MINISTERUL EDUCAȚIEI NAȚIONALE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

SIMULATOR DE COZI

Alin Matean

Facultatea de Automatica si Calculatoare

Grupa 30228

Cuprins

1.Obiectivul temei.....3
2.Analiza problemei, modelare, scenarii, cazuri de utilizare.....3
3.Proiectare.....5
4.Implementare.....8
5.Rezultate.....12
6.Concluzii.....13
7.Bibliografie.....14

ASD

Obiectivul temei

Obiectivul acestei teme este proiectarea și implementarea unei aplicații de simulare ce analizează sistemele bazate pe cozi(FIFO: first in first out) pentru determinarea și minimizarea timpului de așteptare. Obiectivul unei cozi este de a furniza un loc unui client astfel încât acesta să aștepte cât mai puțin pentru procesarea serviciului său.

Analiza problemei, modelare, scenarii, cazuri de utilizare

Analiza problemei

Problema care se impune este cea de a distribui și de a obține un timp cât mai scurt pentru fiecare client care așteaptă la coadă, în funcție de strategia aleasă(cel mai scurt timp/cea mai scurtă coadă). În implementarea, am considerat coada ca fiind un server și clientul ca fiind un task care are un anumit timp de procesare.

Scenarii

Scenariul de utilizare a aplicației presupune interacțiunea cu interfața grafică creată. Astfel utilizatorul are câteva câmpuri unde poate să introducă anumite date care vor genera testele random și vor porni aplicația. Datele pe care utilizatorul le introduce sunt: time limit, number of servers(numărul de cozi), number of tasks(numărul de clienți), min arrival time, max arrival time, min processing time, max processing time, poate sa aleagă una din cele două strategii(queue / time), output file(unde se vor stoca toate datele simulării) și mai are butonul de start care pornește efectiv simularea.

Scenariu de utilizare:

- se pornește aplicația
- se adaugă date corecte, valide in câmpurile rezervate
- se selectează strategia de distribuire a task-urilor către servere
- se adaugă un nume de fișier unde se vor stoca log-urile, toate datele despre simulare
- se apasă butonul start
- se va afișa la fiecare timp starea curentă a simulatorului
- la final, pentru a vedea toate datele generate deschidem fișierul in care am ales să se stocheze
- dacă se va dori o nouă simulare, se vor introduce noi date, altfel se va inchide aplicația de pe butonul corespunzator
- în cazul în care utlizatorul nu introduc date valide(de ex.: în loc să introducă numere introduce alte caractere, se va genera o excepție de la parsare)

