**Tehnici de programare**

**Tema 2:**

**Queuing based systems Application**

Student: Pop Tudor Andrei

Grupa:20229

Data: 21.03.2019

Cuprins:

1. Obiectivul temei
2. Analiza problemei, modelare, scenarii, cazuri de utilizare
3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)
4. Implementare
5. Rezultate
6. Concluzii
7. Bibliografie
8. Obiectivul temei

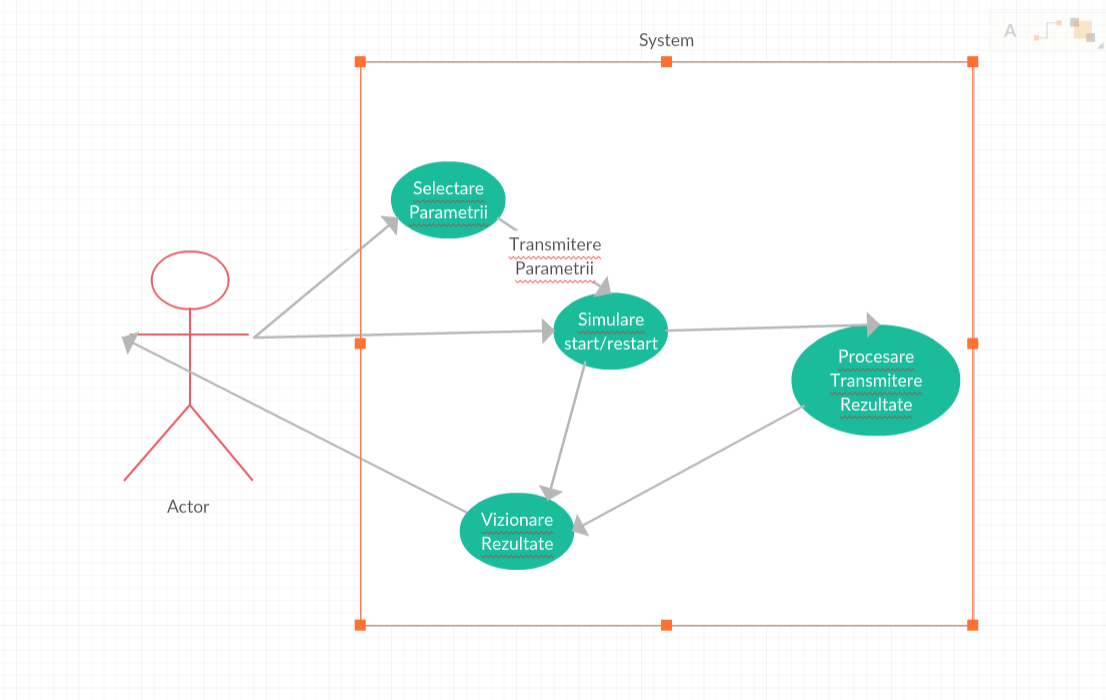
**Objective**

Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients’ waiting time.

**Description**

Queues are commonly used to model real world domains. The main objective of a queue is to provide a place for a "client" to wait before receiving a "service". The management of queue based systems is interested in minimizing the time amount their "clients" are waiting in queues before they are served. One way to minimize the waiting time is to add more servers, i.e. more queues in the system (each queue is considered as having an associated processor) but this approach increases the costs of the service supplier. When a new server is added the waiting customers will be evenly distributed to all current available queues. The application should simulate a series of clients arriving for service, entering queues, waiting, being served and finally leaving the queue. It tracks the time the customers spend waiting in queues and outputs the average waiting time. To calculate waiting time we need to know the arrival time, finish time and service time. The arrival time and the service time depend on the individual clients – when they show up and how much service they need. The finish time depends on the number of queues, the number of clients in the queue and their service needs. Input data: - Minimum and maximum interval of arriving time between customers; - Minimum and maximum service time; - Number of queues; - Simulation interval; - Other information you may consider necessary; Minimal output: - The average of waiting time, service time and empty queue time for 1, 2 and 3 queues for the simulation interval and for a specified interval (other useful information may be also considered); - Log of events and main system data; - Queue evolution; - Peak hour for the simulation interval;

