

**UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA**  
**FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE**

**Tehnici de programare fundamentale**

**Tema 2**

# **Simulator de cozi**

**- Documentație -**

*Realizat de:*

Duca David Alex – grupa 30223

## *Cuprins*

1. Obiectivele temei .....	3
2. Analizarea problemei .....	4
3. Proiectarea temei .....	5
4. Implementarea temei .....	10
5. Testarea temei și rezultate .....	13
6. Concluzii și dezvoltări ulterioare .....	16
7. Webografie .....	16

## 1. Obiectivele temei

**Obiectivul principal** al temei este realizarea unui simulator de cozi, la care se prestează un anumit serviciu unor clienți. Simulatorul trebuie să aibă o interfață grafică, care trebuie să îi permită utilizatorului să introducă un set de date de intrare și să vadă rezultatele simulării pe baza acelor date.

**Obiectivele secundare** ale temei sunt următoarele:

- Analizarea problemei și identificarea resurselor necesare realizării temei propuse – *capitolul 2*
- Realizarea design-ului simulatorului de cozi – *capitolele 3 și 4*
- Implementarea simulatorului – *capitolul 4*
- Testarea simulatorului – *capitolul 5*

## 2. Analizarea problemei

### 2.1. Cerințe funcționale

Simulatorul de cozi trebuie să îi permită utilizatorului să introducă un set de date de intrare (timpul de simulare, timpii minim și maxim de sosire, timpii minim și maxim de procesare/prestare a serviciului, numărul de cozi și numărul de clienți), să selecteze strategia de selectare a cozilor (cel mai scurt timp de așteptare sau cea mai scurtă coadă), să vadă simularea generată pe baza acelor date de intrare în timp real și rezultatele acesteia într-un fișier text, după terminare.

### 2.2. Cerințe non-funcționale

Simulatorul de cozi trebuie să vină însoțit de o interfață intuitivă, ușor de folosit de către utilizator, care să îi permită acestuia să introducă datele de intrare și să vadă rezultatele simulării. De asemenea, interfața grafică trebuie să atenționeze utilizatorul în cazul terminării simulării sau a introducerii greșite a datelor de intrare.

### 2.3 Cazuri de utilizare

Pentru toate cazurile de utilizare prezentate, *actorul principal este utilizatorul simulatorului de cozi.*

#### 1. *Caz de utilizare: simulare*

**Scenariu principal** (simulare realizată cu succes):

1. Utilizatorul introduce, prin interfață, setul de date de intrare pentru simulare:
  - Timpul total de simulare
  - Timpul minim de sosire
  - Timpul maxim de sosire
  - Timpul minim de procesare/prestare a serviciului
  - Timpul maxim de procesare/prestare a serviciului
  - Numărul de clienți
  - Numărul de cozi
2. Utilizatorul selectează strategia de alegere a cozilor pentru procesare:
  - alegerea cozii cu cel mai mic număr de clienți  
sau
  - alegerea cozii cu cel mai mic timp de așteptare
3. Utilizatorul apasă butonul de simulare (“Simulare”)
4. Se deschide o nouă fereastră pentru afișarea simulării
5. Se derulează și afișează în fereastra nouă simularea, în timp real, în felul următor:
  - Momentul de timp – între 0 și timpul maxim de simulare
  - Numărul de clienți care așteaptă să fie plasați într-o coadă
  - *Opțional – dacă numărul clienților este mai mic sau egal cu 10:* Clienții, cu timp de sosire și de procesare generate aleator
  - Cozile și starea lor – “Închisă”, dacă e goală; altfel, se afișează clienții din acea coadă
6. Se așteaptă terminarea simulării
7. Se generează un fișier cu toată derularea simulării și cu statisticile finale
8. Se deschide o nouă fereastră care indică terminarea simulării și generarea fișierului simulării

**Scenariu secundar** (Date de intrare neintroduse):

- Utilizatorul nu a completat toate câmpurile din interfața grafică, necesare pentru pornirea simulării
- Se deschide o nouă fereastră de atenționare pentru utilizator
- Se revine în scenariul principal la pasul 1.