Cazuri de utilizare – diagrama Use-case

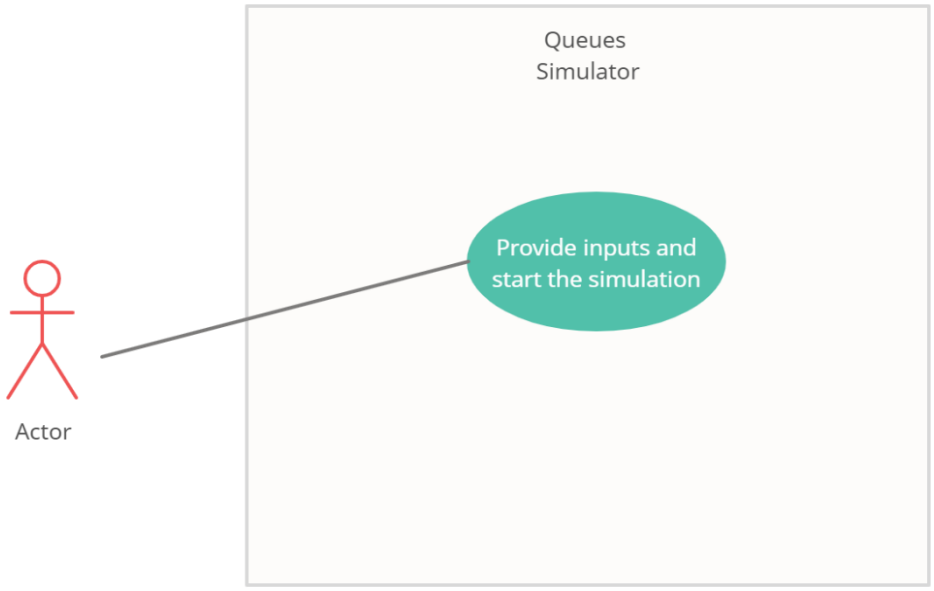

Figura 1. Diagrama use-case

Utilizatorul va efectua practice o singura operație și anume va porni simularea după ce introduce datele de intrare și va crea un fișier unde se vor stoca datele simulării.

Proiectare

Decizii de proiectare

Un fir de execuție (thread) este cea mai mică secvență de instrucțiuni programate care poate fi gestionată independent de un scheduler (planificator), fiind parte a sistemului de operare. Implementarea thread-urilor și proceselor diferă de la un sistem de proiectare la altul, dar în cele mai multe cazuri un thread este o

component a unui process. În Java thread-urile sunt obiecte ca oricare alte obiecte Java, acestea sunt instanțe ale clasei java.lang.Thread sau instanțe ale subclaselor acestei clase. Mai multe fire de execuție pot exista în cadrul unui singur process, executând simultan și împărțind resurse, cum ar fi memoria. În special, firele unui process impart codul executabil și valorile variabilelor în orice moment.

În cadrul pachetului "src" unde se află tot codul necesar funcționării și testării aplcației se găsește directorul "main". În directorul main găsim folder-ul "java" care conține pachetul prinipal și anume "tuc.tp.tema2", în care se află de fapt tot conținutul pentru funcționarea corectă a calculatorului. Acest pachet conține la rândul lui alte 3 pachete și o clasă separate, care pornește aplicția. Celelalte pachete sunt:

- dataModels: care conține clasele Task, Server și Scheduler, modelele aplicației
- logic: unde este implementată logica aplicției prin clasele StrategyTime, StrategyQueue care ambele implementează interfața Strategy
- userInterface: conține clasele necesare creării interfeței grafice cu utilizatorul, View și Controller

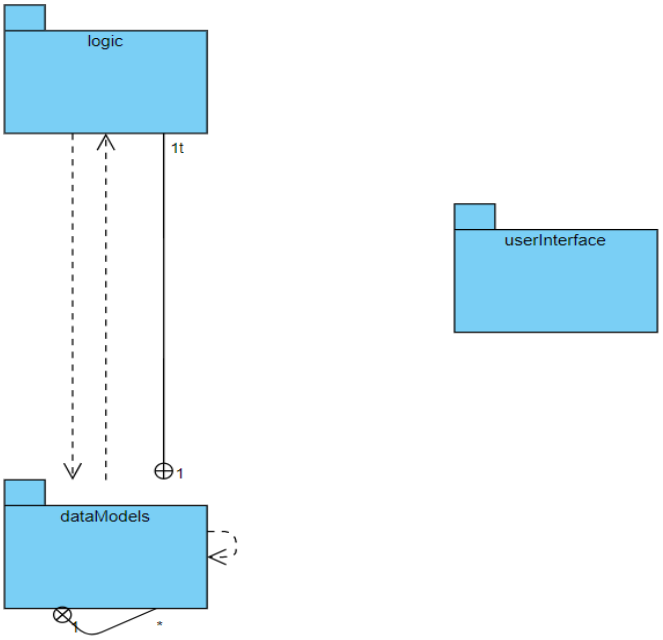

Figura 2. Diagrama de pachete [2]

Diagrama UML de clase

Clasele proiectate:

În pachetul tuc.tp.tema2 se găsesc:

- Scheduler care conține și StrategyPolicy (enum) – pachetul dataModels
- Server – pachetul dataModels
- Task – pachetul dataModels
- Strategy (interaface) – pachetul logic
- StrategyQueue și StrategyTime – pachetul logic
- Controller – pachetul userInterface

- View – pachetul userInterface
- SimulationManager - nu e inclusă în alt pachet separate

