**Documentație Tema 2**

Realizator:

Stroia Lucian Dorin

**Cuprins**

1. *Obiectivul lucrării*
2. *Scopul realizării aplicație*
3. *Abordarea problemei*
4. *Diagrame UML*
5. *Implementarea programului*
6. *Rezultatele obținute*
7. *Concluzii*
8. *Bibliografie*

**Obiectivul lucrării**

*Obiectivul acestei lucrări este reprezentat de simularea mai multor cozi care primesc un anumit număr de clienți, acești clienți fiind distribuiți la cozi în funcție de timpul pe care clientul este nevoit să îl aștepte în momentul în care acesta este generat. Programul a fost realizat prin lucrul cu Thread-uri, fiind esențiale pentru generarea automată si distribuirea clienților la cozi astfel încât să aștepte cât mai puțin.*

*Ne putem imagina evenimentele din viața reală, atunci când mai multe persoane ajung la un magazin, după ce fac anumite cumpărături, ajung la o casă de marcat unde trebuie să achite. Modul în care aleg casa la care să meargă reprezintă o Strategie, care va fi detaliată pe parcursul documentației.*

**Scopul realizării acestei aplicații**

*Această aplicație a fost creată pentru a observa în timp real evoluția simulării mai multor cozi care primesc niște clienți, fiind vizibile detalii precum timpul în care clientul ajunge la magazin, timpul de procesare respectiv timpul de terminarea tuturor activităților, fiind momentul în care clientul părăsește magazinul.*

*Vizualizarea simulării este posibilă prin intermediul unei interfețe, realizată în doua etape, mai exact, etapa inițială, în care sunt introduse datele necesare pentru crearea aplicației, si etapa finală când utilizatorul nu mai introduce nimic în program, ci privește evoluția cozilor, fiind puse în evidență informații despre clienți dar si timpul mediu de așteptare a acestora, până când urmează să fie procesați.*

**Abordarea problemei**

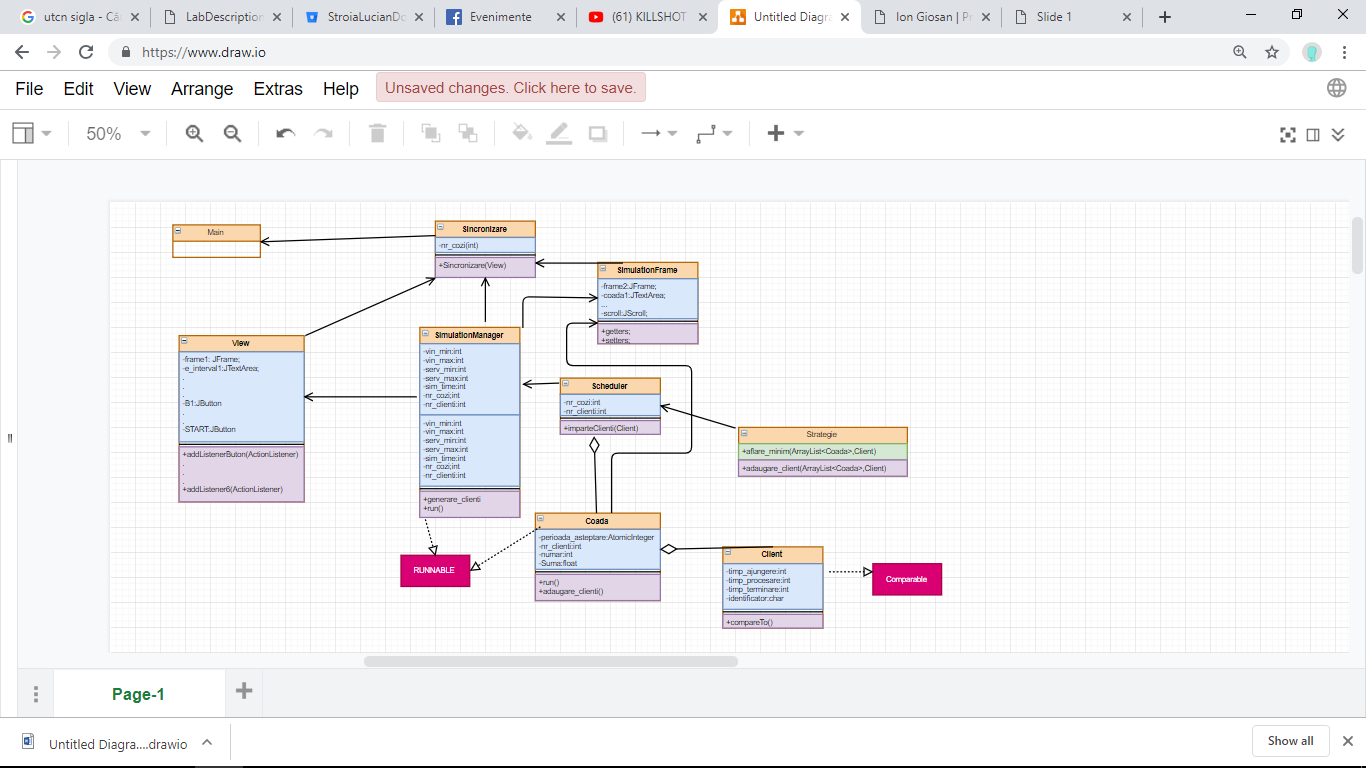
*Pentru realizarea programului, sarcinile interne ale acestuia au fost distribuite astfel încât să se realizeze o sincronizare si egalitate in ceea ce privește fiecare lucru pe care casele de marcat îl aveau de făcut.*

