**DOCUMENTAȚIE TEMA 2**

**SIMULATOR DE COZI**

**Nume prenume: Copoț Raluca**

**Grupa: 30227**

**Profesor laborator: Mitrea Dan**

Cuprins

1. **Obiectivul temei3**
   1. Obiectivul principal3
   2. Obiective secundare3
2. **Analiza problemei3**

2.1 Analiza principală 4

2.2 Cerințe funcționale 4

2.3 Cerințe non-funcționale 4

1. **Proiectare5**

3.1 Determinarea structurilor de date5

3.2 Împărțirea în clase și pachete5

3.3 Algoritmi6

1. **Implementare7**

4.1 Clase și metode7

4.2 GUI9

1. **Rezultate10**

5.1 Funcționare10

5.2 Testare10

1. **Concluzii11**
2. **Bibliografie11**
3. **Obiectivul temei**
   1. **Obiectivul principal**

Proiectarea și implementare unei aplicații care analizează sistemele bazate pe cozi și determină timpul minim de așteptare pentru clienți

* 1. **Obiective secundare**

|  |  |  |
| --- | --- | --- |
| **Obiectiv secundar** | **Descriere** | **Capitol** |
| Analizarea problemei și identificarea cerințelor | Realizarea diagramei use-case și determinarea scenariilor posibile | 2 |
| Determinarea structurilor de date | Pentru ca aplicația să funcționeze corect trebuie să definim tipuri de corespunzătoare execuției cu thread-uri | 3 |
| Împărțirea în clase si pachete | Determinarea unui scop bine definit pentru fiecare clasă și distribuirea acestora în diferite pachete, în funcție de rol | 3 |
| Determinarea și dezvoltarea algoritmilor | Pentru a putea efectua operații pe cozi, precum adăugarea unui client, trebuie dezvoltați și implementați atât algoritmi, cât și strategii | 3 |
| Implementarea simulatorului de cozi | Scrierea propriu-zisă a codului, implementarea efectivă a algoritmilor determinați anterior | 4 |
| Testarea | Verificarea corectitudinii simulatorului pe cele trei cazuri de testare propuse în îndrumător | 5 |

1. **Analiza problemei**

**2.1. Analiza principală**

Exemplu de use-case pentru introducerea datelor:

**Use case**: introducerea datelor

**Actorul primar**: utilizatorul

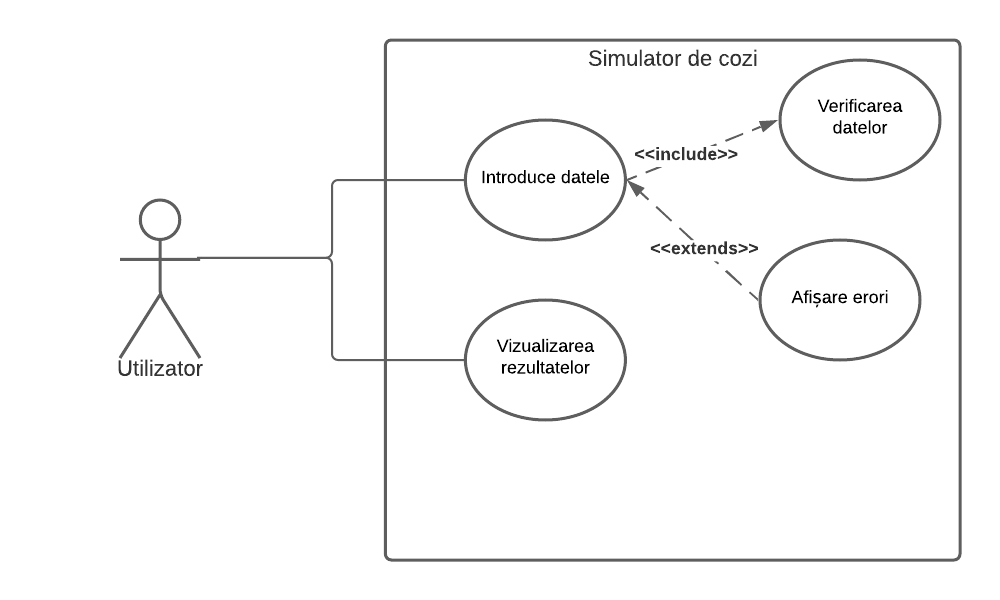
**Scenariul principal în caz de succes**:

1. Utilizatorul introduce datele de intrare: numărul de clienți, numărul de cozi, timpul total de simulare, intervalul de sosire, respectiv intervalul de procesare pentru clienți
2. Utilizatorul apasă pe butonul care pornește simularea „Submit”
3. Simularea pornește, se deschide o nouă fereastră în care utilizatorul poate urmării evoluția în timp real a simulării

**Scenariul alternativ**: Introducerea unor date de intrare invalide

* Utilizatorul introduce un interval negativ de timp sau nu introduce un număr
* Aplicația afișează un mesaj de eroare
* Scenariul se întoarce la pasul 1

**Diagrama use case:**



**2.2. Cerințe funcționale**

Proiectați și implementați un aplicație de simulare cu o interfață grafică, care ca scop analiza sistemelor bazate pe cozi, pentru a determina timpul minim de așteptare al clienților. Aplicația trebuie să poată realiza:

* Citirea datelor de la tastatură;
* Rularea efectivă a simulării;
* Afișarea pe ecran și în fișier a rezultatului
* Revenirea la starea inițială;

**2.3. Cerințe non-funcționale**

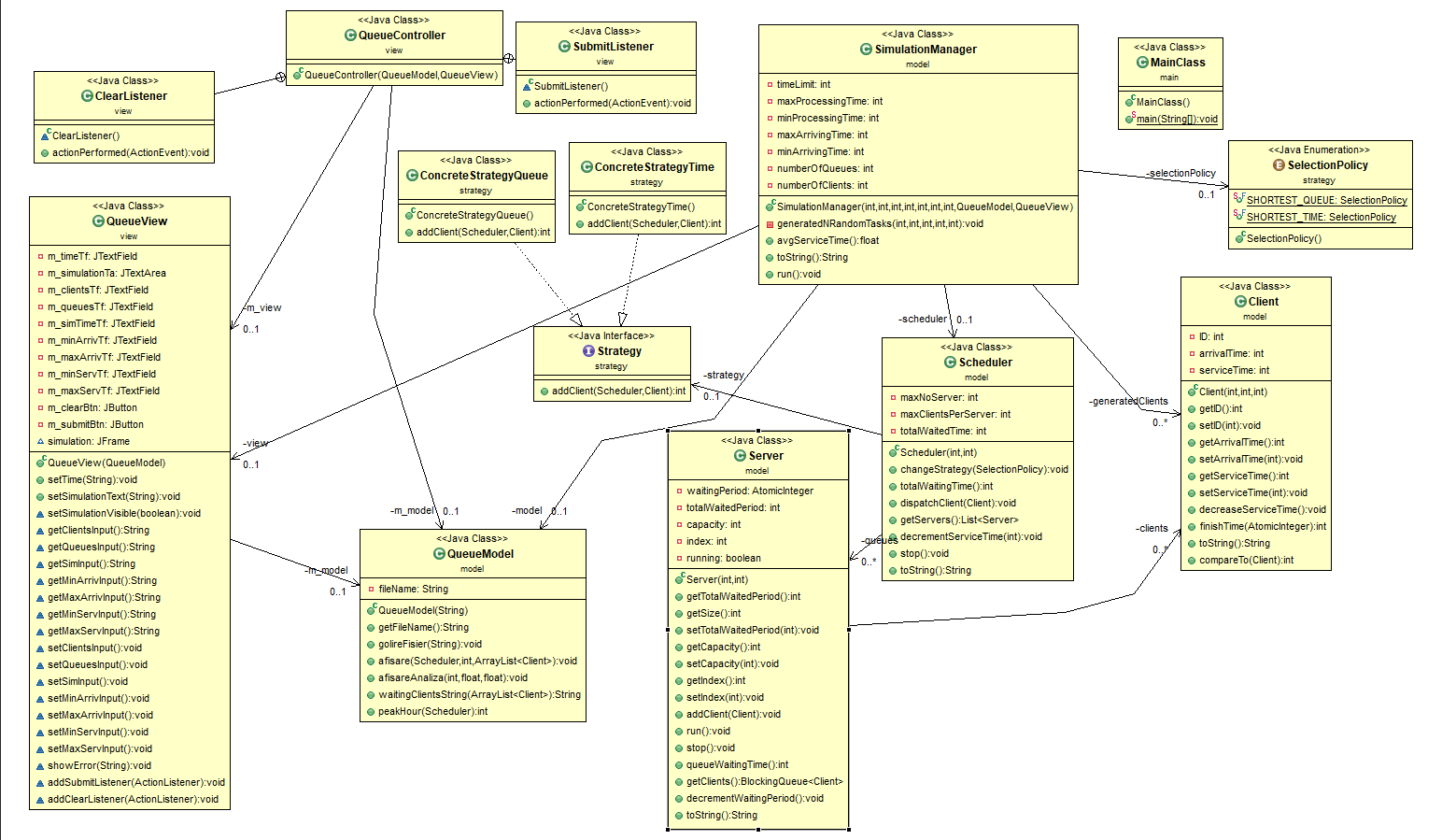
* Simulatorul de cozi ar trebui să fie intuitiv și ușor de folosit de utilizator;
* Designul ar trebui să fie unul compact, dar care conțină toate elementele dorite, fără să devină prea încărcat;
* Aplicația ar trebui să atenționeze utilizatorul atunci când introduce date de intrare invalide și să nu efectueze pornească simularea decât după ce primește datele corespunzătoare;
* Rezultatul afișat ar trebui sa fie lizibil, ușor de înțeles și interpretat;

