

gestionarea cozilor

Tema 2



Popesc ariadna-Ioana

Grupa 30228

Cuprins

Obiectivul temei2

Analiza problemei2

Proiectare4

Implementare5

Rezultate………...…....…………………….…………………………….…13

Concluzii..............................................................................................13

Bibliografie14

1. **Obiectivul temei**

*Obiectiv principal*

Scopul principal al temei "Simularea sistemelor bazate pe cozi" constă în crearea și dezvoltarea unei aplicații software care să simuleze în detaliu un sistem de gestionare a sarcinilor atribuite unor servere, pentru a fi procesate, urmată de prezentarea unor statistici relevante care să descrie eficiența și modul de funcționare a acestuia.

*Obiective secundare*

* Analizarea problemei și identificarea cerințelor
* Proiectarea aplicației de simulare
* Implementarea aplicației de simulare
* Testarea aplicației de simulare

1. **Analiza problemei**

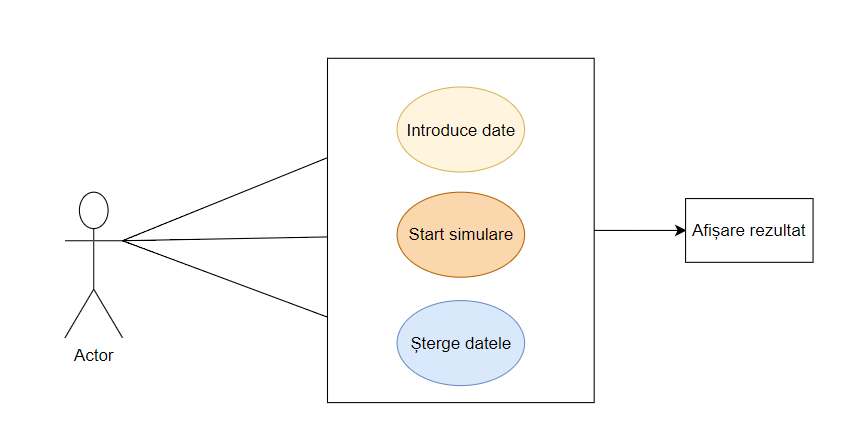
Această temă are ca obiectiv propunerea, proiectarea și implementarea unui sistem de procesare a cozilor gestionate prin rezolvarea sarcinilor atribuite unor servere.

Cerințele non-funcționale sunt:

* Simularea aplicației ar trebui să fie intuitivă și ușor de folosit de către utilizator
* Interfața ar trebui să fie cât mai ușor de parcurs de către utilizator

Cerințele funcționale sunt:

* Simularea aplicației ar trebui să permită utilizatorilor să seteze datele simulării
* Simularea aplicației ar trebui să permită utilizatorilor să pornească aplicația
* Simularea aplicației ar trebui să afișeze evoluția în timp a cozilor
* Aplicația ar trebui să creeze un fișier cu toate datele introduse și rezultatul așteptat
* Caz de utilizare: simulare de configurare
* Actor principal: utilizator
* Scenariul principal de succes:
* Utilizatorul introduce valorile pentru: numărul de clienți, numărul de cozi, timpul minim și maxim de sosire și timpul minim, maxim de serviciu și intervalul de simulare
* Utilizatorul face clic pe butonul de start, adică cel de a datelor de intrare și începe să afișeze simularea dorită
* Secvență alternativă:
* Valori nevalide pentru parametrii de configurare
* Utilizatorul introduce valori nevalide pentru parametrii de configurare ai aplicației
* Aplicația afișează un mesaj de eroare și solicită utilizatorului să introducă valori valide-Scenariul revine la pasul 1

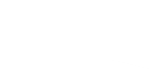
 **Diagrama Use-Case pentru aplicația “Procesarea cozilor”**

1. **Proiectare**

* Structuri de date folosite:
* BlockingQueue<Task> tasks 🡪 pentru a reține task-urile
* List<Server> servers 🡪 pentru a reține fiecare server
* AtomicInteger waitingPeriod 🡪 pentru a reține timpul de așteptare la fiecare coadă în parte.
* Arhitectură conceptuală:

O imagine care conține diagramă

Descriere generată automat



* Diagrama de pachete

O imagine care conține diagramă

Descriere generată automat

* Diagrama UML de clase

O imagine care conține text, de interior

Descriere generată automat

1. **Implementare**

Proiectul este împărțit în 3 pachete de clase. Pachetele sunt:

1. ***Model***, în care se găsesc următoarele clase:
2. **Task**:

* În care există un id, timp de sosire și timp de servire pentru fiecare Task/Client în parte. Sunt generate setterele și getterele pentru fiecare obiect în parte, precum și constructorul. Și am implementat metoda comparteTo:
* public int compareTo(Task task)  
  {  
   if(this.arrivalTime == task.getArrivalTime()) return 0;  
   else  
   if (this.arrivalTime > task.getArrivalTime()) return 1;  
   else return -1;  
  }

unde verific condițiile pentru sortarea din arrayList-ul în care se regăsesc task-urile și returnez valorile 0, 1 și -1, în funcție de caz.

* Și metoda toString() pentru a afișa fiecare task de forma (id, timp de sosire, timp de servire)
* public String toString() {  
   return " (" +  
   " " + id +  
   ", " + arrivalTime +  
   ", " + serviceTime +  
   ") ";  
  }

1. **Server:**

* Am declarat obiectele:
* private BlockingQueue<Task> tasks;  
  private AtomicInteger waitingPeriod;  
  private Thread thread;

unde BlockingQueue<Task> tasks folosesc pentru a reține fiecare task în parte, AtomicInteger waitingPeriod pentru a reține timpul de servire/așteptare, iar Thread thread pe care îl pornim în constructorul clasei.

public Server()  
{  
 this.tasks = new ArrayBlockingQueue<Task>(1000);  
 this.waitingPeriod = new AtomicInteger(0);  
 this.thread = new Thread(this);  
 thread.start();  
}

* Am creat metoda addTask în care vom adăuga fiecare task în coadă și

incrementăm timpul total de așteptare din coada când se adaugă acest nou task (client).

* Metoda run() procesează primul task/Client din coadă, dacă aceasta conține clienți, și apoi îl va scoate în urma procesării serviciului dorit.
* public void run()  
  {  
   while(true)  
   {  
   if (tasks.size() > 0) // daca avem cel putin un element  
   {  
   Task t = this.tasks.peek(); // se ia varful cozii  
   try {  
   // oprim thread-ul timp de cate secunde are fiecare task in parte  
   // inmultim cu 1000 pentru ca este in milisecunde  
   thread.*sleep*(t.getServiceTime()\*1000);  
   } catch (InterruptedException e){  
   e.printStackTrace();  
   }  
   tasks.remove(); // din coada, vom scoate primul task dupa ce este procesat  
   }  
   }  
  }
* Apoi adaugăm toate task-urile într-o listă în metoda getTask():
* public ArrayList<Task> getTasks()  
  {  
   // adaug toate task-urile intr-o lista  
   ArrayList<Task> vect = new ArrayList<Task>();  
   for(Task x:tasks) {  
   vect.add(x);  
   }  
   return vect;  
  }
* Iar la final generăm settere și gettere.

1. ***BussinessLogic***, în care se găsesc urmatoarele clase:
2. **Strategy**:

