**SIMULATOR COZI DE AȘTEPTARE MAGAZIN**

**Nume prenume: Dulău Marius Cristian**

**Grupa: 30225**

**Profesor laborator: Asist. Antal Marcel**

Contents

[1. Cerințe funcționale 3](#_Toc476131445)

[2. Obiective 3](#_Toc476131446)

[2.1. Obiectiv principal 3](#_Toc476131447)

[2.2. Obective secundare 3](#_Toc476131448)

[3. Analiza problemei](#_Toc476131449) 4

[4. Proiectare](#_Toc476131450) 5

[4.1. Structuri de date](#_Toc476131451) 5

[4.2. Diagrama de clase](#_Toc476131452) 5

[4.3. Algoritmi](#_Toc476131453) 5

[5. Implementare](#_Toc476131454) 10

[6. Testare](#_Toc476131455) 13

[7. Concluzii si dezvoltări ulterioare](#_Toc476131456) 14

[8. Bibliografie](#_Toc476131457) 15

# Cerințe funcționale

Dezvoltați o aplicație care să permită următoarele operații:

* Citire date de intrare
  + Intervalul minim și maxim al timpului de sosire al clienților la casa de marcat
  + Timpul minim și maxim de servire al unui client
  + Numărul de case de marcat
  + Intervalul de simulare
* Reprezentarea rezultatelor în funcție de datele de intrare
* Timpul mediu de servire
* Simularea cozilor de clienți pentru fiecare casă de marcat

# Obiective

## Obiectiv Principal:

Proiectați și implementați o aplicațiw de simulare cu scopul de a analiza sistemele bazate pe așteptare pentru determinarea și minimizarea timpului de așteptare al clienților.

## Obective Secundare:

|  |  |  |
| --- | --- | --- |
| **Obiectiv Secundar** | **Descriere** | **Capitol** |
| Dezvoltarea de use case-uri și scenarii | Descrierea în limbaj natural a scenariilor de utilizare și funcționare aplicației, însoțite de o diagramă. | 3 |
| Alegerea structurilor de date | Ca structuri de date s-au folosit List și BlockingQueue | 4 |
| Dezvoltarea algoritmilor | S-au conceput algoritmi pentru simulări, generare clienți aleator, poziționarea unui client în coadă, scriere în fișier, returnarea datelor prelucrate | 4 |
| Împărțirea pe clase | Proiectul este împărțit în 7 clase: ConcreteStrategyQueue, ConcreteStrategyTime, Scheduler, Server, SimulationFrame, SimulationManager și Task | 5 |
| Implementarea soluției | Implementare interfață grafică și a simulării | 5 |
| Testare | Testarea efectuării simulării în interfața grafică | 6 |

# Analiza Problemei

Utilizatorul introduce de la tastatură numărul de servere (case de marcat), numărul de clienți, intervalul de procesare a cumpărăturilor pentru fiecare client, timpul maxim la care pot ajunge clienții într-o coadă la un server. De asemenea, se poate alege tipul de strategie de poziționare pentru clienți într-o coadă de așteptare. Utilizatorul va apăsa pe butonul “Start” pentru a începe simularea. Programul va prelua datele și va crea o simularea reprezentată printr-o animație.

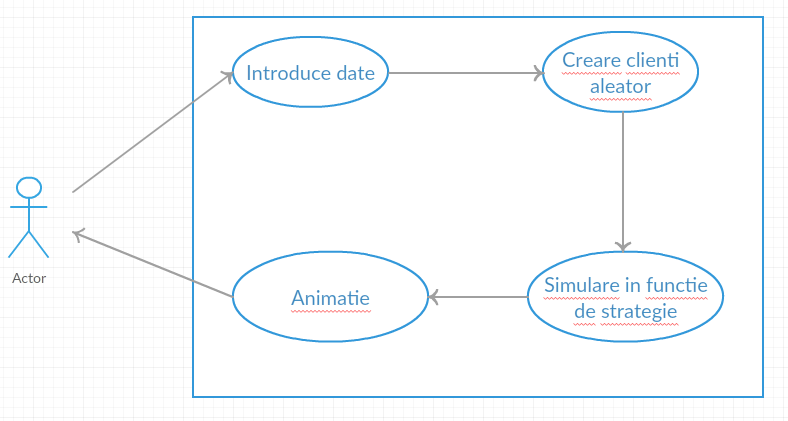


Fig 1. Diagrama use case

# Proiectare

## Structuri de date

În cadrul proiectului s-au folosit următoarele structuri de date:

* List : această structură este utilizată pentru a stoca servere și clienți;
* BlockingQueue : stochează clienții când ajung la un server. Acest tip de structură este folosit în concurență, când un thread produce un obiect și un alt thread îl folosește.

## Diagrama de clase

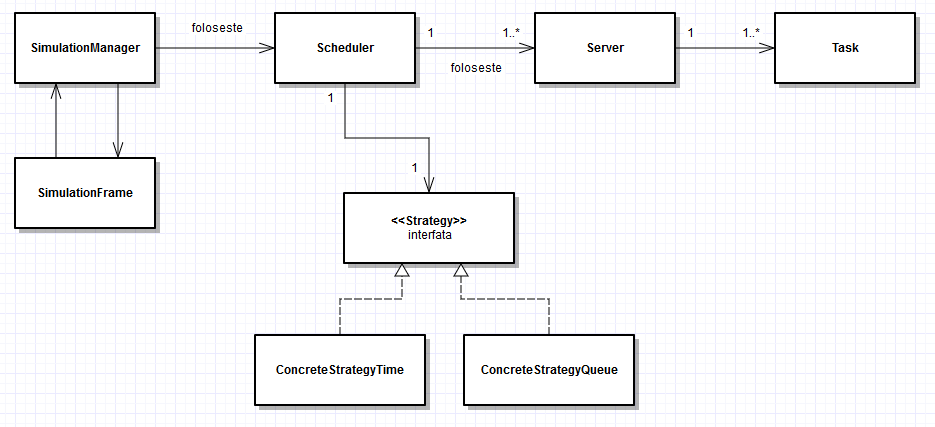


Fig 2. Diagrama de clase

## Algoritmi

Pentru a implementa aplicația a fost nevoie de o serie de algoritmi.

Pentru a realiza plasarea unui client într-o coadă s-au implementat doi algoritmi, fiecare reprezentând o strategie diferită. Strategiile sunt:

1. Timp mai puțin de așteptare la coadă

**public** **void** addTask(List<Server> servers, Task t)

{

// **TODO** Auto-generated method stub

**int** index = 0, minim = Integer.***MAX\_VALUE***;

**for**(**int** i = 0; i < servers.size(); i++)

{

Task[] ts = servers.get(i).getTasks();

**int** sumaTimp = 0;

**for**(**int** j = 0; j < ts.length; j++)

{

sumaTimp += ts[j].getProcessingPeriod();

}

**if**(sumaTimp < minim)

{

minim = sumaTimp;

index = i;

}

}

servers.get(index).addTask(t);

}

1. Cozi de așteptare mai scurte

**public** **void** addTask(List<Server> servers, Task t)

{

// **TODO** Auto-generated method stub

**int** index = 0, minim = Integer.***MAX\_VALUE***;

**for**(**int** i = 0; i < servers.size(); i++)

{

**int** serverSize = servers.get(i).getTasks().length;

**if**(serverSize < minim)

{

minim = serverSize;

index = i;

}

}

servers.get(index).addTask(t);

}

Pentru a simula cum așteaptă clienții la coadă și li se procesează bonul de la o casă de marcat folosim următorul algoritm:

**public** Task[] getTasks()

{

Task[] clienti = **new** Task[**this**.tasks.size()];

**return** (Task[]) tasks.toArray(clienti);

}

**public** **void** run()

{

// **TODO** Auto-generated method stub

**while**(**true**)

{

**try**

{

**if**(tasks.size() != 0)

{

Task task = **this**.tasks.peek();

Thread.*sleep*(task.getProcessingPeriod() \* 1000);

tasks.remove(task);

}

}**catch** (InterruptedException e)

{

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

Pentru a trimite datele despre o coadă de așteptare la interfața grafică folosim următorul cod:

Pentru a returna datele de la toate server-ele se folosește codul de mai jos.