**Scenariu secundar** (Date de intrare incorecte):

- Utilizatorul a introdus date de intrare în mod greșit: timpul minim pentru sosire (sau procesare) este mai mare decât timpul maxim pentru sosire (sau procesare)
- Se deschide o nouă fereastră de atenționare pentru utilizator
- Se revine în scenariul principal la pasul 1.

## 2. Caz de utilizare: verificarea rezultatelor simulării

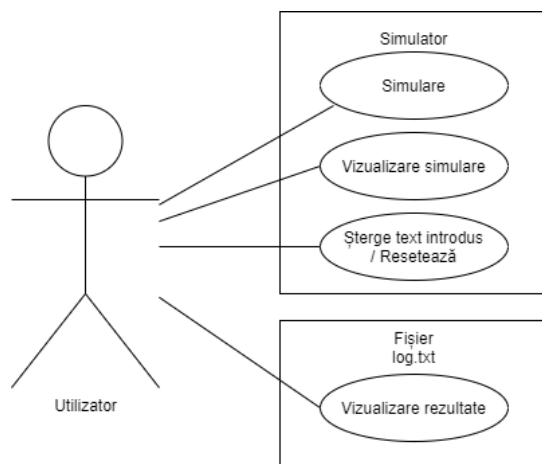
### Scenariu principal:

1. Se realizează **scenariul principal** din **Cazul 1**. de utilizare, care are ca și consecință generarea fișierului cu datele complete ale simulării
2. Utilizatorul deschide fișierul **log.txt**, generat în directorul în care este plasat simulatorul
3. Utilizatorul poate citi datele complete ale simulării:
  - Fiecare moment de timp detaliat (ca în **pasul 5**. al **scenariului principal** din **Cazul 1**. de utilizare)
  - La finalul fișierului – statistici ale simulării
    - Timpul mediu de așteptare la cozi
    - Timpul mediu de procesare a serviciilor
    - Timpul de vârf (momentul de timp cu cei mai mulți clienți la cozi)

## 3. Caz de utilizare: ștergerea datelor de intrare introduse

### Scenariu principal:

1. Utilizatorul introduce în câmpurile disponibile text de orice tip
2. Utilizatorul apasă butonul de ștergere al câmpurilor (“Resetare”)
3. Fereastra de setare a simulatorului șterge textele introduse de utilizator



## 3. Proiectarea temei

Pentru realizarea calculatorului de polinoame am ales modelul arhitectural Model–View–Controller (MVC). Acest model împarte aplicația în trei părți principale: partea de date (Model), partea de ieșiri (View) și partea de intrări (Controller).

**Partea de date (“Model”)** – reține datele utilizate, funcționalitățile și lucrează cu acestea

**Partea de ieșiri (“View”)** – afișează rezultate și informații utilizatorului

**Partea de intrări (“Controller”)** – primește input sub formă de evenimente (mișcări de mouse, apăsări de butoane, etc.) și le traduce în cereri de servicii, care sunt adresate fie părții de date, fie părții de ieșire

Modelul MVC este avantajos, deoarece permite structurarea ușoară a proiectului, conectarea logică între părți, accesul mai rapid la diferite elemente și reutilizarea acestora.

Primul pachet definit va fi pentru *modelul de date*. Simulatorul trebuie să lucreze cu unul sau mai mulți clienți și cu una sau mai multe cozi. Fiecare coadă poate avea unul sau mai mulți clienți, dar procesează serviciul doar pentru primul client la un moment dat. Primele clase care trebuie implementate sunt clasele pentru **client** și pentru **coadă**. Pentru abstractizare, vom redenumi aceste clase în felul următor: clientul se va numi **task (sarcină)** iar coada se va numi **server**. Așadar, serverele vor procesa sarcini.

Prima clasă care trebuie implementată este clasa pentru **sarcină**. O sarcină trebuie să aibă un timp de sosire pentru plasarea în server și un timp de procesare a ei. De asemenea, o sarcină ar trebui să aibă un număr de ordine (ID), influențat de timpul de sosire, pentru afișarea simulării. Clasa va conține constructori și metode de accesare și setare a atributelor.

