**Universitatea Tehnică Cluj-Napoca**

2015

15

**Sistem de gestiune cozi**

**Tehnici de programare – Tema 2**

**Bolba Raluca Maria – grupa 30225**

Cuprins

1. Obiectivul temei ................................................................................................................2
2. Analiza problemei, modelare, scenarii, cazuri de utilizare ................................................2
3. Proiectare (diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfață utilizator) ...........................................................................4
4. Implementare și testare...................................................................................................12
5. Rezultate ..........................................................................................................................15
6. Concluzii, dezvoltări ulterioare ........................................................................................16
7. Bibliografie .......................................................................................................................16

1.Obiectivul temei

*Enunțul problemei :* Proiectați și implementați o aplicație de simulare, care să analizeze sistemele bazate pe cozi de așteptare pentru determinarea și minimizarea timpului de așteptare al clienților.

*Detalii :* Cozile sunt observate frecvent, atât în lumea reală cât și în modele. Obiectivul principal al unei cozi este de a oferi un loc pentru un "client" să aștepte înainte de a primi un "serviciu". Gestiunea sistemelor bazate pe cozi este interesată de minimizarea timpului în care "clienții" așteaptă în cozi. O modalitate de a reduce la minimum timpul de așteptare este de a adăuga mai multe servere, adică mai multe cozi in sistem (fiecare coadă este considerată ca având asociat un procesor), dar această abordare sporește costurile furnizorului. Când un nou server se adaugă, clienții în așteptare vor fi distribuiți în mod egal la toate cozile disponibile în prezent.

Sistemul ar trebui să simuleze o serie de clienți care sosesc pentru serviciu; intrarea în cozi, așteptarea, servirea și în cele din urmă, ieșirea din coadă. Sistemul urmarește timpul în care clienții asteaptă în cozi și determină un timp mediu de așteptare. Pentru a calcula timpul de așteptare trebuie să cunoaștem timpul de sosire, timpul de plecare și timpul de servire. Timpul sosirii și timpul de servire depind de clienții individuali - când apar și cât de multe servicii doresc. Ora de plecare depinde de numărul de cozi, numărul de clienți în coada de așteptare și de serviciile de care au nevoie ceilalți clienți.

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

Pentru o bună implementare a sistemului de gestiune a cozilor, trebuie mai întâi să definim conceptele folosite, să stabilim datele de intrare și cum sunt ele furnizate și procesate pentru a fi folosite în cadrul sistemului pentru a rezulta date de ieșire corecte.

O coadă este o structură de date, mai exact o particularitate a listei, care are un comportament de tipul FIFO (First In First Out). Operațiile permise asupra unei asemenea cozi sunt cele de adăugare a unui element la sfârșitul acesteia și de extragere a unui element din capul listei. Pentru efectuarea acestor operații avem nevoie, de asemenea, să știm dacă există elemente în coadă.

În cadrul aplicației noastre, cozile sunt formate din clienți, caracterizați printr-un timp de sosire în coadă și un timp de servire. Întrucât vorbim de o simulare, acești timpi vor fi generați în mod aleator, cu condiția să se găsească într-un interval stabilit de către utilizatorul aplicației. De asemenea, am stabilit ca numărul de clienți și numărul de cozi să fie cunoscut la momentul începerii simulării. Bineînțeles, timpul pentru care se face simularea este și el o dată de intrare.

Cozile sunt independente, viteza de evoluție a uneia dintre ele nu depinde de viteza de evoluție a celorlalte, așadar ele trebuie implementate astfel încât să funcționeze independent și concurent în același timp. Pentru acest lucru vom folosi așa numitele *Thread-uri* sau fire de execuție care contribuie la eficiența programului. Ele sunt caracterizate tocmai prin faptul că în interiorul aceluiași proces, execută porțiuni diferite de cod în mod paralel.

Pentru ca simularea să fie cât mai apropiată de realitate, atunci când un client dorește să se așeze la o coadă, el este repartizat la coada care are timpul de servire cel mai mic. De asemenea când se deschide o nouă casă, clienții de la cozile cu timpul mai mare de servire se reorientează spre acea casă, până când aceasta nu este casa cu cel mai mic timp de servire. În cazul ștergerii unei cozi, clienții care au rămas la coadă respectivă vor fi repartizați uniform spre restul cozilor.