* În clasa Strategy se va găsi strategia propiu-zisă pe care o vom folosi în proiect, implementată într-o metodă singură în această clasă. Și anume găsește coada ce are timpul minim de servire, pentru a fi cât mai optima distribuirea task-urile la fiecare server în parte.
* public class Strategy {  
   public int addTask(List<Server> servers, Task t) {  
   int minim = 10000; // initializez cu un nr mai mare  
   // luam primul element din coada la care ii luam timpul de asteptare  
   Server coada = servers.get(0); // initializam cu prima coada  
   for (Server svr :servers) {  
   if (svr.getWaitingPeriod().get() < minim) {  
   minim = svr.getWaitingPeriod().get();  
   coada = svr;  
   }  
   }  
   //frame.setCozi("Task-ul "+t.getTaskID()+" a fost adaugat la serverul "+(servers.indexOf(sv)+1)+".\n");  
   coada.addTask(t);  
   return servers.indexOf(coada);  
   }  
  }
* această metodă va adauga efectiv coada, dar va și returna indexul cozii pentru a știi unde se află coada cu timpul minim

1. **Scheduler:**

* Aici am declarat obiectele:
* private List<Server> servers;  
  private int maxNoServers;  
  private int maxTasksPerServer;  
  private Strategy strategy;  
  private int id;

unde primul obiect este o listă de servere, apoi vom avea nevoie de numărul maxim de servere, id, dar și numărul maxim de task-uri ce se pot face per server și o strategie.

* Constructorul clasei inițializează obiectele, dar și adaugă serverele într-un array.
* public Scheduler(int maxNoServers, int maxTasksPerServer) {  
   this.maxNoServers = maxNoServers;  
   this.maxTasksPerServer = maxTasksPerServer;  
   this.strategy = new Strategy();  
   servers = new ArrayList<Server>();  
   for (int i = 0; i < maxNoServers; i++)  
   {  
   servers.add(new Server());  
   }  
  }
* metoda dispatchTask ia id-ul cozii cu cel mai mic timp de procesare, ceea ce returneaza metoda din clasa strategy
* public int dispatchTask(Task t)  
  {  
   // id-ul cozii cu timpul de procesare cel mai mic  
   id = strategy.addTask(servers, t);  
   return id;  
  }
* generăm settere și gettere, apoi metoda toString, numită listServers() în care se vor afișa fiecare coada, cu fiecare timp de așteptare/servire.
* public String listServers(){  
   String string = new String();  
   for(Server svr: servers){  
   string = string + "Server " +(servers.indexOf(svr) + 1) + ":";  
   for (Task x: svr.getTasks()){  
   string = string + " " + x.toString() + " ";  
   }  
   string = string + "---- Waiting period = " + svr.getWaitingPeriod().get() + "\n";  
   }  
   return string+"\n";  
  }

1. **SimulationManager**

* Am obiectele:
* private int n = 4; // numarul N de task-uri  
  private int q = 2; // numarul Q de cozi  
  private int timpSimulare = 60; // timpul de simulare a proiectului  
  private int minArrivalTime = 2; // timpul minim de sosire  
  private int maxArrivalTime = 30; // timpul maxim de sosire  
  private int minServiceTime = 2; // timpul minim de servire  
  private int maxServiceTime = 4; // timpul maxim de servire  
  private int totalServiceTime = 0; // timpul total de servire  
  private int timpCurent = 0; // timpul curent  
  private int maxTaskPerServer = 0; // nr max task-uri per server  
  private int nrMaxTask = 0; // nr maxim de task-uri  
  private int timpMax = 0; // timpu maxim pt un task  
  private int sumaTaskWait = 0; // suma timpilor de asteptare  
  private List<Task> taskuri; // lista de task-uri  
  private Task taskCurent;  
  private Scheduler scheduler; // scheduler  
  private JTextArea textArea; // textArea pentru a-l lega de interfață
* am generat contructorul cu parametrii fiecare obiect în parte și inițializarea lor
* public SimulationManager(int n, int q, int minArrivalTime, int maxArrivalTime, int minServiceTime, int maxServiceTime, int timpSimulare, JTextArea textArea) {  
   this.n = n;  
   this.q = q;  
   this.minArrivalTime = minArrivalTime;  
   this.maxArrivalTime = maxArrivalTime;  
   this.minServiceTime = minServiceTime;  
   this.maxServiceTime = maxServiceTime;  
   this.timpSimulare = timpSimulare;  
   this.textArea = textArea;  
  }
* apoi am creat metoda generateTask în care vom genera random task-urile cu timpul de sosire si de servire, fiecaren generându-se random. Apoi adăugăm task-urile pe rând și calculăm timpul total de servire pentru afișarea rezultatelor dorite. La final afișăm task-urile generate, atât în consolă, cât și în interfață.
* public void generateTasks() {  
   ArrayList<Task> tasks = new ArrayList<Task>();  
    
