DOCUMENTATIE

TEMA 2

NUME STUDENT: Tarța Antonia-Maria
GRUPA: 30229

CUPRINS

1. Obiectivul temei

Obiectivul principal al acestei teme este proiectarea și implementarea unei aplicații care are ca scop analiza sistemelor de procesare a unor cozi la care sosesc clienți la diferite momente de timp.

Obiectivele secundare ale acestei teme sunt:

- Analizarea problemei și identificarea cerințelor
- Proiectarea aplicației în care se va face simularea
- Implementarea aplicației
- Testarea aplicației prin simulare

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

Analiza problemei

Cerințe funcționale:

- Aplicația de simulare trebuie să permită utilizatorului să configureze simularea;
- Aplicația de simulare trebuie să permită utilizatorului să pornească simularea;
- Aplicația de simulare trebuie să afișeze evoluția cozilor în timp real.

Cerințe non-funcționale:

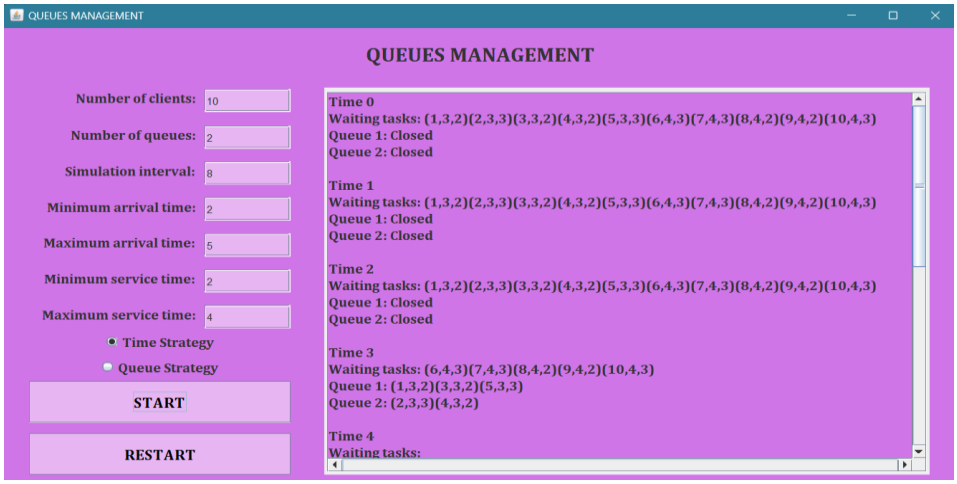
- Aplicația de simulare trebuie să fie intuitivă și ușor de folosit de către utilizator.
- Aplicația ar trebui să fie precisă și să efectueze corect simularea.
- Aplicația trebuie să fie eficientă din punct de vedere al timpului de execuție și al memoriei utilizate.

Modelarea problemei

Utilizatorul va trebui să introducă în spațiile rezervate 7 valori pentru a furniza numărul de clienți, numărul de cozi, timpul de simulare, timpul minim și maxim la care un client poate să ajungă, timpul minim și maxim de procesare a nevoilor unui client, iar ulterior se poate selecta strategia dorită. Acesta va putea să pornească simularea și să resete interfața. Simularea va fi afișată în timp real.

Scenarii și cazuri de utilizare

Cazurile de utilizare și scenariile sunt în strânsă legătură cu acțiunile utilizatorului. Interfața îi permite acestuia să îmbine utilul cu plăcutul deoarece folosește o aplicație cu care este ușor de interacționat pentru a obține simularea dorită. Pentru o mai bună înțelegere a scenariului o să atașez o imagine cu interfața:



Actor principal: utilizatorul.