Task
+ id: int
+ arrivalTime: int
+ processingTime: int

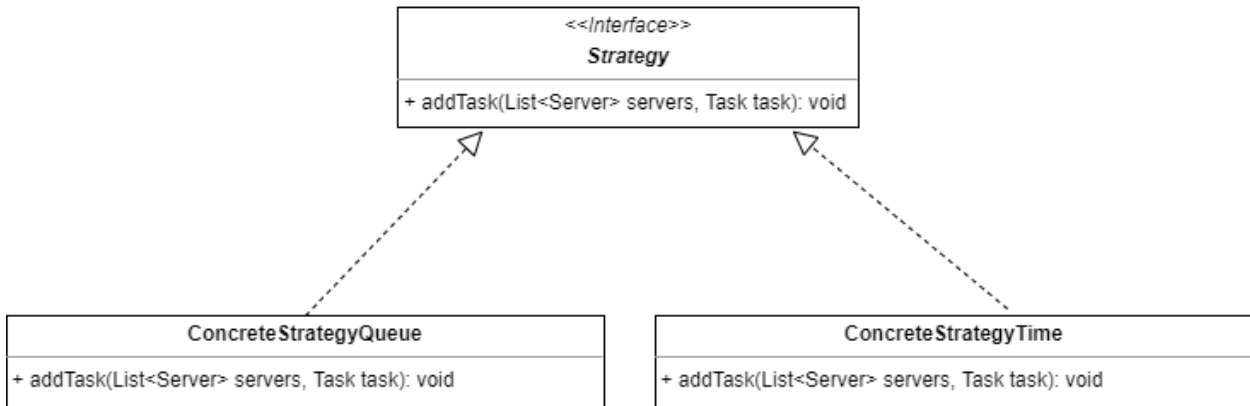
Următoarea clasă care trebuie implementată este clasa pentru **server**. Un server va avea de lucrat cu o listă de sarcini. Simulatorul va conține unul sau mai multe servere care trebuie să lucreze în mod concurent. Din această cauză, vom utiliza un *BlockingQueue* pentru lista de sarcini. Acest tip de listă va bloca introducerea sau procesarea de sarcini dacă nu se pot realiza operațiile. Vom mai adăuga un câmp pentru timpul de așteptare al serverului. Acest timp arată timpul total de procesare a tuturor sarcinilor din server și crește la adăugarea unei noi sarcini în server cu timpul de procesare al sarcinii adăugate. Vom avea nevoie de acest timp pentru calculul timpului mediu de așteptare și pentru strategia de alegere a serverului cu cel mai scurt timp de așteptare. Timpul de așteptare va avea tipul *AtomicInteger*, deoarece acest timp se modifică permanent, cât timp există sarcini de executat. Această clasă va avea un constructor și metode pentru adăugarea unei sarcini în server și pentru procesarea serviciilor. În consecință, clasa va implementa interfața *Runnable*.

Server
+ tasks: BlockingQueue<Task>
+ waitingTime: AtomicInteger
+ addTask(): void
+ run(): void

Vom avea nevoie de o clasă care să lucreze cu o listă de servere și să pună un task în serverul potrivit, în funcție de strategie. Această clasă se va numi **scheduler (programator)**. Această clasă va avea ca attribute lista de servere, numărul maxim de servere, numărul maxim de sarcini care pot fi pe un server și strategia de alegere a unui server pentru adăugarea unei sarcini. Clasa va conține metode pentru adăugarea unui task într-un server în funcție de strategie (atribut).

Scheduler
+ servers: List<Server>
+ maxNumServers: int
+ maxTasksPerServer: int
+ strategy: Strategy
+ dispatchTask(Task task): void
+ changeStrategy(SelectionPolicy policy): void

Pachetul *model* va conține un subpachet pentru **strategii**. Avem două strategii de implementat: cea mai scurtă coadă și cel mai mic timp de așteptare la coadă. Aceste strategii vor avea clasele lor proprii: **ConcreteStrategyQueue** și **ConcreteStrategyTime**. Aceste clase vor implementa o interfață **strategie**, care va avea o singura metodă de adăugare a unei sarcini într-un server dintr-o listă de servere. Parametrii acestei metode vor fi, așadar: o listă de servere și o sarcină care trebuie adăugată. Această metodă va apela, în final, pentru fiecare strategie, metoda de adăugare a unei sarcini într-un server concret (serverul ales ca fiind cel mai bun pentru strategie).