   for(int id = 1; id <= n; id++)  
   {  
   // generare random pt time  
   int arrivalTime = (int)Math.*floor*(Math.*random*() \*(maxArrivalTime - minArrivalTime + 1) + minArrivalTime);  
   int serviceTime = (int)Math.*floor*(Math.*random*() \*(maxServiceTime - minServiceTime + 1) + minServiceTime);  
    
   Task task = new Task(id, arrivalTime, serviceTime);  
   tasks.add(task);  
   totalServiceTime += serviceTime; // timpul total de servire  
   }  
   taskuri = tasks;  
   n = taskuri.size();  
    
   // afisam task-urile generate  
   for (Task x : taskuri) {  
   System.*out*.println(x);  
   textArea.append(x.toString()+"\n");  
   }  
  }
* metoda run() este cea mai amplă metodă, în care se desfășoară simularea în sine. Acesta generează serverele și clienții, apoi distribuie prin intermediul scheduler-ului clienții în cozi, afișează în timp real atât în consolă, cât și în interfață coziile. Iar la finalul simulăriile, după ce toate coziile au fost afișate, se vor lista și timpul mediu de așteptare în coadă al fiecărui client, timpul mediu de servire și ora de vârf a simulării. La finalul metodei se vor introduce toate aceste date și în fișier, nu doar în interfață și consolă, conform cerinței.
* public void run() {  
   // creeam un scheduler nou cu un numar dat de servere si nr maxim de task-uri per server  
   scheduler = new Scheduler(q, maxTaskPerServer);  
   // generam primul task  
   generateTasks();  
   // parcurgem pana cand timpul de simulare ajunge la limita  
   while (timpCurent < timpSimulare) {  
   // verificam daca vreun task nou a ajuns la timpul curent  
   for (Task tsk : taskuri) {  
   if (tsk.getArrivalTime() == timpCurent) {  
   // vom pune/asigna un task la un server folosind scheduler  
   int idServer = scheduler.dispatchTask(tsk);  
   // apoi updatam timpul total de asteptare pentru task  
   sumaTaskWait += scheduler.getServers().get(idServer).getWaitingPeriod().get() - tsk.getServiceTime();  
   }  
   }  
   // calculam numarul total de task-uri din toate serverele  
   int sumTasks = 0;  
   for (Server svr : scheduler.getServers()) {  
   sumTasks += svr.getTasks().size();  
   }  
   // actualizam numarul orelor maxime/de varf dacă ora curentă are mai multe task-uri decât ora de varf anterioara  
   if (sumTasks > nrMaxTask) {  
   nrMaxTask = sumTasks;  
   timpMax = timpCurent; }  
   System.*out*.println(timpCurent);  
   textArea.append(String.*valueOf*(timpCurent) + "\n");  
   System.*out*.println(scheduler.listServers());  
   textArea.append(String.*valueOf*(scheduler.listServers()) + "\n");  
   // crestem timpul de simulare curent cu unu  
   timpCurent++;  
   // apoi vom actualiza timpul de asteptare pentru fiecare server in parte  
   for (Server svr : scheduler.getServers()) {  
   if (svr.getWaitingPeriod().get() > 0) {  
   svr.setWaitingPeriod(svr.getWaitingPeriod().get() - 1);  
   }  
   }  
   try {  
   // adormim thread-ul pentru o secunda pentru a simula in timp real  
   Thread.*sleep*(1000);  
   } catch (InterruptedException e) {  
   e.printStackTrace();  
   }  
   }  
   // vom genera statistica ceruta intr un string pentru a l afisa pe ecran  
   String s = new String("~ Valoarea timpului mediu de asteptare a fost " + ((double) sumaTaskWait / n) + "secunde. ~\n" +  
   "~ Ora de varf a fost " + timpMax + " secunde, cand s-au aflat " + nrMaxTask + " task-uri la servere. ~\n" +  
   "~ Timpul de servire/asteptare mediu a fost de " + ((double) totalServiceTime / n) + " secunde. ~");  
    