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

UML Case Diagram: 

Actorul – reprezinta utilizatorul, acesta introduce datele de intrare pentru simularea scenariului

SelectareParametrii – datele introduse de catre actor sunt salvate in variabile si transmise mai departe prin apasarea butonului “Submit”. Aceste date sunt validate pentru o simulare corecta. Simularea nu va putea avea loc cu date invalide

Simulare start/restart – in aceasta fereastra, actorul are acces la 3 butoane pentru a porni simularea sau a o reporni, sau pentru a porni o alta simulare cu alte date de intrare. Mai exista inca un buton “stop” care ar fi trebuit sa poata opri simularea sa un moment ales de user, dar momentan nu functioneaza

Procesare Transmitere Rezultate – aceasta parte va prelua datele introduse de catre actor, de asemenea si comanda de pornire a simularii, iar rezultatele aceteia prin intermediul variabileleor statice si a metodelor vor fi transmise mai departe

Vizionare Rezultate – aceata parte va afisa in timp ce se intampla in timpul simularii : stare cozilor intr-un panou separate, care Task a fost indeplinit si de care TaskSolver a fost servit. Tot aici Actorul poate observa timpul mediu de asteptare si peak time-ul updatandu-se in timp real precum si numarul de taskuri servite de tot programul

Un posibil mod de utilizare:

1. Se introduce timpul minim de generare al Task-urilor (in secunde)
2. Se introduce timpul maixm de generare al Task-urilor (in secunde)
3. Se introduce timpul minim de servire al Task-urilor (in secunde)
4. Se introduce timpul maxim de servire al Task-urilor (in secunde)
5. Se introduce numarul de cozi/threaduri/TaskSolvere
6. Se introduce cat timp va avea loc simularea (in secunde)
7. Se apasa butonul “Submit”
8. Se apasa butonul “Start” si se observa rezultatele
9. Se apasa butonul “Start” pentru a observa alte rezultate cu aceiasi parametrii
10. Se apasa butonul “Again” pentru se introduce alti parametrii (se reiau tot pasii propusi anterior)

Daca datele de intrare sunt introduse gesit (min > max, numarul de threaduri > 8 (apare specificat) sau daca se introduce caratere necunoscute, se va afisa un mesaj de eroare in interfata. Fereastra pt pornirea simularii nu va aparea doar daca datele sunt valide

Daca datele introduse sunt valide se poate incepe simularea prin intermediul butonului “start”. Butonul “Stop” nu face nimic… in curs de dezvoltare. Butonul “Again” va readuce fereastra cu datele de intrare.

Pentru a se reseta in totalitate datele de la simularile anterioare, este nevoie de o repornire a programului.

Pentru a avea posibilitatea unei rulari cu adevarat paralela, numarul de threaduri introdus nu ar trebui sa depasasca 5 sau 6

3. Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfete, relatii, packages, algoritmi, interfata utilizator)

Pentru proiectare am organizat in pachetul “core” clientii in clasa “Task”. Aceasta clasa contine un indentificator unic (id) pentru a ajuta la citire si timpul de servire/procesare, valoare acestuia fiind utilizata mai tarziua in clasa TaskSolver. Ca metode importante am toStrin impreuna cu gettere si settere

public class Task {

public static int randomTaskNumber = 0;

private int id;

private int servingTime;

--

}

Clasa TaskSolvere reprezinta casele care proceseaza clientii, aici se afla majoritatea metodelor care foloses in generarea datelor de iesire, precum si a statisticilor afisate

public class TaskSolver extends Thread{

public static String logThings = "";

private int solverNumber;

private int waitingTime;

private int totalTimePassed;

private int nrTasksSolved;

