**DOCUMENTATIE TEMA 2**

**APLICATIE PENTRU SIMULAREA ACTIVITATII COZILOR**

**Student: Cibu Mihai**

**Grupa: 30226**

**Profesor Laborator: Marin Oana Andreea**

**Cuprins:**

1. Obiective
   1. Obiectiv principal
   2. Obiective secundare
2. Analiza Problemei, modelare, scenarii, cazuri de utilizare
   1. Analiza problemei, modelare
   2. Scenarii
   3. Cazuri de utilizare

3. Proiectare

4. Implementare

5. Rezultate

6. Concluzii si Dezvoltari Ulterioare

7. Bibliografie

**1.Obiective**

**1.1 Obiectiv principal**

Propuneti, proiectati si implementati o aplicatie cu scopul de a analiza cozile, bazandu-se pe determinarea si minimizarea timpului de asteptare al clientilor.

Cozile sunt intalnite in toate domeniile din viata de zi cu zi, iar principalul lor obiectiv este de a furniza pentru clienti un loc de a astepta inainte de a primi serviciul dorit. Gestionarea acestor cozi se bazeaza pe un sistem care are ca obiectiv minimizarea timpului mediu in care clientii sunt in asteptarea serviciului dorit. O modalitate de minimizare presupune crearea mai multor cozi( implica procesarea concurenta a clientilor de catre „servere” diferite), dar aceasta abordare creste costul procesarii. Prin adaugarea unui nou server, clientii sunt redistribuiti spre serverul disponibil.

Aplicatia presupune simularea sosirii la servere pentru o anumita deservire a unui numar de clienti, asteptarea, procesarea serviciului propriu-zisa, si in final parasirea serverului. Pentru a calcula timpul de asteptare trebuie sa stim momentul de timp cand clientul a ajuns la coada, timpul de finalizare a actiunii si durata timpului de procesare a serviciului. Timpul de sosire si cel de procesare sunt specifice fiecarui client, iar timpul de finalizare depinde de numarul de servere, numarul de clienti care sun deja in asteptare si serviciul dorit.

Pentru simularea activitatii descrise se va folosi interfata grafica, utilizatorul introducand datele necesare simularii si se va putea vizualiza procesul descris.

De asemenea, aplicatia trebuie sa respecte paradigmele Programarii Orientate-Obiect, clasele si metodele trebuie sa respecte numarul de linii impus (maxim 300, respectiv 30) si sa se respecte conventiile Java legate de denumirea claselor, metodelor si variabilelor.

**1.2 Obiective secundare**

|  |  |  |
| --- | --- | --- |
| **Obiectiv secundar** | **Descriere** | **Capitol** |
| Dezvoltarea de use case-uri si scenarii | Prezentarea modalitatilor de folosire si functionare a aplicatiei de simulare a cozilor in cazurile posibile plus situatiile in care utilizatorul greseste la introducerea datelor sau introduce date invalide | **2** |
| Alegerea de structuri de date | Selectarea structurilor de date abstracte folosite pentru stocarea datelor | **3** |
| Impartirea pe clase | Structurarea in clase si pachete | **4** |
| Dezvoltarea algoritmilor | Implementarea propriu-zisa a procesului in cod Java | **4** |
| Implementarea solutiei | Prezemtarea pe larg a solutiei si a metodelor folosite | **4** |

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

**2.1 Analiza problemei, modelare**

Pentru asigurarea corectitudinii datelor de intrare se vor face anuimite verificari pe datele introduse de utilizator. Pentru ca aplicatia sa functioneze corect datele introduse trebuie sa fie numere intregi reprezentand valori corespunzatore etichetei fiecarui camp(intervalele minimum - maximum sa fie valide, numarul de cozi sa fie cel putin 1, timpul de simulare trebuie sa fie mai mare decat valoarea maxima a timpului de sosire, altfel vor exista clienti care ajunge cand simularea este deja terminata si nu vor putea intra la coada).