În următoarele rânduri vor fi descriși algoritmiile celor două strategii posibile.

**ConcreteStrategyQueue** descrie strategia alegerii celei mai scurte cozi. Algoritmul va fi implementat în metoda de adăugare a unei sarcini din interfața de *strategie* în felul următor:

Fie *task* sarcina care trebuie introdusă și *servers*, o listă de servere în care trebuie să introducem sarcina respectivă

Fie *minNumTasks* o variabilă care va reține numărul minim de sarcini ale unui server din listă și *bestServerIndex*, indexul în listă al celui mai bun server pentru această strategie

```

minNumTasks <- numărul de sarcini din primul server din listă
bestServerIndex <- indexul primului server (0)
pentru fiecare server din servers execută
    currentNumTasks <- numărul de sarcini din serverul curent
    dacă currentNumTasks < minNumTasks atunci
        minNumTasks <- currentNumTasks
        bestServerIndex <- indexul serverului curent
sfârșit dacă
sfârșit pentru
adaugă în servers[bestServerIndex] sarcina task // apelează metoda de
adăugare din clasa server (addTask)
  
```

**ConcreteStrategyTime** descrie strategia alegerii cozii cu cel mai scurt timp de așteptare. Algoritmul este asemănător cu algoritmul descris la strategia anterioară:

Fie *task* sarcina care trebuie introdusă și *servers*, o listă de servere în care trebuie să introducem sarcina respectivă

Fie *minWaitingPeriod* o variabilă care va reține timpul minim de așteptare pentru un server și *bestServerIndex*, indexul în listă al celui mai bun server pentru această strategie

```

minWaitingPeriod <- timpul de așteptare al primului server din listă
bestServerIndex <- indexul primului server (0)
pentru fiecare server din servers execută
    currentWaitingPeriod <- timpul de așteptare al serverului curent
    dacă currentWaitingPeriod < minWaitingPeriod atunci
        minWaitingPeriod <- currentWaitingPeriod

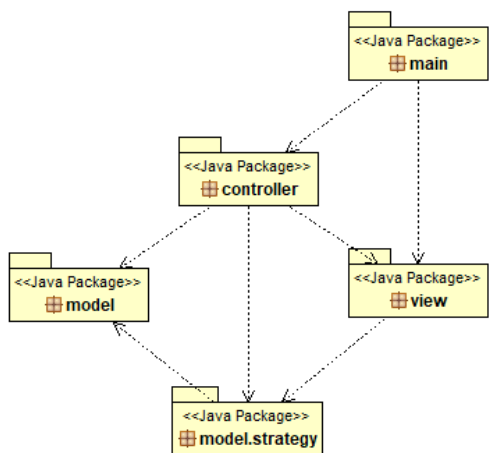
        bestServerIndex <- indexul serverului curent
sfârșit dacă
sfârșit pentru
adaugă în servers[bestServerIndex] sarcina task // apelează metoda de
adăugare din clasa server (addTask)
  
```

Acestea au fost clasele care rețin date și operează cu acestea. Următorul pachet îi corespunde părții de ieșire a aplicației (“View”). Pachetul conține view-ul pentru fereastra de setare (introducere a datelor de intrare) și un view pentru fereastra de simulare.

Ultimul pachet principal îi corespunde părții de intrare (“Controller”). Acest pachet conține două clase “Controller” pentru fereastra setării parametrilor de simulare și pentru managerierea simulării. Ultimul controller este una dintre cele mai importante clase, deoarece setează parametrii de simulare, generează sarcini ale căror atribute au valori aleatoare, pornește serverele și scrie în timp real evoluția simulării.

