

Tehnici Fundamentale de Programare

Tema 2

Simulator de cozi

Proiect realizat de:

Moldovan Andrei

Grupa 30227

Profesor îndrumător: Dan Mitrea

Contents

[1. Obiectul temei 2](#_Toc69388177)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 2](#_Toc69388178)

[3. Proiectare 3](#_Toc69388179)

[4. Implementare 4](#_Toc69388180)

[5. Rezultate 8](#_Toc69388181)

[6. Concluzii 9](#_Toc69388182)

[7. Bibliografie 9](#_Toc69388183)

# Obiectul temei

Principalul obiectiv al acestei teme a fost de a implementa un simulator de cozi care. Acesta a trebuit să conțină o interfață grafică care să îi permite utilizatorului să introducă datele dorite, cum ar fi numărul de clienți, numărul de cozi, timpul maxim de rulare etc. Simularea generează aleator clienții care vor fi puși în cozi, iar mai apoi aceștia sunt adăugați și le va fi procesata comanda în timp real.

Obiective secundare:

* Analizarea problemei și dezvoltarea unei soluții optime
* Proiectarea unui program care să se muleze pe toate cazurile introduse de utilizator
* Implementarea soluției prin scrierea codului și dezvoltare a interfeței
* Observarea rezultatelor și formularea concluziilor

# Analiza problemei, modelare, scenarii, cazuri de utilizare

Avem mai multe cerințe funcționale care necesită să fie tratate, precum:

* Introducerea datelor de către utilizator
* Posibilitatea utilizatorului de a deschide din aplicație fișierul log
* Începerea simulării

1. Diagram

   Description automatically generatedCaz de utilizare: Începerea simulării

Actor principal: Utilizatorul

Principalul scenariu de funcționare:

* Utilizatorul introduce datele dorite pentru simularea pe care dorește sa o facă
* Acesta apasă pe butonul de ”Begin Testing”
* Aplicația începe simularea cu datele primite și va avea afișa în timp real cum vor evolua cozile si modul în care sunt procesați clienții

Secvență alternativă:

* Utilizatorul introduce date greșite care nu reprezintă numere
* Se va deschide o nouă căsuță în care se va afișa un mesaj de eroare

1. Caz de utilizare: Deschiderea fișierului

Actor principal: Utilizatorul

Principalul scenariu de funcționare:

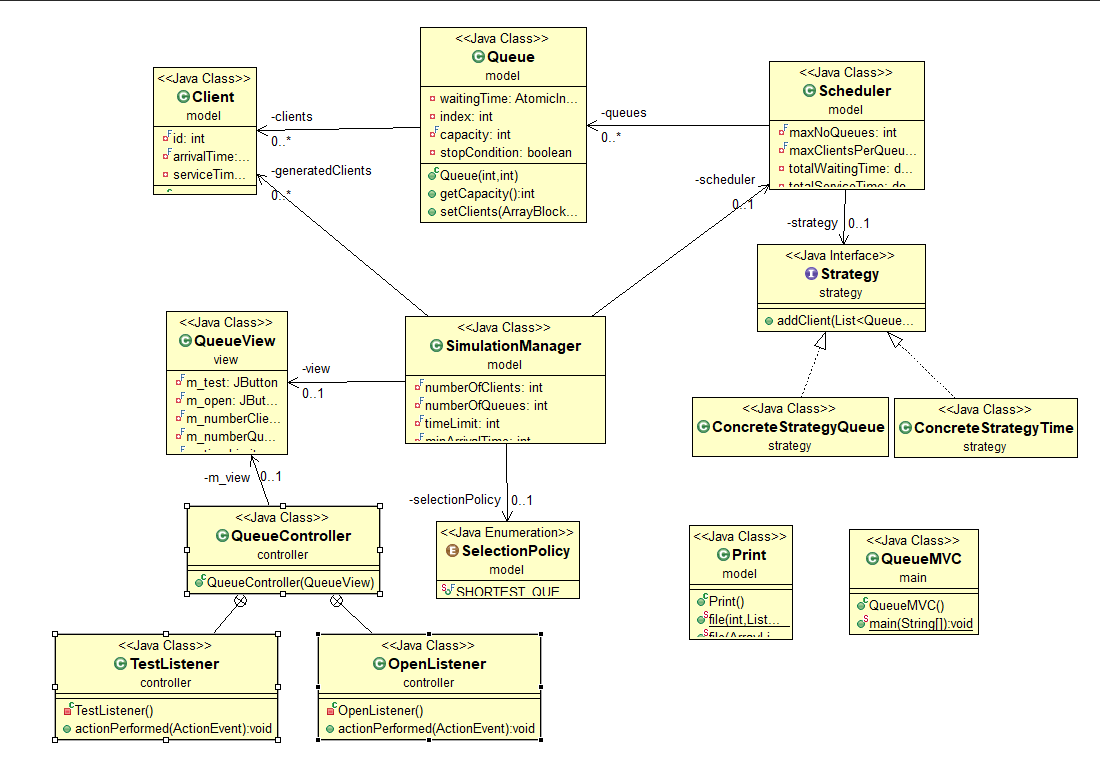
* Utilizatorul începe simularea și datele din simularea curentă sunt salvate în fișierul log
* Acesta apasă pe butonul de ”Open log file”
* Aplicația va deschide fișierul

