****

**FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE**

**CATEDRA CALCULATOARE**

**ASSIGNMENT 2**

**QUEUES SIMULATOR**

**DOCUMENTATIE**

Student : Petcu Sorina

Grupa : 302210

Cuprins

* Obiectivul Temei . . . . . . . . . . . . . . .. . . . . . . . . . . . 3;
* Analiza problemei, asumptii, erori . . . . . . . . . . . . . 4;
* Proiectare . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5;
* Implementare . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6;
* Testare . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11;
* Rezultate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11;
* Concluzii . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15;
* Bibliografie . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16;

1. Obiectivul temei

Cozile sunt utilizate in mod frecvent pentru a modela situatii din lumea reala. Principalul obiectiv al unei este de a genera un loc pentru un “client” pentru a astepta inainte de a fi servit. Scopul unui sistem de manageriere a unor cozi este acela de a reduce timpul de asteptare al clientilor in cozi inainte de a fi serviti. O modalitate de a minimiza timpul de asteptare este aceea de a creste numarul de cozi, dar aceasta solutie este costisitoare din punct de vedere al spatiului ocupat si implicit costul. O alta solutie este aceea de a verifica in fiecare secunda timpul de asteptare in fiecare coada si de a directiona clientul urmator in coada cu cel mai mic timp de asteptare.

Aplicatia ar trebui sa simuleze o serie de client care ajung pentru a li se face un anumit serviciu, acestia intrand in cozi, asteptand si fiind serviti, in final parasind coada. Programul urmareste timpul pe care clientul il petrece asteptand in cozi si returneaza timpul de asteptare mediu in final. Pentru a calcula timpul de asteptare avem nevoie de timpul sosirii, cel al plecarii si timpul de care este nevoie pentru fiecare client pentru a fi servit. Timpul sosirii si cel al servirii depinde individual de fiecare client generat si este creat atunci cand ei apar in sistemul de simulare. Timpul final depinde de numarul de client, numarul de cozi si timpul de care au nevoie pentru a fi serviti.

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

Prin analiza problemei, ne referim la un prim set abstract de operatii si proprietati prin care incercam sa depistam eventualele insusiri si comportamente ale proceselor necunoscute. Programarea orientata ne ofera aici un avantaj clar, tocmai fiindca ea permite sa taclm problema de la un nivel superiror, fara a mai fi constrasi, intr-o asa masura, de caracteristicile tehnice.

Aceasta strategie de conceptualizare, mai poarta numele și de bottom-up design. Este foarte avantajoasa din prisma gasirii componentelor constituente, deoarece pot fi gasite, relativ usor, structuri cu o legatura directa in lumea reala( obiecte, actiuni etc.). Din pacate aceasta versatilitate vine cu pretul complexitatii, ea crescand pe masura ce se avanseaza pe nivelele inferioare.

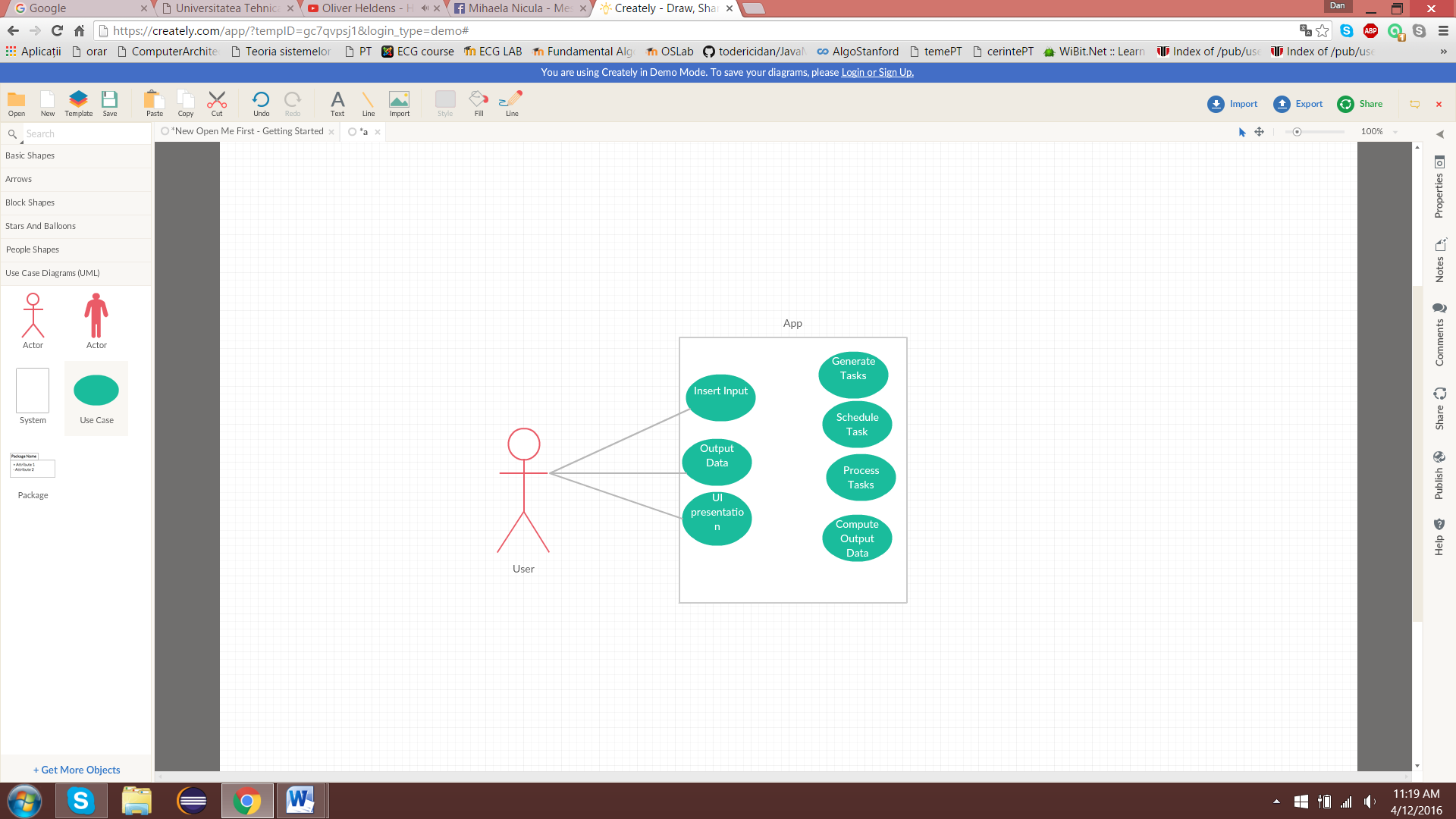
Pornind de la cerintele de functionare ale acestui proiect ne dam seama imediat ca acesta va fi folosit de un numar mare de persoane. De aceea am ales sa implemetez si sa utilizez si interfata grafica care va fi o componenta extrem de importanta pentru acest proiect pentru a intelege mai bine cum functioneaza.