**public** Task[][] serversData()

{

Task[][] matrix = **new** Task[servers.size()][];

**int** i = 0;

**for**(Server s : servers)

{

matrix[i++] = s.getTasks();

}

**return** matrix;

}

Timpul mediu de așteptare îl scriem într-un fișier text. Pentru a realiza acest lucru se utilizează următorul cod:

**public** **void** writeData(**int**[][] data)

{

**try**

{

PrintWriter pw = **new** PrintWriter("Logger.txt");

**for**(**int** i = 0; i < data.length; i++)

{

**if**(data[i][1] != 0) pw.println("Coada " + (i + 1) + " --> Timp mediu asteptare = " + data[i][0] / data[i][1]);

}

pw.close();

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

Pentru a simula cum un client alege coada mai scurtă aplicăm următorul algoritm:

**public** **void** run()

{

// **TODO** Auto-generated method stub

**int** currentTime = 0;

**while**(currentTime < 3 \* timeLimit / 2)

{

currentTime++;

System.***out***.print(currentTime);

**for**(Task t: generatedTasks)

{

**if**(t.getArrivalTime() == currentTime)

{

scheduler.dispatchTask(t);

System.***out***.print(" " + t.getName() + "[" + t.getProcessingPeriod() + "]" + " la coada ");

}

}

System.***out***.println("");

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

frame.displayData(scheduler.serversData());

frame.setTimp(currentTime);

**if**(currentTime == 3 \* timeLimit / 2) frame.writeData(scheduler.dateAsteptare());

}

}

Generarea datelor clienților se face în felul următor:

**private** **void** generateNRandomTasks(**int** n)

{

**int** j = 1;

**for**(**int** i = 0; i < n; i++)

{

**int** arrTime = rand.nextInt(timeLimit) + 1;

**int** procTime = rand.nextInt(maxProcessingTime - minProcessingTime) + minProcessingTime;

generatedTasks.add(**new** Task(arrTime, procTime, j++));

}

}

# Implementare

Proiectul conține 7 clase, o interfață și o enumerare.

Cele 7 clase sunt:

* ConcreteStrategyQueue
* ConcreteStrategyTime
* Scheduler
* Server
* SimulationFrame
* SimulationManager
* Task

Enumerația SelectionPolicy conține opțiunile de simulare pentru această aplicație, timp scurt (SHORTEST\_TIME) sau cozi scurte (SHORTEST\_QUEUE).

Interfața Strategy prezintă o singură metodă neimplementată, ci anume addTask(List<Server> servers, Task t). Această interfață este implementată în clasele ConcreteStrategyQueue și ConcreteStrategyTime, unde metoda addTask are un corp diferit în funcție de clasă din cauza alegerii optiunii din SelectionPolicy.

Atât clasa ConcreteStrategyTime, cât și clasa ConcreteStrategyQueue preiau inforrmațiile de la fiecare casă de marcat. ConcreteStrategyQueue poziționează clientul la casa cu cei mai puțini oameni la coadă, pe când ConcreteStrategyTime pune un cumpărător la acea coadă la care timpul de procesare al cumpărăturilor este minim.

Clasa Task stochează informații despre un anumit client, ci anume timpul de sosire la coadă, timpul de procesare al coșului de cumpărături și numele.

Câmpurile clasei sunt:

* **private** **int** arrivalTime : reprezintă timpul de sosire al clientului la o coadă de așteptare;
* **private** **int** processingPeriod : reprezintă timpul de procesare al cumpărăturilor clientului;
* **private** **int** nr : numele clientului identificat printr-un număr.

Metodele aferente clasei sunt:

* **public** **void** setArrivalTime(**int** arrTime) : setează valoarea variabilei arrivalTime cu valoare variabilei arrTime transmisă ca parametru;
* **public** **int** getArrivalTime() : returnează valoarea câmpului arrivalTime;
* **public** **void** setProcessingPeriod(**int** procTime) : setează valoarea variabilei processingTime cu valoare variabilei procTime transmisă ca parametru;
* **public** **int** getProcessingPeriod() : returnează valoarea câmpului processingTime;
* **public** **int** getName() : returnează valoarea câmpului nr.

