**TEHNICI DE PROGRAMARE**

**TEMA 2**

**SIMULAREA ACTIVITATII UNUI MAGAZIN**

**DOCUMENTATIE**

**Nume și prenume student: Marchis Raul**

**Profesor coordonator: Pop Cristina**

**Grupa: 30229**

1. **Cerinta temei………………………………………………………………………………………….…pag 3**
2. **Obiectivul temei………………………………………………………………………………….......pag 3**
   1. **Obiectivul principal…………………………………………………………………………pag 3**
   2. **Obiectivul secundar………………………………………………………………………..pag 4**
3. **Analiza problemei, modelare, scenarii, cazuri de utilizare…………………………..pag 4**
4. **Proiectare…………………………………………………………………………………………………..pag 6**
5. **Implementare…………………………………………………………………………………………….pag 11**
6. **Rezultate…………………………………………………………………………………………………….pag 13**
7. **Concluzii……………………………………………………………………………………………………..pag 13**
8. **Bibliografie………………………………………………………………………………………………….pag 14**
9. **Cerinta temei**

Propuneti, proiectati si implementati un simulator avand scopul analizarii unui sistem bazat pe cozi de executie pentru a minimiza timpul de asteptare al clientilor.

Mod de rezolvare: Se vor citi datele aferente rezolvarii din interfata grafica, precum :

* Timpul minim si maxim intre care clientii pot sosi la casele de marcat
* Timpul minim si maxim de procesare, in care clientii sunt serviti la casele de marcat
* Numarul de case de marcat
* Numarul de clienti
* Durata in secunde a timpului de simulare

Pentru dezvoltarea aplicatiei, vom folosi tehnica multithreading, astfel incat fiecare casa de marcat va fi reprezentata de cate un thread, fiecare avand propriile ei actiuni de efectuat : adaugarea de noi clienti la coada, respectiv servirea clientilor. Impartirea aplicatiei pe clase si scopul fiecareia dintre ele vor fi explicate mai jos, in capitolul 3.

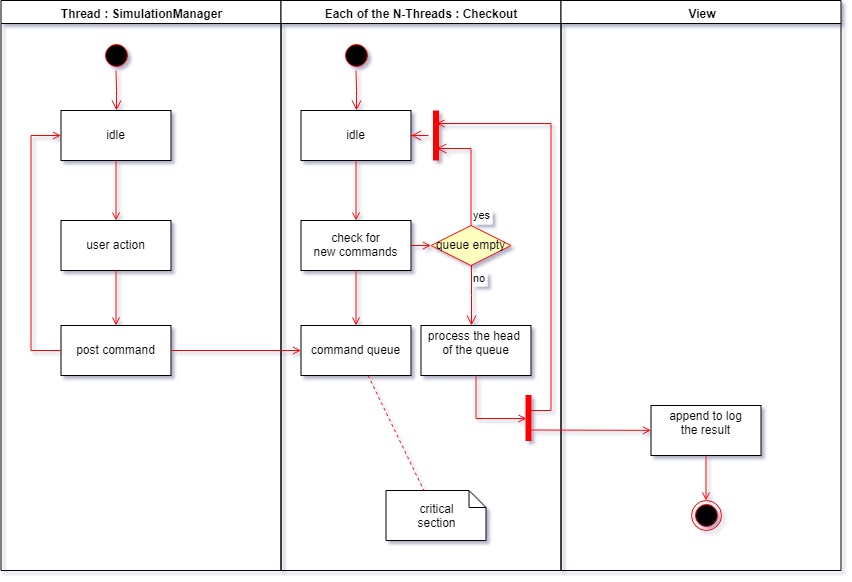
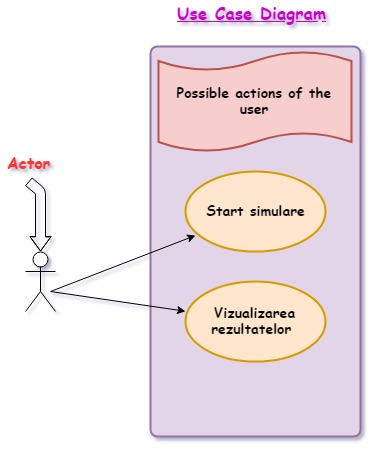
1. **Obiectivul temei**
   1. **Obiectiv principal**Obiectivul principal al aplicatiei esti simularea cat mai aproape de realitate a unui magazin obisnuit. Clientii sosesc la casele de marcat la timpuri diferite, sau de multe ori in acelasi moment. In general, fiecare client “arunca o privire” pentru a vedea casa de marcat unde sunt cei mai putini clienti, pentru a scurta timpul de asteptare si pentru a iesit cat mai repede din magazin. Totodata, ne dorim afisarea timpului mediu de asteptare in cozi, timpul mediu de procesare a clientilor, ora de varf in care casele de marcat au fost cel mai ocupate cat si durata in care cozile au fost goale.
   2. **Obiectiv secundar**