1. **Proiectare**
   1. **Determinarea structurilor de date**

Pentru acest proiect, principalul element este clientul. Deoarece nu există un astfel de timp de date în Java, am decis să creez o clasă care să îndeplinească rolul de client. În ceea ce privește cozile, ele pot conține unul sau mai mulți clienți, sau pot fi goale. Din această cauză și având în vedere faptul ca voi lucra cu threaduri, am decis că am nevoie de o clasă „Serve”, care va conține un ArrayBlockingQueue de clienți. Iar, pentru că am nevoie de mai mult de o singură coadă, voi folosi și o lista de cozi, aceasta fiind conținută în „Scheduler”.

Precum este menționat în cerință, pentru operația de adăugare a unui client pot avea două strategii diferite, una care este în favoarea timpului minim de așteptare și una care este în favoarea cozii celei mai scurte. Așadar, folosesc un enum, numit „SelectionPolicy”, care conține cele două strategii, respectiv „SHORTEST\_QUEUE” și „SHORTEST\_TIME”.

* 1. **Împărțirea în clase și pachete**

**Diagrama UML:**

Pasul următor după stabilirea tipurilor de date pe care le voi folosi, este împărțirea în clase, respectiv în pachete. Două dintre aceste clase reies din analiza efectuată la pasul precedent, acestea fiind clasele „Client”, respectiv „Server”. Pentru a rula simularea în timp real trebuie să lucrez cu fire de execuție. Astfel, precum este recomandat în suportul pentru assignment mă voi folosi și de clasele „Scheduler” și „SimulationManager”.

După cum este menționat în cerință și după cum am menționat la analiza anterioară, trebuie să mă folosesc de două strategii de adăugare. Așadar, pentru a le implementa voi folosi interfața „Strategy”, care va fi implementată de clasele „ConcreteStrategyQueue” și „ConcreteStrategyTime”.

De asemenea, mă folosesc de arhitectura de tip MVC, deci voi avea nevoie de clasele „QueueView”, „QueueController”, „QueueModel”. În cele din urmă, am nevoie de o clasă main, „MainClass”.

Pentru împărțirea în pachete am folosit ca și criteriu funcționalitatea claselor, ajungând la următoarele pachete:

* main: conține clasa „MainClass”;
* model: care conține logica aplicației, respectiv clasele QueueModel, Client, Scheduler, Server și SimulationManager;
* strategy: acest pachet conține elementele care implică strategiile de adăugare, respectiv ConcreteStrategyTime, ConcreteStrategyQueue, SelectionPolicy și Strategy;
* view: în care se află elementele legate de interfață, respectiv QueueView si QueueController;
  1. **Algoritmi**

**Shortest time:** folosindu-mă de un obiect de tipul scheduler compar timpul de așteptare, respectiv variabila „waitingPeriod” din Server, salvând mereu minimul și indexul cozii care are timpul minim de așteptare. După parcurgere, în cazul în care index-ul este valid, voi adăuga clientul cu metoda de addClient din server, care adaugă clientul respectiv în coadă. În caz contrar, va returna -1.

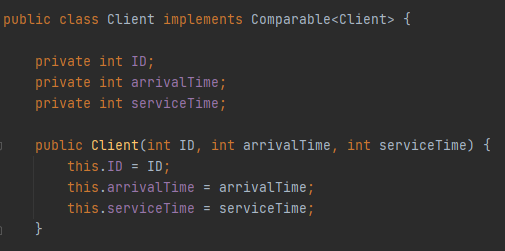
**Shortest queue:** utilizez și în acest caz un obiect de tipul scheduler, comparând de această data numărul de clienți și salvând, precum în cazul precedent, numărul minim și indexul. Verific apoi validitatea indexului, pentru a putea adăuga clientul în coada cu indexul respectiv, în caz contrar returnând -1.