Secvență alternativă:

* Fișierul nu există sau nu poate fi deschis
* Se deschide un o nouă fereastră în care se afișează mesajul de eroare

# Proiectare

Diagrama UML:



Soluția pe care am utilizat-o în rezolvarea cerinței necesită 3 clase diferite de stocare a datelor. Cum toate cozile trebuie să evolueze în timp real a fost necesară implementarea pe fire de execuție, fiecare coadă reprezentând câte unul. Astfel firul de execuție principal începe simularea prin generarea aleatoare a clienților, apoi prin crearea fiecărei cozi, cât și pornirea firului de execuție asociat acesteia, iar apoi adaugă clienții în cozi în funcție de strategia de adăugare pe care dorim să o folosim.

Am adăugat clasele după criteriul de funcționare, astfel să fie mai ușor de înțeles funcționalitatea fiecăreia.

Clasa pentru clienți este cea mai simplă, în ea voi memora datele despre fiecare persoană, acestea fiind id-ul asociat acesteia, apoi timpul când va sosi la cozi, dar și timpul necesar pentru a procesa comanda dorită de client.

Clasa pentru o coadă conține lista de clienți care se află în aceasta, conține indexul acesteia, pentru că lucrăm pe mai multe cozi, mai cuprinde și capacitatea maximă a cozi, iar la final o variabila care îmi spune când s-au terminat de procesat toata simularea.

Clasa pentru programarea cozilor( Scheduler) este cea responsabilă pentru crearea cozilor, crearea și pornirea firelor de execuție asociate acestora, cât și adăugarea clienților în cozi în funcție de criteriul de adăugare dorit.

Ca și algoritmi nu am utilizat nimic sofisticat, toate adăugările se fac prin parcurgeri simple și prin comparații, iar calculul timpilor de așteptare sunt calculați prin sume simple care se realizează la adăugarea fiecărui client nou.

# Implementare

1. Clasa Clienți



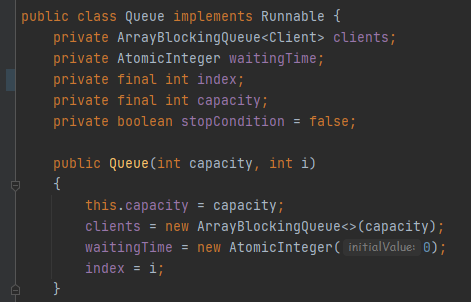
Cum am specificat deja și la secțiunea 3, am nevoie de 3 variabile care să memoreze datele despre clienți, id-ul unic asociat clientului, timpul când acesta a ajuns la cozi, cât și timpul de procesare a comenzii, iar cum id-ul si timpul de sosire nu se va schimba pe durata simulării, acestea vor primii și atributul de final.

Constructorul este simplu, în el doar memorez datele clientului care au fost primite ca și parametrii, adică id-ul, timpul de sosire și timpul de procesare a comenzii.

Metodele din această sunt cele de baza prin care luam datele de la client care sunt private pentru celelalte clase. Am mai implementat si metoda compareTo pentru a o putea utiliza în sortarea listei de clienți generată, dar asta se va regăsi și mai jos la clasa SimulationManager.

Ultima metodă din clasa aceasta este pentru a afișa clienții.