   for (Server svr : scheduler.getServers()) {  
   System.*out*.println(svr.getTasks());  
   textArea.append(String.*valueOf*(svr.getTasks()) + "\n");  
   }  
   // apoi afisam acest string de statistica si lista de task-uri pentru fiecare server in parte  
   textArea.append(s);  
    
   try {  
   FileWriter myWriter = new FileWriter("test3.txt");  
   myWriter.write(textArea.getText());  
   myWriter.close();  
   System.*out*.println("S-a executat cu succes!");  
   } catch (IOException exceptie) {  
   System.*out*.println("!!!EROARE!!!");  
   exceptie.printStackTrace();  
   }  
   }  
  }

1. Ultima clasă este SimulationFrame, care este interfața programului

* Sunt implementate 7 textField-uri, pentru a introduce toate datele cerute în enunț, și 7 jlabel-uri pentru a indica utilizatorului ce trebuie să introducă în fiecare câmp.
* De asemenea exista încă un jlabel în partea de sus central, în care se află titlul, iar unul este situat deasupra textArea, pentru a indica unde se vor afișa datele/simularea dorită, și anume în textArea.
* În partea de jos, fix sub textArea există 2 butoane. Butonul de start care va porni simularea, în actionListener-ul său apelându-se constructorul din simulationManager pentru a porni programul, folosind un thread care primește obiectul de simulationManager. Iar butonul de delete șterge ceea ce este scris în textArea în urma simulării.
* SimulationManager s = new SimulationManager(n, q, minArrivalTime, maxArrivalTime, minServiceTime, maxServiceTime, timpSimulare, textArea);  
  Thread t = new Thread(s);  
  t.start();

**Modelul interfeței implementate**

O imagine care conține masă

Descriere generată automat

1. **Rezultate**

O imagine care conține text

Descriere generată automat

Rezultatele pentru fiecare test în parte sunt:

* **Test 1:**

~ Valoarea timpului mediu de așteptare a fost 0.0 secunde. ~

~ Ora de vârf a fost 27 secunde, când s-au aflat 2 task-uri la servere. ~

~ Timpul de servire/așteptare mediu a fost de 2.75 secunde. ~

* **Test 2:**

~ Valoarea timpului mediu de așteptare a fost 1.3 secunde. ~

~ Ora de vârf a fost 29 secunde, când s-au aflat 10 task-uri la servere. ~

~ Timpul de servire/așteptare mediu a fost de 4.02 secunde. ~

* **Test 3:**

~ Valoarea timpului mediu de așteptare a fost 103.744 secunde. ~

~Ora de vârf a fost 100 secunde, când s-au aflat 715 task-uri la servere.~

~ Timpul de servire/așteptare mediu a fost de 5.998 secunde. ~

1. **Concluzii**

În urma acestei teme am învătat despre fire de execuție și cum se folosesc acestea în java. De asemenea am învățat și despre variabile sincronizate, dar am și recapitulat coziile și principiile POO de încapsulare și nu numai.

O posibilă dezvoltare a acestei teme poate să fie pentru a putea utilizatorul să aibă mai multe servicii pe care vrea ca server-ul să le rezolve, introducând eventual încă un câmp în care utilizatorul sa introducă pentru fiecare task în parce câte servicii adiționale dorește.

1. **Bibliografie**

<https://www.w3schools.com/java/java_files_create.asp>

<https://dsrl.eu/courses/pt/>

<https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html>

<https://www.tutorialspoint.com/java/util/timer_schedule_period.htm>

<https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>