Clasa Server modelează conceptul casei de marcat de la magazin. Ea implementează clasa Runnable deoarece se lucrează cu thread-uri.

Câmpurile aferente clasei sunt:

* **private** BlockingQueue<Task> tasks : listă de clienți;
* **private** Thread t : fir de execuție;

Metodele clasei sunt:

* **public** **void** addTask(Task newTask) : adaugă un client nou la lista tasks;
* **public** **void** run() : metodă ce realizează simularea;
* **public** Task[] getTasks() : returnează conținutul listei de clienți tasks;
* **public** **void** setProcTime(**int** nr) : incrementează timpul total de procesare al cumpărăturilor de la o casă de marcat (server);
* **public** **int** getProcTime() : returnează timpul total de procesare de clienți de la un server;
* **public** **void** setNrClienti() : incrementează numărul de clienți de la o casă de marcat (server);
* **public** **int** getNrClienti() : returnează numărul de clienți de la un server.

Clasa Scheduler decide în ce coadă (server) va pune un nou client(task).

Câmpurile clasei sunt:

* **private** List<Server> servers : listă ce conține case de marcat (obiect de tipul Server) ;
* **private** **int** maxNoServers : numărul maxim de case de marcat;
* **private** Strategy strategy : obiect de tip interfață;
* **public** **int**[][] dateAsteptare() : returnează date referitoare la numărul total de clienți ce au fost la o casă de marcat și timpul mediu de procesare al comenzilor.

Metodele aferentei clasei sunt:

* **public** **void** changeStrategy(SelectionPolicy policy) : stabileșete pe baza parametrului policy dacă variabila de tip interfață strategy va avea instanța clasei ConcreteStrategyTime sau a clasei ConcreteStrategyQueue;
* **public** **void** dispatchTask(Task t) : cu ajutorul acestei metode se plasează un obiect Task (client) la un server (sau la coada unei case de marcat) pe baza strategiei alese;
* **public** List<Server> getServers() : returnează lista de server-e (case de marcat) ;
* **public** Task[][] serversData() : returnează o matrice de obiecte tip Task (client) pe baza conținutului listei servers.

Clasa SimulationManager este clasa principală (main) a acestui proiect. Ea preia informații de la interfață și creează date pentru simulare. După crearea datelor începe simularea propriu-zisă. Câmpurile acestei clase sunt următoarele:

* **public** **int** timeLimit : timpul maxim la care poate ajunge ultimul client la o coadă ;
* **public** **int** maxProcessingTime : limita superioară pentru timpul de procesare a cumpărăturilor clientului;
* **public** **int** minProcessingTime : limita inferioară pentru timpul de procesare a cumpărăturilor clientului;
* **public** **int** numberOfServers : numărul de case de marcat;
* **public** **int** numberOfClients : numărul de clienți;
* **public** String option : opțiunea aleasă de utilizator;
* **public** SelectionPolicy selectionPolicy : strategia de poziționare la o coadă în funcție de câmpul option;
* **private** Scheduler scheduler : planificator pentru a introduce un client într-o coadă;
* **private** SimulationFrame frame : interfața cu utilizatorul;
* **public** List<Task> generatedTasks : clienții generați aleator cu timp de sosire în coadă și timp de procesare a comenzii;
* **private** Random rand : acest câmp este utilizat pentru a crea valori aleatoare pentru timpul de sosire și de procesare;
* **private** Thread t : fir de execuție;

Metodele aferente clasei sunt:

* **private** **void** generateNRandomTasks(**int** n) : în această metodă se creează valori aleatoare pentru timpul de sosire și de procesare pentru fiecare client în parte;
* **public** **void** run(): metodă ce realizează simularea;
* **public** **static** **void** main(String[] args) : în această metodă se pornește execuția programului;

Această clasă implementează un ascultător pentru butonul “Start” din interfața grafică, fiind astfel un MVC (Model View Controller).