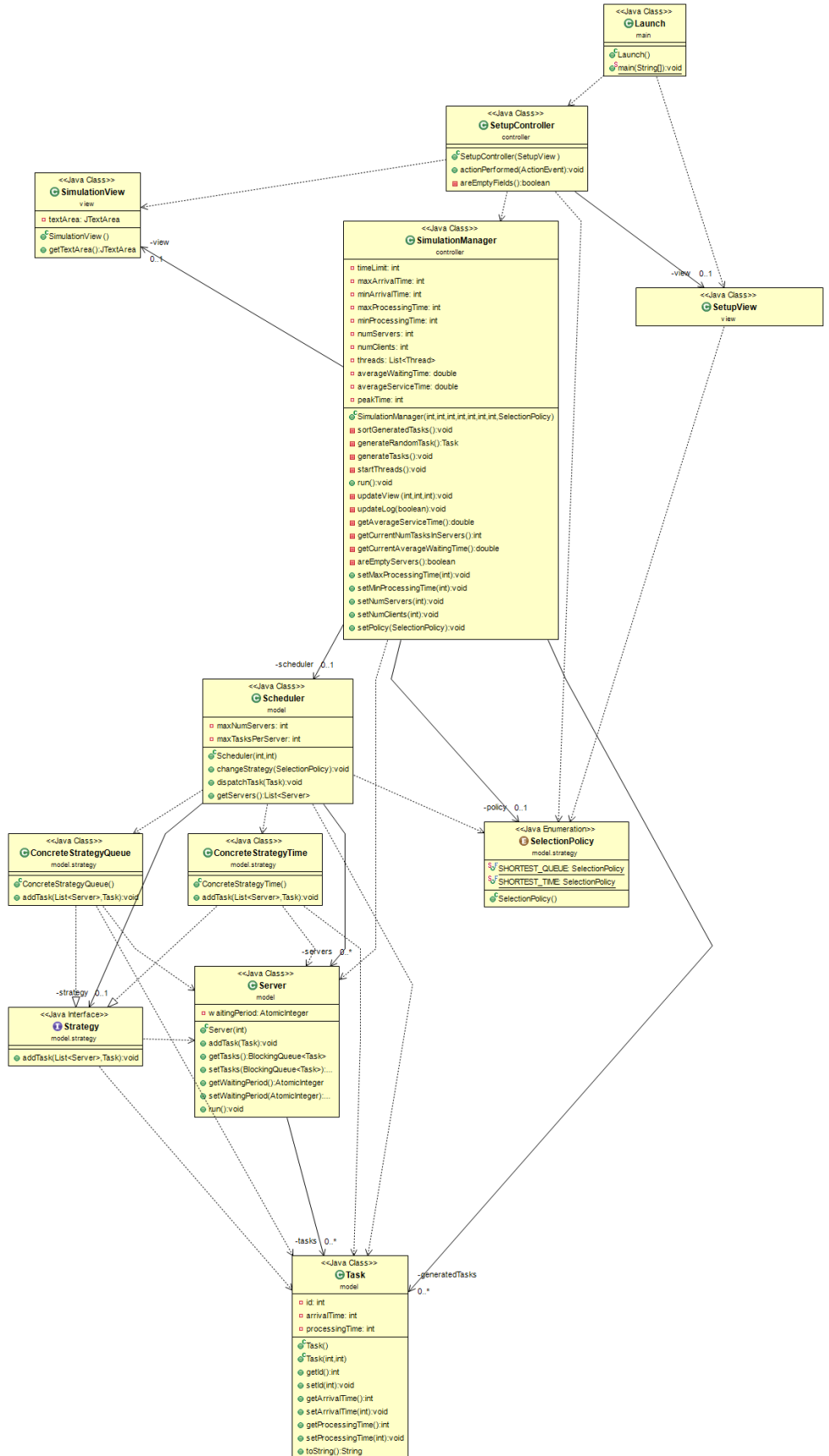
*Diagrame UML:*

- diagrama de pachete





- diagrama de clase



## 4. Implementarea temei

În acest capitol vor fi descrise toate clasele din aplicație împreună cu atributele și metodele cele mai importante.

### 4.1. Pachetul *model*

Clasa **Task** – reține datele unei sarcini de executat

#### *Attribute*

- id – numărul de ordine
- arrivalTime – timpul de sosire
- processingTime – timpul de procesare