*Practic, prin intermediul Thread-urilor, in momentul in care am creat o coadă (casă sau server, poate fi pusă în evidență în mai multe moduri, având însă aceeași semnificație) coada respectivă a fost pusă într-un Thread, acesta fiind pus pe sleep sau pe start în funcție de proprietățile clienților.*

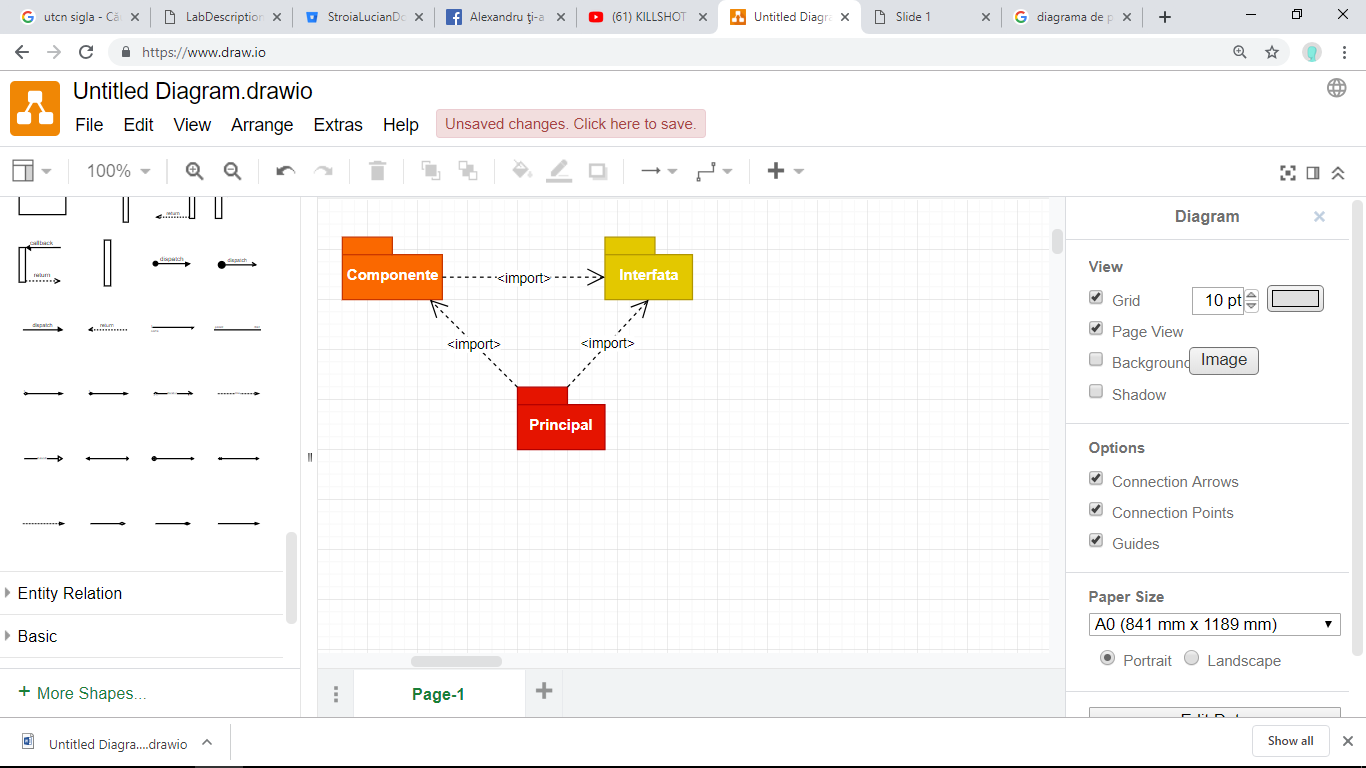
*Toată acțiunea este controlată de un Thread mai mare, care primește șirul de Taskuri și îi distribuie la fiecare coadă în funcție de timpul de așteptare, fiind posibil să facem afirmația că s-a folosit procedeul de Multi-Threading, un Thread controlând mai multe Thread-uri.*