**USE – CASE:**

Utilizarea normala a programului presupune introducerea datelor pe rand in cele 3 fisiere text care trebuiesc generate si avem: min Arrival Time, max Arrival Time, min Service Time, max Service Time , Number of Clients, Number of Queues, Simulation Interval. Pentru o buna functionare valorile de min vor fi mereu mai mici ca cele de max si se vor introduce mereu numere. De asemenea, programul mai ofera informatii suplimentare: average waiting time( care ne ofera informatii cu privire la timpul mediu de asteptare in programul nostru pentru numarul de cozi si intervalul de timp introdus de catre utilizator).

1. Proiectare

User Diagram:



Proiectul realizat are la baza cerinta de a simula si analiza un magazin in care clienti vin, asteapta la coada, sunt serviti si parasesc coada. El presupune folosirea de threaduri si timere pentru monitorizarea timpilor de sosire la coada, de asteptare, servire si timpul de parasire a magazinului. Pentru simulare am folosita o fereastra grafica JFrame care contine un panou: unul pentru datele de intrare, adica numarul de cozi, perioada de simulare, si clientii care intra si ies din coada in momentul terminarii timpului propriu de asteptare. Clienti sunt reprezentati printr-un patrat de culoare albastru si id-ul fiecaruia scris in interiorul patratului. Casele, maxim cinci la numar, sunt dreptunghiuri de culoare verde dupa care sunt aliniati clientii si sunt deschise in functie de numarul specificat inainte de pornirea simularii. Simularea dureaza in functie de intervalul de simulare dat si daca dupa secunda specificata mai sunt clienti in coada acestia vor fi tratati si casele se vor inchide.

**Proiectare clase:**

Tema contine 13 entitati java si respecta tiparul Model View Controller cu specificatia ca divizarea atributilor s-a facut in functie de modelul ales. Thread-urile sunt distribuite pe fiecare casa si exista un thread “principal” care comanda simularea, toate fiind sincronizate dupa un timer global afisat pe fereastra. Totul a fost creat cu ajutorul librariilor “javax.swing”, “java.awt” si “java.util”, etc. De asemenea am folosit 4 pachete diferite pentru a diviza mai bine fiecare functie.

1. Implementare
2. Clasa **Client -** este clasa ce retine datele referitoare la un client: idClient, arrivalTime, serviceTime, finishTime, waitingTime si un speficicator care transmite daca clientul a fost servit sau nu (initializat cu false): “served”. Ea detine doi constructori si metode de tipul “get” si “set” pentru obtinerea variabilelor private declarate in clasa. Clasa implementeaza interfata Comparable si suprascrie metoda compareTo in functie de timpul de sosire, si va fi folosita ulterior pentru sortarea liste de clienti generata aleator;
3. Clasa **ServiceConsumer -** clasa retine date referitoare la un client, de afisat pe fereastra grafica. Obiectele detinute sunt: un client, pozitia acestuia pe axa x si pe axa y in fereastra, forma lui (Rectangle), casa la care clientul a ales sa astepte si un speficicator care transmite daca clientul a fost desenat. Clasa detine metode de obtinere a obiectelor private si un constructor care care instantiaza Rectangle-ul “shape” si clientul in cauza.
4. Clasa **HomeStore** -detine o coada de clienti aflati intr-un LinkedBlockingDeque, un int care reprezinta numarul casei . Ea are un singur constructor in care coada este initializata si 8 metode printre care metoda de run(), cea responsabila pentru comportarea thread-urilor. Metoda de adaugare in coada pune un client in capatul cozii si afla timpul care il are de asteptat prin findWaitingTime, in functie de timpul in care a ajuns si timpii de servire a celorlalti clienti din fata lui. HomeStore descrie metoda nextServed, care va fi folosita doar in interiorul clasei de catre metoda run. Aceasta metoda, atat timp cat casa este deschisa, pentru fiecare client seteaza timpul de terminare, faptul ca a fost servit si dupa ce asteappta o perioada de timp am setat thread-ul intr-o blucla nula care va rula pana la venirea unui nou client
5. Clasa **Scheduler** -clasa contine partea “hard” a modelului, fiind entitatea ce va reprezenta logic intreg magazinul. Ea detine cate o lista de case (HomeStore) si o lista de threaduri, precum si o coada de tipul ServiceQueue (de alegere pentru clienti casa la care se vor plasa). Constructorul ei initializeaza listele si atribuie fiecarei case disponibile un thread propriu. Clasa are o metoda standar de schimbare a strategiei de adaugare si gettere si settere pentru obiectele de tipul listelor. In plus, clasa contine si metoda pentru determinarea average Time-ului.
6. Clasa **SimulationManager** - implementeaza interfata Runnable si suprascrie implicit metoda run(). Rolul major al acestei clase este cel de Controller, ea find cea care distribuie sarcinile celorlalte clase din proiect. Are un obiect de tip Random din Java.util care este folosit pentru generarea aleatoare de clienti, un obiect scheduler, un altul public static de tipul Frame, o lista unde vor fi plasati clientii generati aleator si un log care scrie informatii despre client. Constructorul creaza obiecte de tipul Scheduler si SimulationFrame. Metoda setServiceCustomers parcurge fiecare clasa deschisa si creaza un nou obiect ServiceConsumer pe care il adauga in fereastra pentru a semnala existenta lui in magazin. StartThreads are obligatia de a porni thredurile ( metoda start() ) daca nu sunt in viata .Un bloc try-catch inglobalizeaza metoda **start** deoarece aceasta poate arunca o exceptie neprevazuta. Metoda generateClients are rolul de a genera clienti cu date aleatoare in functie de parametri descrisi la intrare. Metoda trateaza cazul in care simularea poate ajunge intr-o stare invalida/nedorita astfel:

- daca timpul minim dintre clienti e mai mare decat timpul maxim

-daca timpul minim de servire e mai mare decat timpul maxim

- daca timpul minim dintre clienti e setat la zero ( pentru 0 s-ar genera toti clientii odata).