**Peak hour:** pentru fiecare coadă din scheduler, îmi adun numărul de clienți în variabila „maxClients”, pe care o voi returna la final. Ulterior, în run, în fiecare secundă a simulării apelez această funcție pentru determina la ce „oră” este numărul maxim de clienți, ora respectivă fiind „peakHour”.

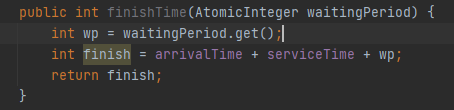
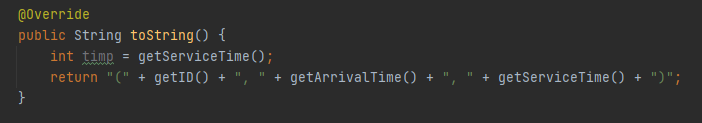
**Average service time:** pentru fiecare client adun într-o variabilă serviceTime-ul, pe care la final îl împart la numărul total de clienți, pentru a obține o medie.

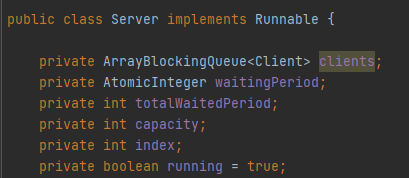
**Average waiting time:** calculez timpul de așteptare pentru ultimul client din coadă ca fiind suma timpului de procesare pentru toți clienții care se află în fața lui în coadă. Calculez acest timp pentru toate cozile din scheduler și împart de această dată la numărul de cozi, pentru a obține o medie. Repet acest algoritm pentru fiecare secundă din simulare, suma finală împărțită la timpul la care simularea s-a oprit, rezultând, astfel, timpul mediu de așteptare.

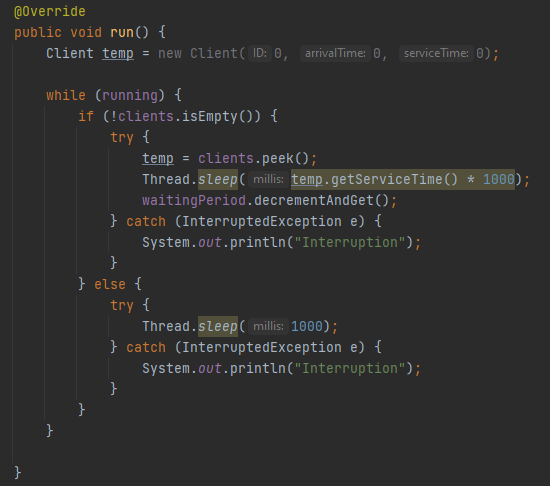
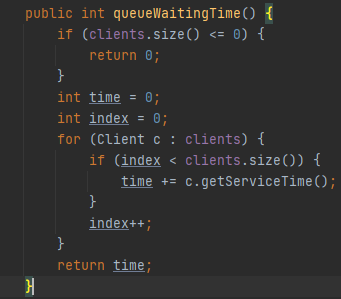
**Scăderea timpului de procesare:** dacă timpul de procesare al primului client din coadă este mai mic sau egal cu unu, atunci voi scoate acel client din coadă, în caz contrar decrementând timpul de procesare. Iar, în cazul în care mai avem clienți în coadă, va scădea timpul de așteptare.

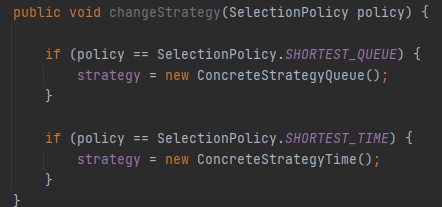
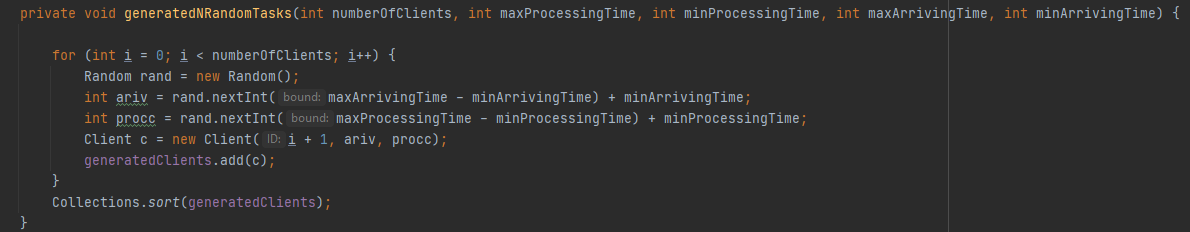