Clasa SimulationFrame reprezintă interfața grafică ce interacționează cu utilizatorul. În figura de mai jos se observă că interfața cu utilizatorul conține 6 label-uri (etichete), 5 casete de text, un combo box, un buton, un separator și scrollPanel. După ce apăsăm

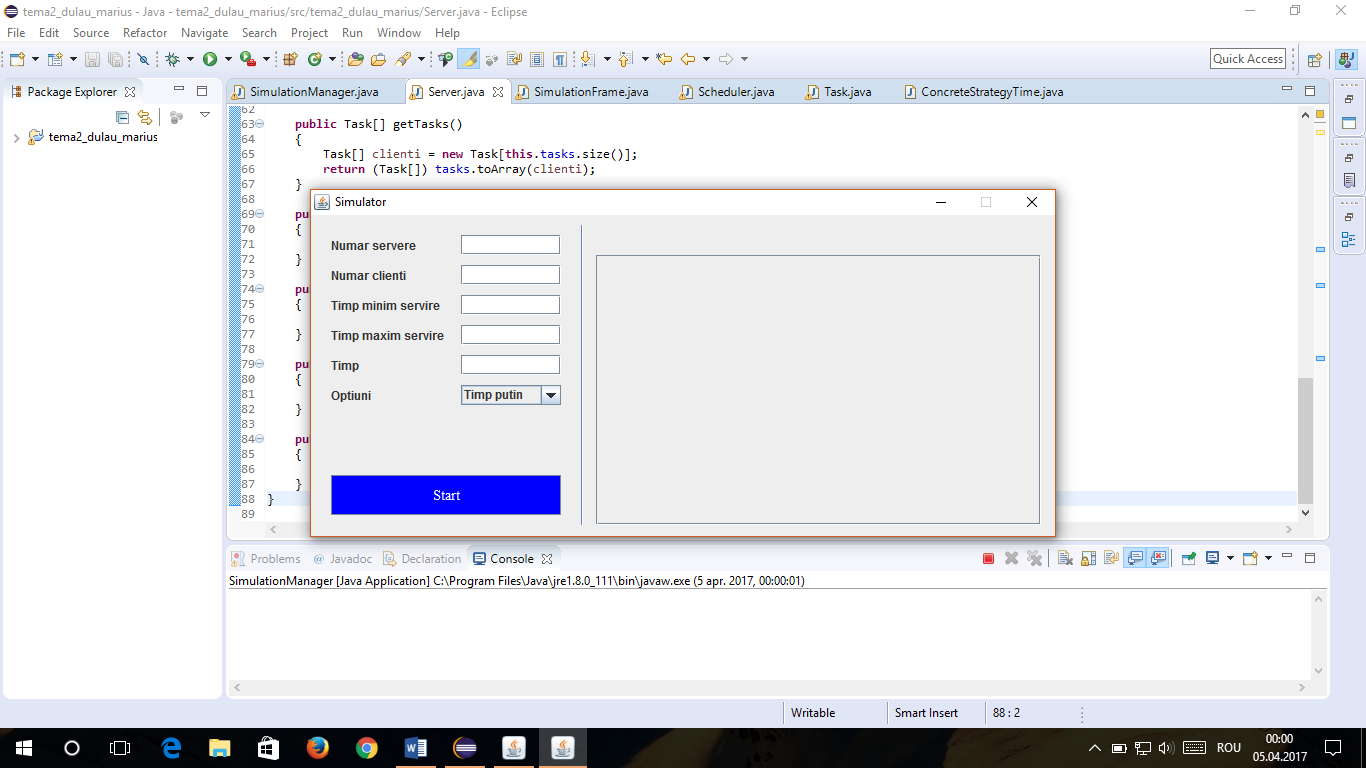