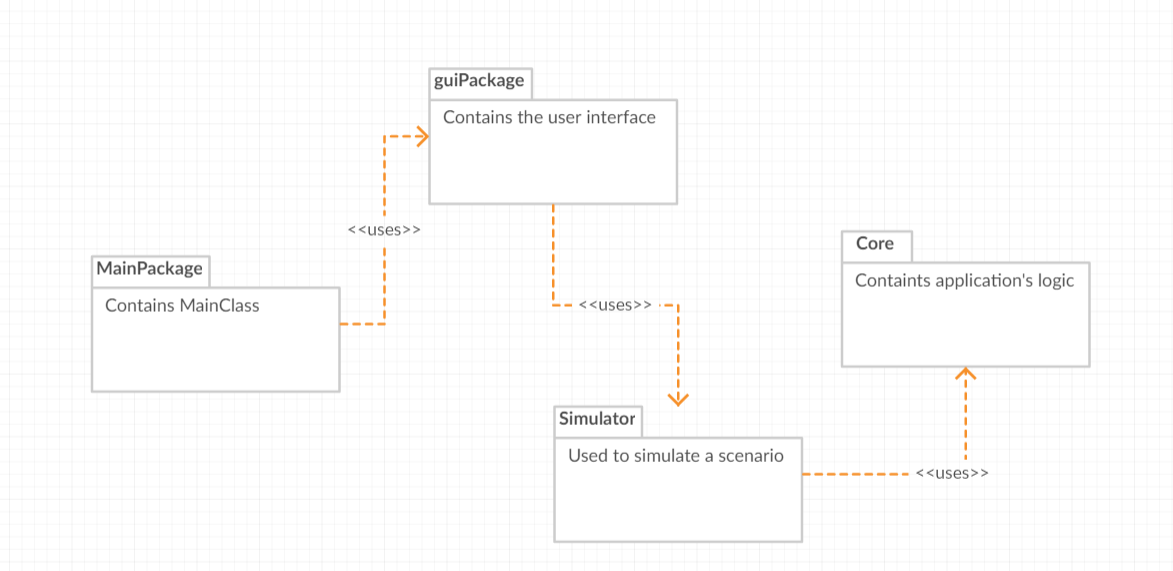
private boolean timeToStop;

public Queue<Task> q;

}

Simulatorul va prelua datele introduse in interfata si fa simula scenariul droit.