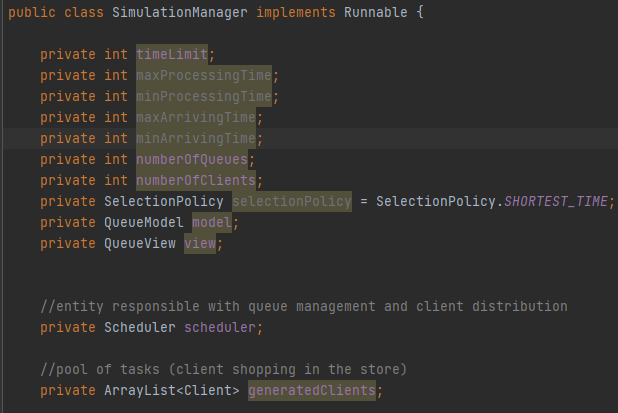
1. **Implementare**
   1. **Clase și metode**

**Client:** conține structura unui client. Atributele sale sunt: ID-ul, timpul de sosire, „arrivalTime” și timpul de servire, „serviceTime”, toate acestea fiind int-uri. Această clasă implementează interfața „Comparable”, pentru a putea sorta clienții crescător, în funcție de arrivalTime. Deorece acest lucru este menționat în cerință, am implementat o funcție care îmi calculează timpul de terminare, „finishTime”, pentru fiecare client. Acesta este calculat precum suma dintre timpul de sosire, timpul de procesare și timpul de așteptare. Pentru ușurință, am implementat și o metodă, care va decrementa timpul de așteptare, atunci când este apelată. De asemenea, am decis să suprascriu metoda „toString”, pentru a putea afișa clienții din cozi.

**Server:** această clasă este cea care conține clienții, doar primul dintre aceștia fiind procesați. În plus, această coadă implementează interfața „Runnable”, așadar, atributele sale sunt: un ArrayBlockingQueue de clienți, structură pe care am ales-o pentru ca acest vector să fie „thread-safe”, un „AtomicInteger”, care reprezintă perioada de așteptare. Am ales acest tip de dată din aceeași cauză. De asemenea, conține si o perioadă totală de așteptare, capacitatea si indexul cozii, acestea fiind int-uri, și o variabilă boolean, numită running. Atunci când valoarea de adevăr a acestei variabile este falsă, rularea cozii se va încheia.

Fiind clasa care conține clienții, am implementat metoda „addClient”, care adaugă un task în listă și incrementează perioada de așteptare a cozii cu perioada de procesare a taskului respectiv. Conform celor menționate anterior, metoda „run” va rula cât timp variabila „running” este true. Cât timp există clienți în listă, firul de execuție va aștepta pe perioada pe procesare a primului client din listă și va decrementa acest timp. În caz contrar, va aștepta o secundă. Metoda „stop” va oferi valorea falsă de adevăr variabilei „running”. De asemenea, am implementat „toString” și metoda menționată anterior, care calculează timpul de așteptare pentru ultimul client din coadă.

**Scheduler:** reprezintă ansamblul de cozi. Atributele sale cuprind o listă „List” de servere și strategia. În constructorul acestei clase, am inițializat lista de cozi utilizând „Collections.synchronizedList”. În funcție de argumentul primit ca parametru, metoda „changeStrategy” va schimba strategia curentă, ce utilizată în mod implicit fiind „SHORTEST\_TIME”. Metoda „dispatchClient” apelează adăugarea clientului în funcție de strategia curentă, iar dacă aceasta returnează minus unu, atunci clientul nu poate fi adăugat și se va arunca o excepție. Pentru a putea scoate clienții din coadă după ce li se termină timpul de procesare, am implementat metoda „decrementServiceTime”. Pentru fiecare coadă, în cazul în care timpul de procesare al primului client este nenul, în decrementez, iar în caz contrar el este scos din coadă. De asemenea, dacă perioada de așteptare e cozii este nenula, atunci o voi decrementa și pe aceasta.



