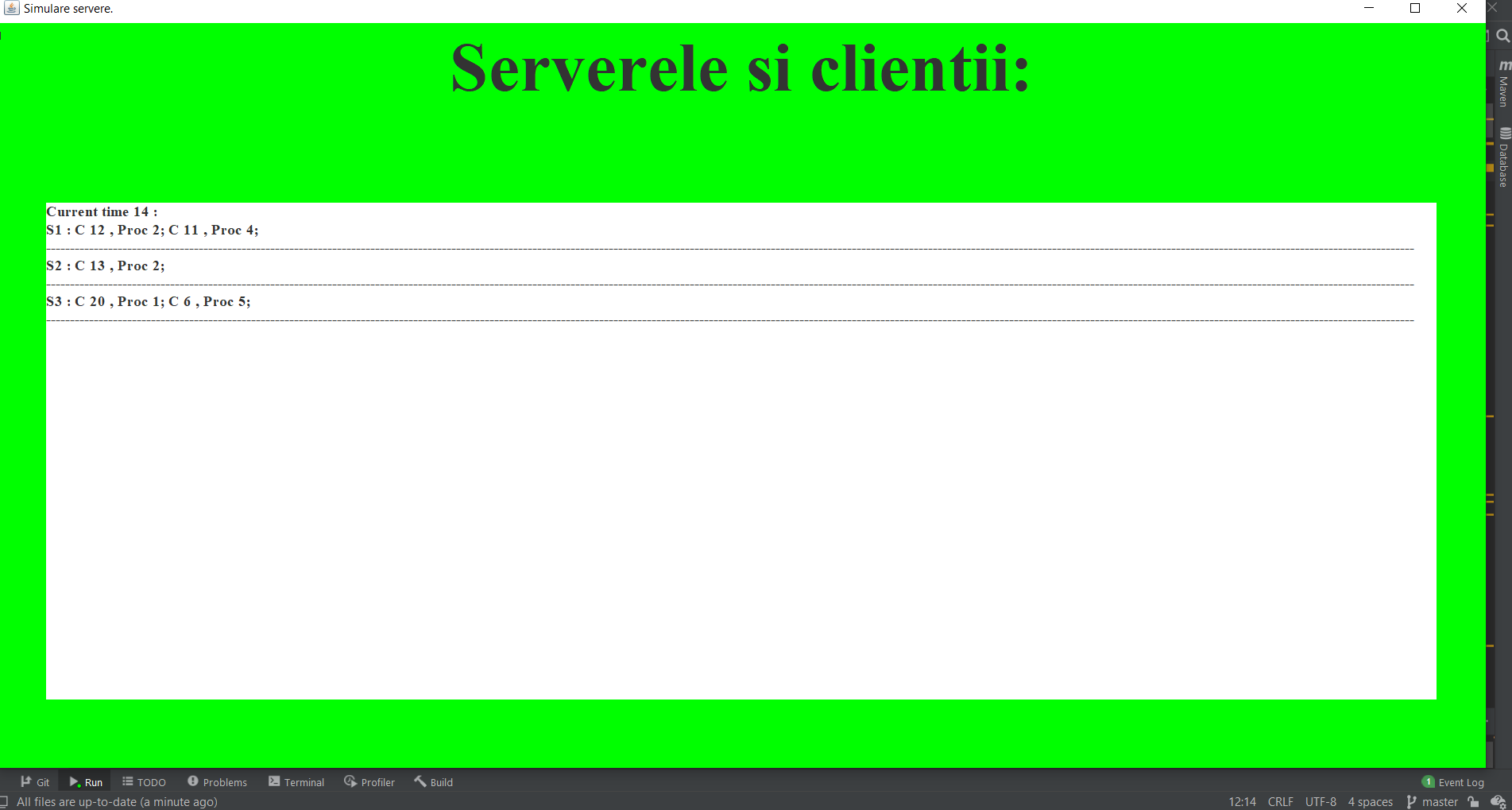
-Asa arata interfata pentru introducerea datelor:

-Se pot introduce cele 7 date de intrare pe text fielduri, dupa care cu butonul se trece la rularea simularii:



-Asa arata programul gui in timpul rularii, la un moment de timp (aici 14, ales la intamplare):

-Se observa lista de servere si dupa lista de clienti la fiecare server:



-Clasa de interfata de data aceasta nu este la fel de complexa ca un MVC, dar isi face in mod corect treaba. Folosesc 2 „view-uri” diferite, se trece de la primul la al doilea printr-un buton, inapoit nu se poate. Structura este asemanatoare cu tema 1 (asa mi se pare cel mai natural sa fac), am 7 getteri pentru a da datele de intrare la SimulationManager si sa foloseasca datele de intrare pentru simularea programului, si pe langa clasa interna pentru buton si alti constructori banali, mai am un „AreaText”, pentru a afisa pe rand, ca intr-un fisier text, serverele impreuna cu clientii. Este o abordare simpla, ce functioneaza. Daca se doreste afisarea a mai multor servere sau clienti, se poate mari fereastra sau micsora fontul de scriere a datelor.