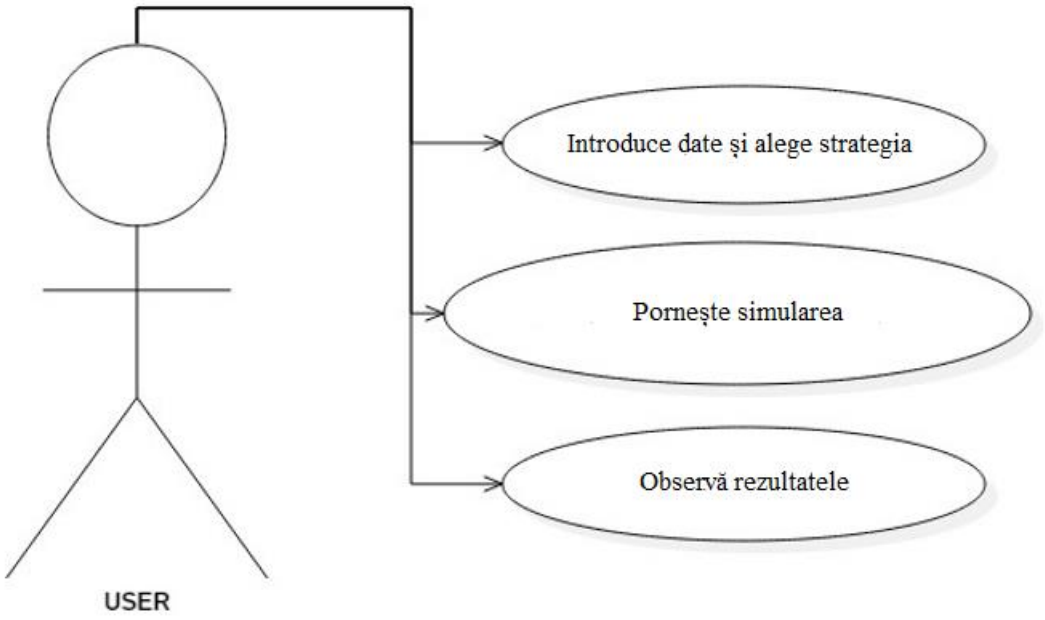
Scenariul principal de success:

1. Utilizatorul va trebui să introducă în spațiile rezervate 7 valori pentru a furniza numărul de clienți, numărul de cozi, timpul de simulare, timpul minim și maxim la care un client poate să ajungă, timpul minim și maxim de procesare a nevoilor unui client.
2. Utilizatorul alege strategia dorită(timp de așteptare minim sau coada mai scurtă)
3. Utilizatorul apasă butonul START.
4. Simularea începe și rezultatele sunt afișate în timp real.

Secvență alternativă:

- Utilizatorul introduce date invalide.

- Opțional se resetează căsuțele de text, iar scenariul revine la pasul 1.



3. Proiectare

Proiectarea POO

Aplicația conține :
- 7 clase :
  - Task
  - Server
  - Scheduler
  - StrategyTime
  - StrategyQueue
  - SimulationManager
  - View
  - Controller

-o interfata :
  - Strategy

-o enumerație:
  - SelectionPolicy

-pachete :
  - BusinessLogic(conține logica din spatele simulării- clasele : Scheduler, StrategyTime, StrategyQueue, SimulationManager)
  - DataModels (conține clasa unde sunt modelate datele – clasele : Task, Server)
  - GUI (conține clasele care se ocupă de interfața grafică – View, Controller)