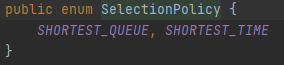
**SimulationManager:** implementează interfața „Runnable”precum celelalte clase care se ocupă cu simularea. Printre atributele acestei clase, precizate deja în suportul pentru această tema, am inclus modelul si view-ul, pentru a putea realiza cu ușurință afișarea în interfață în timp real. Utilizez, de asemenea, o metodă pentru a genera aleator un „ArrayList” de clienți, pe care îl voi sorta. În plus, implementez metoda de run. Inițial, aceasta îmi golește fișierul, pentru a pentru a nu suprascrie datele. Prima condiție de oprire apare în cazul în care timpul curent depășește intervalul de timp alocat. A doua condiție care poate încheia simularea, este în cazul în care au fost procesați toți clienții. În cazul în care aceste condiții sunt false, thread-ul își urmează execuția normal, adăugând în coada adecvată clienții, dacă timpul lor de sosire corespunde cu timpul curent, iar apoi acesta este scos din lista inițială. Indiferent dacă a avut sau nu loc o adăugare, thread-ul va aștepta o secundă, pentru ca utilizatorul sa poată înțelege datele afișate în gui. În plus, pentru fiecare pas al simulării, sunt calculate averageWaiting, peakHour și averageServiceTime, iar la finalul execuției sunt afișate, în cele din urmă thread-urile fiind oprite. Pentru a putea afișa clienții care așteaptă într-un format mai ușor de înțeles, am implementat toString.

**QueueModel:** conține ca și atribut numele fișierului. Metodele „golireFisier”, „afisare” și „afisareAnaliza” sunt cele ca realizează scrierea efectivă în fișier, prima golind-ul, a doua afișând cozile și clienții în așteptare, iar cea de a treia va afișa cele trei rezultate calculate, iar apoi îl va închide. Am implementat, de asemenea, o metoda care calculează ora de vârf și una care asigură o afișare lizibilă.

**Strategy:** este o interfață care conține metoda de adăugare a unui client.

**ConcreteStrategyQueue:** implementează interfața „Strategy” și implicit metoda de adăugare. Aceasta este implementată pentru a putea adăuga clientul în cea mai scurtă coadă.

**ConcreteStrategyTime:** implementează interfața „Strategy” și implicit metoda de adăugare. Aceasta este implementată pentru a putea adăuga clientul în coada care are cel mai scurt timp de așteptare.



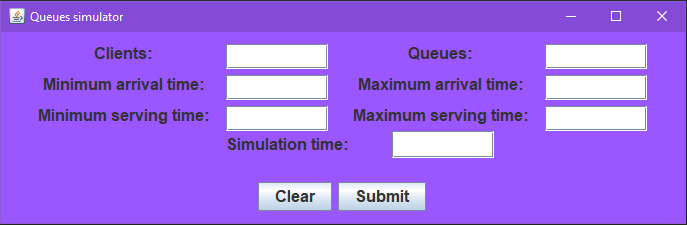
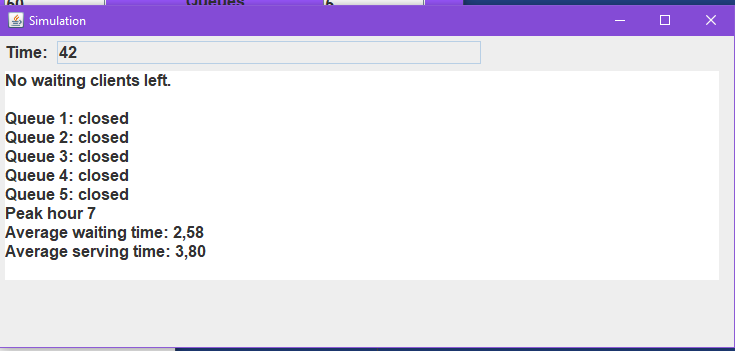
**SelectionPolicy:** este un enum care conține cele două strategii de adăugare în coadă.

**QueueController: :** controlează logica aplicației, coordonând View-ul și Modelul. Acesta primește un input prin intermediul vederii, procesează datele cu ajutorul modelului si returnează rezultatul vederii.

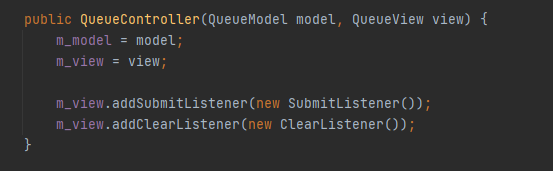
**QueueView:** conține elementele care formează interfața grafică, precum butoanele sau text-field-urile, dar și câte un ActionListener pentru fiecare buton.

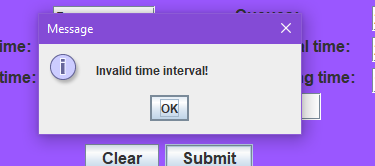
**4.2. GUI**

GUI (Graphical User Interface) din Java reprezintă mecanismul cu ajutorul căruia utilizatorul poate interacționa cu un program. Acesta conține componente grafice, cum ar fi butoane, etichete, ferestre etc., pe care utilizatorul le poate folosi pentru a interacționa cu ușurință cu aplicația. GUI are un rol important în construirea cu ușurință a unei interfețe pentru aplicațiile Java.