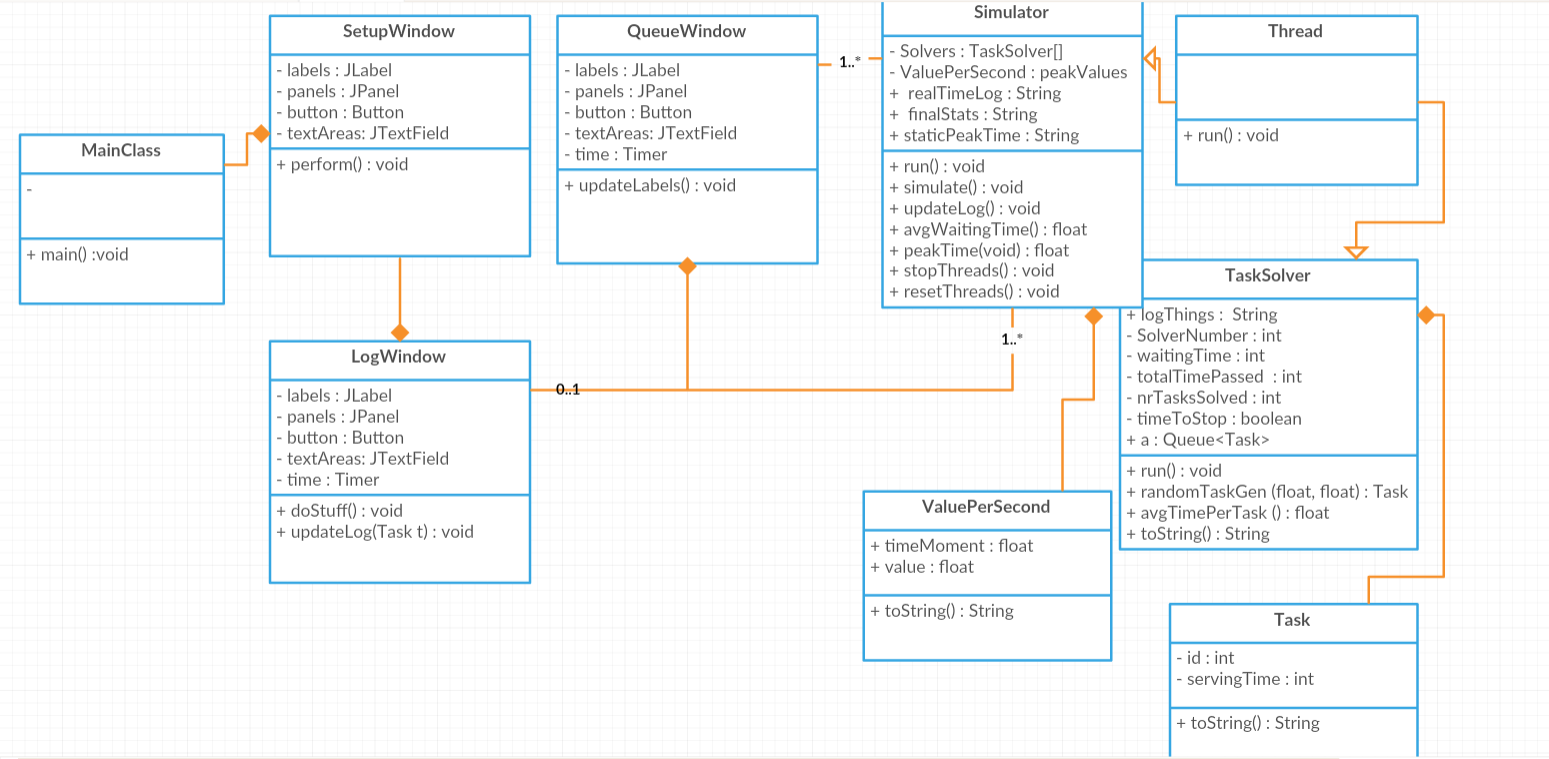
**Package Diagram:**



Dupa cum se poate observa din diagrama pachetelor utilizatorul interactioneaza direct doar cu casa pachetul MainPackage care la randul sau formeaza un lant care ajunge pana la “core” unde se prelucreaza datele.

Pachetul interfetei grafice impreuna cu pachetul simulatorului preiau datele de intrare si le folosesc pentru a simula scenariul droit. Aceste pachete ofera user-ului interactiune cu aplicatia, aducand de asemenea si rezultatele obtinute pentru a foi observate.

**Class Diagram:**

****

Dupa cum se poate observa din ilustratia de mai sus, clasele Simulator si TaskSolver mostensc clasa Thread din java pentru a realiza rularea programului in parallel, si pentru a aduce utilizatorului o interactiune “real time” cu starea interna a programului

4. Implementare

Varianta acutala acestei aplicatii este rezultatul unui sir lung de incercari si esuari in care am incercat diferite metode, diferite abordari, variante proprii implementand runnable, Thread, variante asemanatoare cu cele prezentate la laborator ( poate chiar curs ) testate pe o perioada de 3 saptamani.