Clasa ConcreteStrategyQueue si Time, impreuna cu interfata Strategy si enumerarea SelectionPolicy:

-Logica aici este ca alegem o stragegie, si folosind acea strategie (una din acea enumeratie), vom putea stabili unde punem acel task. Cu ajutorul interfetei strategy, cele 2 clase de Queue si Time adauga, in mod diferit, prin suprascrierea metodei (schimband continutul metodei), taskul la servere. Folosesc logica la Time cu preluarea cu getter a waitingPeriod-ului fiecarui server, pentru a vedea care are cel mai putin timp de asteptare, si in acel server introducem taskul nou venit, cu dorinta de a nu sta prea mult la coada. De asemenea, cu Queue, punem acel task nou la rand cu cele mai putine alte taskuri, nu ne intereseaza timpul lor de terminare. (folosind de data asta size la lista de taskuri, tot cu un getter)

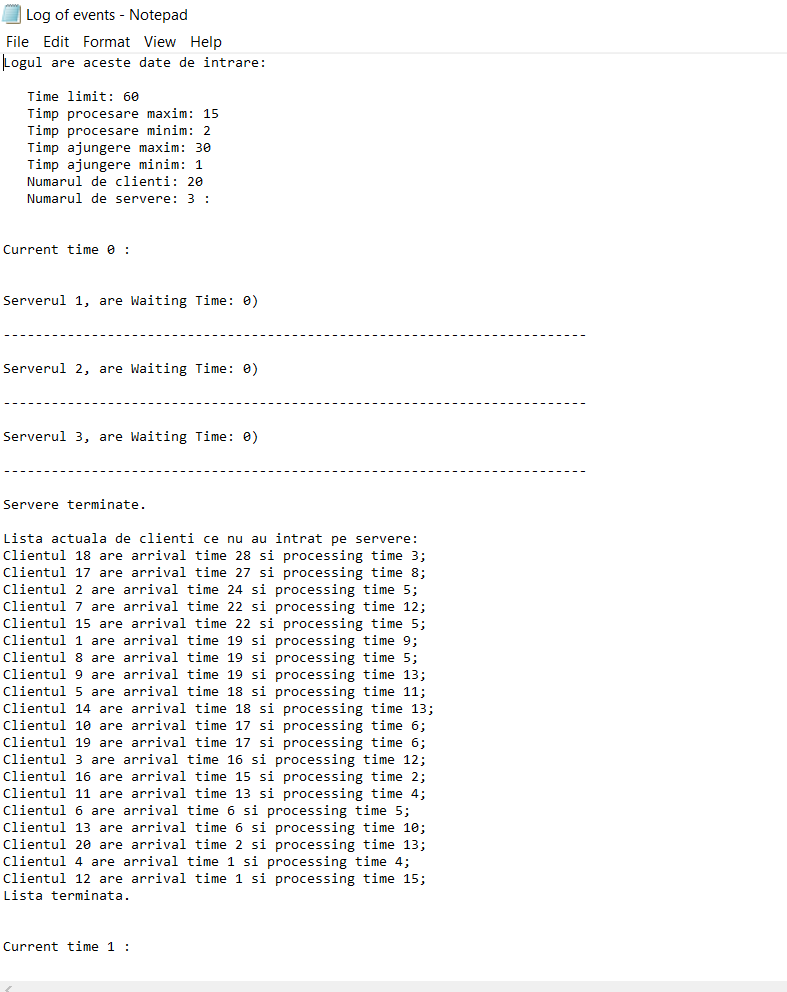
-Ambele strategii sunt bune, dar am implementat folosind Time, pentru ca asa se specifica si pentru ca asa pare cel mai eficient. Dupa aceasta strategie, si implementarea ei, se adauga taskul nou la unul din serverele alese.