Pe lângă simularea evoluției cozilor, mai sunt furnizate date de ieșire precum ora de vârf, timpul de servire mediu pe coadă și timpul de așteptare mediu pentru un client. Timpul de așteptare pentru un client este calculat ca diferența dintre momentul ieșirii din coadă și suma timpului de sosire în coadă și timpul de servire.

Interfața grafică cu utilizatorul trebuie să fie una simplă, intuitivă și care să permită utilizatorului să observe cum are loc consumarea cozilor, ce se întâmplă când se deschide o nouă coadă și când se închide una. Tocmai pentru a facilita acest lucru, am stabilit ca momentul când are loc adăugarea sau ștergerea unei cozi să depindă de utilizator.

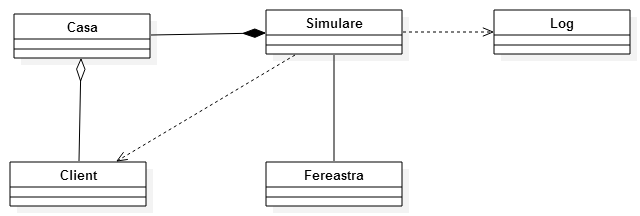
3. Proiectare (diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfață utilizator)

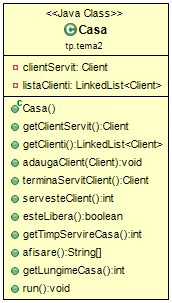
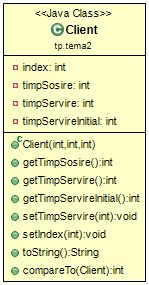
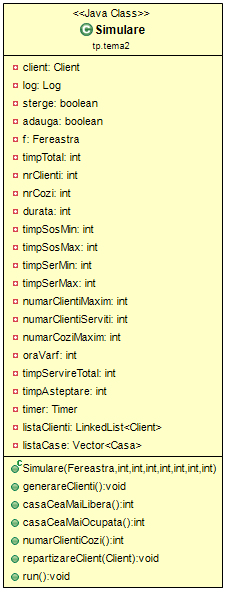
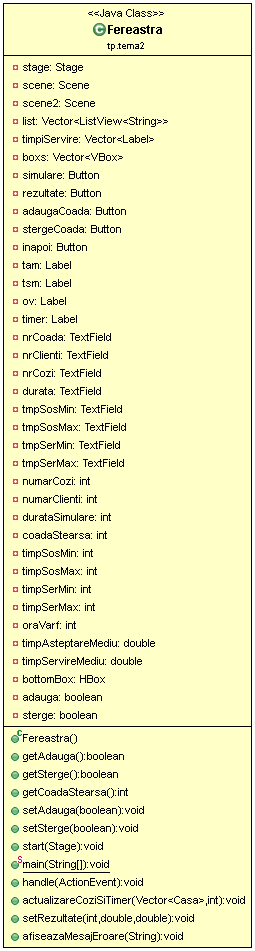
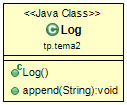
3.1 Proiectare clase

Pentru implementarea acestei probleme, am considerat necesare următoarele clase : *Client, Casa, Fereastra, Simulare* și *Log.*