**Diagrame UML**

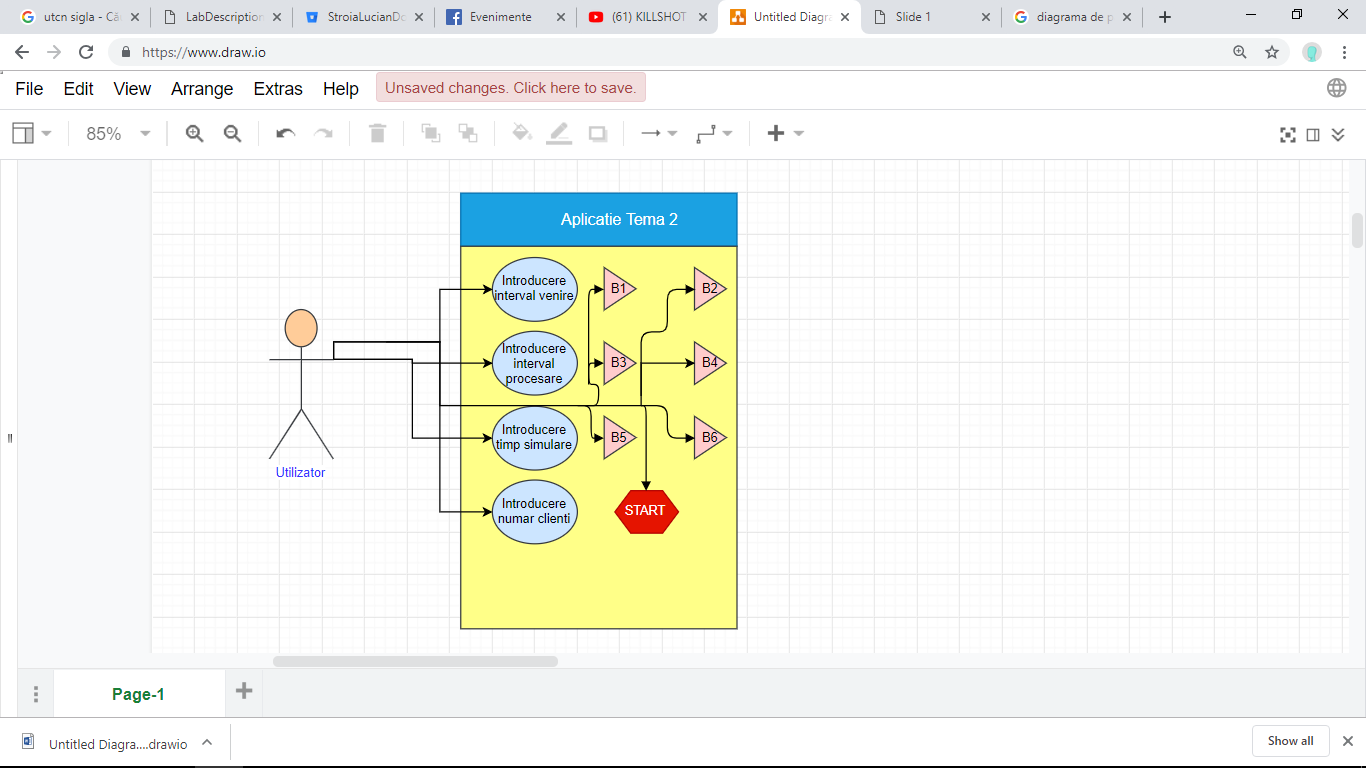
*1.DIAGRAMA DE CLASE*



*2.DIAGRAMA DE PACHET*



*3.DIAGRAMA USE-CASE*



**Implementarea programului**

*Implementarea propriu-zisă a programului a fost organizată în 3 pachete, fiecare pachet având o funcționalitate și niște obiective de realizat din momentul în care a pornit aplicația.*