1. Rezultatele obtinute:

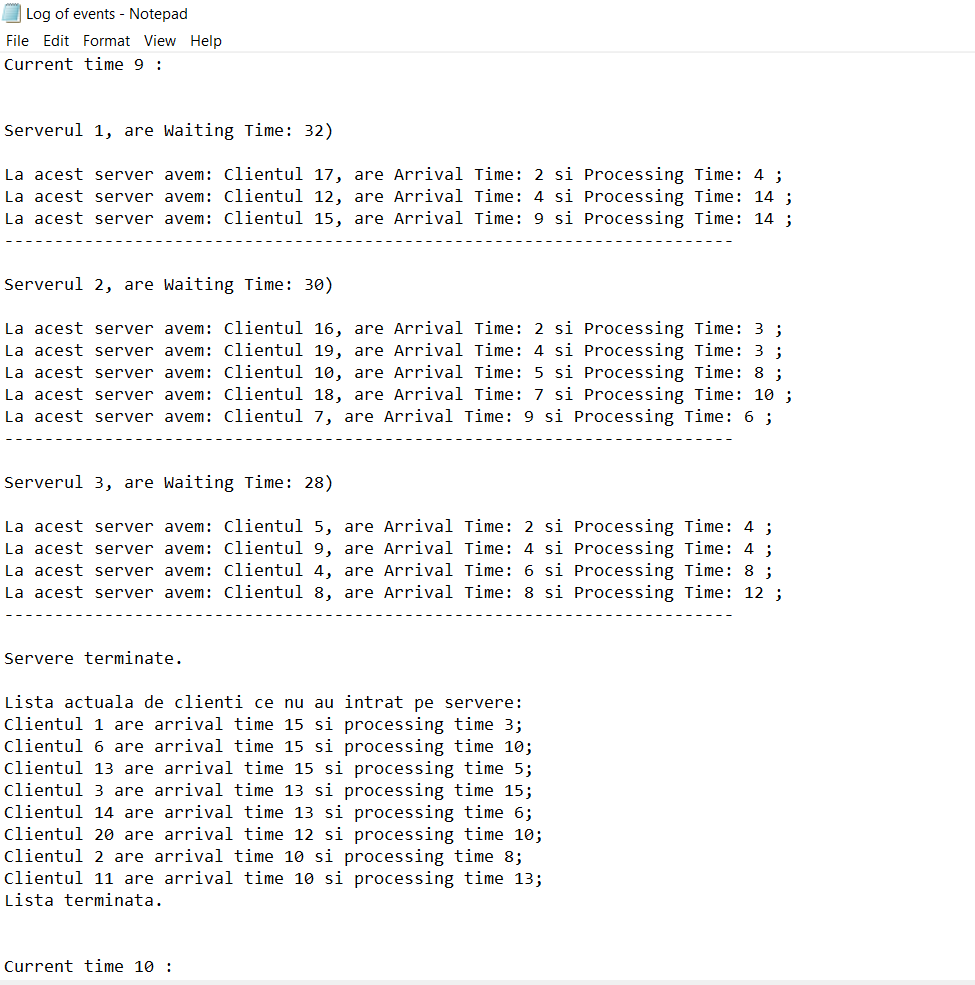
Nu am folosit Junit pentru testare la aceasta tema, ci am testat pe mai multe cazuri exceptionale, majoritatea parand sa dea rezultate logice, sau cele care nu au dau fiind tratate intr-un final;

La aceasta tema, o cerinta in plus a fost afisarea intr-un fisier text a logului cu informatii legate de fiecare timp de executie, in care threadurile rulau, si afisarea lor in ordine in acel fisier (in acel log). In gui am afisat id-ul clientului, timpul de rulare si timpul de procesare a fiecarui client, pentru ca mi se pareau informatiile cele mai relevante. In plus, se puteau afisa acei timpi de average (service si waiting), dupa care peak hour, la finalul unuia din cele 2 abordari. Puteam in ambele la fel de usor, am ales sa afisez la finalul fisierului text cele 3, dupa care se termina executia.

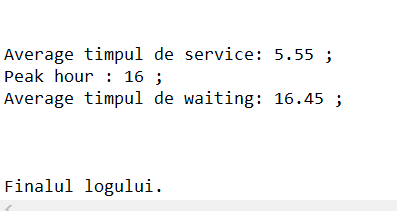
-La inceputul fisierului, extrag datele de intrare, dupa care, pentru fiecare current time, afisez serverul, cu cat waiting time are, si fiecare client de la acel server, cu informatiile despre el. Folosesc linia --- pentru despartire. Dupa afisarea serverelor, afisez lista generata, in ordine arrival time, a clientilor ramasi in afara serverelor (pe moment), pentru a stii cine urmeaza sa fie introdus si la ce moment.



-La fiecare moment de timp se poate observa cat de mult mai trebuie procesat un client, si cat de mult se sta la fiecare server la coada.



-La finalul fisierului, afisez cele 3 informatii legate de timpii de executie si de average intre clienti, calculate cu metodele descrise mai sus. Chiar daca executia nu s-a terminat cu toti clientii terminati de generat, tot se calculeaza dupa anumita logica acesti timpi si se afiseaza aici:



O ultima observatie, ar fi faptul ca cele 3 fisiere text de testare, sunt puse in folderul incarcat pe GitLab, cat si aici:

1)



2)



3)



1. Concluzii:

De la aceasta tema am invatat cum sa lucrez in paralel cu threaduri, un lucu ce este in sine complex, dar dupa studierea problemei, este chiar pe inteles si pare foarte eficient sa lucrezi in asemenea fel. Mi-a placut sa rezolv tema, mi s-a parut ca totul a avut sens, si ca am ramas cu experienta acumulata fata de cum eram la inceput.

Ca dezvoltari ulterioare, as dori sa finisez unele cazuri exceptionale, pentru a asigura o rulare corecta din toate punctele de vedere, insa nu am avut timp sa rulez absolut fiecare caz. Dar, in rest, as dori sa incerc sa maresc numarul de servere pe care se poate rula programul (momentan pare sa fie careva sute, dar vreau sa ajung si mai sus), cu algoritmi de eficientizare, sau poate o afisare mai organica, sa poata sa incapa toata informatia gasita.

In concluzie, a fost o tema interesanta, am invatat multe de la ea, si sper sa mai am oportunitatea sa lucrez cu threaduri de acum incolo!

1. Bibliografie:

* <https://www.geeksforgeeks.org/>
* https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html
* Materialul ajutator oferit pentru aceasta tema.