Fig 3. Interfața grafică

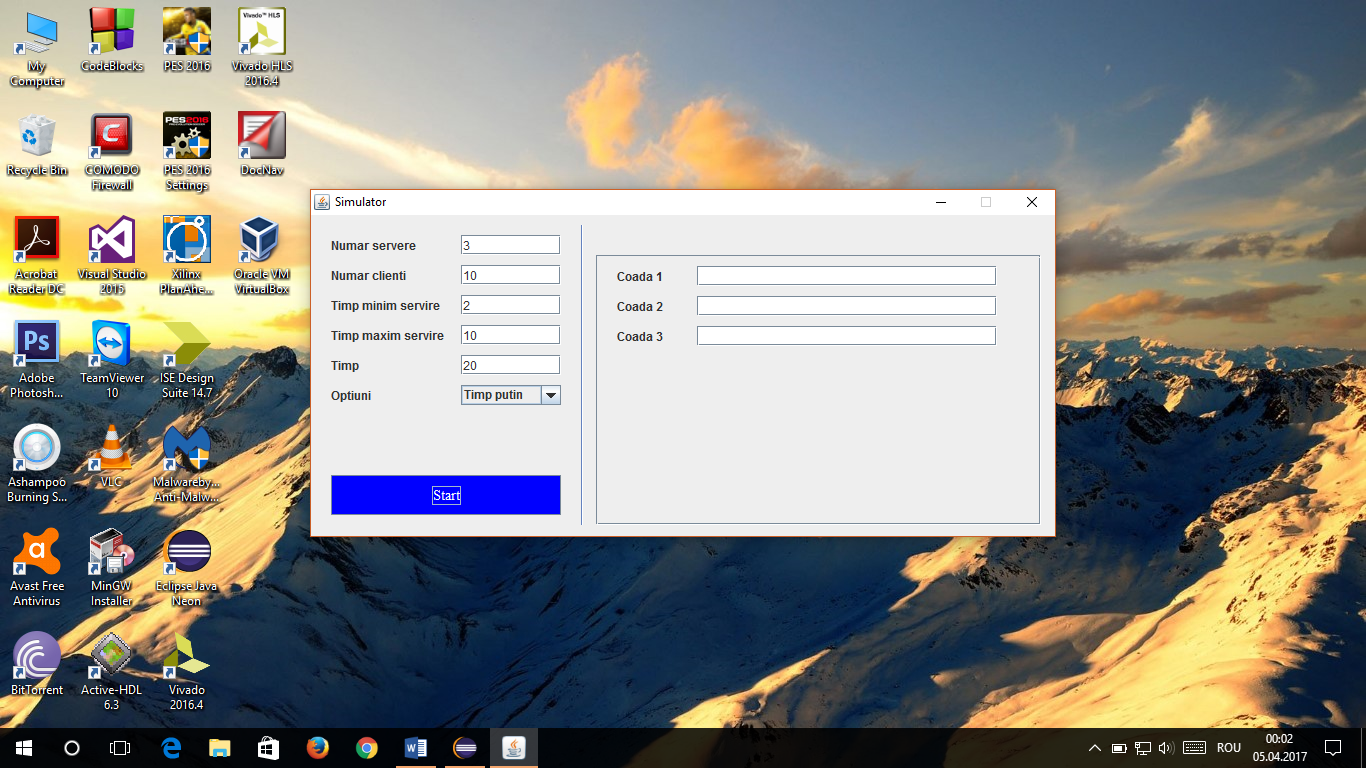


Fig 4. Interfața grafică – Completare date și apăsare buton “Start”

# Testare

Pentru a urmări corectitudinea simulării, testarea a avut loc observând ceea ce se întâmplă în interfața grafică cu ce se întâmplă în consola programului Eclipse.