1. ***Pachetul COMPONENTE***

*Acest pachet conține mai multe clase care fac posibilă rularea aplicației, fiind elementele cu care programul lucrează în mod intern.*

*CLASA CLIENT*

*În clasa Client, avem ca și variabile de instanță*

*timpul de ajungere al clientului la magazin, timpul de procesare, adică face referire la timpul pe care fiecărui client îl petrece la casă, după ce acesta a fost distribuit la una din ele, timpul de terminare, care face referire la timpul în care clientul a plecat practic din magazin, și un identificator, care e de tipul Char, acest identificator făcând referire la ”numele” clientului, fiind reprezentat doar pe o literă. Prin intermediul acestui identificator, se poate observa evoluția clientului la coadă, în interfața grafică fiind posibilă recunoașterea lui prin intermediul literei specifice.*

*Avem definit un constructor care primește ca și parametri timpul de ajungere, timpul de procesare si identificatorul. Timpul de ajungere si timpul de procesare este generat aleatoriu, între 2 capete de interval specific fiecărei variabile de instanță.*

*Această clasă implementează interfața Comparable, pentru a se face posibilă sortarea clienților, care au fost generați si stocați într-o structura de tipul LIST<>, în funcție de timpul de procesare pe care aceștia l-au primit într-un mode aleatoriu.*

*Sunt create 2 metode de toString(), una dintre acestea fiind necesară pentru afișarea datelor clientului in fereastra de informații a celui de-al doilea frame, iar a doua metodă toString(), are rolul de a afișa identificatorul clientului în coada specifică din cel de-al doilea frame.*

*CLASA COADA*

*Această clasă are ca și variabile de instanță un ArrayBlockingQueue care primește ca elemente niște obiecte de tipul Client, un AtomicInteger, care pune în evidență timpul de așteptare pe care trebuie să îl petreci la casă atunci când ajungi, numărul de clienți, cele 2 vederi, una din primul frame unde utilizatorul introduce datele necesare pentru rularea programului, si una din cel de-al doilea frame, în care utilizatorul observă evoluția a unui număr de cozi ales anterior de către acesta.*

*În constructor, sunt inițializate variabilele de instanță și este*

*alocat ArrayBlockingQueue-ul cu numărul maxim de clienți pe care coada respectivă l-ar putea primi.*

*Este definită o metodă de adăugare a clienților în coadă, metodă care este apelată în altă clasă, adăugarea clienților făcându-se in funcție de timpul de așteptare pe care aceștia urmează să îl petreacă la coadă. În această metodă, se setează perioada de așteptare a clientului, se adaugă clientul în coadă si se actualizează perioada de așteptare specifică cozii respective, informație necesară pentru următorul client care este distribuit la casa respectivă. Metoda aceasta returnează numărul cozii, pentru a fi posibilă actualizarea informațiilor din cel de-al doilea View.*

*Această clasă implementează Runnable, fiind necesară și obligatoriu de definit metoda run(), metodă în care practic, este scos din coadă fiecare client, si este pus thread-ul pe sleep pentru un timp egal cu timpul de procesare al clientului înlăturat din coadă, astfel, coada noastră simulează servirea propriu-zisă a clientului prin punerea pe sleep, fiind permisă procesarea celui de-al doilea client doar după trecerea acestei perioade de timp. Practic, clientul ordonează cozii cât timp să fie pusă pe sleep, iar apoi pleacă, coada revenind după trecerea timpului indicat de către client. Aici se calculează și perioada medie de așteptare, și sunt aduse actualizări ale celui de-al doilea frame, în care utilizatorul observă evoluția cozilor*

*CLASA SCHEDULER*

*In această clasă, în interiorul constructorului sunt create cozile propriu-zis si sunt puse într-un ArrayList de cozi, pentru a fi folosite pe parcursul programului. Tot aici, după crearea cozii si adăugarea în ArrayList-ul de cozi, fiecare coadă este introdusă într-un thread, acesta urmând să fie porni cu metoda thread.start().*

*Este inițializată si o strategie, necesară pentru deciderea numărului specific cozii în care clientul urmează să fie distribuit, toată această strategie bazându-se pe timpul de așteptare pe care clientul trebuie să îl aștepte.*

*Clasa Scheduler are implementată si metoda de imparteClienti(), care apelează metoda de adăugare client din Strategie.*

