

SIMULATOR DE COZI

Student: Nemțanu Alexandru-Vasile

Grupa: 30227

Cuprins

1. Obiectivul temei.
2. Analiza problemei
3. Proiectare și diagrama UML
4. Pachete și clase
5. Rezultate
6. Concluzii
7. Bibliografie
8. Obiectivul temei:

Obiectivul temei este de a proiecta și de a implementa o aplicație de gestionare a cozilor care atribuie clienții la cozi astfel încât timpul de așteptare să fie minimizat.

1. Analiza problemei:

Cozile sunt utilizate frecvent pentru a modela domenii din lumea reală. Obiectivul principal al unei cozi este de a furniza un loc în care un "client" să aștepte înainte de a primi un "serviciu". Managementul sistemelor bazate pe cozi urmărește minimizarea timpului în care "clienții" așteaptă în cozi înainte de a fi serviți.

Una dintre modalitățile de minimizare a timpului de așteptare este de a adăuga mai multe servere, adică mai multe cozi în sistem (fiecare coadă este considerată ca având un procesor asociat), dar această abordare crește costurile furnizorului de servicii.

Aplicația de gestionare a cozilor trebuie să simuleze (prin definirea unui timp de simulare 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛) o serie de N clienți care sosesc pentru a primi un serviciu, intră în Q cozi, așteaptă, sunt serviți și în cele din urmă părăsesc cozile. Toți clienții sunt generați când simularea începe și sunt caracterizați de trei parametri: ID (un număr între 1 și N), 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 (timpul de simulare când sunt pregătiți să intre în coadă) și 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 (intervalul de timp sau durata necesară pentru a servi clientul; adică timpul de așteptare când clientul este în fața cozii).

Aplicația urmărește timpul total petrecut de fiecare client în cozi și calculează timpul mediu de așteptare. Fiecare client este adăugat în coadă cu timpul minim de așteptare atunci când timpul lor 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 este mai mare sau egal cu timpul de simulare (𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 ≥ 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛).

Urmatoarele date trebuie considerate ca date de intrare pentru aplicatie si trebuie introduse de utilizator in interfata aplicatiei:

* Numarul de clienti (N);
* Numarul de cozi (Q);
* Intervalul de simulare (timpul maxim de simulare);
* Timpul minim si maxim de sosire (timpul de sosire minim ≤ timpul de sosire ≤ timpul de sosire maxim);
* Timpul minim si maxim de serviciu (timpul minim de serviciu ≤ timpul de serviciu ≤ timpul maxim de serviciu).

1. Proiectare:

În proiectarea OOP a acestui proiect, s-au utilizat threaduri în Java, preferându-se implementarea interfeței Runnable în defavoarea extinderii clasei Thread, deoarece aceasta oferea o mai mare libertate de exprimare.

Multithreading în Java se referă la un program care execută simultan mai multe threaduri, partajând o zonă de memorie comună. Fiecare fir de lucru își începe executia la o locație bine cunoscută și predefinită. Fiecare fir de lucru își execută codul independent de celelalte fire de lucru din program, părând a avea un anumit grad de simultaneitate în execuție și având acces la diferite tipuri de date.

Un fir de lucru în Java începe prin crearea unei instanțe a clasei java.lang.Thread. Metodele din clasa Thread pentru manipularea firelor de executie sunt, de exemplu, start(), yield(), sleep() și run(). Acțiunea firului de executie începe la invocarea metodei run(). Fiecare fir de executie are propria sa stivă de apeluri.

Definirea și instanțierea unui fir de execuție poate fi realizată prin extinderea clasei java.lang.Thread sau prin implementarea interfeței Runnable. Singurul motiv pentru a extinde clasa Thread este cazul în care se dorește realizarea unei versiuni mai specializate a clasei Thread. În restul cazurilor, când se dorește doar specificarea a ceea ce trebuie să execute firul de executie, se definește o clasă care implementează interfața Runnable.

Implementarea interfeței Runnable oferă flexibilitatea de a extinde orice altă clasă, păstrându-și proprietatea de a putea fi executată într-un fir de lucru separat. Un obiect care implementează interfața Runnable este trimis ca argument către un obiect de tip Thread nou creat pentru ca firul de executie să știe a cui metodă run o va executa când este pornit.