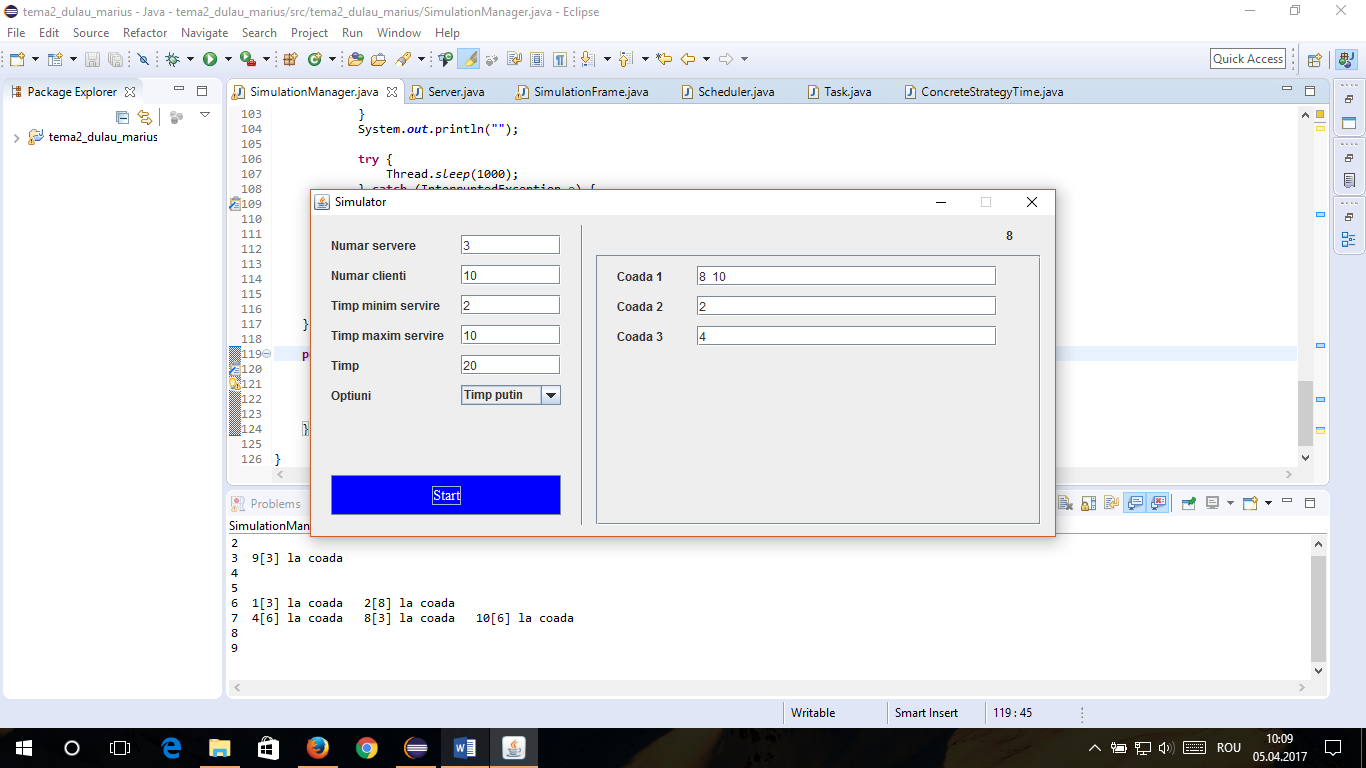


Fig 5. Testare aplicație prin interfața grafică

În consolă se scrie la fiecare moment de timp când apare un client în coadă și timpul său de procesare. Când acesta ajunge la casă (server) se urmărește timpul t la care a ajuns să i se proceseze coșul de cumpărături. După, se urmărește dacă la momentul t + timpProcesareClient acesta pleacă din coadă.

# Concluzii și dezvoltări ulterioare

În urma dezvoltării acestui proiect am învățat să lucrez cu threaduri și să preiau informații de la interfața grafică dintr-o altă clasă. Am înțeles mai bine prin implementare procedeul MVC(Model View Controller).

Printre dezvoltările ulterioare se numără:

* Crearea de animație cu imagini, nu cu numere
* Timpul să fie reprezentat sub forma ore:minute:secunde, nu cu un număr întreg
* Afișarea timpul de așteptare pentru fiecare server
* Afișarea orelor de vărf

# Bibliografie

<http://stackoverflow.com/questions/11496700/how-to-use-printwriter-and-file-classes-in-java>

<http://coned.utcluj.ro/~marcel99/PT/3%20-%20Presentation%20-%20Java%20Concurrency-3.pdf>

<http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html>

<http://stackoverflow.com/questions/9572795/convert-list-to-array-in-java>