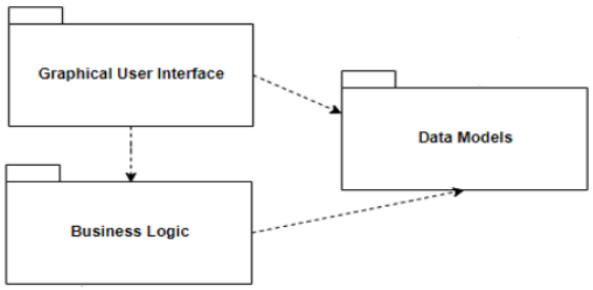
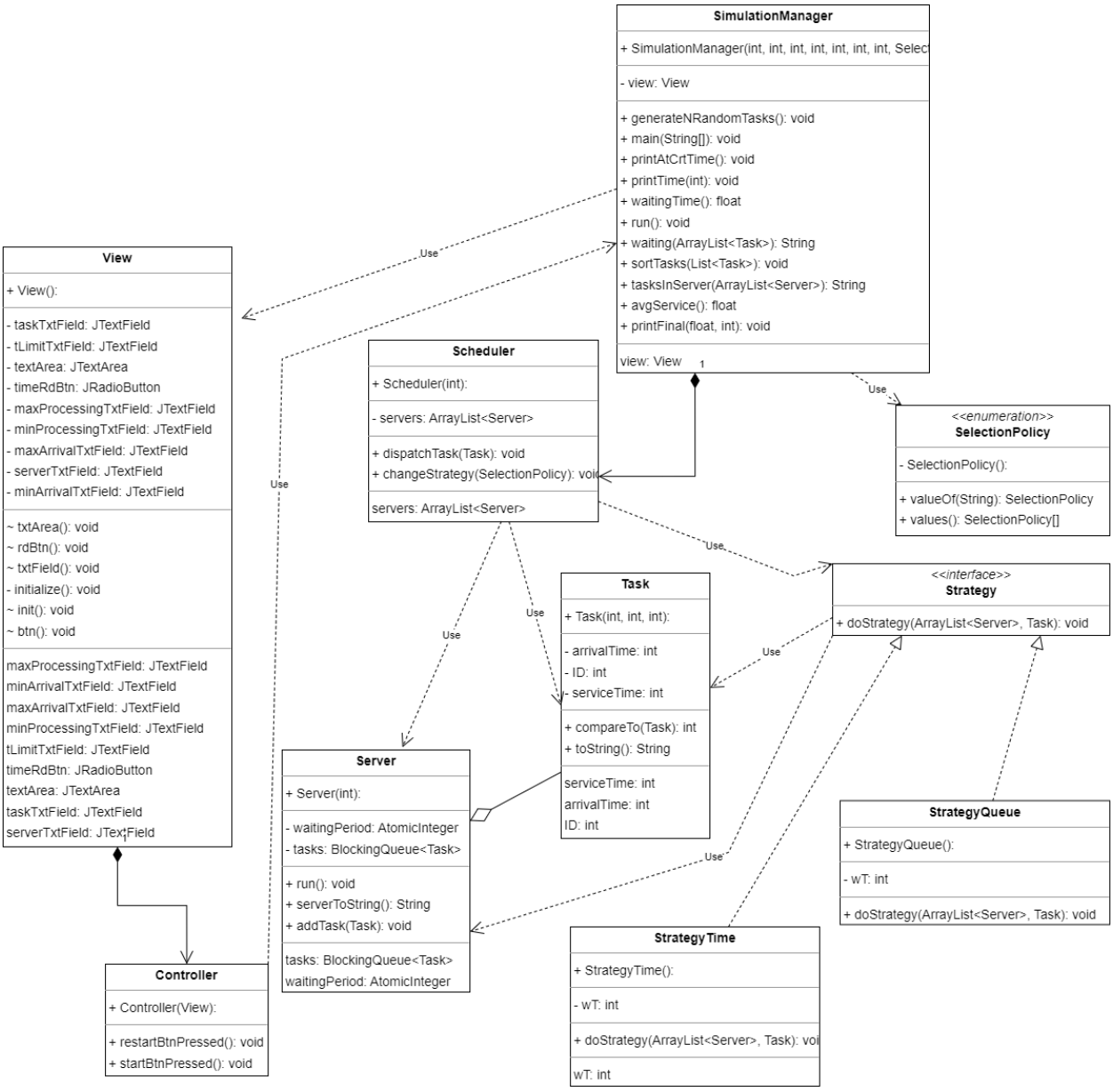


Diagrama UML



Structuri de date

În acest proiect am folosit structurile: BlockingQueue și AtomicInteger. BlockingQueue este folosit pentru a adauga și a șterge concurent clienți, acesta fiind capabil sa blocheze firele de lucru care încearcă să insereze sau să ia elemente din coadă. Clasa AtomicInteger oferă o variabilă int care poate fi citită și scrisă atomic și care conține, de asemenea, operații atomice avansate.

4. Implementare

Clasa Task este definită în pachetul "Model", implementează interfața "Comparable".