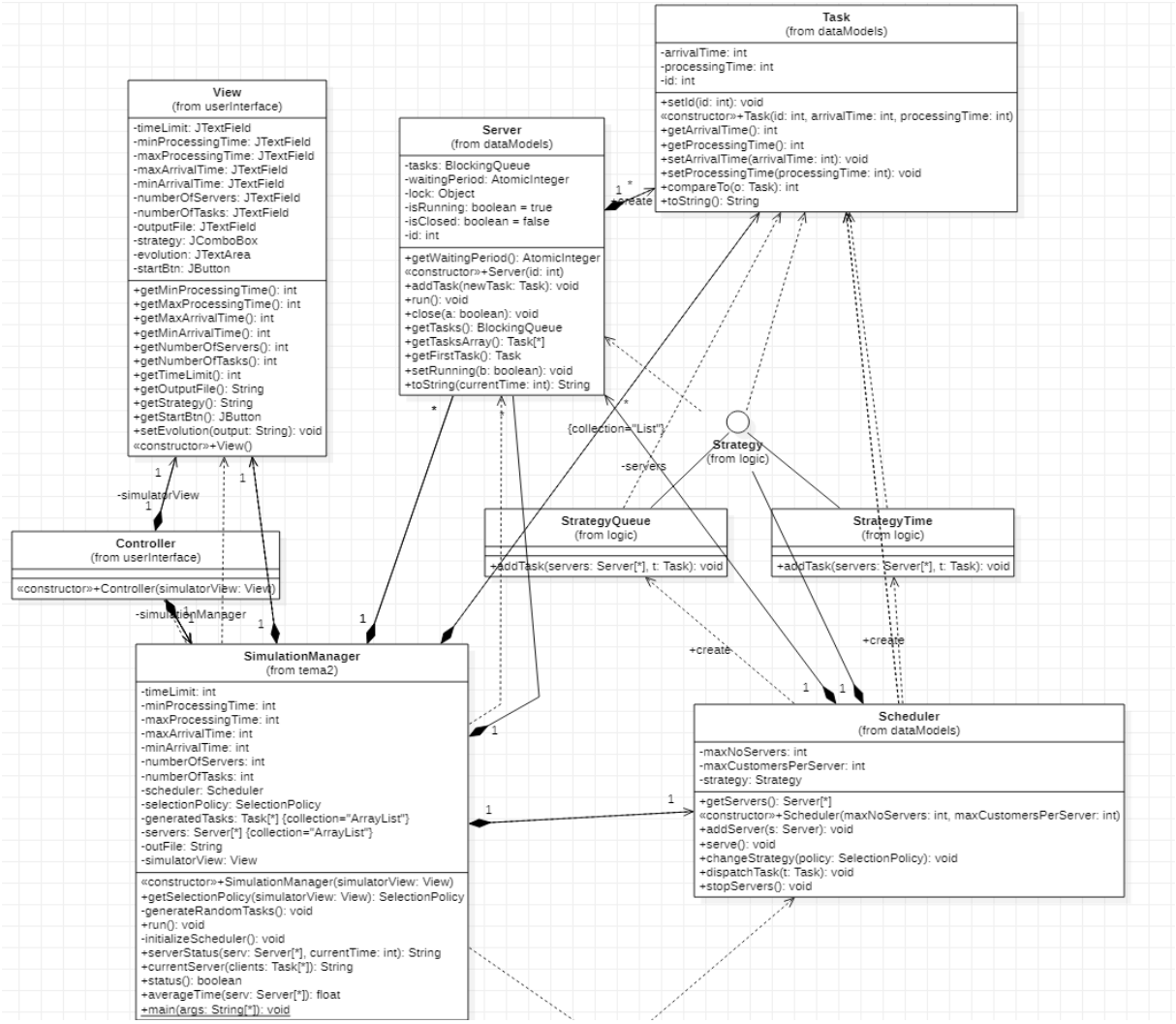

Figura 3. Diagrama UML de clase

Interfața grafică cu utilizatorul:

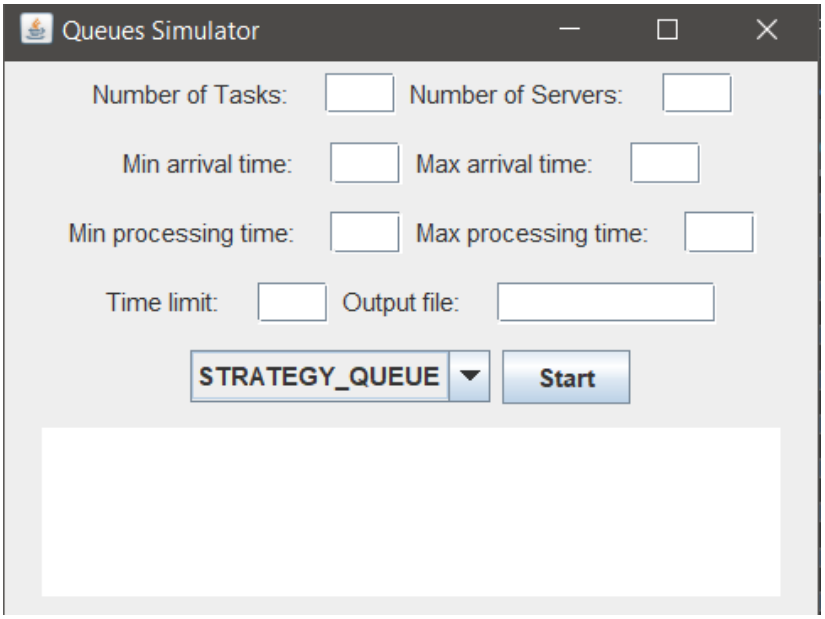

Figura 4. Interfața grafică cu utilizatorul

Utilizatorul va introduce datele cu care se vor genera teste random și va porni aplicația de pe butonul "Start". Conținutul cozilor și situația în timp real a simulării va aparea în partea de jos, iar rezultatul complet, log-urile vor apărea in "output file" dat de utilizator.[1]

Implementare

Clasele

Pachetul dataModels:

- Task
  - Conține 3 variabile instanță și anume id, arrivalTime și processingTime care reprezintă datele despre fiecare task sau client
  - Avem metode de get și set pentru fiecare, un constructor prin care instanțiem obiecte
  - Clasa implementeaza interfața Comparable pentru a putea ordona clienții după arrivalTime
  - Am suprascris și metoda toString pentru a afișa task-urile
- Server
  - Conține un tablou tasks de tip BlockingQueue, o vriabilă Atomic Integer waitingPeriod

- Conține niște metode care să ne returneze primul client din coadă sau primul task, metoda toString care afișează coada
- Metoda addTask este cea care adugă un nou client la coadă, tot aici se adaugă și la waitingPeriod timpul de procesare al task-ului curent.
- Clasa implemetează Runnable și evident suprascrie metoda run() invocată automat de thread. Aici verificăm dacă există clienți la coadă și daca timpul lor de procesare e mai mare de 0, atunci punem thread-ul pe sleep pentru a putea fi procesat și pentru a nu se continua cu un alt client instant, nu ar fi posibil.[3]

➢ Scheduler

- Această clasă dupa cum ii spune și numele este cea care planifică și ditribuie clienții către cozi
- Aici se găsește un enum de tip SelectioPolicy prin care vom selecta modul de distrbuire al task-urilor catre servere

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

- când inițializăm un obiect de tipul Scheduler trimitem ca și argumente numărul maxim de servere și numărul maxim de clienți la fiecare coadă
- metoda dispatchTask cheamă strategia de adăugare in coadă (cea mai scurtă coadă sau cel mai scurt timp până la procesare)
- metoda serve() pornește câte un thread pentru fiecare server

Pachetul logic:

➢ interfața Strategy

- conține doar metoda addTask care va fi implementată de cele doua strategii ulterior

➢ StrategyQueue

- Implementează interfața Strategy și astfel conține metoda addTask. În această metoda atribuim un task unui server și alegem server-ul cu size-ul cel mai mic, după ce le parcurgem pe toate.