* Clasa *Client,* după cum sugerează și numele, modelează conceptul de client care așteaptă la o coadă pentru a fi servit. Acesta este caracterizat de un timp de sosire la coadă și un timp necesar pentru servire. Acești parametri diferă de la un client la altul, și așa cum am mai menționat sunt generați aleator.Atributele acestei clase sunt așadar : timpul de sosire, timpul de servire, timpul de servire inițial și un index, util pentru identificarea clienților.Obiectele acestei clase sunt instanțiate în constructorul cu același nume, care primește ca parametrii timpul de sosire, cel de servire și indexul. Aceasta clasă implementează interfața *Comparable*, pentru a putea sorta un set de clienți pe baza timpilor de sosire la coadă. Acest lucru este realizat prin suprascrierea metodei *compareTo(Object arg).*
* Clasa *Casa* modelează conceptul de casă, la care se formează o coadă de clienți pentru a primi un anumit serviciu. Inițial o casă este goală, așadar constructorul cu același nume nu primește parametrii. Atributele acestei clase constă într-o listă de clienți, care se va modifica pe parcursul simulării, și un client servit, care ne este util atunci când trebuie să calculăm timpul de așteptare pentru clienții serviți. Întrucât putem avea un număr mai mare de case, iar acestea lucrează independent de celelalte, însă paralel cu ele, această clasă extinde clasa *Thread.* Acest lucru ne obligă să suprascriem metoda *run()* în care definim ”jobul” pe care trebuie să îl execute această clasă.
* Clasa *Fereastra* reprezintă clasa care modelează interfața grafică cu utilizatorul (GUI) prin care un utilizator poate interacționa cu aplicația propriu-zisă. Ca bibliotecă de dezvoltare a interfeței grafice am folosit *JavaFX*. O aplicație JavaFX trebuie să extindă clasa *javafx.application.Application*. Aceasta este o clasă abstractă, deci suntem obligați să definim metoda *public void start(Stage primaryStage)*, care este metoda de start a aplicației.
* Clasa *Simulare* este clasa în care realizăm simularea aplicației. De lângă calculul anumitor date de ieșire, ea reprezintă în fond legătura dintre interfața grafică cu utilizatorul și evoluția cozilor. Constructorul cu același nume primește următorii parametri : un obiect de tip *Fereastra* prin intermediul căruia putem actualiza periodic evoluția cozilor, un număr de clienți și un număr de cozi pentru care se face simularea, o durata a simulării și capetele intervalelor pentru timpii de sosire și de servire a clienților. Întrucât vrem să vedem evoluția programului în timp real, am utilizat clasa *TimerTask*, pe care clasa *Simulare* o extinde, și care ne permite să definim un task pe care să îl repetăm o dată sau de mai multe ori. În cazul nostru, el va fi repetat în funcție de durata simulării. Printre atributele acestei clase se află un obiect de tip *Timer* , un vector de obiecte de tip *Casa*, a cărui dimensiune va fi initial numărul de case primit ca parametru în constructor, un obiect de tip *Log* prin care vom scrie într-un fișier evoluția cozilor, un număr întreg reprezentând ora de vârf, un număr întreg reprezentând timpul de așteptare total pentru clienții serviți, un număr întreg reprezentând timpul de servire total pentru case etc.
* Clasa *Log* prin intermediul căreia scriem într-un fișier *„Log.txt”* toate operațiile care au loc în cadrul simulării.

3.2 Diagrama UML





3.3 Relațiile între clase

Pe baza diagramei UML se observă relațiile între cele cinci clase : *Client, Casa, Fereastra, Simulare și Log.*

Între clasa *Client* și clasa *Casa* există relația de agregare, care reprezintă o relație specială de asociere ce modelează relația întreg – parte între un agregat(întreg) și părțile sale. Agregatul reprezintă în cazul nostru clasa *Casa*, iar partea casa *Client.*

Între clasa *Simulare* și clasa *Casa* există relația de compoziție, care reprezintă o relație specială de asociere, mai exact o formă de agregare cu posesiune puternică conform căreia părțile nu pot supraviețui agregatului. În cazul nostru, clasa *Simulare* reprezintă agregatul, ale cărei părți, respectiv instanțele clasei *Casa* nu pot exista independent.