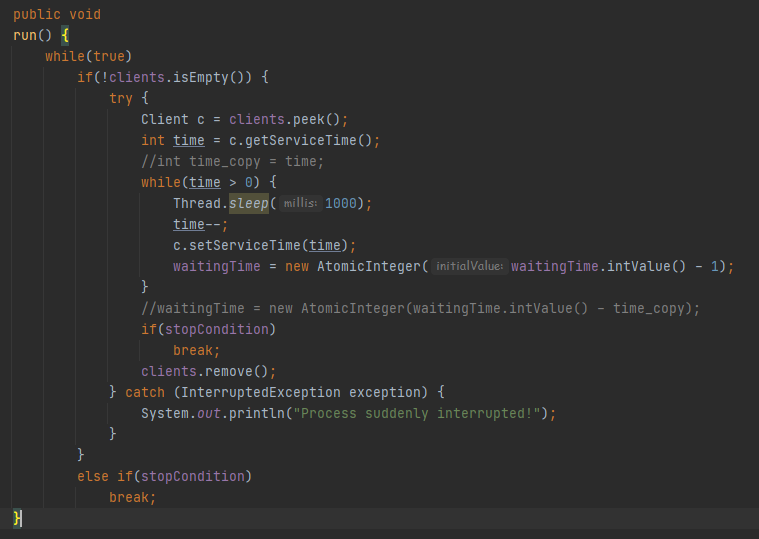
1. Clasa Queue

Aici se regăsește lista de clienți pentru care am folosit ArrayBlockingQueue, iar pentru waitingTime am folosit AtomicInteger, pentru a fi thread-safe, iar pentru celelalte date reprezintă id-ul cozi respective, capacitatea maximă de clienți pe care o poate procesa coada, iar mai apoi și condiția prin care se oprește firul de execuție al cozi.

Constructorul este unul simplu, dau valori pentru capacitate și index pe care nu le voi mai modifica, îmi creez coada unde vor putea fi adăugați clienții și inițializez waitingTime cu 0.

Următoarele metode pe care le am sunt simple, ori returnez valori prin ele, ori setez cate o valoare, cum ar fi la waitingTime, pe care îl cresc la fiecare venire a câte a unui client nou în coada respectivă.

O metoda care este puțin diferita este stopQueue, prin ea variabila booleana stopCondition va deveni adevărată, iar in metoda run, ea va fi cea care va opri execuția acestuia.

Cea mai complexă metoda din această clasa este run. Codul din această metodă va rula pe firul de execuție care este pornit în clasa Scheduler. Aceasta conține o buclă care se tot repetă la infinit dacă nu se iese din ea cu un break. Pentru a economisi timp și și pentru a evita viitoare erori, întreb dacă coada este goală, iar dacă este goală întreb dacă de fapt nu s-au terminat de adăugat clienții generați. În cazul în care coada nu este goală, atunci încep să procesez comanda primului client din coadă, iau timpul de procesare din acesta pentru a fi mai ușor de înțeles ce face bucla aceasta și continui. Pentru a face ca simularea să funcționeze în timp real, firul de execuție își va întrerupe execuția cu câte secunde durează procesarea comenzii clientului curent. Tot aici decrementez timpul pentru a se afișa corect la fiecare client noul timp de procesare. După ce iese din buclă verifică dacă nu trebuia să își oprească rularea până acum, iar daca stopCondition e tot fals, atunci continuă și elimină clientul din listă. În caz că e adevărat, atunci nu îl elimină, iar clientul va rămâne în listă.

Ultima metodă din clasa aceasta este pentru afișare.

1. Clasa Scheduler

Această clasă este responsabilă de crearea și pornirea cozilor, modul in care sunt adăugați clienții, dar si de calcularea timpilor finali necesari la afișare. Aici se regăsește lista de cozi din simularea curentă, numărul maxim de cozi, numărul maxim de clienți din fiecare coadă, strategia după care vor fi adăugați clienții în cozi, timpul total de așteptare, timpul total de procesare al comenzilor clienților, și numărul de clienți de la ora de vârf.

Constructorul nu diferă foarte mult de celelalte de până acum, dau valori pentru fiecare variabilă din clasă, îmi creez lista de cozi, iar mai apoi inițializez strategia de adăugare a clienților. Odată ce am făcut asta, pornesc firele de execuție ale fiecărei cozi.

Prima metodă care se regăsește după constructor este de schimbare a strategiei care este destul de simplă, întreb care strategie se folosește și o pun în variabilă.

În funcția de adăugare a clienților actualizez timpul de așteptare total, timpul total de procesare și adaug clientul în coada care se potrivește cu criteriul strategiei selectate.

Următoarele două metode sunt pentru calcularea timpului mediu de așteptare și de procesare.