Structurile de date utilizate au fost în principal array-uri generale și concurente implementate de Java, cum ar fi ArrayList și ArrayBlockingQueue etc. Interfața implementată este Strategy, responsabilă cu metoda addTask.

În interiorul unui fir de executie lucrurile se petrec într-o ordine predictibilă, dar acțiunile firelor de executie multiple pot fi amestecate într-o ordine neprevăzută. Dacă același cod se rulează de mai multe ori sau pe mașini diferite, rezultatul poate să fie diferit.

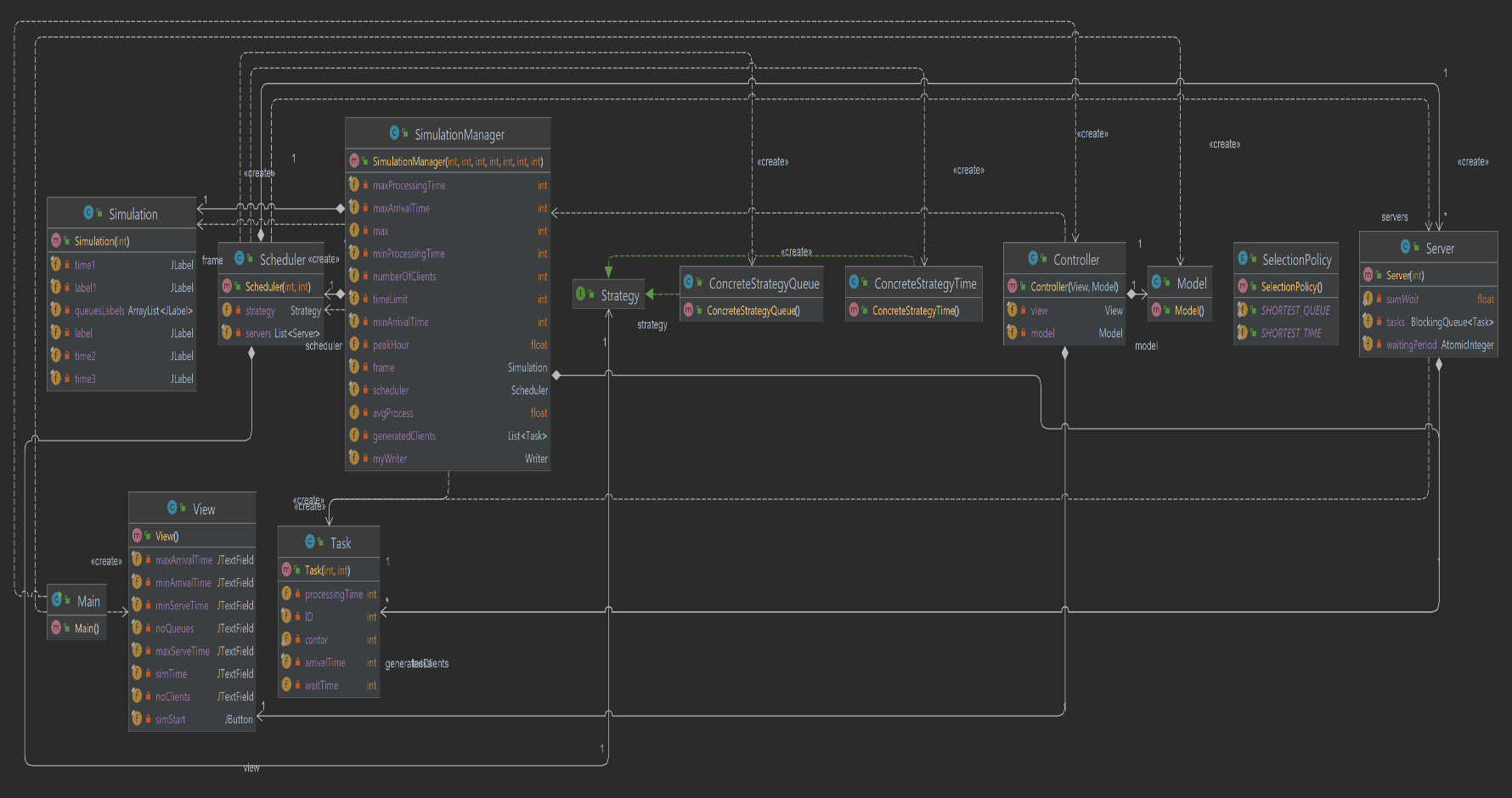


Diagrama UML:

1. Pachete și clase:

* Pachetul **Server**, clasa **Server**:

Aceasta clasă reprezintă un server care poate procesa Task-uri. Clasa implementează interfața Runnable, ceea ce înseamnă că poate fi utilizată pentru a rula într-un fir separat de execuție. Clasa are o coadă de Task-uri (reprezentată de un obiect BlockingQueue<Task>), o variabilă atomică pentru perioada de așteptare a Task-urilor și o variabilă statică pentru a păstra suma perioadelor de așteptare a tuturor Task-urilor procesate de instanțele Server-ului.

Constructorul primește ca parametru numărul maxim de Task-uri pe care Server-ul le poate gestiona.

Metoda addTask adaugă un nou Task în coadă.

Metoda run() conține logica de procesare a Task-urilor: Server-ul așteaptă ca primul Task din coadă să fie disponibil pentru procesare, apoi îl procesează și îl elimină din coadă, actualizând perioada de așteptare.

Metoda getTasks() convertește coada de Task-uri într-un array de Task-uri și returnează array-ul.

De asemenea, clasa Server are alte câteva metode, cum ar fi getWaitingPeriod(), care returnează perioada totală de așteptare a Task-urilor din coadă, getNumberOfTasks(), care returnează numărul de Task-uri din coadă, și clear(), care elimină toate Task-urile din coadă. Metoda getSumWait() returnează suma perioadelor de așteptare a tuturor Task-urilor procesate de instanțele Server-ului, iar această valoare este statică și se aplică tuturor instanțelor Server.

* Pachetul **Simulation**, clasa **Simulation Manager**:

Clasa SimulationManager implementează interfața Runnable și este responsabilă pentru gestionarea simulării cozilor de procesare. Această clasă generează clienți cu diferite intervale de sosire și timpi de procesare, îi distribuie pe acești clienți pe servere și menține evidența stării acestora de-a lungul timpului.

Constructorul primește o serie de parametri necesari pentru inițializarea simulării, cum ar fi durata simulării, timpul maxim și minim de procesare și intervalul maxim și minim de sosire a clienților. De asemenea, constructorul inițializează o instanță de Scheduler, care este folosit pentru a distribui clienții la servere, și o instanță de Simulation, care este folosită pentru a afișa starea curentă a simulării.

Metoda generateNRandomTasks generează un număr specific de clienți ale căror intervale de sosire și timpi de procesare sunt aleatorii. Acești clienți sunt sortați ulterior după timpul lor de sosire.

Metoda run reprezintă codul principal al simulării. Aceasta implementează bucla principală care rulează simularea până când timpul limită a fost atins. În fiecare iterație, metoda parcurge lista de clienți generați și, dacă un client a sosit la momentul curent, îl trimite la unul dintre servere utilizând Scheduler. De asemenea, calculează timpul mediu de așteptare pentru clienții aflați în coadă și afișează starea curentă a coziilor și a clienților așteptând în coadă.

* Pachetul **Model,** clasa **Model:**

Această clasă conține o metodă numită "validator" care primește un șir de caractere și verifică dacă acesta conține doar cifre. Dacă șirul de intrare este nul, gol sau conține doar spații albe, metoda returnează -1. Dacă șirul de intrare conține doar cifre, metoda returnează 0. În caz contrar, metoda returnează -1. Metoda folosește clasa "Pattern" din pachetul "java.util.regex" pentru a crea un șablon regex care potriveste șirurile de caractere care conțin doar cifre.

* Pachetul **Strategy**, clasa **ConcreteStrategyQueue**:

Aceasta clasa este o implementare concretă a interfeței Strategy și are ca scop adăugarea unei sarcini (Task) in coada unui server dintr-o lista de servere (servers).

Metoda addTask implementeaza logica de selectie a serverului cu cel mai mic numar de sarcini si adauga sarcina la coada acestui server.

Se utilizeaza metoda min din clasa Collections pentru a determina serverul cu cel mai mic numar de sarcini. Se utilizeaza interfata Comparator pentru a compara serverele dupa numarul de sarcini pe care le detin.