|  |  |  |
| --- | --- | --- |
| Obiectivele secundare | Descrierea obiectivelor | Capitolul |
| Diagrame si descrieri de use-case | Se vor propune posibile intrari de date (nu neaparat optime ) si se va explica motivul acestui lucru prin intermediul unui flow-chart. | Capitolul 2 |
| Structurile de date folosite | Se vor explica structurile de date utilizate si motivul pentru care au fost alese tocmai acestea. | Capitolul 3 |
| Deciziile de proiectare / implementare | Se va explica modul de gandire ales atat pentru interfata grafica cat si pentru implementare. | Capitolul 3 |
| Algoritmi utilizati | Se vor descrie algoritmii folositi pentru trimiterea clientilor in cozi, cat si modul in care este aleasa cate o coada pentru fiecare client in parte. | Capitolul 3 |
| Pachete folosite | Se va descrie motivul alegerii pachetelor in care este structurata aplicatia. | Capitolul 3 |
| Rezultate | Se va discuta putin despre rezultatele obtinute in urma testarii mai multor intrari de date posibile. | Capitolul 5 |
| Concluzii | Se vor trage concluziile si vor fi subliniate lucrurile invatate in urma temei. | Capitolul 6 |

1. **Analiza problemei, modelare, scenarii, cazuri de utilizare**Use-case pentru posibilele date de intrare ale aplicatiei.   
   **1. Utilizatorul(actorul) introduce datele de intrare**  
    Normal flow: se introduc toate datele de intrare valide si se apasa butonul start.  
    Bad flow: utilizatorul introduce date de intrare invalide, ca exemplu: in locul introducerii un interval in secunde, sau un numar de cozi / persoane, care reprezinta un numar intreg strict pozitiv, acesta introduce numere negative sau chiar litere ale alfabetului / caractere.  
    Tratarea **bad flow-ului** / cazuri speciale: se vor verifica datele introduse in fiecare text field disponibil, iar in cazul in care parsarea lor din string-uri in valori intregi nu este posibila (sau rezultatul parsarii este un numar negativ), se va afisa un mesaj de eroare prin care utilizatorul va fi informat de greseala efectuata.  
   **2. Utilizatorul(actorul) apasa butonul de start**  
    **Normal flow**: se apasa butonul de “START” al aplicatiei o singura data, dupa care se asteapta expirarea timpului de simulare, timp in care se urmareste evolutia programului.  
    **Bad flow / cazuri speciale**: singurul caz special il reprezinta apasarea butonului de “START” de doua ori.

**Tratarea bad flow-ului / cazuri speciale** : In acest caz, se va recepta numarul de apasari ale butonului iar in cazul in care acesta a apasat deja o data pe buton, o a doua apasare nu va avea niciun efect asupra aplicatiei.   
   