Metodele care se regăsesc după sunt pentru actualizarea orei de vârf, unde verific numărul de clienți din fiecare coadă, iar daca totalul este mai mare decât altul anterior, actualizez ora, cât și numărul de clienți. Urmează metoda prin care returnez aceasta valoare, iar mai în față metoda prin care returnez listele de clienți.

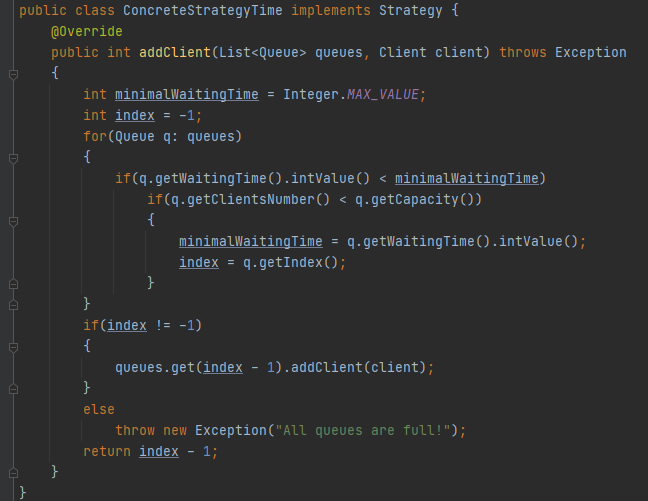
Penultima metodă verifică dacă se mai afla clienți în cozi și returnează numărul de cozi active. Ultima metodă oprește firele de execuție, actualizând la fiecare prin metoda stopQueue a variabilei stopCondition.

1. Enumerarea SelectionPolicy

În ea re regăsesc denumirile strategiilor care se pot aplica pe listele de clienți, în cazul nostru cea mai scurtă coadă sau timpul cel mai scurt de așteptare.

1. Interfața Strategy

În această interfață se regăsește formatul metodei care va fi implementat de către alte clase pentru adăugarea clienților



1. Clasele ConcreteStrategyQueue și ConcreteStrategyTime

Ambele clase au același scop, de a implementa interfața Strategy. In ConcreteStrategyTime, verific prin parcurgerea cozilor care este cel mai scurt timp de așteptare, iar dacă mai este loc în coada actualizez indexul cozi la care voi face adăugarea.

Cam același lucru se va face și in cealaltă clasă, doar că diferă criteriul după care este adăugat clientul în coadă, în aceasta veriificându-se care coadă are cei mai puțini clienți în ea, astfel făcându-se o împărțire mai uniformă a clienților în toate cozile disponibile.

1. Clasa SimulationManager

Această clasă reprezintă firul principal de execuție al simulării noastre, aici primim datele pentru simulare și se creează resursele, cât și datele necesare simulării pe care dorim să o facem.

Constructorul este destul de complex, cu foarte mulți parametrii, dar care sunt necesari pentru a inițializa datele simulării. Aici se generează apelează metoda de generare a clienților și se stabilește strategia dorită.

Următoarea metodă conține generarea clienților. Aici inițializez lista de clienți, iar mai apoi cu fiecare iterație acesta va primi id-ul, apoi timpul de sosire și timpul de procesare generate aleator între parametrii introduși de către utilizator în interfața grafică. Apelez metoda sort din clasa Collections pentru a ordona crescător clienți după timpul de sosire al fiecăruia, iar mai apoi îi afișez în aplicație și în fișierul log.

Ultima metodă din clasa este run, Aici se întâmplă simularea propriu zisă. Inițializez timpul curent, iar mai apoi setez o variabila booleană cu adevărat pentru a spune că listele sunt pornite. În buclă se verifică dacă nu s-a ajuns la timpul maxim pentru simulare, iar dacă nu, se intră în ea. Verific dacă mai am clienți în lista de așteptare, iar dacă mai sunt, verific dacă timpul de sosire este mai mic sau egal cu timpul curent al simulării, iar dacă este, adaug atâția clienți câte cozi am, doar dacă timpul de sosire este egal sau mai mic ca și cel curent. După ce l-am adăugat într-o coadă, îl elimin din lista clienților care sunt în așteptare. Acum pot să afișez lista cu clienții care sunt în așteptare, cât și lista de clienți din fiecare coadă. După acestea verific dacă mai am clienți în lista de așteptare, cât și în cozi, iar daca nu, pot sa opresc simularea. Apoi verific ora de vârf, iar la final aștept o secundă ca simularea să funcționeze în timp real, iar mai apoi incrementez timpul curent. După ce se iese din această buclă, verific dacă mai sunt cozi active, dacă mai sunt, atunci le opresc, si afișez clienții care nu au apucat să le fie procesată comanda. Iar la final afișez detaliile despre timpul mediu de așteptare, de procesare, cât și ora de vârf.