Daca starea ferestrei este una invalida atunci pana cand va deveni valida este blocata intr-o bucla goala {}. Se calculeaza:

- timpul de servire dupa metoda forService = frame.getMinServiceTime() + random.nextInt(frame.getMaxServiceTime() - frame.getMinServiceTime() )

-timpul de sosire: inInterval = random.nextInt(frame.getMaxInterval() - frame.getMinInterval()) + frame.getMinInterval()

-arrival = lastArrival + inInterval

-lastArrival = arrival

Id e incrementat la fiecare client daca timpul de sosire este mai mare ca si timpul pe care simularea trebuie sa il intretina - timpul de servire al ultimului client atunci se va iesi din bucla while cu break. Datele create sunt asociate unui nou client care este adaugat listei de clienti.

Metoda waitRun “adoarme” threadul curent atat timp cat fereastra se afla intr-o stare de asteptare ( timpul in care se introduc datele de intrare ). Metoda **run()** este o metoda sincronizata care asteapta datele de intrare prin metoda waitRun, creaza lista de obiecte cu clienti generati, deschide casele si generaza clientii. Ca si strategie de selectie a cozii este setata metoda de adaugare prin minimizare a timpului de asteptare. Clienti sunt scosi din lista generata si in blocul catch folosit pentru adaugarea clientilor pana la limita null pointer exception sunt adaugati in lista consumatorilor ce urmeaza a fi desenati. De asemenea, aici apelam si afisam averageWaitingTime daca variabila control are valoarea 1.

Threadurile sunt pornite si in cazul in care timpul de simulare dat depaseste timpul actual al timer-ului ferestrei atunci metoda run este parasita.

1. Clasa **MenuPanel –** extinzand clasa JPanel, clasa devine un panou cu atribute specifie care vor fi desenate in fereastra principala. Ca obiecte, sunt folosite un boton, un combo box. In constructor sunt setete dimensiunile panoului, culoarea si este apelatao metoda privata private din aceasta clasa:

* action() – aceasta metoda specifica atributile pe care trebuie sa le aiba butonul si combo box-ul. Lor li se adauga un nou ActionListener si in interiorul parantezelor acestuia se suprascrie metoda actionPerformed. Butonul, in fereastra grafica seteaza starea ferestrei (timer) la true, prin metoda getText() ia datele introduse (timpii de simulare, si numarul cozilor de deschis) si seteaza in clasa de timpul JFrame-ul obiectele aferente. Dupa introducerea datelor in fiecare container i se transmite prin setInvalidData( false ) faptul ca in timpul de rulare nu se pot modifica datele de intrare, lucru care, nefiind tratat poate duce la comportarea incorecta a simularii. In cazul in care datele nu au fost introduse corect se va transmite pentru a putea fi afisat un mesaj pentru reintroducerea input-urilor.

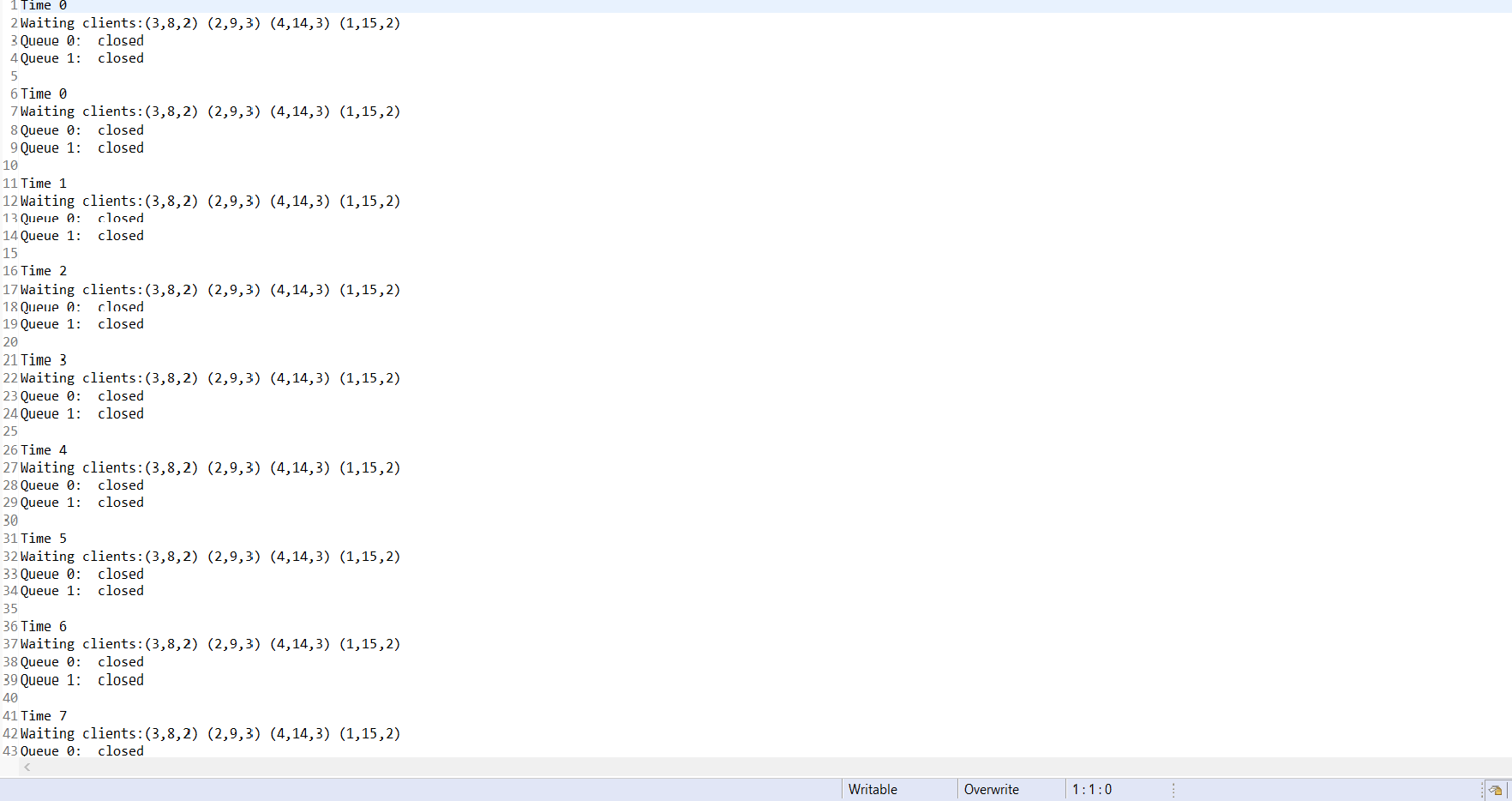
1. Clasa **Renderer** – aceasta clasa este celalalt panou adaugat ferestrei. Are o singura metoda “paintComponent” care seteaza dimensiunea ferestrei, seteaza layout-ul la null si apeleaza metoda paintComponet din parinte (super) si metoda repaint(din fereastra grafica). Metoda de desenare din parinte are rolul de a desena o scena de obiecte.
2. Clasa **SimulationFrame** – implementeaza interfata actionListener. Ea ara o lista de Consumer ( cei ce urmeaza a fi desenati ), variablie care retin datele de intrare, obiectul ferestrei in sine, obiecte de tipul panoului, constante pentru dimensiunea ferestrei, doua variabile booleene ce specifica daca fereastra trebuie sa simuleze si daca datele sunt corecte si doua timere, unul standard si unul de tipul clasei TimerSec. In constructorul acestei clase sunt create obiectele de tipul celor declarate, sunt adaugate caracteristici ferestei: vizibilitate, titlu, dimensiune, sunt adaugate cele doua panouri ferestei, se seteaza starea simularii la false (clientii nu vin la cozi) si timer-ul pornit prin metoda start(). Acest timer are doi parametri: 500 si this; primul reprezinta un numar de milisecunde si specifica la cat timp este apelat (automat) un Listener, iar “this” indica faptul ca acel listener este clasa in sine. Metode:

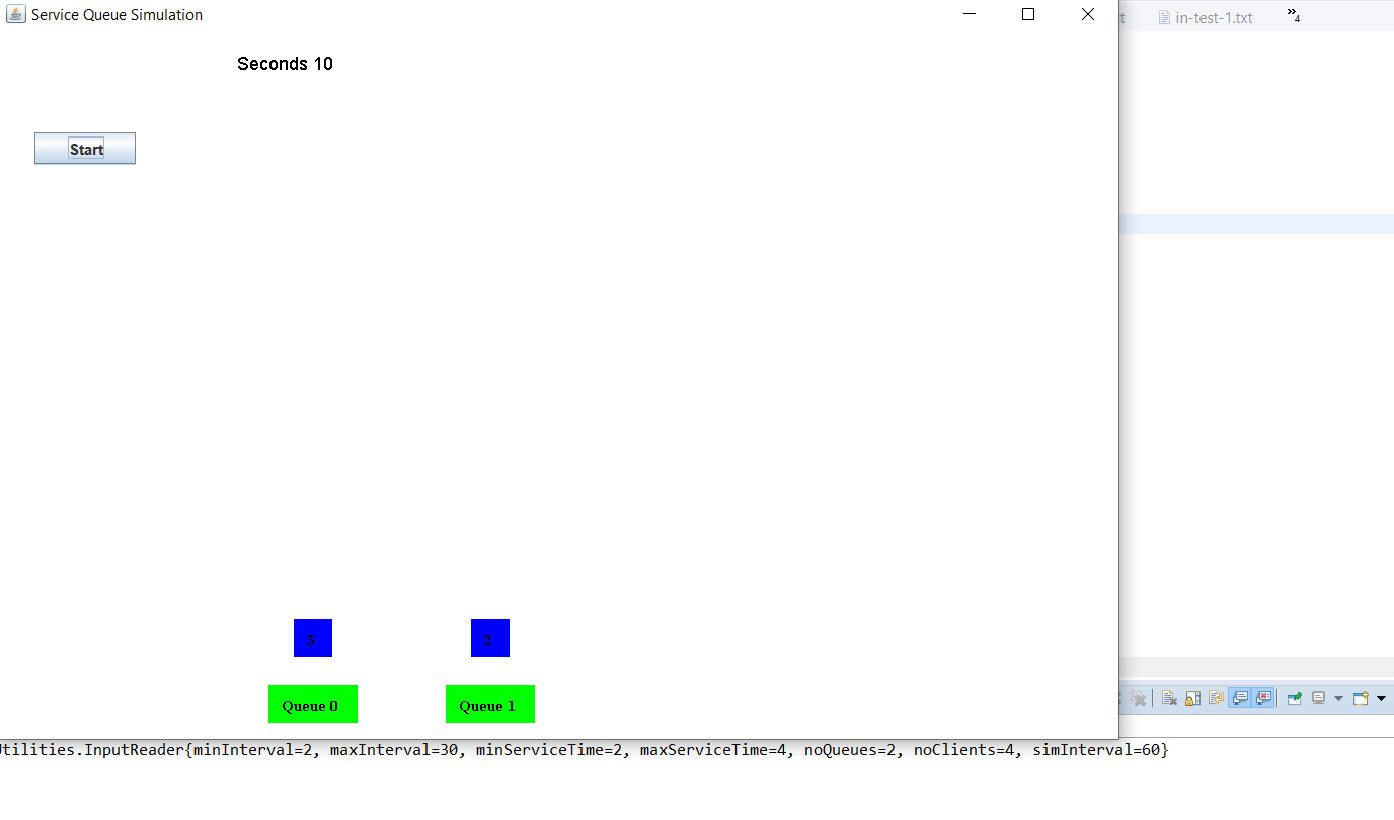
* paintHomeStores - verifica numarul de case pe care metoda trebuie sa le deseneze. Fiecare din cele cinci case sunt dreptunghiuri de culoarea verde si sunt plasate la baza ferestrei grafice in pozitie orizontala;
* Metoda detXPosition are rolul de a determina pozitia pe axa Ox pentru fiecare client in parte.
* detYPosition() calculeaza pozitia pe axa Oy dintre clienti. Porneste de la o pozitie initiala si scade acea pozitie cu spatiul necesar incadrarii unui alt client la coada. Atunci cand coada se schimba se replaseaza primul client la pozitia initiala;
* Metoda responsabila cu adaugarea formei addForm(), parcurge multimea de clienti si seteaza caracteristicile rectangle-ului “shape” la pozitile din clasa;
* paintClients seteaza pentru un client culoarea albastra, si umple un patrat dupa atributele figurei ”shape” din clasa Consumer. In interiorul acestui patrat este desenat un string ce reprezinta id-ul clientului
* actionPerformed este metoda ce se va executa incontinu pe parcursul simularii. Initial verifica daca fereastra trebuie sa simuleze si daca da, atunci timerul de tipul TimerSec este pornit prin metoda start(). Pentru fiecare client se verifica daca este timpul ca el sa asjunga la coada si seteaza la true variabila mustPaint ( din Consumer) daca acesta trebuie desanat. Pe urma se verifica daca exista clienti care si-au terminat timpul de servire si ii elimina pe acestia din lista de clienti din magazin. Se apeleaza metodele de determinare a pozitiei fiecarui client, se seteaza forma si in cazul in care totul este in regula se apeleaza metoda repaint() fara parametri pe obiectul de tipul clasei panoului de animatie.
* Metoda REPAINT primeste un obiect grafic si pe el deseneaza tot ce trebuie pus in panou. Seteaza un rectangle alb pe toata fereastra grafica, atata timp cat simularea ruleaza un string ce reprezinta secundele scurse de la inceput este pictat. In cazul in care datele sunt invalide se printeaza un “Invalid data set” pe fereastra si daca exista clienti care trebuie pictati prin metoda paintClients acestia sunt generati;
* Clasa mai contine metode pentru preluarea si setarea obiectelor private.