 **3. Utilizatorul asteapta si urmareste demersul caselor de marcat si procesarea clientilor**

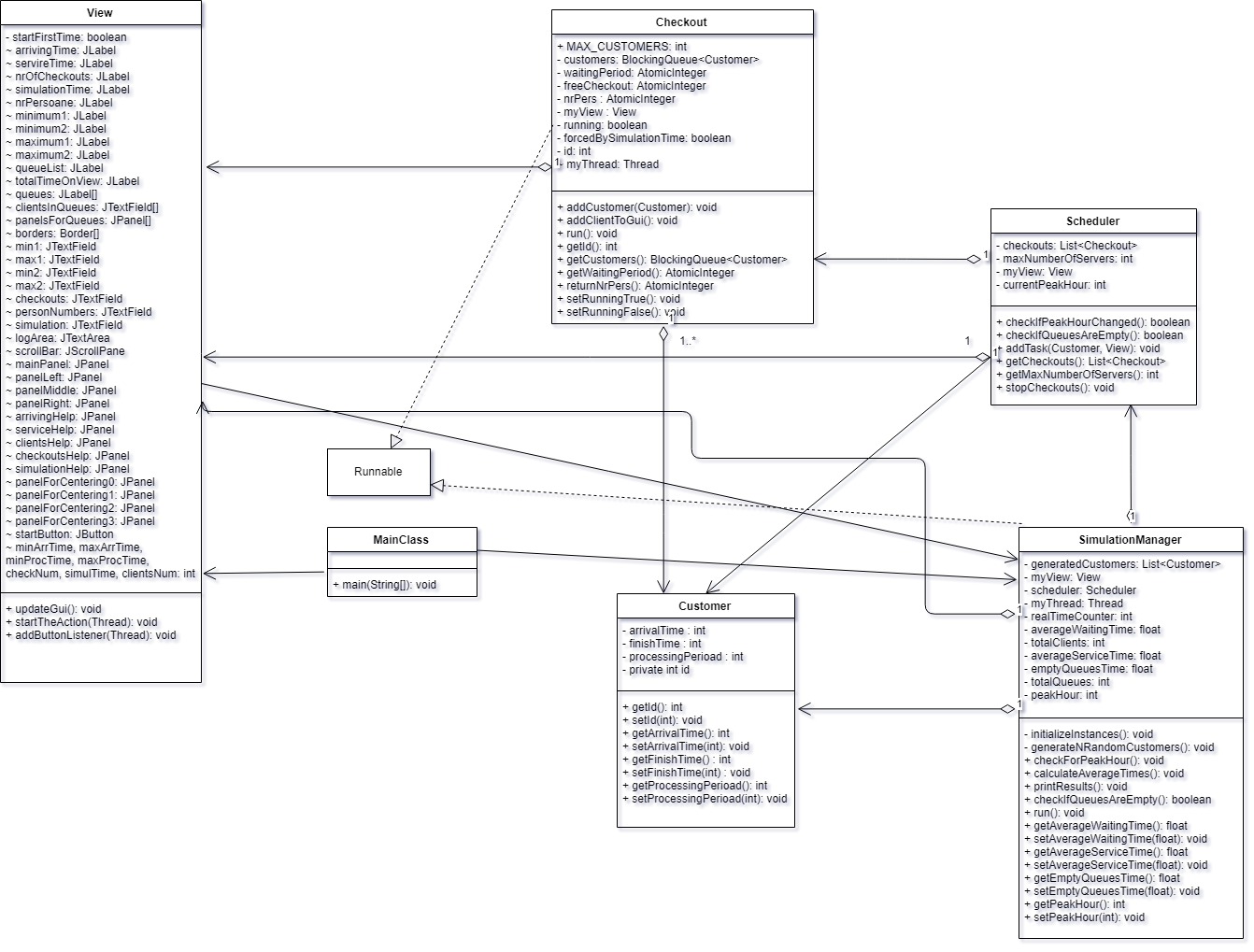
**Normal flow**: dupa incheierea timpului de simulare, se va verifica urmatorul lucru: precum intr-un magazin normal, daca magazinul are ora de inchidere la 22:00, iar la ora 21:45, se aseaza 10 clienti la casele de marcat, este de bun simt ca vanzatorii sa aiba rabdare si sa le ofere serviciile necesare, chiar daca vor sta cu cateva minute in plus fata de programul obisnuit. Asa si aici, in cazul in care timpul de simulare a expirat, insa mai avem cativa clienti la casele de marcat, se va astepta procesarea lor urmand ca rezultatele dorite sa fie afisate printr-un mesaj utilizatorului.  
 **Bad flow / cazuri speciale**: -  
 **Tratarea bad flow-ului / cazuri speciale :** -