**2.2 Scenarii**

Pe parcursul implementarii am verificat mai multe scenarii posibile, atat favorabile cat si nefavorabile. Scenariile favorabile se refera la introducerea corecta a datelor (dupa formatul prezentat mai sus) si observarea rezultatului in campurile corespunzatoare( modificara in timp real a cozilor, afisara de mesaje corespunzator fiecarei inserari in coada, respectiv parasirea cozii si vizualizarea statisticilor la sfarsitul simularii). In ceea ce priveste scenariile nefavorabile s-au facut verificari legat de omiterea introducerii de date in toate campurile(se va observa un mesaj de eroare, introducerea unui numar necorespunzator de de cozi, sau posibilitatea ca anumiti clienti sa nu poata lua parte la simulare).

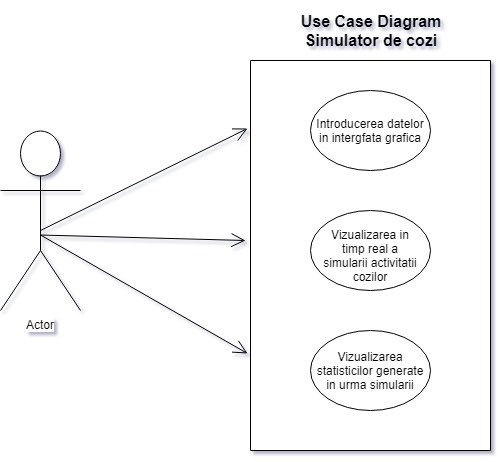
**2. 3 Cazuri de utilizare:**

Use case: Simularea activitatii a doua case/servere.

Actor: Utilizatorul care doreste simularea activitatii de distribuire a clientilor in cozi.

Preconditii: Utilizatorul introduce in campurile specificate din interfata grafica date corespunzatoare intervalelor de timp, numarului de clienti si numarului de case. Dupa introducerea datelor se va apasa butonul „Submit Data”. Acum se poate da start simularii prin apasarea butonului „Start Simulare”. In zona rezervata log-urilor vor aparea mesaje reprezentand date despre clientii care intra in case, specificandu-se timpul de sosire, timpul de procesare si casa la care a fost distribuit. Postconditii: Se va observa ca in interfata grafica vor aparea casete speciale reprezentand casele(in functie de numarul introdus). De asemenea se va putea observa in timp real inserarea si stergerea persoanelor de la case in campul specific fiecarei case.

Flow: Datele introduse de utilizator vor fi preluate, verificare, procesate si in cazul in care se indeplinesc conditiile respective se va incepe simularea..



**3. Proiectare**

In ceea ce priveste proiectarea propriu zisa a proiectului, am structurat aplicatia in pachete, fiecarui pachet fiindu-i intrebuintate responsabilitati comune/ inrudite. Modelul este reprezentarea obiectuala a clientilor, gui este reprezentat de interfata grafica, care faciliteaza interactiunea utilizatorului cu aplicatia, iar simulaarea face legatura intre cerintele utilizatorului transmise prin intermediul interfetei si modelul. Tot in simulare se implementeaza ascultatorii pentru butoanele din interfata. Pachetele create sunt:

* simulare: compus din clasele Scheduler si SimulationManager se ocupa cu gestiunea clientilor si face legatura intre model si interfata grafica (preluarea datelor introduse de utilizator, analiza lor).
* model: cu clasele reprezentative Client si Casa. Acest pachet reprezinta modelul conceptual pe care se bazeaza aplicatia.
* gui: in cadrul acestui pachet se realizeaza implementarea interfetei grafice.