➢ StrategyTime

- Implementează interfața Strategy și astfel conține metoda addTask. În această metoda atribuim un task unui server și alegem server-ul astfel încât un presupus client ar avea de așteptat cel mai puțin până ar ajunge să fie procesat, după ce le parcurgem pe toate.

Pachetul userInterface:

➢ Controller

- Conține două variabile instanță de tipul View și de tipul SimulationManager
- În constructor practic se ia cu ajutorul unui getter butonul de Start din VIew și i se adaugă funcționalitatea și anume de a porni aplicația cu thread-uri.

➢ View

- Este clasa care realizează interfața grafică din punct de vedere vizual.

- Conține 8 variabile de tipul JTextField, o variabilă JComboBox, una JTextArea și un buton JButton. În câmputrile number of tasks, number of servers, min arrival time, maz arrival time, min processing time, max processing time, time limit utilizatorul introduce date pe baza cărora se vor genera teste aleatoare, în câmpul output file se va scrie rezultatul simulării, din combo box se alege strategia ditribuirii clienților către cozi și de pe butonul cu "Start" se va porni simularea. Starea curentă a cozii va aparea in text area de mai jos la fiecare timp.
- In constructorul clasei se genereaza interfața
- Avem metode de get pentru a obține datele introduse
- Conține metoda de setEvolution() cu ajutorul căreia se va scrie textul la fiecare moment.

➢ SimulationManager

- Aceasta este practic clasa care realizează simularea
- Tot aici se găsește și metoda main care pornește aplicația
- În constructorul clasei care primește View ca și parametru se inițializează câmpurile necesare generării testelor random
- Metoda getSelectionPolicy e necesară pentru a obține câmpul selectat de utilizator privind strtegia aleasă ca mai apoi cu ajutorul metodei changeStrategy din Scheduler să setăm metoda de adăugare la coadă
- initializeScheduler realizeză crearea cozilor și adăugarea acestora in scheduler
- metoda generateRandomTasks este cea care genereaza testele aleatoare referitoare la clienți, la timpii de sosire și de procesare
- metodele currentServer și serverStatus sunt metode care returnează String, necesare afișării stării curente a cozilor și a scheduler-ului
- metoda status ne indică dacă mai sunt clienți de procesat sau dacă mai există clienți in orice coadă
- clasa SimulationManager implementează interfața Runnable, deci suprascrie metoda run() necesară simulării, necesară thread-ului

```
@Override
public void run() {
    FileWriter fileWriter = null;
    StringBuilder result;
    try { fileWriter = new
FileWriter(System.getProperty("user.dir") + "\\" +
this.outFile);
    }
    catch (IOException e) {
        e.printStackTrace();
    }
    try{
        fileWriter.write("Queues Simulator\n");
    }
    catch (IOException e){
        e.printStackTrace();
    }
```

```
        int currentTime = 0;

        while (currentTime <= timeLimit){
            boolean ok = true;
            result = new StringBuilder();
            result.append("Time:
").append(currentTime).append("\n");
            System.out.println("Time: " + currentTime);

            while(generatedTasks.size() > 0 && ok){
                ok = false;
                if(generatedTasks.get(0).getArrivalTime() <=
currentTime){
                    Task t = generatedTasks.get(0);
                    scheduler.dispatchTask(t);
                    generatedTasks.remove(0);
                    ok = true;
                }
                scheduler.serve();
            }
            try{
                Thread.sleep(1000);
            }
            catch (InterruptedException e){
                e.printStackTrace();
            }
            result.append("Waiting:
").append(currentServer(generatedTasks)).append("\n");
            result.append(serverStatus(scheduler.getServers(),
currentTime)).append(("\n"));
            System.out.println("Waiting: " +
currentServer(generatedTasks));
            System.out.println(serverStatus(scheduler.getServers(),
currentTime));
            simulatorView.setEvolution(result.toString());
            try{
                fileWriter.write("\n" + "Time: " + currentTime +
"\n" + "Waiting: " + currentServer(generatedTasks) + "\n" +
serverStatus(scheduler.getServers(), currentTime));
            }
            catch (IOException e){
                e.printStackTrace();
            }
            currentTime++;
            if(!status())
                break;
        }
        scheduler.stopServers();
        System.out.println("Average waiting time: " +
averageTime(scheduler.getServers()));

        try{
            fileWriter.write("\n" + "Average waiting time: " +
averageTime(scheduler.getServers()));
        }
```

```
        catch (IOException e){
            e.printStackTrace();
        }

        try {
            fileWriter.close();
        }
        catch (IOException e){
            e.printStackTrace();
        }
    }
```