*CLASA STRATEGIE*

*Această clasă are un rol foarte important, deoarece aici se decide în ce coadă să fie distribuit clientul care ajunge la magazin. Acesta nu dorește să petreacă mult timp la coadă, motiv pentru care timpul de așteptare care este indicat de către fiecare coadă este un aspect primordial pentru fiecare client.*

*Metoda de aflare\_minim() primește ca și argument un ArrayList de cozi, și indică returnează numărul specific al cozii care are timpul de așteptare cel mai mic, și apelează metoda din coadă, pentru adăugare a clientului în coadă.*

*CLASA SIMULATIOMANAGER*

*Această clasă unește componentele si preia informațiile din Vederea inițială, lucru care se întâmplă încă de la apelarea constructorului SimulationManager() care primește ca și parametri numărul de cozi, vederea principală și vederea secundară. Tot aici, în interiorul constructorului se apelează si constructorul din Scheduler, cu nr\_cozi, nr\_clienti, vederea principală si vederea secundară, iar mai apoi se apelează metoda internă din clasa SimulationManager, metoda de generare a clienților care stochează clienții generați într-un ArrayList de Clienți, acest ArrayList fiind necesar pentru adăugarea cliențior în coadă.*

*Clasa SimulationManager implementează Runnable, motiv pentru care este definită și metoda run() specifică acestei interfețe, metodă în care se parcurge ArrayList-ul de clienți generați automat, iar în funcție de timpul de ajungere al clientului, este distribuit in coadă prin metoda imparteClienți din Scheduler, metodă care apelează la rândul ei strategia, și tot așa până ajunge la metoda propriu-zisă de adăugare din Coadă.*

*Tot în interiorul acestei metode, din sunt actualizate si informațiile din vederea 2, odată cu adăugarea clientului într-o coadă dar sunt făcute actualizări în timp real și în fiecare coadă, fiind posibilă observarea evoluției cozii la fiecare secundă, până la scurgerea timpului de simulare introdus de către utilizator. Actualizarea cozilor se face in funcție de numărul cozii în care clientul a fost adăugat.*

***2.Pachetul Interfată***

*In acest pachet, sunt definite doua clase, care dovedesc funcționarea programului, prin vizualizarea informațiilor introduse de către utilizator dar și vizualizarea rezultatelor obținute fiind posibilă compararea cu evenimentele din viața reală, atunci când ajungem la un magazin.*

*Prima clasă definită este clasa View, clasă în care au fost setate pentru a fi afișate și a informat utilizatorul diferite enunțuri care fac referire la datele pe care acesta trebuie sa le introducă de la tastatură.*

*Alegerea numărului de cozi este limitată, această alegere fiind posibilă prin acțiunea asupra unui buton, care indică numărul de cozi pe care programul urmează să le genereze.*

*Butonul START face posibilă pornirea aplicației, închide frame-ul principal si în deschide pe cel secundar.*

*Cea de-a doua clasă, clasa SimulationFrame informează utilizatorul despre acțiunile care se petrec odată cu ajungerea în magazin a unui număr de clienți, indicat de către utilizator în frame-ul principal. Aici se poate observa și fereastra care conține informațiile a ceea ce se petrece în interiorul programului(Ex: ”Clientul A ajunge în coada z la timpul x având timpul de procesare egal cu y”, ”Clientul A pleacă din coada x, având timpul de terminare egal cu y”).*

*Tot în acest frame, sunt vizibile si cozile, în care se adaugă si se scot clienții în funcție de strategia aleasă in interiorul programului.*

***3.Pachetul PRINCIPAL***

*În acest pachet, există 2 clase, clase din care pornește toată activitatea programului nostru.*

*CLASA SINCRONIZARE*

*Această clasă are rolul de a sincroniza cele 2 clase principale, una care are contact in mod vizual cu utilizatorul, adică clasa SimulationFrame, si cea de-a doua, clasa SimulationManager, unde are loc toată coordonarea internă a programului.*

*Constructorul acestei clase, primește ca și argument o vedere, adică vederea principală, în care utilizatorul introduce datele necesare pentru procesarea cozilor si a întregului program. Tot aici, sunt adăugate si listener-ele de la cele 7 butoane care sunt definite in vederea principallă.*