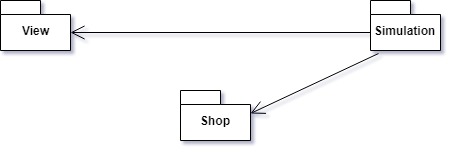
1. **Proiectare**

**Decizii de proiectare** Aplicatia este impartita in trei pachete : **View**, **Shop** si **Simulation**.   
 Pachetul **View** contine o singura clasa, ce are acelasi nume : **View**. Aceasta descrie implementarea interfetei grafice si obiectele alese pentru design-ul GUI-ului.  
 Pachetul **Shop** este alcatuit din clasa **Checkout** si clasa **Customer**, care impreuna alcatuiesc un necesar ca simularea unui magazin sa poata avea loc.   
 Pachetul **SimulationManager** este alcatuit din doua clase: SimulationManager si Scheduler, ambele avand rol de a tine evidenta timpului real scurs din simulare, cat si de a organiza intregul proces.  
  
**Proiectare clase** Pentru inceput, am considerat ca un inceput necesar este proiectarea interfetei grafice, intrucat o aplicatie in care “nu poti vedea ce se intampla” este putin cam neinteresanta pentru utilizatori. Astfel, clasa ***View*** implementeaza intreaga interfata grafica, care este alcatuita din trei parti:   
 - panoul din stanga, in care utilizatorului ii sunt puse la dispozitie campuri pentru a introduce datele de intrare ale aplicatiei : timpul minim si maxim de sosire a clientilor, timpul minim si maxim de procesare al clientilor, numarul de cozi cat si numarul de clienti. Campurile se termina cu un buton de “START”, care va porni aplicatia.  
 - panoul central, care la inceputul aplicatiei este gol. Acesta se va updata in urma apasarii butonului “START”, cu un numar de case egal cu cel introdus de utilizator. Se va putea urmari adaugarea, respectiv procesarea clientilor din cozi in timp real.   
 - panoul din dreapta, care contine LOG-ul aplicatiei: va fi afisata fiecare aparitie a unui nou client in cozi cat si momentul in care clientii pararesc casa de marcat, afisandu-se si timpul lor de finalizare.  
  
 Clasa ***Customer*** reprezinta un client, care are trei variabile instanta: un **arrivalTime**, un **processingPeriod** si un **finishTime**. Atat **arrivalTime**-ul cat si **processingPeriod**-ul sunt ambele generate random, fiecare in parte fiind reprezentat de un numar intreg pozitiv, aflat intre limitele de interval introduse de utilizator in interfata grafica. Campul **finishTime** reprezinta timpul de finalizare, care se stabileste cand un client paraseste coada (se calculeaza ca fiind timpul de sosire <**arrivalTime**> + timpul de procesare <**processingPeriod**> + timpul de procesare al tuturor persoanelor din fata lui.  
  