Clasa **Server** – reține o listă de sarcini și operează asupra lor

#### *Attribute*

- tasks – lista de sarcini

#### *Metode*

- addTask (Task task) – adaugă o sarcină (parametrul task) în lista de servere și crește timpul de așteptare al serverului cu valoarea timpului de procesare al sarcinii adăugate
- run ( ) – metoda executată la pornirea unui nou thread asupra unui obiect server. Modul de funcționare este următorul: se execută totul într-o buclă infinită (în realitate, se va opri când expiră timpul de simulare)
  1. dacă lista de sarcini este goală, așteaptă 1 secundă (în realitate, se va aștepta până când există cel puțin o sarcină în listă) și se revine la pasul 1
  2. altfel, ia prima sarcină din listă
  3. cât timp timpul de procesare al sarcinii este mai mare decât 1, decrementează timpul de așteptare al serverului, decrementează timpul de procesare al sarcinii și așteaptă 1 secundă
  4. elimină primul element din listă, deoarece a fost procesat la pasul 3
  5. așteaptă 1 secundă – are rolul de a opri începerea imediată a următoarei sarcini din listă, dacă există
  6. se revine la pasul 1

Clasa **Scheduler** – programează adăugarea de sarcini în servere pe baza strategiei alese

#### *Attribute*

- maxNumServers – numărul maxim de servere
- maxTasksPerServer – numărul maxim de sarcini per server
- servers – lista de servere
- strategy – strategia aleasă

#### *Metode*

- changeStrategy (SelectionPolicy policy) – schimbă strategia de adăugare în funcție de o politică (policy)
- dispatchTask (Task task) – adaugă o sarcină într-unul din serverele din listă în funcție de strategia aleasă; în ea se apelează metoda *addTask* din clasa *ConcreteStrategyQueue* sau *ConcreteStrategyTime*, care implementează interfața *Strategy*

### 4.1.1. Subpachetul *strategy*

Conține elementele referitoare la strategia de selectare a serverelor.

#### Interfața **Strategy**

#### *Metode*

- addTask (List<Server> servers, Task task) – adaugă o sarcină (parametrul *task*) într-unul din serverele din parametrul *servers*

Enumerația **SelectionPolicy** – politica de alegere a strategiei (metoda *changeStrategy* din *Scheduler*)

- SHORTEST\_TIME

- SHORTEST\_QUEUE

Clasa **ConcreteStrategyQueue**

**Metode**

- addTask (List<Server> servers, Task task) – adaugă o sarcină (parametrul *task*) în serverul cu cele mai puține sarcini din parametrul *servers*. Algoritmul este descris în capitolul 3.

Clasa **ConcreteStrategyTime**

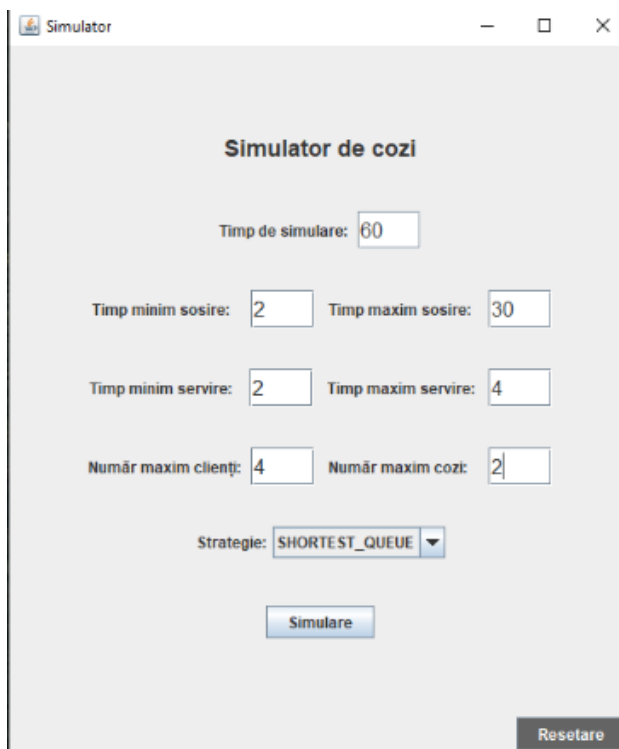
**Metode**

- addTask (List<Server> servers, Task task) – adaugă o sarcină (parametrul *task*) în serverul cu cel mai mic timp de așteptare din parametrul *servers*. Algoritmul este descris în capitolul 3.

#### 4.2. Pachetul *view*

Conține view-urile pentru fereastra de setări ale datelor de intrare și pentru fereastra de simulare.