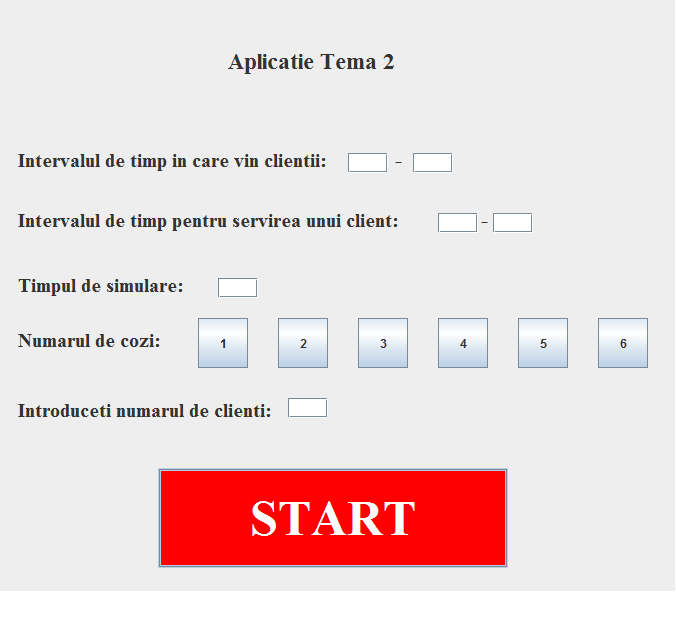
*Prin definirea clasei care face referire la listener-ul butonului START, se instanțiază cele 2 clase principale care fac legătura cu utilizatorul, adică vederea secundară si SimulationManager, se creează un Thread nou, care face referire la obiectul de tipul SimulationManager si se pornește Thread-ul principal, astfel sincronizânddu-se cu celelalte thread-uri si cu utilizatorul care introduce datele in frame-ul principal, iar prin acțiunea asupra butonului START, practic pornește Thread-ul.*