 Clasa ***Checkout*** reprezinta clasa care descrie o casa de marcat. Ea in sine este un Thread, intrucat fiecare casa de marcat (fiind condusa de un vanzator), lucreaza individual de celelalte case de marcat.  
Variabilele instanta de importanta ridicata ale clasei sunt: Un **ArrayBlockingQueue** de clienti numit **customers**, doua variabile de tip **AtomicInteger**, ce reprezinta timpul de asteptare pentru casa de marcat respectiva cat si timpul in care casa de marcat este goala. Motivul alegerii acestor tipuri de date va fi explicat intr-o subsectiune urmatoare, dedicata structurilor de date folosite. Restul variabilelor instanta sunt si ele foarte importante, intrucat aplicatia nu ar functiona fara ele.  
Dupa cum spuneam, intrucat fiecare coada reprezinta un Thread, voi explica in continuare metodele implementate in aceasta clasa, incluzand metoda specifica unui Thread : **run**().  
Metoda **addCustomer** seteaza finishTime-ul clientului trimis ca parametru, dupa care il adauga in **ArrayBlockingQueue**-ul casei de marcat. Se modifica **waitingPeriod**-ul pentru casa de marcat respectiva, dupa care se face update la log-ul din clasa **View**, apeland metoda implicita a unui **textArea**. Totodata, in aceasta metoda se incrementeaza variabilele statice **averageWaitingTime** si **averageServiceTime** din clasa ***SimulationManager***, care urmeaza la finalul simularii sa fie impartite la numarul total de clienti pentru a determina average time-ul celor doua.  
Metoda corespunzatoare unui Thread, numita **run**() are cea mai mare importanta in aceasta clasa. Pentru a rula metoda intr-o bucla continua, pana cand se decide inchiderea Thread-ului, m-am folosit de o variabila numita **running,** care este setata pe false cand se doreste oprirea Thread-ului. Aceasta bucla mare, mai contine inca doua bucle while : una determina cat timp coada se mentine goala, iar cealalta continua sa verifice contrariul : cat timp avem persoane, le vom procesa, modificand la parasirea unui client si waitingPeriod-ul pentru casa de marcat. In continuare, se face update la interfata grafica pentru a simula in timp real plecarea clientilor din coada.  
  
 Clasa ***Scheduler*** este folosita pentru a determina care este casa cu numarul minim de persoane, astfel incat urmatorul client va fi distribuit acolo. Acest lucru se realizeaza prin intermediul metodei **addTask**().  
 Variabilele instanta de importanta ridicata sunt: un **LinkedList**, care retine toate casele de marcat; Totodata, intrucat avem nevoie de cozi pentru a verifica la fiecare adaugare de client daca ora de varf este la momentul curent, am implementat aici metoda **checkIfPeakHourChanged**(), cat si metoda **checkIfQueuesAreEmpty**() pentru a determina starea cozilor (metoda ajutatoare pentru afisarea rezultatelor, intrucat nu vor fi afisate decat in momentul cand toate cozile sunt goale).   
  
 Clasa ***SimulationManager*** , dupa cum ii spune si numele, este clasa care porneste intreaga aplicatie. Scopul acestei clase, este de a genera numarul total de clienti specificat in interfata grafica de catre utilizator. Fiecarui client generat i se asigneaza un **arrivalTime** si **processingTime** random, dupa care intreaga lista este sortata in ordinea crescatoare a sosirii clientilor. Acest lucru se realizeaza prin intermediul metodei **generateNRandomCustomers**(), clientii fiind retinuti in lista de clienti numita **generatedCustomers**.  
Intrucat si aceasta clasa este un Thread, metoda **run**() are urmatorul scop: atata timp cat timpul de cand s-a dat start simularii nu a depasit timpul total de simulare, parcurgem intreaga lista pentru a trimite clasei ***Scheduler*** clientii ce au arrivalTime-ul egal cu timpul curent de simulare, pentru a fi distribuiti in cozi.  
Cand timpul de simulare s-a incheiat si toate cozile sunt goale, adica nu mai exista clienti de procesat, se apeleaza metoda **printResults**() pentru a afisa utilizatorului rezultatele calculate in urma simularii.