1. Clasa QueueView

Aici am creat interfața grafică a aplicației. Sunt 6 zone de scriere accesibile de utilizator pentru a introduce datele, o zona de afișare a tuturor clienților generați și după se mai regăsește încă o zonă de afișare a simulării curente.

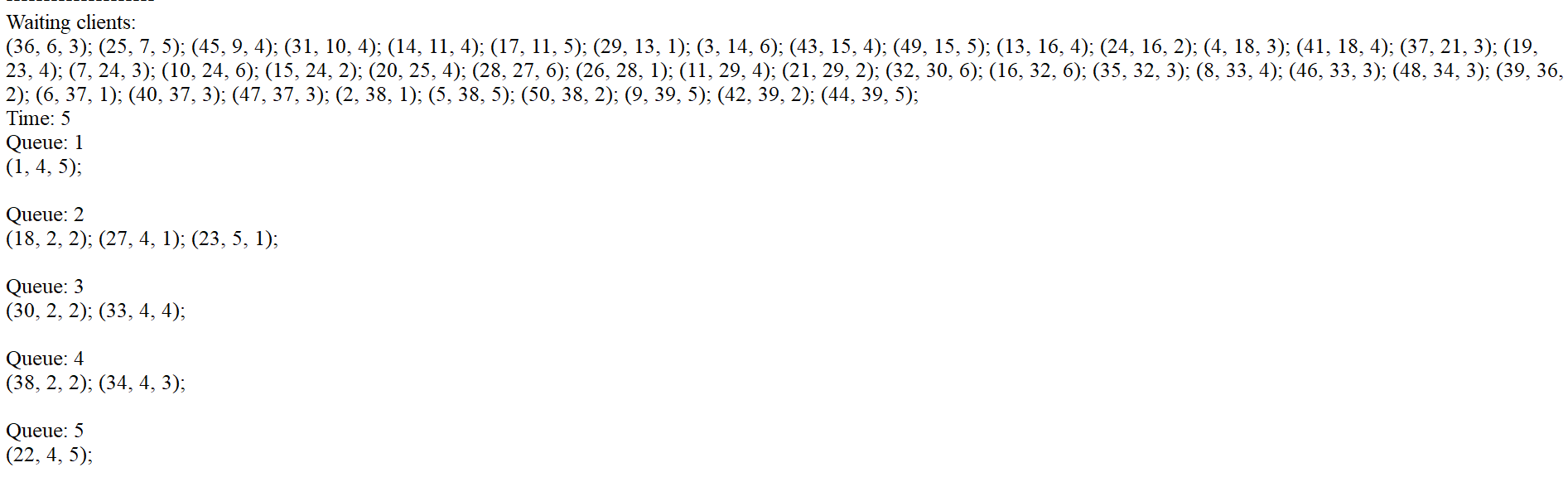
1. Clasa QueueController

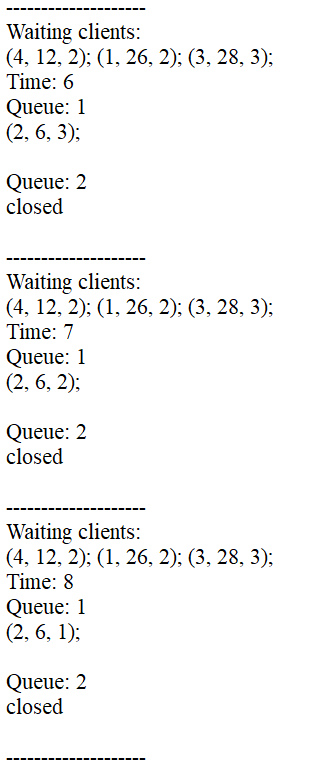
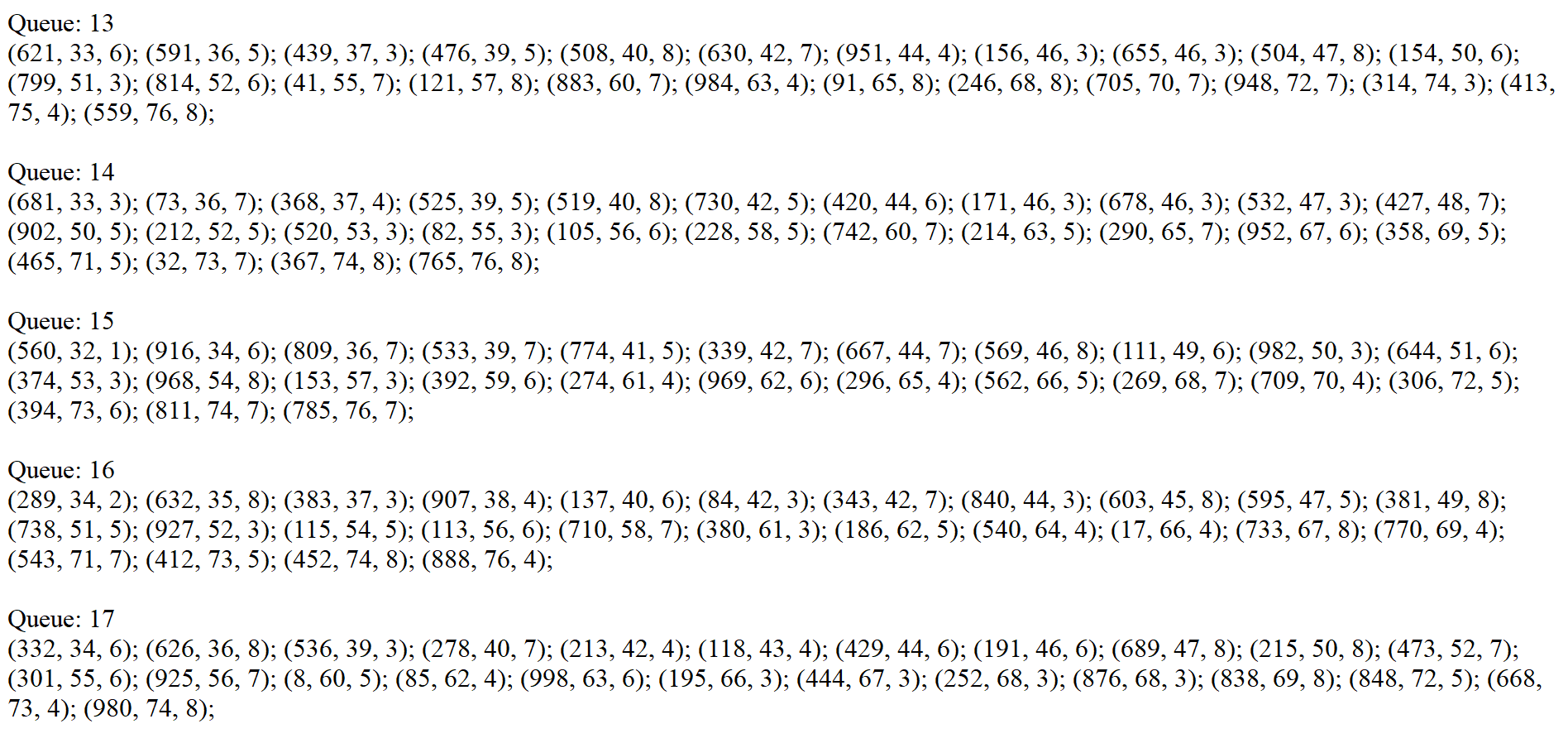
În această clasă se găsește ce va face fiecare buton, cât și pornirea și transmiterea parametrilor introduși de către utilizator.

1. Clasa QueueMVC

Este clasa main de unde se începe pornirea programului.

# Rezultate

Am reușit să rulez toate testele recomandate fără probleme și am salvat rezultatele fiecărei simulării în câte un fișier separat. Nu am sesizat să fie vreo problemă în simulările respective, astfel programul oferind soluția corectă.



# Concluzii

A fost o cerință destul de greu de realizat, pentru că nu am mai lucrat cu fire de execuție, dar am reușit să învăț modul de utilizare al acestora, cât și modul prin care pot depana un program care lucrează pe mai multe fire de execuție. Am reușit să învăț cum să folosesc noi tipuri de structuri, cât și cum creez un program care să fie thread-safe.

# Bibliografie

* <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>
* <https://www.javatpoint.com/how-to-open-a-file-in-java>
* <https://stackoverflow.com/questions/27476838/open-any-file-from-within-a-java-program>