* Pachetul **Strategy**, clasa **ConcreteStrategyTime**:

Această clasă implementează interfața Strategy și definește o strategie de alegere a unui server pentru a adăuga o sarcină în funcție de cel cu cel mai mic timp de așteptare.

Metoda addTask primește o listă de servere și o sarcină. Se parcurge lista de servere pentru a găsi serverul cu cel mai mic timp de așteptare și se salvează în variabila chosenServer. Timpul de așteptare este calculat folosind metoda getWaitingPeriod a clasei Server. Sarcina este apoi adăugată la chosenServer și timpul de așteptare al sarcinii este actualizat cu timpul de procesare al sarcinii plus timpul de așteptare al serverului.

* Pachetul **Strategy**, clasa **Scheduler**:

Această clasă reprezintă un planificator de sarcini, care este responsabil pentru atribuirea sarcinilor către servere într-un mod eficient. Planificatorul este creat cu un număr maxim de servere și un număr maxim de sarcini per server, care sunt stocate într-o listă de obiecte Server. Planificatorul utilizează o strategie pentru a decide cum să distribuie sarcinile în funcție de politicile de selecție alese. Există două politici de selecție disponibile: "Shortest Queue" și "Shortest Time", care sunt implementate în clasele ConcreteStrategyQueue și ConcreteStrategyTime.

Metoda changeStrategy() este responsabilă pentru schimbarea strategiei utilizate de planificator. Dacă politica selectată este "Shortest Queue", atunci planificatorul va crea un obiect ConcreteStrategyQueue. Dacă politica selectată este "Shortest Time", atunci planificatorul va crea un obiect ConcreteStrategyTime.

Metoda dispatchTask() este responsabilă pentru adăugarea sarcinii la unul dintre servere, folosind strategia selectată.

Metoda getServers() returnează lista de servere.

Metoda check() este responsabilă pentru verificarea dacă există sarcini rămase de executat. Verifică numărul total de sarcini din toate serverele și returnează true dacă numărul total de sarcini este 0, altfel returnează false.

* Pachetul **Strategy**, clasa **SelectionPolicy**:

Aceasta este o enumerare cu două valori posibile: SHORTEST\_QUEUE și SHORTEST\_TIME. Este utilizată pentru a indica politica de selectare a serverelor în cadrul clasei Scheduler. Valoarea SHORTEST\_QUEUE indică faptul că se va selecta serverul cu cel mai mic număr de sarcini în coadă, în timp ce valoarea SHORTEST\_TIME indică faptul că se va selecta serverul cu cel mai scurt timp de așteptare pentru a procesa o sarcină nouă.

* Pachetul **Strategy**, interfața **Strategy:**

Acesta este o interfață pentru strategiile de planificare a sarcinilor. Interfața definește o metodă abstractă numită addTask, care ia o listă de servere și o sarcină și adaugă sarcina la unul dintre servere, în funcție de o anumită strategie. Această interfață este implementată de clasele ConcreteStrategyQueue și ConcreteStrategyTime, care definesc diferite strategii de planificare a sarcinilor.

* Pachetul **Task,** clasa **Task:**

Clasa are un constructor cu doi parametri, arrivalTime și processingTime, care setează valorile atributelor corespunzătoare și incrementează contorul static pentru a genera un ID unic.

Clasa are câteva metode publice care oferă acces la atributele private ale obiectului. Metoda getArrivalTime returnează timpul de sosire al sarcinii, getProcessingTime returnează timpul necesar pentru procesarea sarcinii, iar metoda getWaitTime returnează timpul de așteptare al sarcinii. Metoda setWaitTime este folosită pentru a seta timpul de așteptare al sarcinii.

Clasa are o metodă toString care returnează un șir de caractere formatat care conține ID-ul sarcinii, timpul de sosire și timpul de procesare. Clasa are și o metodă decrementProcessingTime care scade processingTime cu o unitate.

* Pachetul **View**, clasa **Simulation**:

Clasa SimulationFrame este unul dintre GUI urile prezente in acest proiect. Acest GUI este cel initializat in SimulationManager și reprezintă interfața interactivă care afișează în timp real procesul de punere în cozi a clienților.

* Pachetul **View**, clasa **View**:

View este cel de-al doilea GUI al aplicației și reprezintă clasa care modelează prima interfață(cea in care utilizatorul introduce datele simularii si instrumentele ca aceasta sa fie pornita).