**Diagrama UML de clase**



**Diagrama UML de pachete**

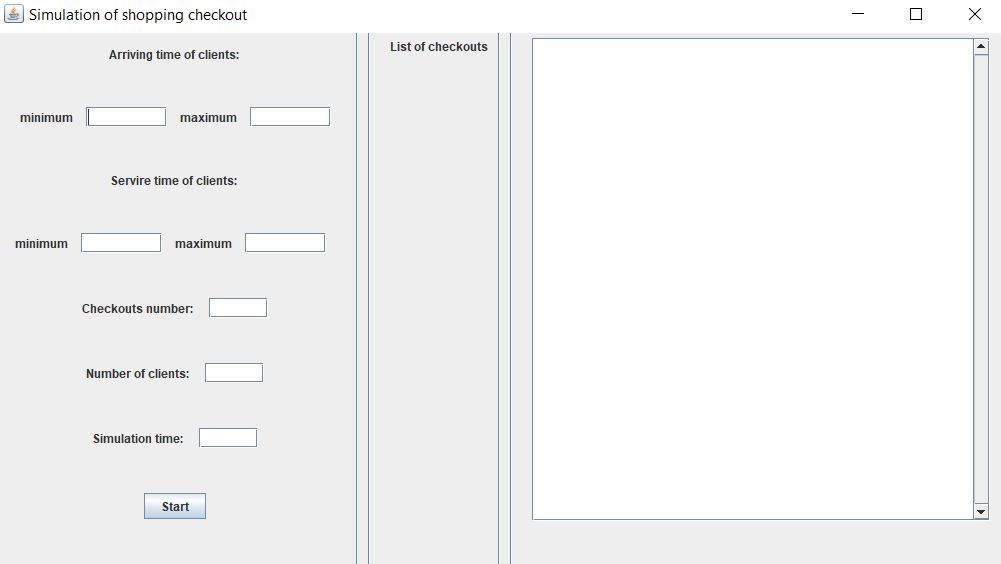


**Structuri de date** In ceea ce priveste structurile de date folosite, am decis sa folosesc :  
 - ArrayBlockingQueue(), prentru a retine clientii de la o anumita coada, intrucat folosind Thread-uri, este o clasa deja implementata si sincronizata, fapt care ne ajuta enorm in ceea ce priveste implementarea aplicatiei.   
 - LinkedList(), pentru a retine clientii generati la inceput de aplicatie, inainte de a-i trimite pe toti spre cozile de marcat. Motivul alegerii este dat de usurinta de a lucra cu o lista, in special datorita faptului ca poate fi sortata.  
 - AtomicInteger, intrucat este o un tip de data sincronizat implicit, care din nou, ne ajuta enorm in dezvoltarea aplicatiei, fara a avea pe cap problema sincronizarii.