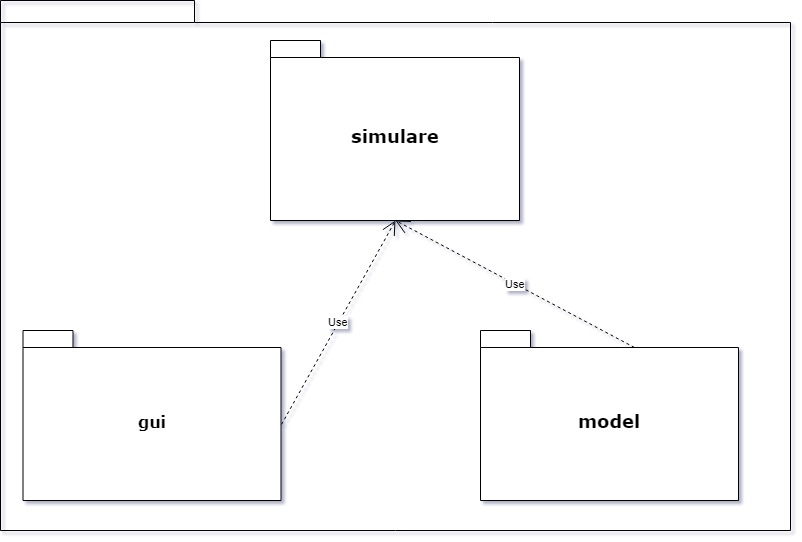


Fig2. Diagrama de pachet

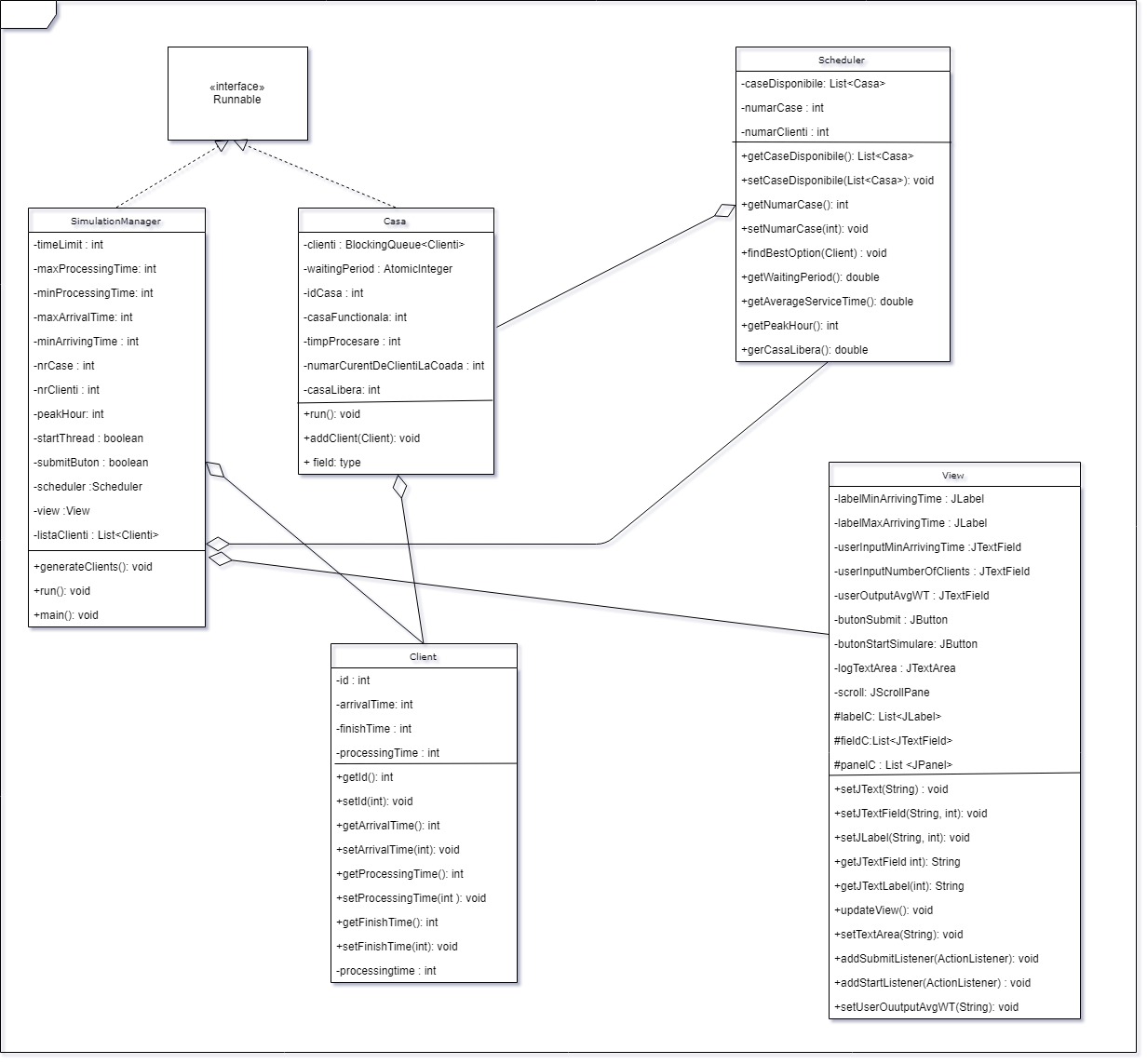


Fig3. Diagrama de clase

In ceea ce priveste structurile de date, am folosit BlockingQueue<Clienti> pentru stocarea clientilor in casele de marcat. Am ales sa folosesc aceasta structura deoarece ea este specifica lucrului cu threaduri(mai multe fire de executie), avand mecanismele de sincronizare deja implementate si nefiind necesare sincronizari. De asemenea sunt implementate si metode pentru extragerea din coada si adaugarea in coada, facilitandu-ne astfel scrierea de cod care necesita aceste mecanisme.