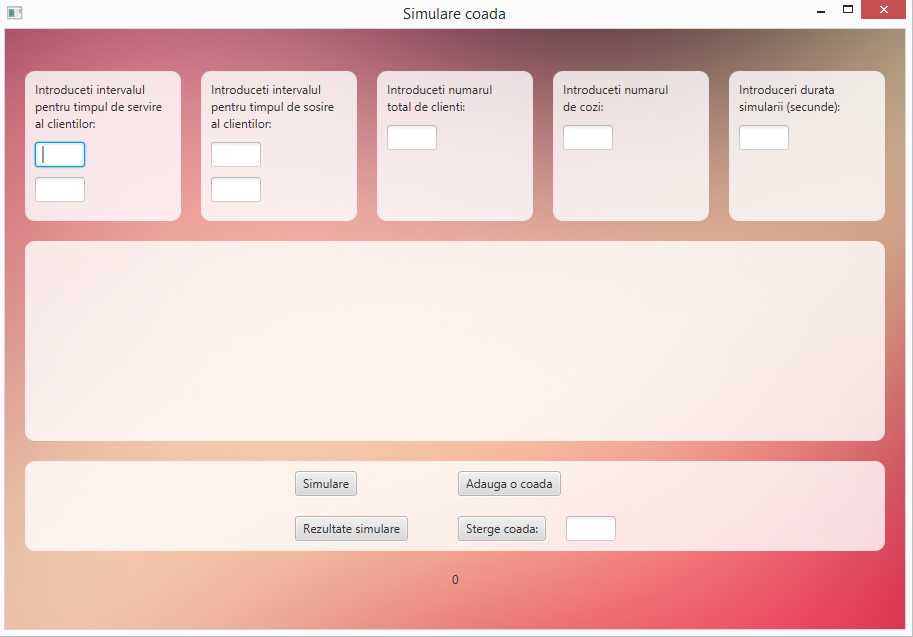
Între clasa *Client* și clasa *Simulare* există o legatură de tip dependență, prin care cea de-a doua clasă foloseșe obiecte ale clasei *Client.*

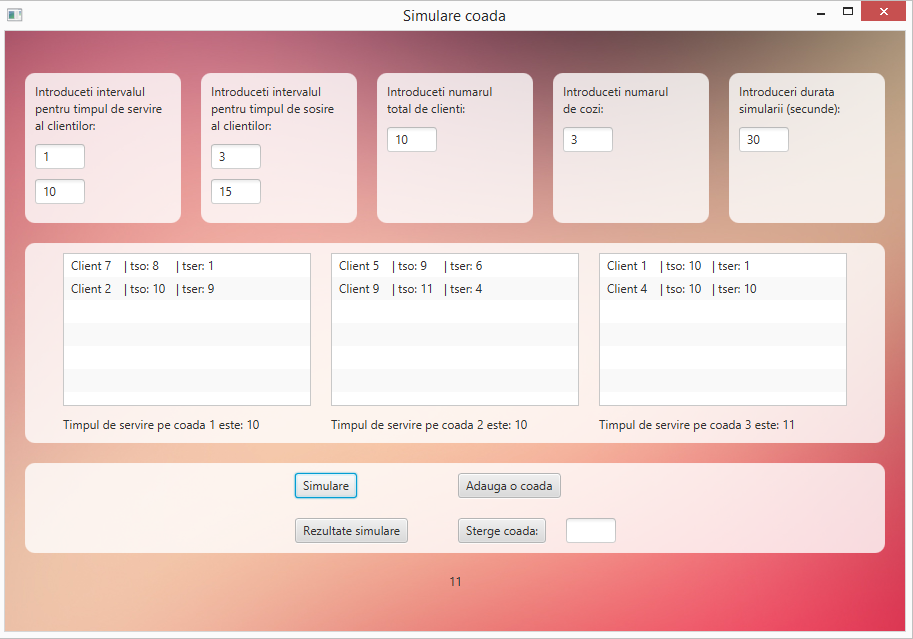
Între clasa *Simulare* și clasa *Fereastra* există o relație de asociere care reprezintă cazul în care un obiect din clasa *Simulare* folosește serviciile unui obiect al clasei *Fereastra,* pentru a realiza simularea și viceversa, caz în care un obiect de tip *Simulare* apelează metode corespunzătoare clasei *Ferestre* pentru a actualiza interfața grafică și a oferi utilizatorului posibilitatea de a vizualiza evoluția cozilor, dar și rezultatele finale .

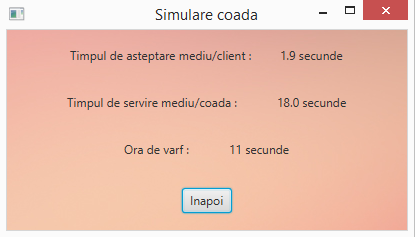
Între clasa *Simulare* și clasa *Log* există, de asemenea o relație de asociere, prin care obiectele clasei *Simulare* folosește obiectele clasei *Log* pentru a scrie într-un fișier text modificările ce au loc în timpul simulării.