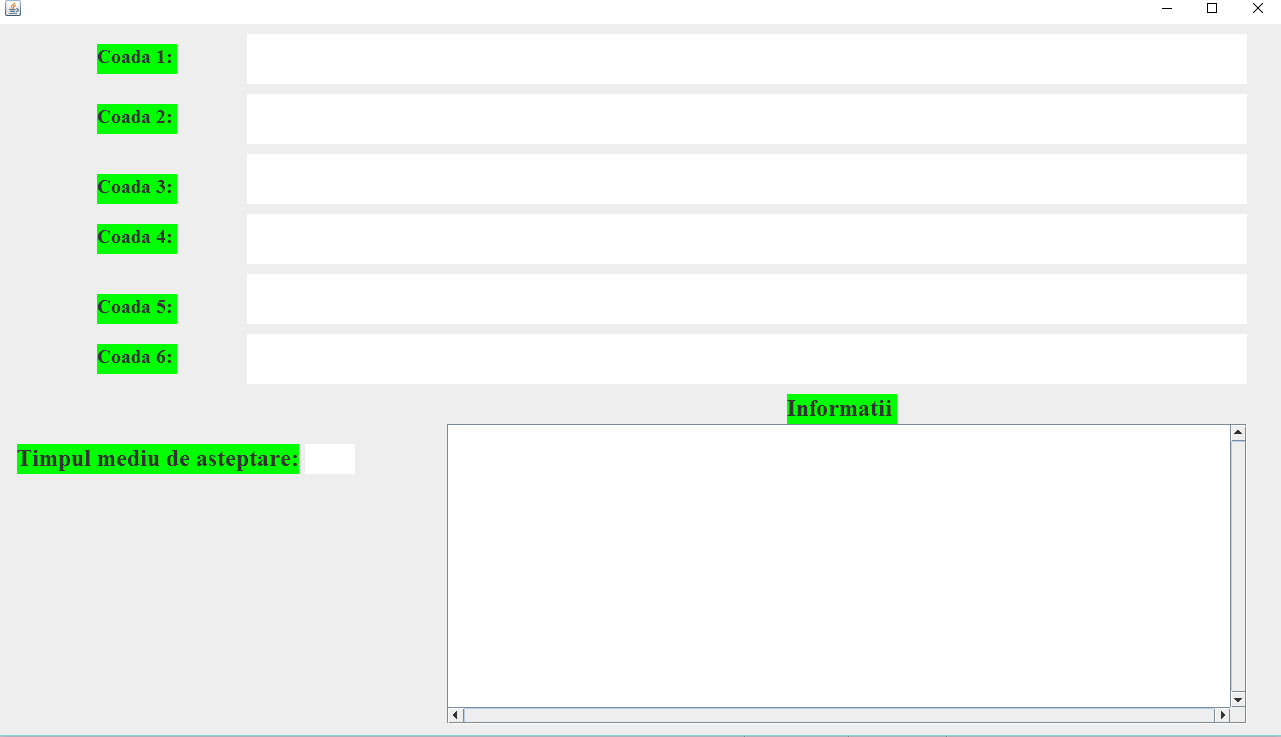
*CLASA MAIN*

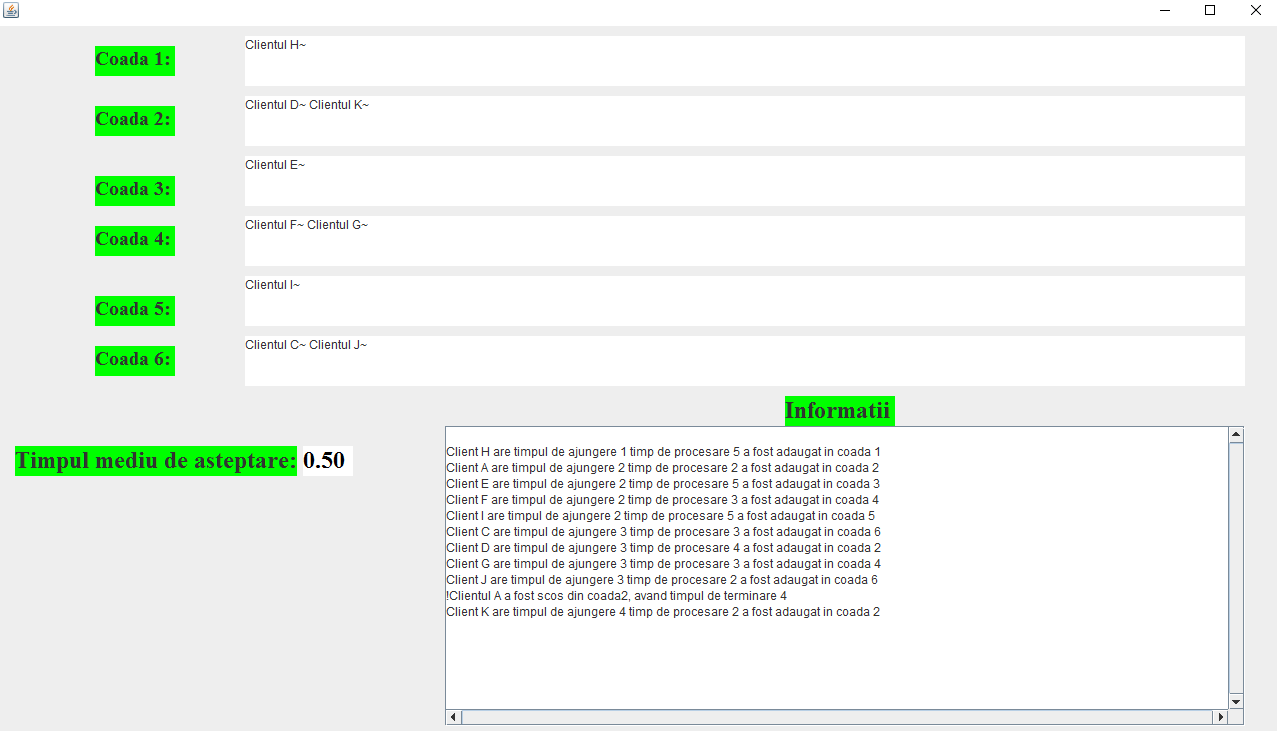
*Această clasă este mai sărăcăcioasă, conținând doar metoda public static void main(), în care se instanțiază un obiect de timpul SINCRONIZARE.*