De asemenea am folosit List si ArrayList pentru stocarea de liste ce nu necesita sincronizare. Pentru a crea threadurile, clasele responsabile implementeaza interfata Runnable, si doar dupa se instantiaza obiectul de tip Thread. Am ales aceasta modalitate de creare a threadurilor deoarece prin extinderea clasei Thread, clasa copil nu mai poate extinde alta clasa (in Java nu este permisa mostenirea multipla). In implementarea aplicatiei nu s-au folosit algoritmi predefiniti, ci s-au implementat propriile metode si algoritmi.

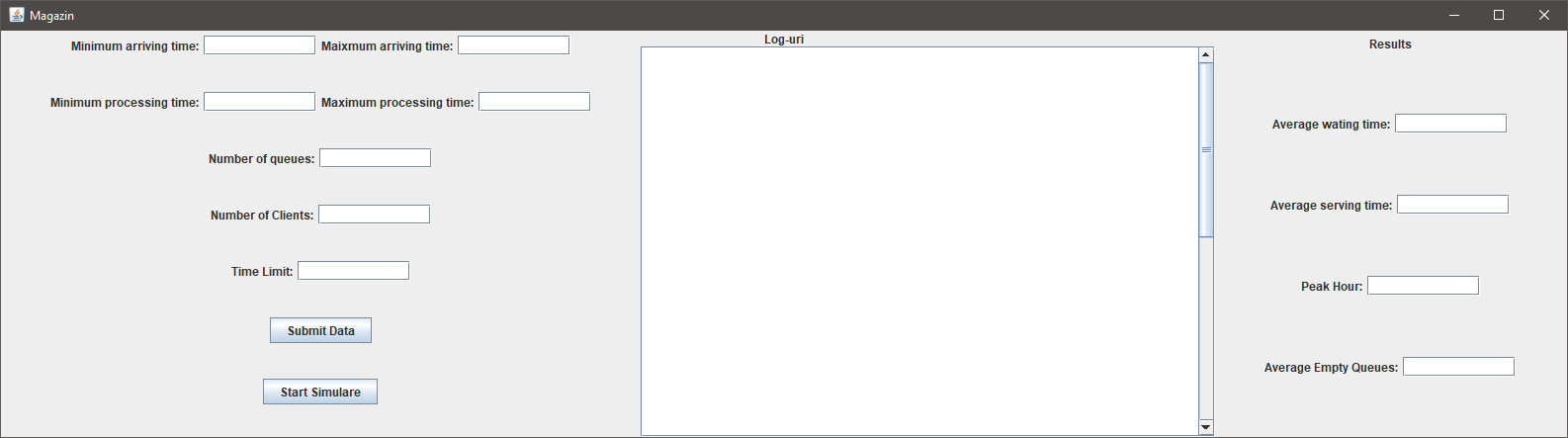
**4. Proiectare**

In continuare voi descrie fiecare clasa, impreuna cu campurile si metodele importante, inclusiv implementarea interfetei utilizator.