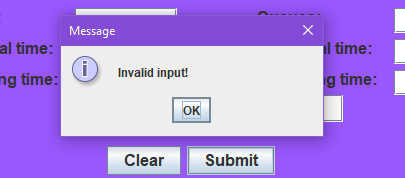
**View:** precum am menționat în secțiunea anterioară, în această clasă se află elementele care formează interfața grafică. În momentul pornirii execuției programului, utilizatorul poate vedea JFrame-ul din dreapta. Aceasta este formar din trei panouri. Primul dintre acestea „input”, conține etichetele si text-field-urile din care vor fi preluate datele de intrare. Al doilea, „butoane”, conține cele două butoane, „Clear” și „Submit”. Cel de al treilea panou, „panouPrinc” le cuprinde pe cele menționate. Pentru a grupa componentele, am folosit layout-ul „GroupLayout”. În cazul utilizării acestui tip de layout componentele trebuie grupate atât pe orizontală, cât și pe verticală, motiv pentru care componentele sunt adăugate de două ori. De asemenea, pentru ca acestea sa fie de aceeași dimensiune, folosesc .linkSize. În cazul în care datele de intrare sunt valide, simularea va porni, moment în care se va face vizibil ce de-al doilea JFrame, care poate fi văzut în stânga. Acesta este format din două panouri, primul fiind „time”, pe care se află un JLabel și un text-field, în care afișez timpul curent al simulării. Al doilea panou, „sim”, conține un JTextArea, la rândul său actualizat în fiecare secundă a simulării, pentru a afișa rezultatele, pe parcurs. De această date am folosit layout-ul „SpringLayout”, aranjând panourile, nu componentele de pe acestea.

Pentru a putea edita această componentă din afara pachetului în care se află, metodele de setare a outputului pentru cel de al doilea JFrame sunt publice, restul având vizibilitatea implicită. De asemenea, am creat câte un ActionListener pentru fiecare buton și metode de preluare a inputului.

******Controller:** conține logica aflată la baza interfeței grafice, îmbinând view-ul cu modelul, motiv pentru care le conține ca atribute. În această clasă am implementat ActionListener pentru fiecare dintre cele două butoane. Acestea suprascriu metoda „actionPerformed” după cum urmează: dacă este apăsat butonul „Submit”, atunci sunt preluate datele de intrare din text-field-uri. În cazul în care sunt valide, este creat un thread si porneste execuția și este făcut vizibil JFrame-ul simulării iar în caz contrat este afișat un mesaj de eroare. Pentru butonul „Clear”, apelez metode care setează String-ul null în text-field-urile din care este preluat inputul și JFrame-ul care conține simularea este ascuns.

1. **Rezultate**
   1. **Funcționare**

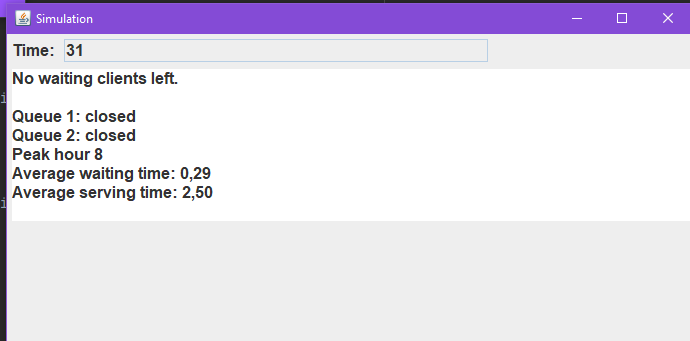
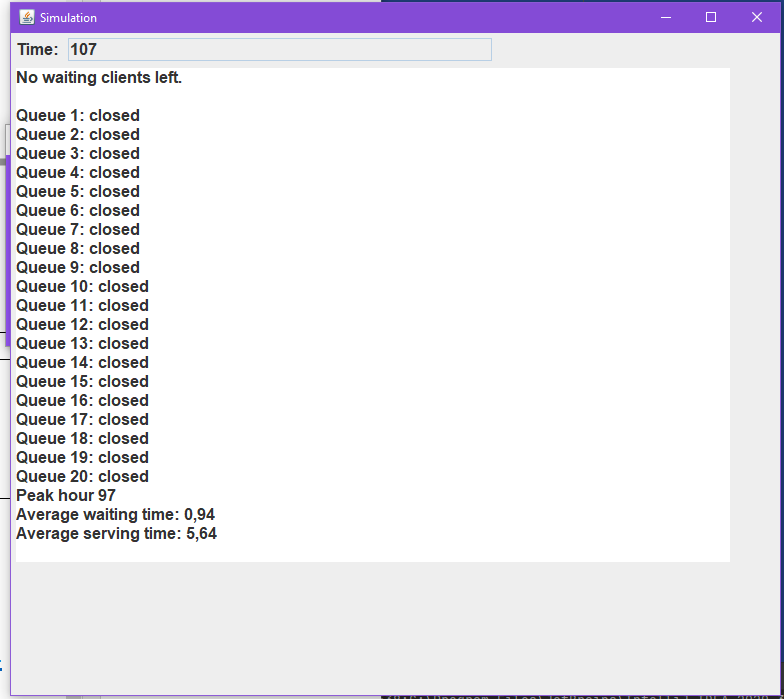
Utilizatorul trebuie să introducă în text-field-uri datele corespunzătoare pentru testare și să apese apoi pe butonul de trimitere. În cazul în care acesta face o greșeală va apărea un mesaj de eroare. În acest caz trebuie să își repare greșeala și să încerce din nou.