* Pachetul **Controller**, clasa **Controller**:

Controller implementează partea tehnică de start a simularii în care gestionează interacțiunea cu primul GUI în care utilizatorul introduce datele.

1. Rezultate

Pentru testarea aplicatiei s-au luat trei cazuri:

* Primul caz :

Avem patru clienti, doua cozi, timpul maxim de simulare este de 60, timpul minim de sosire este 2, timpul maxim de sosire este 30, timpul minim de procesare este 2 iar timpul maxim este 4.

* Al doilea caz:

50 de clienti, cinci cozi, 60 de secunde timpul de simulare ,timpul minim de sosire este 2 iar timpul maxim este 30,timpul minim de procesare este 1 iar timpul maxim este 7.

* Al treilea caz:

1000 de clienti,20 de cozi,200 de secunde timpul de simulare,timpul minim de sosire este 10 iar timpul maxim 100,timpul minim de procesare este 3 iar timpul maxim este 9.

Mai jos avem un exemplu al primului caz:

Simulation Time: 0

Waiting Clients: (4,10,2)(1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 1

Waiting Clients: (4,10,2)(1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 2

Waiting Clients: (4,10,2)(1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 3

Waiting Clients: (4,10,2)(1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 4

Waiting Clients: (4,10,2)(1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 5

Waiting Clients: (4,10,2)(1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 6

Waiting Clients: (4,10,2)(1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 7

Waiting Clients: (4,10,2)(1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 8

Waiting Clients: (4,10,2)(1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 9

Waiting Clients: (4,10,2)(1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 10

Waiting Clients: (1,17,2)(2,26,2)(3,29,3)

Queue 1: (4,10,2)

Queue 2:

Simulation Time: 11

Waiting Clients: (1,17,2)(2,26,2)(3,29,3)

Queue 1: (4,10,1)

Queue 2:

Simulation Time: 12

Waiting Clients: (1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 13

Waiting Clients: (1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 14

Waiting Clients: (1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 15

Waiting Clients: (1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 16

Waiting Clients: (1,17,2)(2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 17

Waiting Clients: (2,26,2)(3,29,3)

Queue 1: (1,17,2)

Queue 2:

Simulation Time: 18

Waiting Clients: (2,26,2)(3,29,3)

Queue 1: (1,17,1)

Queue 2:

Simulation Time: 19

Waiting Clients: (2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 20

Waiting Clients: (2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 21

Waiting Clients: (2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 22

Waiting Clients: (2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 23

Waiting Clients: (2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 24

Waiting Clients: (2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 25

Waiting Clients: (2,26,2)(3,29,3)

Queue 1:

Queue 2:

Simulation Time: 26

Waiting Clients: (3,29,3)

Queue 1: (2,26,2)

Queue 2:

Simulation Time: 27

Waiting Clients: (3,29,3)

Queue 1: (2,26,1)

Queue 2:

Simulation Time: 28

Waiting Clients: (3,29,3)

Queue 1:

Queue 2:

Simulation Time: 29

Waiting Clients:

Queue 1: (3,29,3)

Queue 2:

Simulation Time: 30

Waiting Clients:

Queue 1: (3,29,2)

Queue 2:

Simulation Time: 31

Waiting Clients:

Queue 1: (3,29,1)

Queue 2:

Simulation Time: 32

Waiting Clients:

Queue 1:

Queue 2:

Queue 0: Simulation finished

Queue 1: Simulation finished

Simulation finished

1. Concluzii

Simulatorul de cozi este construit pe baza arhitecturii model-view-controller, care separă responsabilitățile de gestionare a datelor, afișare și control. Această separare face ca codul să fie mai ușor de întreținut și mai modular, permițând ca părțile componente să fie dezvoltate și testate separat.

În plus, prin utilizarea unor mecanisme de validare a datelor, cum ar fi blocuri try-catch și metodele de validare implementate în clasa Model, utilizatorii sunt preveniți să introducă date greșite și să se confrunte cu erori în timpul simulării.

În concluzie, acest proiect a reprezentat o adevărată provocare, dar a adus cu el și o serie de beneficii în dezvoltarea competențelor mele.

1. Bibliografie

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

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

<https://www.geeksforgeeks.org/>