Codul contine diferite bucati de cod commentate, variabile nefolosite, metode neapelate sau care pur si simplu nu am nicio aplicabilitate toate din incercariile anterioare care pentru un motiv sau altul nu si-au gasit loc in variant “finala”

Clasa MainClass : in interiorul aceste clasa se afla metoda “public static void main(String[] args)” in interiorul acesteia declar un obiect din clasa “SetupWindow” si apelez cu acel obiect metoda “perform()”. Desigur, acest apel este inconjurat de o structura try/catch pentru a intercepta eventualele exceptii ce inevitabil pot aparea

Clasa SetupWindow : in interiorul acestei clasa construiesc efectiv interfata grafica si implementez clasa “ProjectListener”. Constructorul acestei clase da forma aplicatiei, un obiect ”JFram” care contine restul obiectelor intr-un “GridLayout”. Fereastra in sine are rol de a procura si valida parametrii care trebuie sa dicteze cum decurge rularea programului. O data cu apasarea butonului Submit aceasta fereastra se inchide si se initializeaza urmatorul elemnt al interfeti grafice (LogWindow). De asemenea in aceasta clasa se afla metoda “perform()” care asociaza un listener butonului de subit care duce mai departe inputurile spre validare.

Clasa LogWindow: initializarea acestei ferestre se afla in clasa prezentata anterior. Aici se poate observa activitatea TaskSolverelor in timp real, timpul mediu de asteptare, peaktime-ul, numarul de taskuri procesate la momentul current. Userul are de asemenea optiunea de a restarta simularea cu datele introduse, sau de a merge inapoi si a porni o simulare cu alte date de intrare. Acesta fereastra deschide ultima componenta a interfeti grafice care ca fi prezentata in continuare.

Clasa **QueueWindow** : este o clasa absolut esentiala acestui proiect deoarece in interiorul el se poate observa stare cozilor care TaskSolverele le proceseaza. Taskurile pot fi observate in timp real cum intra (sunt generate) si cum ies (sunt procesate)

Clasa **Task:** Aceasta clasa reprezinta clientii. Acestia au fiecare un numar de cu care sunt identificati in mod unic, un id si timpul de procesare care va merge mai departe in TaskSolver

Clasa **TaskSolver:** Este o clasa foarte importanta din acest proiect. Aceasta clasa contrine o coada de Task-uri care sunt generate in mod aleator.

Metoda run() este defapt metoda run suprascrisa din clasa Thread care este mostenita in aceasta component. Aici thread-ul extrage din coada un element si asteapta (in milisecunde) timpul necesar de procesare care apare In fiecare task. Tot in aceasta metoda ne sunt generate si updatate niste variabile statice care au rol de a aduce datele legate de procesarea taskurilor in timp real spre clasele din User Interface pentru a fi afisate si mai apoi evaluate.

Code sample:

if(!this.emptyQueue()) {

t = this.removeTask();

try {

sleep(t.getServingTime());

} catch (InterruptedException e) {

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

System.out.println("la sleep.. int taskSolver");

}

this.nrTasksSolved ++;

this.updateTimePassed(t);

this.updateServingTime(t);

System.out.println("The task :"+"Id: "+t.getId() + ", Serving time:"+(t.getServingTime()/1000f)+" seconds Solved by Thread: "+this.solverNumber);

this.updateLogThings(t);

}

Metoda RandomTaskGen – genereaza un nou Task care are timpul de procesare o valoare aleatoare luata din intrervalul specificat la incepulul simularii

public static synchronized Task randomTaskGen(float min, float max) {

min\*=1000;

max\*=1000;

Task.randomTaskNumber++;

return new Task(Task.randomTaskNumber,TaskSolver.randomServingTime(min, max) );

}

Tot in aceasta clasa apare si metoda toString care are rolul de a afisa sub un anumit format lista pe care threadul current o are in procesare

Clasa **Simulator** : Aceasta este cea mai importanta clasa din tot proiectul. Mosteneste obiectul Thread pentru a permite afisarea in timp real a datelor proiectului. Tot in aceasta clasa ajung toate datele de intrare introduse de user in prima interfata.