3.4 Interfața utilizator

Așa cum am mai menționat, interfața grafică cu utilizatorul am implementat-o cu ajutorul bibliotecii JavaFX. Principalele clase folosite în cadrul realizării acestei interfețe sunt : *Button, RadioButton, Label, TextField, TextArea, ListView<T>, Vbox, Hbox etc.*

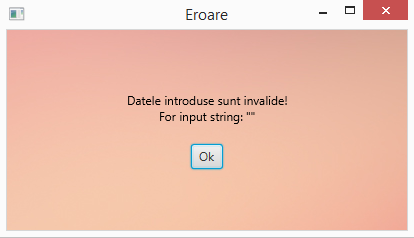
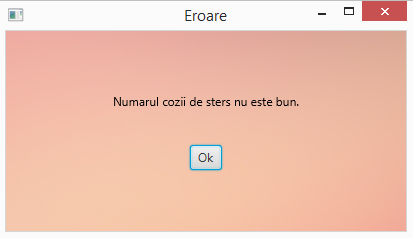
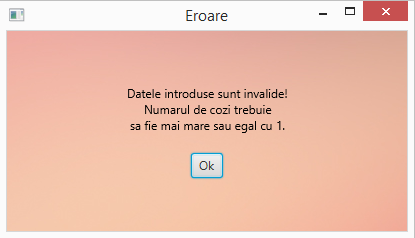
Fereastra de început la rularea aplicației este următoarea :

După cum se observă, în partea de sus a ferestrei clientul trebuie să completeze o serie de câmpuri pentru a putea începe simularea. Aceștia sunt timpii minimi și maximi de sosire și de servire pentru clienți, numărul total de clienți, numărul de cozi inițiale și durata simulării. Toți timpii se consideră scriși în secunde. În partea centrală observăm un chenar, care în cazul în care datele introduse de utilizator sunt valide în timpul apăsării butonului *Simulare*, el va furniza evoluția cozilor, în felul acesta :

În partea de jos se află câteva butoane și un câmp pe care utilizatorul poate să îl completeze dacă vrea să șteargă sau să închidă o anumită casă. De asemenea, în josul ferestrei se află un „Label” care reprezintă un timer, ce se modifică odată la o secundă. La finalul simulării, utilizatorul poate vedea rezultatele finale apăsând butonul *Rezultate simulare*, care îl va redirecționa spre următoarea fereastră :

Dacă acesta dorește să se reîntoarcă la fereastra inițială, va apăsa butonul *Inapoi*.

În cazul în care utilizatorul introduce date invalide, precum : durata simulării, numărul cozilor și al clienților egale mai mici sau egali cu zero, timpii de sosire sau servire maximi mai mici decât timpii minimi de sosire sau servire, neintroducerea unor date sau introducerea unor date care nu sunt numerice, acesta va fi atenționat prin mesaje de eroare :



3.5 Algoritmi

Operațiile care trebuiesc efectuate în cazul sistemului de gestiune a cozilor sunt :