1. Clasa **InputReader** si **WriteOutput –** cele 2 clase sunt folosite pentru scrierea in fisierul txt (path1) a datelor introduce de la tastaura, cat si pentru afisarea datelor rezultate in urma rularii aplicatiei.
2. Clasa **ServiceQueue** – in aceasta clasa se calzuleaza suma timpilor de servire pentru clientii aflati deja la coada, setand ulterior numarul casei si adaugand clientul in coada.
3. Clasa **TimerSeconds** – aceasta clasa detine un int seconds initializat cu 0 si prin metoda run din TimerTask incrementeaza aceasta variabila. Incrementarea se face odata la o secunda astfel incat clasa implemeteaza un timer care incepe din secunda 1 si monitorizaeaza secundele trecute de la apelul metodei start().
4. Clasa **Simulation**- Clasa detine metoda principala main de unde se simuleaza proiectul. Aceasta contine un obiect de tipul clasei SimulationManager si un Thread pe acest obiect care porneste la apelul metodei start().
5. Rezultate si Testare

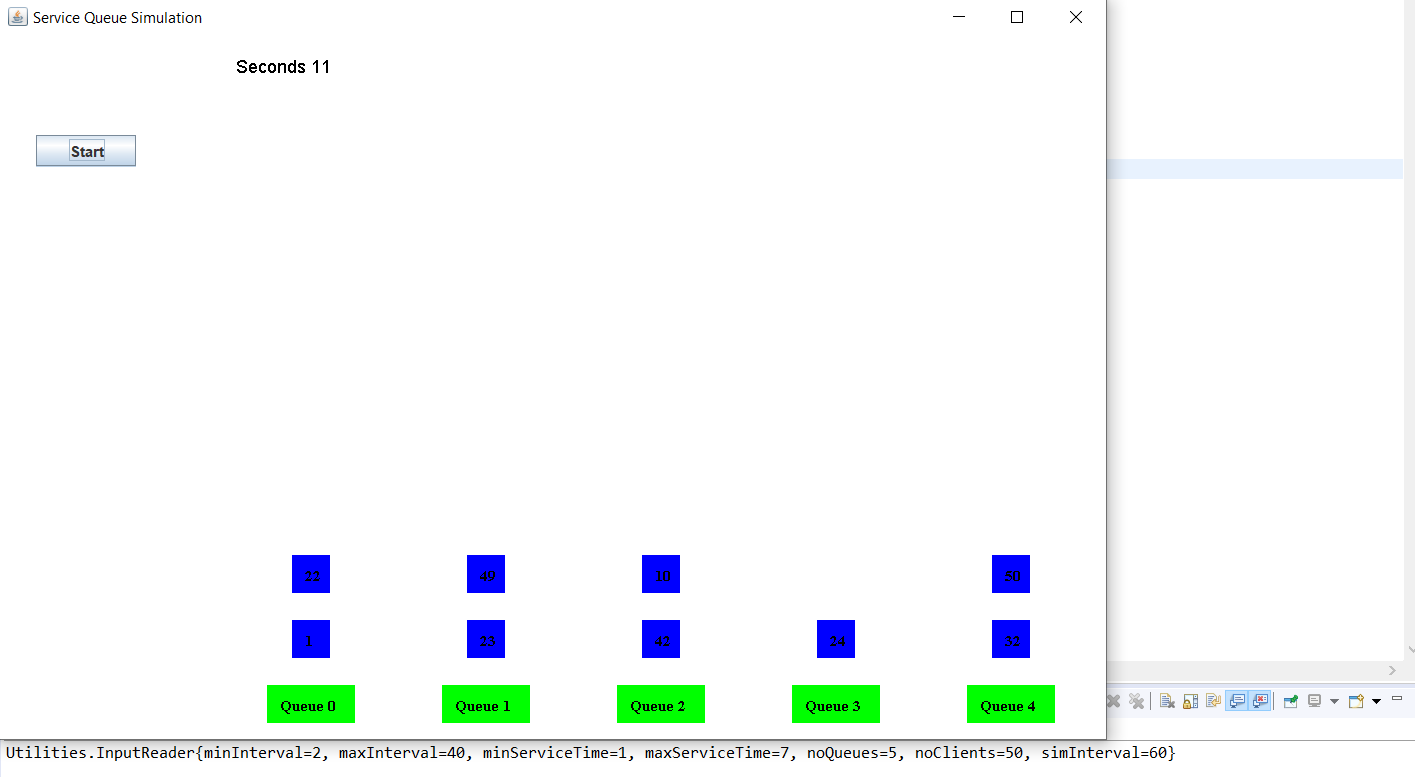
Etapa de testare din acest proiect a fost implementata doar fizic, verificand daca timpii generati sunt valizi si clienti ocupa cozile in care ar trebui sa ajunga.