* **Client**: Clientul reprezinta entitatea principala pentru modelarea aplicatiei, scopul aplicatiei fiind minimizarea timpului petrecut de client in asteptarea serviciului dorit, astfel este necesar ca in aceasta clasa sa stocam informatii legate de id-ul fiecarui client(identificator unic), timpul la care clientul a ajuns la coada (arrivalTime), timpul pe care clientul il va petrece in scopul obtinerii serviciilui dorit (processingTime), si timpul final, reprezentand timpul la care clientul paraseste casa dupa asteptarea la coada si mai apoi primirea serviciului dorit. Legat de metodele acestei clase, pe langa cele accesaore si mutatore avem metoda toString() pentru afisarea in interfata grafica detaliile clientului.
* **Casa**: Aceasta clasa contine reprezentarea propriu zisa a concepului de coasa de asteptare si procesare a clientului. Stocarea clientilor se face sub forma de BlockingQueue, astfel fiecare casa are o coada asociata. Aceasta clasa implementeaza interfata Runnable, ea fiind un thread. Astfel, fiecare casa care va fi instantiata va lucra independent fata de cealalta fiecare executand metoda run(). Aceasta modalitate de lucru in paralel permite minimizarea timpului petrecut de catre clienti in asteptarea serviciului dorit. Campurile acestei clase sunt coada de clienti, timpul de asteptare la coada(de tipul AtomicInteger, tip specific lucrului cu threaduri) care specifica cat de mult ar trebui un client sa astepte la coada respeectiva pentru a-si primi serviciul. De asemenea fiecare casa are un id, un timp care contorizeaza numarul de secunde in care casa este goala, un numar curent de clienti in coada, un timp de procesare al casei, echivalent cu suma timpilor de proceare a tuturor clientilor care au trecut prin acea casa si un timp care specifica cat de mult au asteptat clientii la casa respectiva. Principalele metode ale acestei clase, pe langa cele accesoare si mutatoare sunt:
* run: metoda pe care clasa este obligata sa o suprascrie, deoarece implementeaza interfata Runnable. In cadrul acestei metode se verifica daca coada este goala, caz in care threadul este pus pe sleep pentru o secunda. In cazul in care casa nu este goala se extrage primul client din BlockingQueue, reprezentand primul client ajuns la casa, se pune thread-ul pe sleep un numar de secunde proportional cu timpul de procesare al clientului extras, waitingPeriod-ul este decrementat cu timpul de procesare al clientului care paraseste casa si pe urma se face update la interfata grafica.
* addClient: in cadrul acestei metode se adauga la sfarsitul cozii Clientul primit ca si parametru si se face update la informatiile necesare(waitingPeriod - este incrementat cu timpul de procesare al noului client, timpProcesare, numarulCurentDeClientiInCoada)
* **Scheduler:**  Aceasta clasa este responsabila cu distribuirea clientilor in cozi conform cu timpul de asteptare al fiecarei cozi, incercandu-se minimizarea timpului de asteptare pentru clienti. Ca si variabile instanta, aceasta clasa are o lista de case(fiecare casa fiind un thread), numarul total de case si numarul de clienti. In constructorul acestei clase se instantiaza casele, se adauga fiecare casa in lista de case si se da start threadurilor. Pe langa metodele accesoare si mutatoare urmatoarele metode descriu aceata clasa:
* findBestOption() : aceasta metoda este apelata atunci cand se doreste inserarea unui nou client in coada, iar aceasta metoda cauta locul optim pentru inserare. Cautarea se face in functie de timpul de asteptare curent la casa, si se alege casa cu timpul minim, in scopul minimizarii timpului de asteptare pentru client. Dupa ce s-a gasit pozitia corepunzatoare se apeleaza metoda de addClient() din clasa Casa, care realizeaza propriu zis inserarea. Tot in cadrul acestei metode se apeleaza metodele corespunzatoare pentru actualizarea interfetei grafice cu datele clientului nou introdus.
* getWaitingPeriod(): aceasta metoda este apelata dupa incheierea simularii si este folosita la calcularea timpului mediu de asteptare la cozi pentru o anumita simulare. Se itereaza prin lista de case si se insumeaza timpii in care s-a asteptat la fiecare casa.
* getAverageServiceTime(): aceasta metoda returneaza numarul mediu de secunde petrecut de clienti atunci cand sunt serviti. Pentru a afla acest timp se itereaza prin lista de case si se insumeaza timpii de procesare, urmand a fi impartiti la numarul total de clienti.
* getPeakHour(): la fiecare moment din timpul de simulare se afla numarul total de clienti din cozi si la final se returneaza maximul dintre acesti timpi.
* getCasaLibera(); la finalul simularii se itereaza prin lista de cozi si se insumeaza timpul fiecarei case in care a fost libera, returnandu-se numarul mediu de secunde