Clasa **SetupView** – interfața grafică a ferestrei de setări a parametrilor pentru simulare; (imagine cu exemplu de utilizare)



Clasa **SimulationView** - interfața grafică a ferestrei de simulare; (imagine cu exemplu de utilizare)



#### 4.3. Pachetul *controller*

Conține controller-ele asociate fiecărui view din pachetul *view*.

Clasa **SetupController** – controller pentru fereastra de setări ale parametrilor simulării

##### *Metode*

- `areEmptyFields ()` – verifică dacă există câmpuri necompletate în interfață
- `actionPerformed ()` – listener pentru butonul de începere a simulării.
  - dacă există câmpuri goale (se apelează metoda *areEmptyFields*), se afișează un mesaj de atenționare pentru utilizator și nu se pornește simularea
  - altfel, se extrag valorile întregi din câmpurile text din interfață
    - dacă există date incorect introduse (capitolul 2, secțiunea 2.3, cazul de utilizare 1, scenariu secundar 2), se afișează un mesaj de atenționare pentru utilizator și nu se pornește simularea
    - altfel, se pornește simularea (se creează controller-ul pentru simulare cu parametrii introduși de utilizator)

Clasa **SimulationManager** – controller pentru simulare

##### *Atribute*

- toate datele de intrare menționate în secțiunea 2.1. din capitolul 2.
- `scheduler`
- `policy`
- `generatedTasks` – listă care va conține sarcinile generate aleator
- `threads` – listă cu threadurile fiecărui server din scheduler
- `averageWaitingTime` – timpul mediu de așteptare
- `averageServiceTime` – timpul mediu de procesare
- `peekTime` – timpul de vârf

##### *Metode*

- `generateRandomTask ()` – generează o sarcină cu timpii de sosire și de procesare aleși aleatori din intervalele introduse de utilizator în interfață
- `sortGeneratedTasks ()` – sortează crescător lista de sarcini generate aleator în funcție de timpul de sosire al fiecărei sarcini
- `generateTasks ()` – adaugă în lista *generatedTasks* sarcini create cu metoda *generateRandomTask*, după care sortează lista, apelând metoda *sortGeneratedTasks*, și setează numărul de ordine (ID) potrivit pentru fiecare sarcină
- `startThreads ()` – pornește threadurile din *threads*
- `updateView (int numWaitingClients, int time, int numSpaces)` – actualizează fereastra de simulare la timpul *time*, restul parametrilor sunt doar pentru cosmetică
- `updateLog (boolean isSimulationDone)`

- dacă *isSimulationDone* e *false*, se scrie în log conținutul din fereastra de simulare la momentul curent
- altfel, se scriu timpii medii de așteptare, servire și timpul de vârf
- *getAverageServiceTime* ( ) – calculează timpul mediu de procesare al serviciilor – suma tuturor timpilor de procesare împărțit la numărul total de sarcini
- *getCurrentAverageWaitingTime* ( ) – calculează timpul mediu de așteptare la un moment de timp – suma tuturor timpilor de așteptare împărțit la numărul total de servere
- *run* ( ) – simularea propiu-zisă
  - timpul curent e 0
  - simularea nu e gata – *isSimulationDone* = *false*
  - cât timp timpul curent e mai mic decât timpul de simulare
    - cât timp există sarcini în *generatedTasks*
      - adaugă o sarcină într-un server (*dispatchTask*) din scheduler
      - elimină sarcina respectivă din *generatedTasks*
    - actualizează fereastra de simulare – *updateView*
    - actualizează *log.txt* cu simularea la momentul curent – *updateLog*
    - dacă nu mai există sarcini de pus și serverele sunt goale, se iese din buclă
    - calculează statisticile curente (timpii medii de așteptare și timpul de vârf)
    - incrementează timpul curent
    - așteaptă 1 secundă
  - am terminat simularea - *isSimulationDone* = *true*
  - calculează timpul mediu de așteptare total – suma tuturor mediilor de așteptare împărțită la numărul de momente de timp (timpul simulării)
  - actualizează *log.txt* cu statisticile finale - *updateLog*
  - afișează un mesaj de terminare în interfață
  - închide fereastra de simulare

#### 4.4. Pachetul *main*

Clasa **Launch** – se execută aplicația

## 5. Testarea temei și rezultate

Simulatorul a fost testat prin adăugarea a trei seturi predefinite de parametri. Rezultatele celor trei simulări vor fi prezentate în acest capitol. Pentru toate cele trei simulări s-a folosit strategia SHORTEST\_QUEUE.