Aceasta clasa are un tablou cu TaskSolvere care fiecare la randul sau are o coada de Taskuri pe care trebuie sa le serveasca.

for(int i=1; i<=this.getNrOfTaskSolvers(); i++) {

getT().add(new TaskSolver(i));

this.Solvers[i-1] = new TaskSolver(i);

this.Solvers[i-1].start();

}

Metoda run() din aceasta clasa, similar clasei TaskSolvere suprascrie metoda run din clasa Thread. In interiorul acestei metode simulatorul isi genereaza un timp aleator de sleep() procurat din intervalul in care trebuie sa se generez taskuri, dupa care va adauga un nou task Solverului care are timpul de asteptare cel mai mic pentru a eficientiza programul. Code sample :

TaskSolver dummy;

int minIdx = this.minIndex();

dummy = this.getSolvers()[minIdx];

dummy.addTask(TaskSolver.randomTaskGen(this.minServ, this.maxServ)); // generare task cu un timp de asteptare random

this.peakTime(timePassed); // genereaza peak time moment

this.updateLog();

tot in aceasta metoda run, simulatorul foloste variabliele statice care se afla in aceasta clasa, precum si cele din clasa TaskSolver pentru a genera correct si in timp real starea in care se afla partea de procesare a datelor din aplicatie.

Nefiind mentionat mai sus, in clasele LogWindow si QueueWindow s-a folosit un timer pentru a updata permanent datele din ferestre la un interval fix de timp ( de 10 ori pe secunda ) pentru a asigura date reale si valide in legatura cu simularea. Aceste date sun preluate direct din simulator care, dupa cum spuneam si mai devreme, genereaza toate aceste date in metoda run()

Clasa ValuePerSecond: este folosita pentru a stoca intr-un mod mai organizat datele despre peakTime. Aceasta clasa nu aduce niciun beneficiu prigramului in sine si s-ar putea elimina, insa metoda toString si folosirea sa in anumite portiuni de simulare face codul mai lizibil, mai compact si mai organiza. Iar pe langa toate acestea, e mai usor de implementat.

5. Rezultate

Conform asteptariilor, programul functioneaza bine, afisaza rezultatele in timp real, ofera informatii suplimentare preacum peakTime/AverageWaitingTime.

Simulare se poate repeta prin apasarea butonului start. De asemenea se pot introduce date noi prin apasarea butonului “Again”.

Din pacate, pentru oprirea thredurilor (respective a simularii) nu am gasit o metoda foarte buna, iar la final TaskSolverele vor mai afisa ultimele task-uri care si mai afla in metoda Run in asteptare, vor procesa datele acestora si abia apoi se vor inchide.

6. Concluzii

- documentatia s-a putut scrie in ultima zi (\*noapte 2:15 AM, Test la EGC imediat dupa lab-ul de PT (y) )

- proiectul se poate inbunatatii la fiecare pas: optimizare, design, implementarea operatiilor pe mai multe threaduri, oprirea threadurilor trebuie revizuita

- codul contine multe ramasite de la incercarile anterioare, incluziv clase care nu au niciun impact asupra logicii in sine, dat daca sunt eliminate apar exceptii, iar programul refuza sa functionze

-am invatat multe despre cum functioneaza functia threadurile : synchronized synchronized synchronized synchronized synchronized synchronized synchronized synchronized synchronized synchronized synchronized synchronized synchronized synchronized

- foarte multe problem cu TreeSet<>, exagerat, nu am resusit sa pastrez relatia de ordine intre threaduri nicicum, nici dupa synchronized la metode compare(). Am renuntat si am folosit un tablou unidimensional simplu.

7. Bibliografie

1) <https://en.wikipedia.org/wiki/Polynomial_long_divisio> -- pentru impartire

2) <https://creately.com/diagram-type/use-case> -- pentru diagrame

3) <https://stackoverflow.com/> -- pentru restul problemelor

4) Toate linkurile care apar prin cod