in care casele sunt libere pe parcursul simularii.

* **SimulationManager:** Reprezinta clasa de baza care implementeaza interfata Runnable, iar in metoda main este instantiat, creandu-se astfel thread-ul principal al aplicatiei, care controleaza restul componentelor. Variabilele instanta ale acestei clase sunt timpul limita pentru simulare, intervalul de genereare a timpului de procesare, respectiv de sosire al clientilor, numarul de case(servere), numarul de clienti (aceste date vor fi introduse de catre utilizator in interfata grafica). De asemenea ca si variabila instanta avem un obiect de tip Scheduler, raspunzator pentru inserarea clientilor in coada si determinarea timpilor statistici. De asemenea aceasta clasa face legatura dintre interfata grafica si modelul propriu zis al aplicatiei, astfel incat se declara si o variabila instanta de tipul View, permitand citirea datelor din interfata si actualizarea acesteia odata cu inceperea simularii. In cadrul acestei clase se declara si clasele care implementeaza ActionListener pentru citirea datelor din interfata grafica si validarea lor. Pe langa metodele accesoare si mutatoare, principalele metode implementate in aceasta clas sunt:
* run(): metoda care implementeaza actiunile thread-ului principal. In cadrul acestei metode se asteapta introducerea datelor de catre utilizator si apasarea butonului de start. Cand s-a apasat butonul de start incepe simularea propriu-zisa. Initial se va genera o lista aleatoare de clienti in functie de datele introduse, dupa care clientii sunt introdusi in cozi in functie de timpul de sosire si timpul de simulare. Tot in cadrul acestei metode, dupa terminarea timpului de simulare se apeleaza metodele corespunzatoare din clasa View pentru afisarea datelor cu caracter statistic in interfata grafica.
* generateClients(): metoda responsabila cu generarea unui anuir numar de clienti, pentru fiecare client generandu-se timpul de sosire si timpul de procesare. Acesti timpi se genereaza in intervalele specificate, valorile fiind generate aleatoriu. Dupa generarea numarului corespunzator de clienti acestia sunt sortati in functie de timpul de sosire.
* **View:** Aceasta clasa este responsabila cu interfata grafica, iar ca variabile instanta avem Jlabel-uri, TextField-uri, in care utilizatorul va introduce datele corespunzatoare etichetelor, un JTextArea in care se vor afisa inserare, respectiv stergerea clientilor , doua Jbutton-uri pentru inserarea datelor introduse si pentru a da start simularii si TextField-uri pentru fiecare casa, putandu-se observa evolutia acesteia. Metodele principale ale acestei clase sunt legate de citirea String-urilor din TextField-uri, setarea de mesaje in TextField-ul pentru rezultat, si cate o metoda pentru adaugarea de ascultator pentru fiecare JButton. In constructorul acestei clase se seteaza aspectul ferestrei. Fereastra este construita din mai multe JPanel-uri, in care se adauga pe rand elementele decrise mai sus.