Câmpuri:

- ID
- arrivalTime
- serviceTime

Constructori:

- Task(ID, arrivalTime, serviceTime)

Metode accesoare:

- getID()
- getArrivalTime()
- getServiceTime()

Metode mutatoare:

- setID()
- getServiceTime()

Metode :

- toString() : returnează obiectul sub forma unui șir de caractere
- compareTo(Task) : compară două obiecte "Task" pe baza timpului de sosire.

```
@Override
public int compareTo(Task t) {
    if(this.getArrivalTime()<t.getArrivalTime()){
        return -1;
    }
    if(this.getArrivalTime()>t.getArrivalTime()){
        return 1;
    }
    return 0;
}
```

Clasa Server este definită în pachetul "Model", implementează interfața Runnable.

Câmpuri:

- tasks
- waitingPeriod
- id
- waitingT: timpul total de asteptare.

Constructori:

- Server(id)

Metode accesoare:

- getTasks()
- getWaitingPeriod()
- getWaitingT()

Metode :

- serverToString() : returnează obiectul sub forma unui șir de caractere
- addTask(Task) : utilizată pentru a adăuga sarcini la coadă
- run() : această metodă este apelată atunci când firul de execuție asociat instanței curente este pornit și începe să proceseze sarcinile din coadă.

```
public void run(){
    while(true){
        if(tasks.size()!=0){
            waitingT+=waitingPeriod.get();
        }
        Task t=tasks.peek();
        if(t!=null){
            while(t.getServiceTime()>0){
                try {
                    Thread.sleep(1000);
                }
                catch(InterruptedException e) {
                    throw new RuntimeException(e);
                }
                t.setServiceTime(t.getServiceTime()-1);
                waitingPeriod.decrementAndGet();
            }
            tasks.remove(t);
        }
    }
}
```

Enumerația SelectionPolicy

```
public enum SelectionPolicy {
    SHORTEST_QUEUE, SHORTEST_TIME
}
```

Interfața Strategy

```
public interface Strategy {
    public void doStrategy(ArrayList<Server> servers, Task t);
}
```

Clasa StrategyTime este definită în pachetul "BusinessLogic", implementează interfața "Strategy" și definește strategia de alocare a task-urilor la servere. Această strategie se bazează pe timpul de așteptare în coadă al fiecărui server, alegându-se serverul cu cel mai mic timp de așteptare.

Metode :

- doStrategy(servers,Task)

Clasa StrategyQueue este definită în pachetul "BusinessLogic", implementează interfața "Strategy" și definește strategia de alocare a task-urilor la servere. Această strategie se bazează pe lungimea fiecărui server, alegându-se serverul cel mai puțin populat.

Metode :

- doStrategy(servers,Task)

Clasa Scheduler este definită în pachetul "BusinessLogic".

Câmpuri:

- servers
- strategy

Constructori:

- Scheduler (maxNoServers)

Metode accesoare:

- getServers()
- getArrivalTime()
- getServiceTime()

Metode :

- changeStrategy(SelectionPolicy)
- dispatchTask(Task)

```
public void changeStrategy(SelectionPolicy policy){
    if(policy==SelectionPolicy.SHORTEST_QUEUE){
        strategy=new StrategyQueue();
    }
    else if(policy==SelectionPolicy.SHORTEST_TIME)
    {
        strategy=new StrategyTime();
    }
}
```

Clasa SimulationManager este definită în pachetul "BusinessLogic", implementează interfața Runnable. Aceasta este clasa în care am implementat metode precum cea care generează clienții random si cea care îi sortează în funcție de timpul lor de sosire. Totodată există un constructor SimulatonManager() care ia din interfață datele necesare simulării și le procesează. Tot această clasă conține si metodele care calculează ora de vârf, timpul mediu de așteptare și procesare.

Clasa View este definită în pachetul "GUI", aici am creat design-ul interfeței.