#### *Simularea 1*

- Timp maxim de simulare: 60 secunde
- Număr de clienți : 4
- Număr de cozi: 2
- Intervalul timpului de sosire: [2,30]
- Intervalul timpului de procesare: [2,4]

Fișierul *log.txt* pentru simularea 1 (sunt arătate numai părțile esențiale):  
Momentul 0

Clienți în așteptare: 4

(1,4,3)

(2,8,5)

(3,12,4)

(4,14,4)

Coadă 1: Închisa

Coadă 2: Închisa

...

Momentul 4

Clienți în așteptare: 3

(2,8,5)

(3,12,4)

(4,14,4)

Coadă 1: (1,4,3);

Coadă 2: Încchisa

Momentul 5

Clienți în așteptare: 3

(2,8,5)

(3,12,4)

(4,14,4)

Coadă 1: (1,4,2);

Coadă 2: Încchisa

...

Momentul 8

Clienți în așteptare: 2

(3,12,4)

(4,14,4)

Coadă 1: (2,8,5);

Coadă 2: Încchisa

Momentul 9

Clienți în așteptare: 2

(3,12,4)

(4,14,4)

Coadă 1: (2,8,4);

Coadă 2: Încchisa

...

Momentul 12

Clienți în așteptare: 1

(4,14,4)

Coadă 1: (2,8,1);

Coadă 2: (3,12,4);

Momentul 13

Clienți în așteptare: 1  
(4,14,4)

Coadă 1: Închisă  
Coadă 2: (3,12,3);  
...  
Momentul 18

Clienți în așteptare: 0

Coadă 1: Închisă  
Coadă 2: Închisă

Timp mediu de așteptare: 0.49  
Timp mediu de servire: 4.00  
Timpul de varf: 12

#### ***Simularea 2***

- Timp maxim de simulare: 60 secunde
- Număr de clienți : 50
- Număr de cozi: 5
- Intervalul timpului de sosire: [2,40]
- Intervalul timpului de procesare: [1,7]

#### ***Rezultate simulare:***

Timp mediu de așteptare: 8.70  
Timp mediu de servire: 4.24  
Timpul de varf: 39

#### ***Simularea 3***

- Timp maxim de simulare: 200 secunde
- Număr de clienți : 1000
- Număr de cozi: 20
- Intervalul timpului de sosire: [10,100]
- Intervalul timpului de procesare: [3,9]

#### ***Rezultate simulare:***

Timp mediu de așteptare: 169.73  
Timp mediu de servire: 7.02  
Timpul de varf: 109

Nu au mai existat clienți în așteptare, dar serverele nu au terminat de executat toate sarcinile înaintea terminării simulării.

Fișierele *log.txt* pentru fiecare simulare sunt incluse în directorul aplicației.

## 6. Concluzii și dezvoltări ulterioare

Realizând această temă, am învățat:

- lucrul cu threaduri, sincronizare, concurență
- cum să scriu în timp real rezultate într-o interfață
- lucrul cu fișiere: creare, actualizare, ștergere

În versiuni ulterioare ale aplicației, aș dori să arăt simularea în timp real într-o manieră mai grafică, să perfecționez lucrul cu threaduri, deoarece am avut dileme pe parcursul realizării temei, și să îmbunătățesc fereastra de setare a parametrilor cu mai multe mesaje pentru utilizator și restricții.

## 7. Webografie

1. Baeldung, “Guide to java.util.concurrent.BlockingQueue” - <https://www.baeldung.com/java-blocking-queue>;
2. HowToDoInJava, “A Guide to AtomicInteger in Java” - <https://howtodoinjava.com/java/multi-threading/atomicinteger-example/>;
3. TutorialsPoint, “Java - FileWriter Class” - [https://www.tutorialspoint.com/java/java\\_filewriter\\_class.htm](https://www.tutorialspoint.com/java/java_filewriter_class.htm);
4. Oracle, “JTextArea” - <https://docs.oracle.com/javase/7/docs/api/javax/swing/JTextArea.html>;