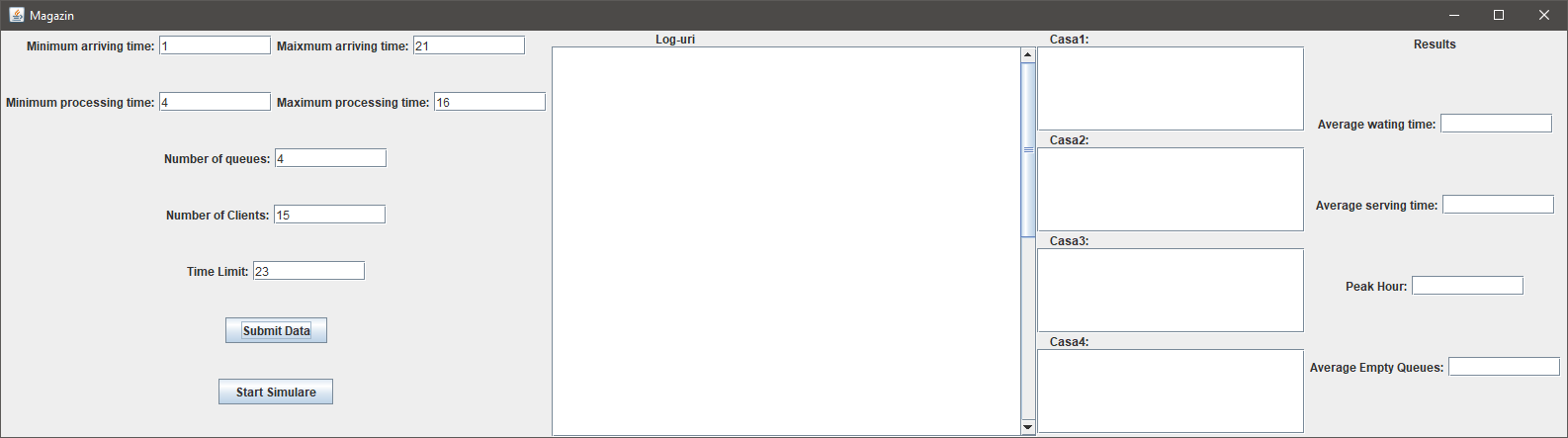


Fig.4 Interfata Grafica (inainte si dupa apasarea butonului de Submit)

Se observa in Fig. 4 campurile in care utilizatorul trebuie sa introduca datele, sugerate de etichetele lor, campul de afisare al rezultatului si casele corespunzatoare numarului introdus. Dupa introducerea datelor este obligatoriu apasarea butonului „Submit Data”, care va actualiza interfata grafica astfel incat evolutia caselor se va fi vizibila in timp real.