* Repartizarea unui client într-o coadă . La fel ca în lumea reală, un client atunci când se așează la o coadă, o alege pe cea la care apreciază el că ar avea mai puțin de așteptat. Așadar, pentru a efectua acest lucru, am calculat timpul de servire total pentru toate casele deschise, adică am însumat timpii de servire a clienților de la acea casă și am ales-o pe cea cu timpul de servire mai mic. Clientul va fi așadar așezat la această casă, la sfârșitul ei.
* Servirea clienților*.* Întrucât simularea se execută într-un pas de o secundă, servirea clienților constă în decrementarea timpului de servire a primului client de la fiecare casă la fiecare secundă. Atunci când timpul se servire a clientului a ajuns la zero, se consideră că acesta a fost servit în totalitate și este eliminat din coada de așteptare a casei respective.
* Închiderea unei case*.* Atunci când se închide o anumită casă, clienții care se află în coada de așteptare a casei respective trebuie să se mute la restul caselor în mod realist și uniform. Pentru aceasta, se reține lista de clienți a casei ce urmează să se închidă, închidem firul de execuție corespunzător acelei case, și cum casele sunt păstrate într-un vector, vectorul trebuie translatat cu o poziție la stânga începând cu poziția corespunzătoare casei șterse, iar pe ultima poziție a vectorului punem *null*. Din lista de clienți a casei închise, alegem câte un client pe rând, pe care îl repartizăm la casa cu cel mai mic timp de servire.
* Deschiderea unei case. Atunci când se deschide o nouă casă, o parte din clienții de la cozile de așteptare existente ar trebui să se mute la casa respectivă. Acești clienți sunt de obicei ultimii clienți de la casele care au cel mai mare timp de servire. Așadar, la adăugarea unei noi case, atâta timp cât aceasta este casa cu cel mai mic timp de servire total, se ia ultimul client de la casa cu cel mai mare timp de servire total și se adăuga la sfârșitul noii cozi formate.
* Determinarea orei de vârf. Ora de vârf reprezintă în cazul nostru secunda din durata simulării când se află cel mai mare număr de clienți la cozile existente. Pentru a calcula acest lucru, la fiecare secundă calculăm numărul de clienți (sau lungimea) cozilor de așteptare existente și îl comparăm cu un maxim existent, iar dacă este mai mare decât acesta, el devine maximul și ora de vârf devine timpul curent din simulare.
* Calcularea timpului de așteptare mediu pe client. Timpul de așteptare pentru un client care a fost servit este definit ca diferența dintre dintre momentul ieșirii din coadă și suma timpului de sosire în coadă și timpul de servire a acestuia. Pentru a putea calcula acest timp de așteptare, trebuie să verificăm la fiecare secundă dacă a fost vreun client servit la fiecare casă și care este acesta. La final calculăm media dintre suma timpilor de așteptare a clienților serviți și numărul acestora.
* Calcularea timpului de servire mediu pe casă. Pentru a calcula acest lucru, de fiecare dată când un client se așează la o coadă, se adună la o variabila numită timpAsteptare timpul lui de servire. Media se calculează între această sumă și numărul maxim de cozi care au existat în timpul simulării.