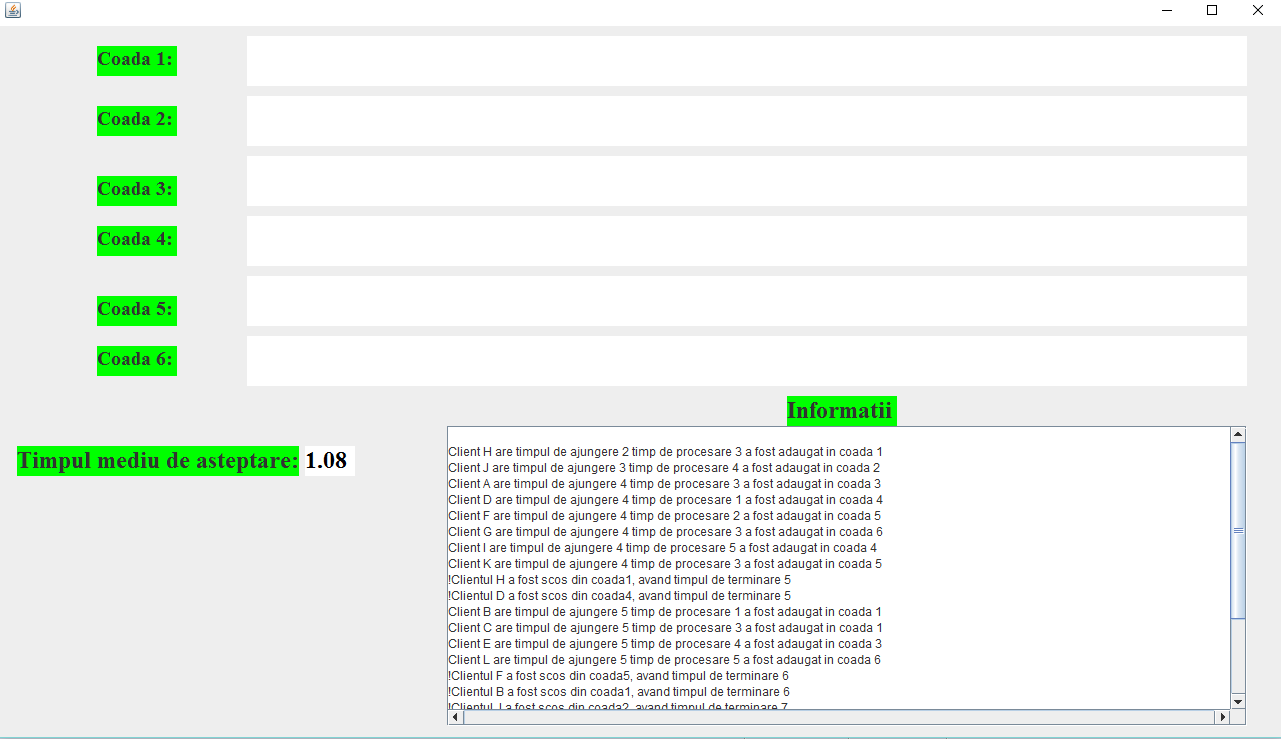
**REZULTATELE OBTINUTE**

*In funcție de datele introduse de către utilizator, se pot observa rezultate in timp real a ceea ce se întâmplă în interiorul programului, fiind vizibilă evoluția clienților care ajung la un moment dat in magazin, sunt distribuiți la o coadă în funcție de timpul de așteptare pe care acest client trebuie să îl petreacă până când ajunge și el la coadă, unde trebuie sa fie procesat. Clientul este înlăturat de la coadă în momentul în care timpul de procesare al acestuia este scurs fiind posibilă primirea unui nou client.*

**

**

**

**

**CONCLUZII**

*In urma realizării acestui proiect, am putut să îmi dau seama cum sunt gestionate activitățile într-o unitate care este dispusă să primească la un moment dat un număr mai mare de clienți, aceștia având timpul de ajungere egal, trebuie definită o anumită strategie astfel încât timpul să fie in folosul ambelor părți, atât pentru unitate cât și pentru client.*

*Folosirea Thread-urilor utilizând si procedeul de MultiThread-ing face posibilă sincronizarea si execuția anumitor pași într-o ordine bine stabilită astfel încât, după cum se poate observa pe Interfața grafică, fiecare client este ”servit” petrecând cât mai puțin timp la coadă în funcție de timpul la care a ajuns. Daca erau clienți care au ajuns mai repede, este și normal ca aceștia să fie procesați, soluția pentru un timp cât mai optim fiind luată ținând cont de timpul de așteptare indicat de fiecare casă.*

*În comparație cu tema trecută, ”Operații cu polinoame” mi se pare că această tema face o strânsă legătură cu lumea reală deoarece îi permite utilizatorului să vizualizeze în timp real cum decurg lucrurile atunci când există mai multe persoane care au același lucru de făcut. Modul de a alege casa în funcție de timpul de așteptare mi se pare cea mai bună modalitate de a salva timp, în comparație cu alte strategii, spre exemplu, mi se pare mult mai avantajoasă această strategie decât strategia de a te așeza la coada care este mai scurtă.*

***BIBLIOGRAFIE***

<http://users.utcluj.ro/~igiosan/teaching_poo.html>

Materialul pus la dispoziție ca model.