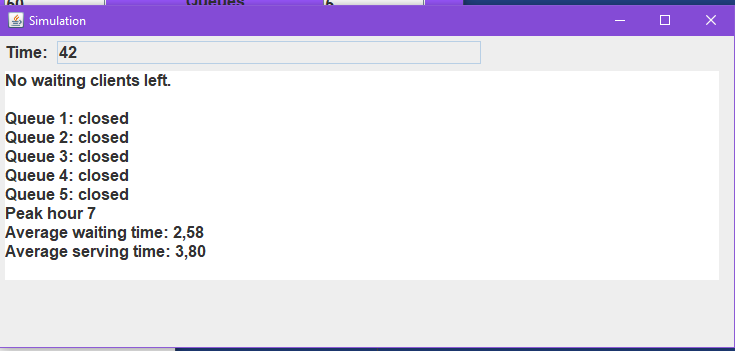
Dacă nu comite nicio greșeală simularea va începe și se vor afișa pașii săi în a doua fereastră. Utilizatorul poate fie să o închidă, fie să apese pe butonul „Clear”. La finalul execuției, în fișierul „output.txt” utilizatorul poate găsi log-ul întregii simulări.

* 1. **Testare**

Pentru acest proiect nu a trebuit să efectuez teste cu JUnit, în schimb fiind nevoie să rulez cele trei teste din pdf-ul cu cerința. Rezultatele celor trei teste, după terminarea rulării sunt următoarele:

Testul 1: cu 4 clienți și două cozi Testul 3: cu 100 de clienți și 20 de cozi

****

Testul 2: cu 50 de clienți și 5 cozi

1. **Concluzii**

Datorită acestui proiect am lucrat pentru prima dată cu thread-uri. Am înțeles, astfel cum funcționează și de ce este important să folosesc structuri de date care sunt thread-safe. De asemenea, am folosit pentru prima dată layout-ul de tip „SpringLayout”. În plus, am încercat pentru prima dată să actualizez în timp real o interfață de tipul GUI și voi știi pe viitor ce tip de componente ar trebui sa folosesc pentru a reuși acest lucru.

Unul dintre lucrurile pe care le-aș schimba este metoda „run” din SimulationManager, toată clasa fiind puțin haotică, pentru că nu am știut cum aș putea organiza mai bine toate metodele necesare. Iar, dacă aș fi avut un nivel mai avansat, aș fi încercat să găsesc modalitatea optimă de a schimba strategiile de adăugare, pentru a avea întotdeauna cel mai mic timp de așteptare.

1. **Bibliografie**

* Definiție GUI (13.04.2021 21:30): [https://www.guru99.com/java-swing-gui.html definiție GUI 17.03](https://www.guru99.com/java-swing-gui.html%20definiție%20GUI%2017.03)
* Layout (13.04.18:00): <https://docs.oracle.com/javase/tutorial/uiswing/layout/group.html>
* AtomicInteger (10.04 19:00): <https://howtodoinjava.com/java/multi-threading/atomicinteger-example/>
* ArrayBlockingQueue(10.04 20:00): <https://www.geeksforgeeks.org/arrayblockingqueue-class-in-java/>
* Afișare în fișier (11.04 19:00):
  + <https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html>
  + <https://docs.oracle.com/javase/7/docs/api/java/io/BufferedWriter.html>