4. Implementare și testare

4.1 Implementare

* Clasa *Client* modelează conceptul de client care se așează la o coadă pentru a primi un serviciu. Metodele acestei clase sunt :
* **public** Client(**int** index, **int** timpSosire, **int** timpServire) – reprezintă constructorul clasei, iar în cadrul acestuia se inițializează indexul clientului, timpul de sosire, timpul de servire și timpul de servire inițial
* **public** **int** getTimpSosire() – returnează timpul de sosire al clientului
* **public** **int** getTimpServire() - returnează timpul de servire al clientului
* **public** **int** getTimpServireInitial() - returnează timpul de servire inițial al clientului
* **public** **void** setTimpServire(**int** timpServireNou) - setează timpul de servire al clientului cu argumentul primit ca parametru
* **public** **void** setIndex(**int** indexNou) - setează indexul clientului cu argumentul primit ca parametru
* **public** String toString() – metodă care returnează un șir de caractere, reprezentând o etichetă a clientului. Aceasta este formată din cuvântul „Client” împreună cu indexul acestuia, timpul de sosire și timpul de servire. O etichetă a unui client cu indexul 2 care sosește în coadă la secunda 3 și are timpul de servire 5 secunde va fi : *Client 2 | tso: 3 | tser: 17*
* **public** **int** compareTo(Client arg0) – metodă care compară obiectul din clasa curentă cu clientul primit ca parametru în funcție de timpul de sosire în coadă. Această metodă suprascrie metoda din interfața *Comparable*.
* Clasa *Casa* modelează conceptul de casă, la care se formează o coadă de clienți pentru a primi un anumit serviciu. Metodele acestei clase sunt :
* **public** Casa() – constructorul clasei, în cadrul căruia se crează o coadă de așteptare goală, iar variabila *clientServit* este inițializată cu *null*
* **public** Client getClientServit() – metodă care returnează un obiect de tip client reprezentând clientul servit la acel moment al simulării, sau null în cazul în care nu s-a servit vreunul
* **public** LinkedList<Client> getClienti() – metodă care returnează o listă de obiecte de tip *Client* reprezentând clienții aflați în coada de așteptare
* **public** **void** adaugaClient(Client c) – metodă care adaugă un client la sfârșitul cozii de așteptare
* **public** Client terminaServitClient() – metodă care termină de servit primul client, adică îl elimină din coadă, îi setează timpul de servire la cel initial și îl returnează
* **public** **int** servesteClient() – metodă care descrește cu o unitate timpul de servire a primului client din coada de așteptare
* **public** **boolean** esteLibera() – metodă care returnează *true* dacă coada este vidă și *false* în caz contrar
* **public** **int** getTimpServireCasa() – metodă care returnează suma timpilor de servire a clienților aflați la coada de așteptare corespunzătoare casei pentru care se apelează metoda
* **public** String[] afisare() – metodă care returnează un vector de șiruri de caractere, în care fiecare element este eticheta unui client din coada de așteptare ; această metodă este utilă atunci când se actualizează forma cozilor în interfața grafică
* **public** **int** getLungimeCasa() – metodă care returnează numărul clienților din coada de așteptare a casei pentru care se apelează metoda
* **public** **void** run() – metodă suprascrisă a clasei *Thread* ; în această metodă se definește ce trebuie să facă thread-ul, respectiv casa, și este apelată de JVM atunci când se apelează metoda *start()*. „Jobul” unei case este de a servi , atâtâ timp cât există, primul client, iar dacă timpul lui de servire a devenit nul, atunci el este eliminat din coadă și atribuit variabilei *clientServit*.
* Clasa *Log* este utilizată pentru a scrie într-un fișier text modificările ce au loc în timpul simulării programului, precum adăugarea unui client în coadă, închiderea unei case, deschiderea unei case etc. Singura metodă a acestei clase este :
* **public** **void** append(String sir) – metodă care adaugă pe linia următoare a fișierului text *Log.txt* șirul de caractere sir
* Clasa *Fereastra* modelează conceptul de interfață grafică cu utilizatorul. Metodele acestei clase sunt :
* **public** **boolean** getAdauga() – metodă care returnează *true* dacă utilizatorul a apăsat butonul de adăugare coadă și *false* în caz contrar
* **public** **boolean** getSterge() - metodă care returnează *true* dacă utilizatorul a apăsat butonul de ștergere coadă și *false* în caz contrar
* **public** **int** getCoadaStearsa() - metodă care returnează numărul cozii pe care utilizatorul vrea să o șteargă
* **public** **void** setAdauga(**boolean** adauga) - metodă care setează variabila *adauga* cu cea primită ca parametru
* **public** **void** setSterge(**boolean** sterge) - metodă care setează variabila *sterge* cu cea primită ca parametru
* **public** **void** start(Stage primaryStage) **throws** Exception - reprezintă metoda de start a aplicației. În cadrul acesteia sunt definite componentele pentru fereastra de start, dar și pentru cea de rezultate
* **public** **void** handle(ActionEvent event) – metodă care conține codul ce va fi executat în funcție de sursa evenimentului. Dacă aceasta este butonul de simulare, atunci de preiau datele introduse de utilizator și sunt procesate pentru a putea fi folosite și este instanțiat un obiect de tip simulare a cărui constructor primește ca parametrii datele procesate și sunt create *ListView-uri* în care se va putea urmări evoluția cozilor. Dacă sursa o reprezintă butonul de adăugare a unei cozi, atunci se mai adaugă un *ListView* și se setează variabila *adauga* la true, iar dacă sursa este butonul de ștergere a unei cozi atunci se șterge *ListView-ul* corespunzător cozii de șters și se setează variabila *sterge* la true. Dacă sursa este butonul *rezultate* atunci se setează acestea și se trece la fereastra a doua.
* **public** **void** actualizareCoziSiTimer(Vector<Casa> listaCozi, **int** timp) – metodă care actualizează forma *ListView*-urilor și valoarea *Label*-ului corespunzător timerului cu valorile primite ca argumente
* **public** **void** setRezultate(**int** oraVarf, **double** timpAsteptareMediu, **double** timpServireMediu) - metodă ce setează ora de vârf, timpul de așteptare mediu și timpul de servire mediu cu valorile primite ca parametri, pentru a le putea folosi ulterior în fereastra de rezultate
* **public** **void** afiseazaMesajEroare(String s) - metodă apelată în cazul apariției unor excepții care afișează o fereastră ce conține un mesaj de eroare, transmis ca parametru.
* Clasa *Simulare* are ca metode :
* **public** Simulare(Fereastra f, **int** nrClienti, **int** nrCozi, **int** durata, **int** timpSosMin, **int** timpSosMax, **int** timpSerMin, **int** timpSerMax) – constructorul clasei care inițializează atributele simulării cu datele extrase din interfață
* **public** **void** generareClienti() – metodă care generează o listă de *nrClienti* clienți, a căror timpi de sosire și servire sunt generați aleatori în intervalele *[timpSerMin, timpSerMax]* și *[timpSosMin, timpSosMax]* și o sortează în funcție de timpul de sosire a clienților
* **public** **int** casaCeaMaiLibera() – metodă care returnează indexul casei cu timpul de servire total cel mai mic
* **public** **int** casaCeaMaiOcupata() – metodă care returnează indexul casei cu timpul de servire total cel mai mare
* **public** **int** numarClientiCozi() – metodă care returnează numărul de clienți care se află la cozile de așteptare existente la momentul actual
* **public** **void** repartizareClient(Client client) – metodă care repartizează un client la casa cea mai liberă, așezându-l la sfârșitul cozii de așteptare a casei respective
* **public** **void** run() – metodă suprascrisă din clasa *TimerTask* care reprezintă task-ul ce trebuie executat de un anumit număr de ori și care are durata de o secundă. În această metodă, se extrag din vârful listei de clienți generate clienții care au timpul de sosire este egal cu momentul actual, și în caz afirmativ, sunt repartizați la cozile existente. De asemenea, în această metodă se pornesc „joburile” caselor care au cel puțin un client adăugat în coadă, se verifică dacă utilizatorul a ales una dintre opțiunile de ștergere sau adăugare de coadă din interfață și se execută operația corespunzătoare, se calculează timpul de așteptare mediu, ora de vârf și timpul de servire mediu, dacă timpul de simulare a expirat se trimit la interfață rezultatele finale și se apelează metoda actualizareCoziSiTimer(Vector<Casa> listaCozi, **int** timp).