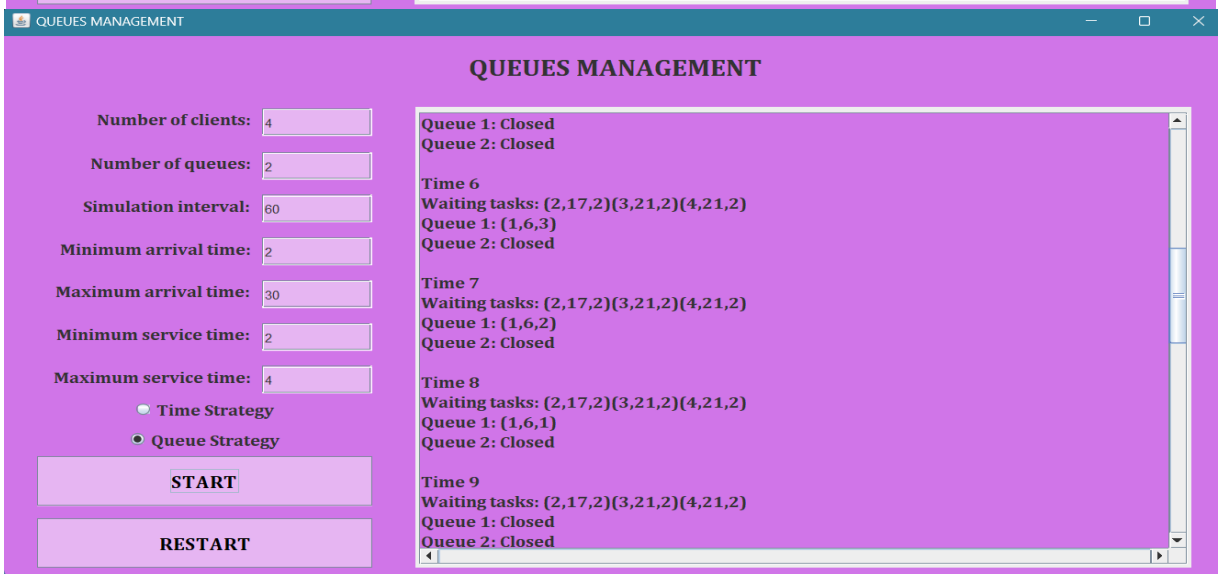
Clasa Controller este definită în pachetul "GUI". Metodele din această clasă primesc semnal din view când utiliztorul interacționează cu obiectele și decid cum se schimbă partea pe care acesta o vede.

5. Rezultate

Test 1	Test 2	Test 3
N = 4 Q = 2 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 30]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [2, 4]$	N = 50 Q = 5 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [2, 40]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [1, 7]$	N = 1000 Q = 20 $t_{simulation}^{MAX}$ = 200 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}] = [10, 100]$ $[t_{service}^{MIN}, t_{service}^{MAX}] = [3, 9]$

Rezultatele acestor teste sunt scrise în fișierele atașate proiectului. Pentru simularea acestora în timp real se poate rula aplicația, iar rezultatele vor fi afișate în timp real. Aceste rezultate vor fi diferite față de cele din fișiere deoarece la fiecare simulare clienții sunt generați random.

Exemple pentru Test 1:



6. Concluzii

Lucrând la acest proiect am dobândit cunoștințe despre thread-uri. Este prima dată când folosesc variabile AtomicInteger și BlockingQueue. Am învățat atât să lucrez cu acestea, dar mai important este că am înțeles de ce este nevoie să le folosesc în contextul acestei teme. Am înțeles conceptul de operație atomică. Programul ar putea fi îmbunătățit prin implementarea unor situații neprevăzute care ar putea duce la creșterea timpului de așteptare sau de procesare. Am realizat că doar testând o aplicație poți să descoperi eventuale detalii care fac aplicația să nu ducă la rezultatul așteptat.

Posibile dezvoltari ulterioare:

- Extinderea implementării pentru a lua în calcul situații neprevăzute care ar putea duce la creșterea timpului de așteptare sau de procesare;
- Îmbunătățirea interfeței grafice.

7. Bibliografie

1) Generating Random Numbers in a Range - **https://www.baeldung.com/java-generating-random-numbers-in-range**

2) Write to a File - https://www.w3schools.com/java/java_files_create.asp

3) https://dsrl.eu/courses/pt/