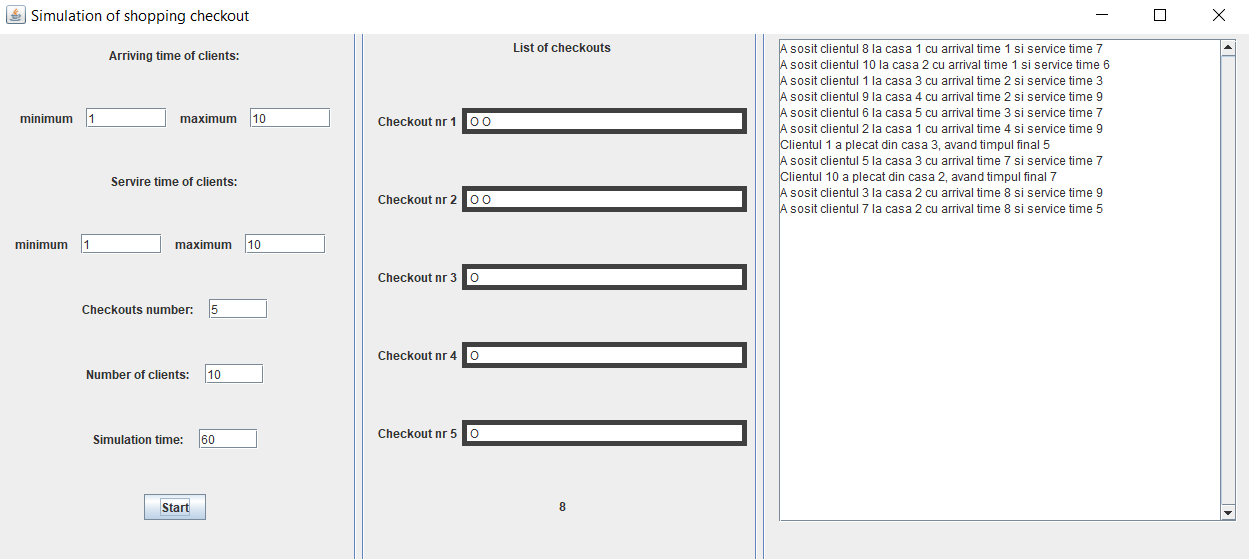
**Pachete folosite** Aplicatia este impartita in trei pachete: **View**, **Simulation** si **Shop**. Intrucat nu este posibil sa avem un magazin fara clienti si case de marcat, am hotarat ca aceste doua clase sunt inrudite intre ele, fapt pentru care ar trebui sa faca parte din acelasi pachet, numit ***Shop***.   
 Totodata, pachetul **Simulation** este format din clasele: **SimulationManager**, **Scheduler**, care functioneaza impreuna si au ca target simularea in timp real a unui magazin. Clasa **Scheduler**, dupa cum a fost descrisa mai sus, are rol de a distribui clientii in cozi, iar clasa **SimulationManager** are grija de intervalul de simulare si trimite clienti spre clasa **Scheduler**. Asadar, cele doua clase fiind inrudite si functionand impreuna, am hotarat sa faca parte din pachetul ***Simulation***.  
 Ultimul pachet se numeste ***View*** si este alcatuit doar de clasa care da viata interfetei grafice.

**Algoritmi** Fiind o aplicatie multi-threading, nu a fost necesara implementarea a mai multor algoritmi, dar totusi am avut nevoie de cativa.  
 Unul din algoritmii implementati il reprezinta gasirea cozii cu cei mai putini clienti. Parcurgand vectorul de liste din clasa Scheduler, am retinut de fiecare data care este coada cu cei mai putini clienti, rezultand astfel in final, coada cautata.  
 Un alt algoritm il reprezinta generarea de clienti random, la inceputul aplicatiei. Intr-o bucla, am creeat numarul de clienti doriti de utilizator, fiecare dintre clienti avand un timp de sosire si un timp de procesare generat aleator. Folosind clasa **Random**(), am generat un numar intre limita inferioara a intervalului si limita superioara a acestuia.  
 Algoritmii pentru determinarea timpilor medii de asteptare, procesare si timpului mediu in care cozile sunt goale, m-am gandit in felul urmator:   
 **- timpul mediu de asteptare** : Daca intr-o coada, avem un singur client, timpul sau de asteptare este egal cu timpul sau de procesare. Daca intre timp, mai apare un client la coada, acest client va avea timpul de asteptare egal cu timpul sau de procesare adunat cu timpul de asteptare al clientului din fata sa. Dat fiind acest lucru, am adunat timpul de asteptare pentru fiecare client in parte si am impartit rezultatul la numarul total de clienti.  
 **- timpul mediu de procesare**: Am adunat timpii de procesare pentru fiecare client in parte, iar rezultatul l-am impartit la numarul total de clienti.  
 **- timpul mediu in care cozile sunt goale:** In clasa Checkout, in metoda run() aferenta Thread-ului, am implementat o bucla while in care, de fiecare data cand coada este goala, se opreste Thread-ul pentru o secunda si se contorizeaza timpul cu 1. Astfel, am adunat timpii in care fiecare coada este goala, iar rezultatul l-am impartit la numarul total de cozi, rezultand un timp mediu.  
 **- ora de varf (peak hour):** M-am gandit asa: la fiecare adaugare de client sa numar cate persoane se afla in toate cozile in momentul acela. Astfel, retinand de fiecare data numarul maxim de clienti si secunda la care avea loc acest lucru, a rezultat in final ora de varf cautata.