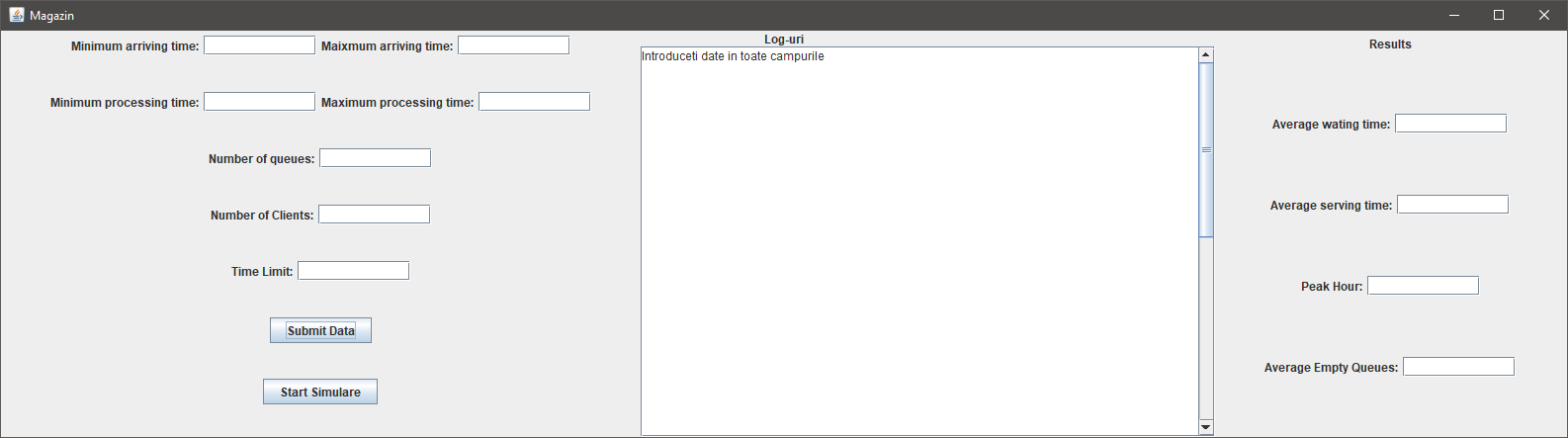


Fig.5 Utilizatorul nu introduce date in campurile specificate

**5. Rezultate**

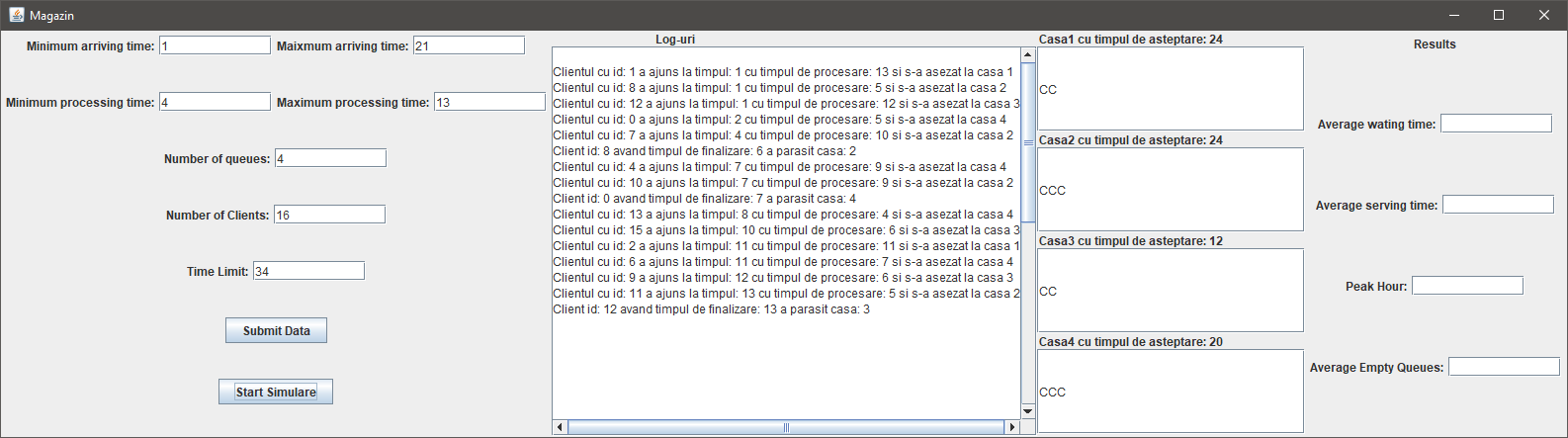


Fig.5 Simularea corecta a aplicatiei

Pentru verificarea corectitudinii aplicatiei se vor introduce date valide in interfata grafica, se va simula activitatea si se va urmari evolutia cozilor(evolutia este vizibila atat in TextField-ul fiecarei case cat si sub forma de text).

**6. Concluzii**

Pe parcursul proiectarii, dezvoltarii si implementarii acestei teme mi-am exersat abilitatile de a scrie cod in Java, am descoperit lucruri noi legate de implementarea anumitor metode, am invat despre thread-uri, cum se folosesc, la ce ajuta si cat de folositoare sunt pentru problemele din viata de zi cu zi, am descoperit BlockingQueue si AtomicInteger si cat de folositoare sunt pentru lucrul cu thread-uri.

**7. Bibliografie**

<https://stackoverflow.com>

draw.io/ - pentru desenarea diagramelor UML