4.2 Testare

Întrucât simularea se bazează pe date aleatoare, nu am putut realiza o testare prin tehnica JUnit. Testarea în acest caz implică rularea aplicației și validarea rezultatelor și verificarea corectitudinii în ceea ce privește evoluția cozilor.

5. Rezultate

În urma testării aplicației, se observă că simulările au loc fără probleme prea mari, aplicația răspunde în timp real la cererile utilizatorului, iar evoluția cozilor se desfășoară corect. Se mai observă că atunci când utilizatorul introduce date de intrare invalide acesta este atenționat cu mesaje de eroare corespunzătoare.

6. Concluzii

În concluzie, se poate spune că am reușit implementarea unui sistem de gestiune a cozilor, în cadrul căruia poate fi observată evoluția cozilor, ce se întâmplă dacă una dintre case este închisă sau dacă alta este deschisă. Deși interfața grafică nu a ridicat probleme prea mari, am întâmpinat dificultăți la actualizarea ei în timp real și gestiunea cozilor pentru ca ele să funcționeze în mod concurent și independent.

Deși aplicația respectă în mare cerințele problemei, ea poate fi îmbunătățită prin :

* Modificare interfeței grafice astfel încât utilizatorul să poată observa mai bine evoluția cozilor
* Adăugarea unei limite maxime de clienți pentru o coadă astfel încât la depășirea acesteia să fie deschisă o nouă casă, acest lucru să nu mai revină utilizatorului aplicației.
  + Definirea unor zone, fiecare caracterizată printr-o capacitate maximă de clienți la o casă.
* Adăugarea unor intervale orare la care să se cunoască numărul de case deschise.

7. Bibliografie

<http://docs.oracle.com/javase/7/docs/api/java/util/TimerTask.html>

<http://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html>

<https://docs.oracle.com/javafx/2/api/javafx/application/Platform.html>