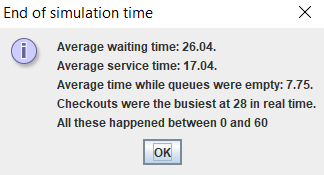
1. **Implementare** In principiu, metodele importante au fost discutate in subsectiunea **algoritmi**, asadar voi descrie **implementarea interfetei utilizator.** La prima rulare a aplicatiei, va aparea urmatoarea interfata grafica.   
    Panoul din stanga, contine informatiile pe care utilizatorul trebuie sa le introduca pentru a porni aplicatia. In panoul din mijloc, dupa cum am explicat intr-o sectiune precedenta, momentan nu se afla nimic intrucat numarul de cozi va fi generat la apasarea butonului “**Start**”. In panoul din dreapta, se afla LOG-ul de informatii generat in timpul simularii. Vor fi afisati clientii care intra in cozi, respectiv momentul cand acestia au fost procesati si parasesc coada.



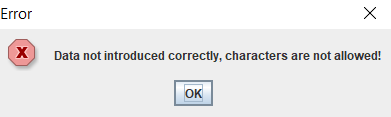
In urmatoarea poza, se poate vedea aplicatia ruland, dupa introducerea datelor dorite de catre utilizator si apasarea butonului “Start”. Au fost afisate cele 5 cozi, in interiorul carora, se afla clientii. Un client este specificat printr-un cerculet.  
 In panoul din dreapta se observa sosirea clientilor, specificata prin arrival time-ul fiecaruia dintre ei. Totodata, se remarca si plecarea clientilor din coada, deja procesati.  
 In panoul din mijloc, in partea de jos a acestuia, am afisat in timp real secunda curenta a aplicatiei, pentru a putea urmari exact timpul in care sosesc sau timpul in care parasesc clientii coada.



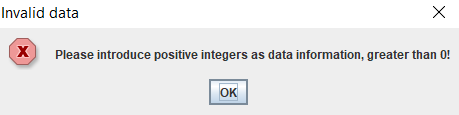
La finalul simularii, utilizatorului ii va fi adresat un mesaj in care vor fi afisati timpii medii rezultati in urma simularii.



***Tratarea posibilelor exceptii***   
 **Introducerea de caractere de catre utilizator**



**Introducerea de numere negative de catre utilizator**



Interfata grafica contine un singur ascultator, care a fost pus pe butonul **Start**. Acesta fiind apasat, se incepe executia aplicatiei: sunt extrase datele introduse in field-urile puse la dispozitia utilizatorului, se verifica corectitudinea acestora, iar in caz afirmativ, sunt generate cozile de asteptare la casele de marcat si incepe simularea in timp real a aplicatiei, urmand ca la finalul timpului de simulare si golirea tuturor cozilor, sa se afiseze rezultatele obtinute.

1. **Rezultate** In ceea ce priveste rezultatele obtinute, in functie de informatiile introduse de utilizator se va genera o serie de rezultate diferite.   
    Rezultatele partiale si corectitudinea acestora pot fi urmarite pe parcursul simularii, intrucat se pot compara timpii de sosire si de procesare calculati de aplicatie, aleator, cu timpul real in care ruleaza aplicatia.
2. **Concluzii**

Foarte sincer, inaintea acestui proiect nu stiam ce face metoda sleep din clasa Thread. Am  
invatat foarte multe, in special lucruri legate de clasele foarte friendly, deja implementate : AtomicInteger si ArrayBlockingQueue.  
 O posibila imbunatatire a aplicatiei o reprezinta clar interfata grafica. Mi-ar fi placut sa implementez cozile in asa fel incat sa se poata vedea cum clientii se deplaseaza spre fiecare coada, si cum acestia parasesc coada in urma procesarii, dar posibil pe viitor.

1. **Bibliografie**

Generarea diagramelor : [www.draw.io](http://www.draw.io)  
Informatii suplimentare despre utilizarea Thread-urilor: <https://www.caveofprogramming.com/categories/java-multithreading/>