- Tot aici se scrie și în fișierul output.txt
- În metodă se verifică daca mai avem task-uri, dacă da se ia primul task(acestea sunt ordonate crescător după timpul de sosire), se verifică timpul lui de sosire și se distribuie către coada potrivită. Prin metoda serve () din clasa Scheduler se "servește" clientul.

Rezultate

Pentru testare am utilizat exemplele din cerință. Rezultatele de la orice test se vor salva într-un fișier trimis în câmpul "output file". În fișier se vor afișa clienții care așteaptă și conținutul cozilor în toți timpii din timpul simulării, iar în interfața grafică în câmpul rezervat se va afișa conținutul cozilor si task-urile ce urmează să fie procesate doar în timpul curent, acest câmp actualizându-se constant.

Exemplu de output:

```
Queues Simulator

Time: 0
Waiting: (1 3 2)(2 4 3)(3 4 3)(4 5 3)(5 5 3)
Queue 1: closed
Queue 2: closed

Time: 1
Waiting: (1 3 2)(2 4 3)(3 4 3)(4 5 3)(5 5 3)
Queue 1: closed
Queue 2: closed

Time: 2
Waiting: (1 3 2)(2 4 3)(3 4 3)(4 5 3)(5 5 3)
Queue 1: closed
Queue 2: closed

Time: 3
Waiting: (2 4 3)(3 4 3)(4 5 3)(5 5 3)
Queue 1: (1 3 2);
Queue 2: closed

Time: 4
Waiting: (4 5 3)(5 5 3)
Queue 1: (1 3 1);(3 4 3);
```

```
Queue 2: (2 4 3);

Time: 5
Waiting:
Queue 1: (3 4 3);(5 5 3);
Queue 2: (2 4 2);(4 5 3);

Time: 6
Waiting:
Queue 1: (3 4 2);(5 5 3);
Queue 2: (2 4 1);(4 5 3);

Time: 7
Waiting:
Queue 1: (3 4 1);(5 5 3);
Queue 2: (4 5 3);

Time: 8
Waiting:
Queue 1: (5 5 3);
Queue 2: (4 5 2);

Time: 9
Waiting:
Queue 1: (5 5 2);
Queue 2: (4 5 1);

Time: 10
Waiting:
Queue 1: (5 5 1);
Queue 2: closed

Time: 11
Waiting:
Queue 1: closed
Queue 2: closed

Average waiting time: 2.8
```

Concluzii

Dezvoltări ulterioare:

- S-ar putea dezvolta interfața grafică cu utilizatorul
- Implementarea unui mod de a pune pe pauză simularea la un moment dat
- Calcularea și afișarea momentului în care cei mai mulți clienți se află la coadă
- Afișarea numărului de clienți care așteaptă

Concluzii:

Cu ajutorul acestui proiect am învățat ce este un thread și cum se folosește. Am învățat și cum se scrie într-un fișier. Această problemă, a minimizării timpului de stat la coadă la magazin, ar putea fi implementat și în viața reală, însă adaptat puțin deoarece nu putem stabili timpul pe care îl vom petrece la partea de procesare.

Bibliografie

[1] https://users.utcluj.ro/~igiosan/teaching_poo.html

[2] https://online.visual-paradigm.com/

[3] https://www.tutorialspoint.com/java/java_thread_synchronization.htm