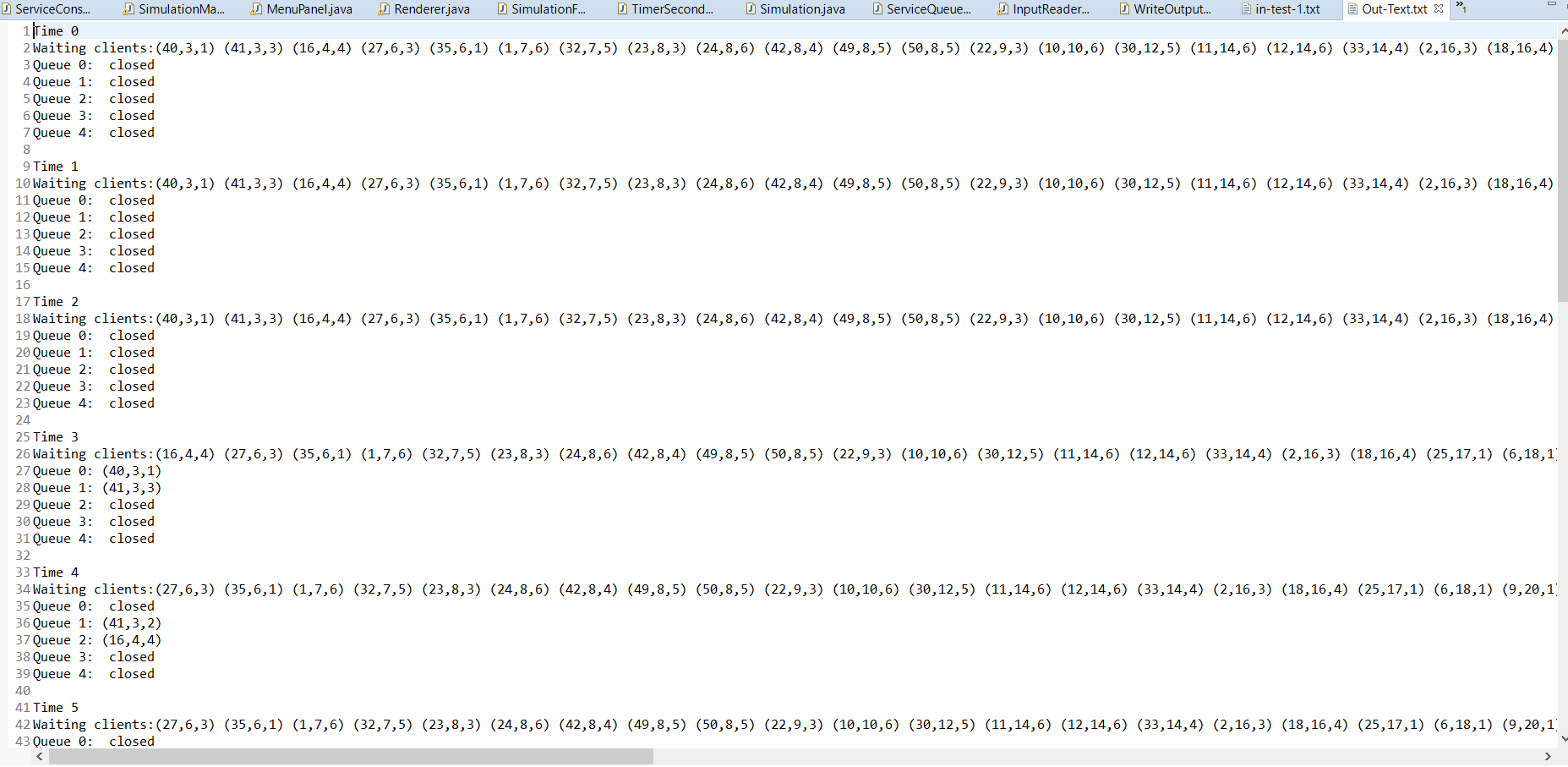
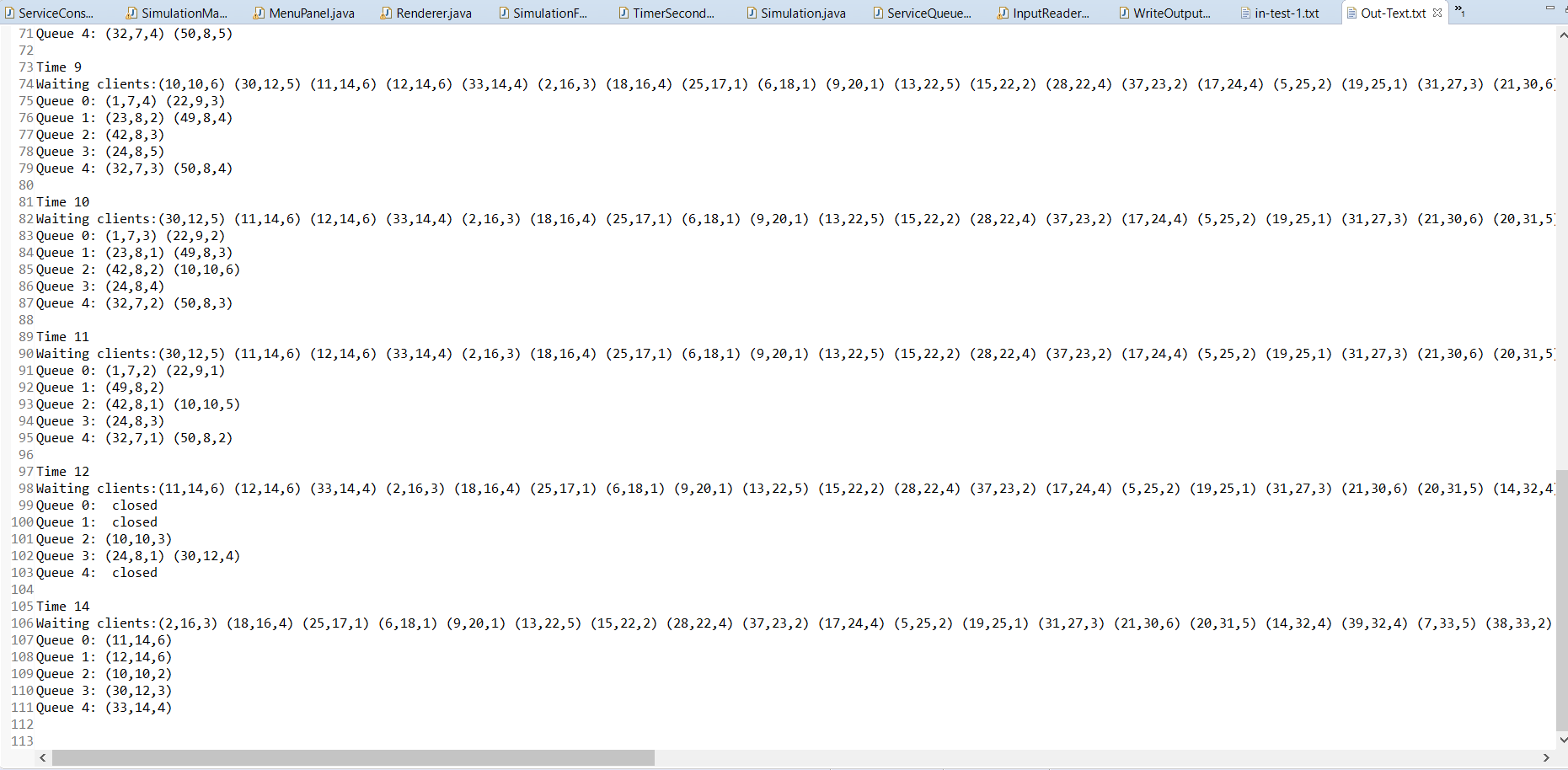
Pentru fisierul txt1:



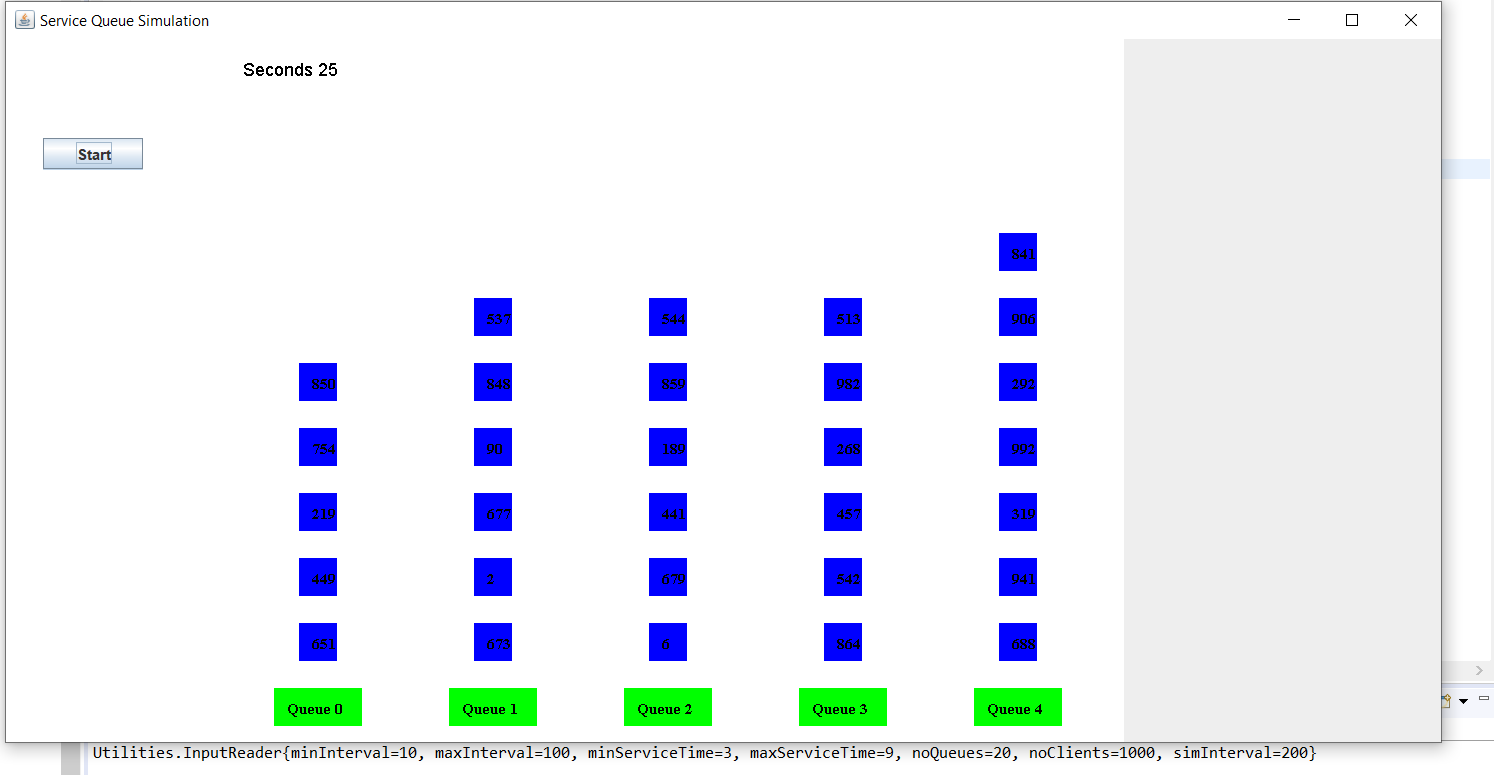


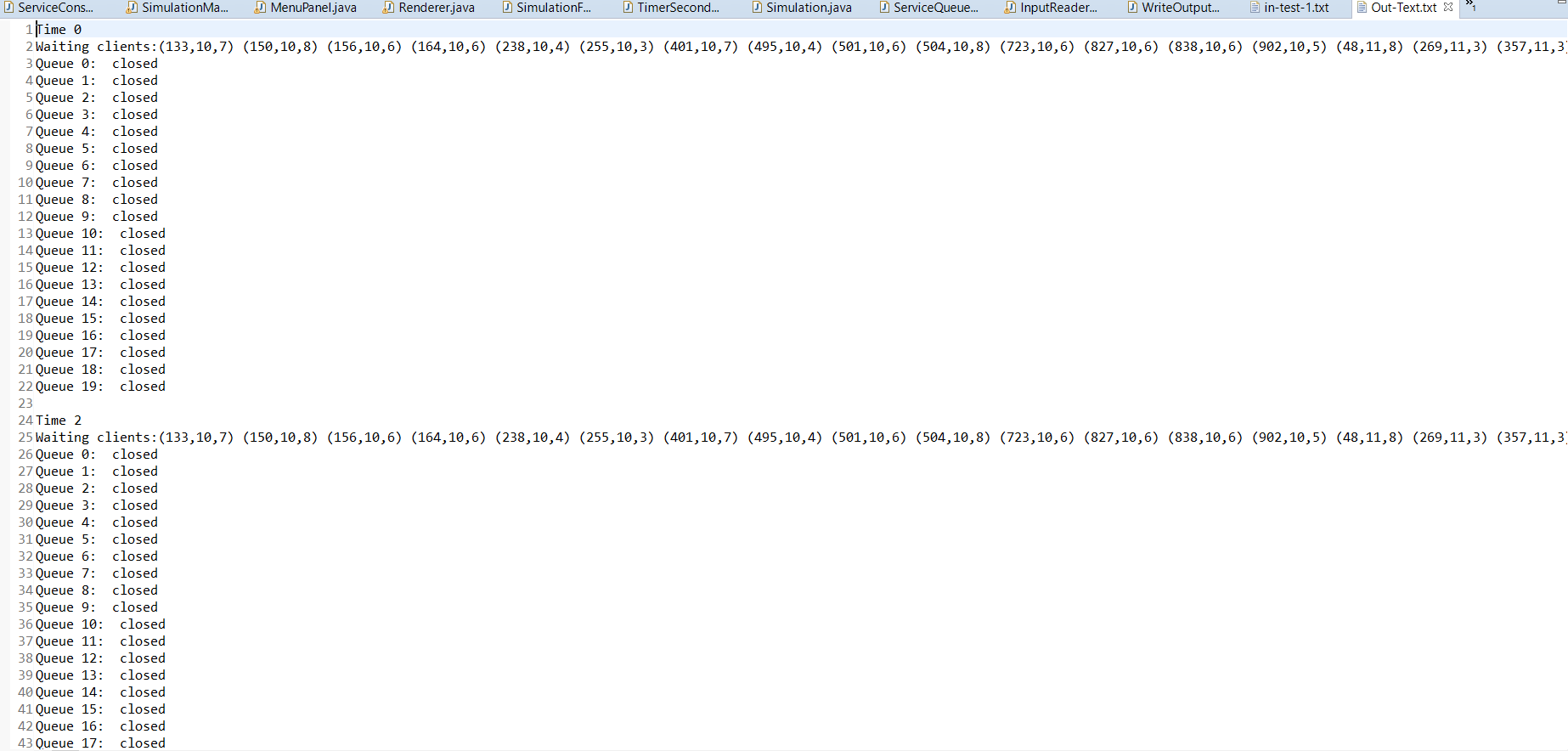
Pentru fisierul txt2:

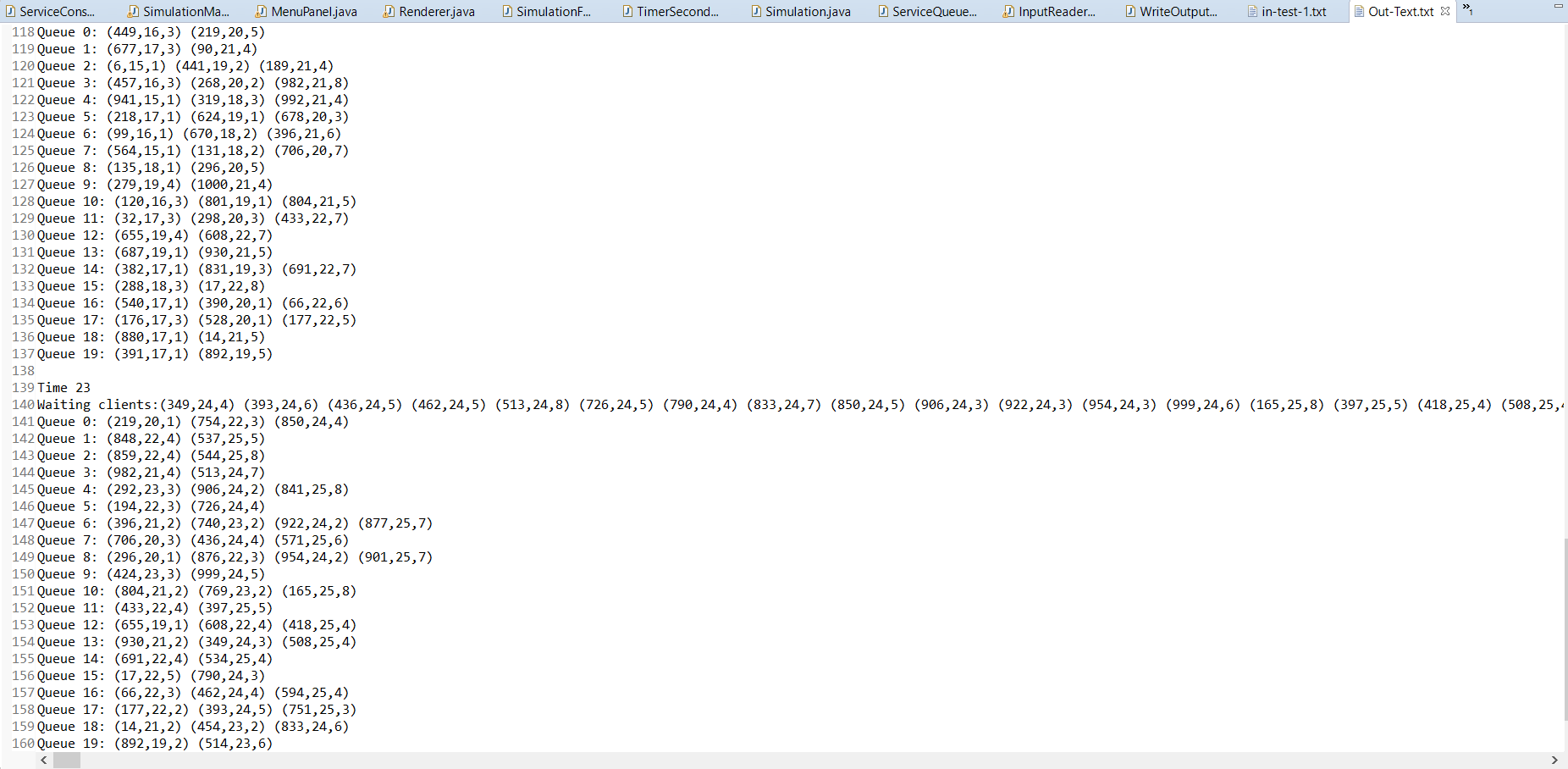




Pentru fisierul txt3:







1. Concluzii

La introducerea datelor de intrare, panoul de clasa Renderer pare a se extinde pe toata fereasta, fapt datorat metodei repaint() apelata pe obiect, cere la randul ei apeleaza repaint(0, 0, 0, width, height) care specifica faptul ca dimensiunea ferestrei incepe de la ( 0, 0 ), coltul din stanga sus.

Threadurile desi au o ordine aleatoare, manifestarea lor depinde intr-o oarecare masura de durata pe care threadul o petrece in starea de stagnare.

Am utilizat de mai multe ori “**synchronized”** desi ar fi trebuit utilizarea acestuia.

De asemenea, in panoul de afisare pentru fisierul txt3 nu vor fi afisate toate casele deoacere limita implementata este de 5 case.

O dezvoltare ulterioara a proiectului ar fi implementarea unui alt sistem de gestionare a caselor prin care acestea se deschid si se inchid sistematic in functie de numarul de clienti, acestia fiind redistribuiti.

Cu ajutorul acestui proiect am inteles mult mai bine ideea de thread si de multithreading si modul in care aceste tehnici pot ajuta la functionarea programelor.

Aceasta tema mi-a imbunatatit tehnica de programare ce respecta pattern-ul MVP. M-a ajutat sa organizez mai bine codul si sa inteleg mai bine conceptul de abstractizare a datelor.

1. Bibliografie

* <http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurre.pdf>
* <http://stackoverflow.com/>
* <https://thenewboston.com/videos.php>
* http://www.programmerinterview.com/index.php/operating-systems/thread-vs-process/