Facultatea de Automatică și Calculatoare

**Documentație**

**Queue Management System**

-Tema II-

Nume student: Ivan Alexandra-Ștefania

Grupa: 30227

**CUPRINS**

[1. Obiectivul temei 3](file:///C:\Users\Stefi\Downloads\PT2023_Template_Documentatie_RO.doc#_Toc95297885)

* 1. Obiectiv principal …………………………………………...3
  2. Obiective secundare ………………………………………...3

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare](file:///C:\Users\Stefi\Downloads\PT2023_Template_Documentatie_RO.doc#_Toc95297886) 3

[3. Proiectare](file:///C:\Users\Stefi\Downloads\PT2023_Template_Documentatie_RO.doc#_Toc95297887) 5

3.1 Diagrama UML de clase ……………………………………...5

3.2 Structuri de date folosite ……………………………………...6

[4. Implementare](file:///C:\Users\Stefi\Downloads\PT2023_Template_Documentatie_RO.doc#_Toc95297888) 7

4.1 Clasa Task…. …………………………………………………8

4.2 Clasa Server… ………………………………………………...9

4.3 Clasa SimulationManager………………..……………………9

4.4 Clasa SimulationFrame…..…………………………..………12

4.5 Clasa Scheduler. …………………………………..…………10

4.6 Clasa Controller……………………………………………...10

4.7 Interfața Strategy……………………………………………..10

[5. Rezultate 1](file:///C:\Users\Stefi\Downloads\PT2023_Template_Documentatie_RO.doc#_Toc95297889)1

[6. Concluzii](file:///C:\Users\Stefi\Downloads\PT2023_Template_Documentatie_RO.doc#_Toc95297890) 12

[7. Bibliografie 12](file:///C:\Users\Stefi\Downloads\PT2023_Template_Documentatie_RO.doc#_Toc95297891)

1. **Obiectivul temei** 
   1. **Obiectivul principal** al temei constă în implementarea unei aplicații care își

propune să analizeze sisteme bazate pe cozi, simulând prin intermediul unei interfețe grafice, o serie de N clienți care sosesc, se așează la Q cozi, sunt serviți și părăsesc în cele din urmă coada. Pe baza acestor date, se calculează timpul mediu de așteptare, timpul mediu de servire a clienților și ora de vârf.

**2.2. Obiectivele secundare** se referă la analiza separată a mai multor aspecte din cadrul problemei:

* Identificarea cerințelor, în urma unui studiu detaliat al aplicației;
* Proiectarea simulării;
* Implementarea simulării;
* Testarea simulării;

1. **Analiza problemei**

Aceast capitol reunește mai multe aspecte care trebuie luate în considerare

vederea unei implementări corecte și eficiente a aplicației. Printre cele mai importante se numără: **funcționalitățile** programului, **constrângerile** și **scenariile de testare**, în vederea stabilirii corectitudinii execuției.

**Cerințele funcționale:**

* Aplicația de simulare ar trebui să permită utilizatorului să configureze aplicația;
* Aplicația de simulare ar trebui să permită utilizatorului să înceapă simularea;
* Aplicația de simulare ar trebui să afișeze evoluția în timp real a cozilor;
* Aplicația de simulare ar trebui să permită utilizatorului să introducă datele necesare, precum: numărul de clienți, numărul de cozi, timpul de simulare, timpul minim/maxim de sosire, timpul minim/maxim de servire;
* Aplicația de simulare ar trebui să permită utilizatorului să selecteze strategia dorită în gestionarea cozilor;
* Aplicația de simulare ar trebui să permită utilizatorului să acceseze informații referitoare la timpul mediu de așteptare, timpul mediu de servire.

**Cerințe non-funcționale:**

* Aplicația de simulare ar trebui să fie intuitivă și ușor de folosit de către utilizator;
* Aplicația de simulare ar trebui să poată procesa un număr mare de date;
* Aplicația de simulare ar trebui să valideze datele introduse de utilizator;
* Aplicația de simulare ar trebui să fie disponibilă în mod constant utilizatorilor pentru a simula diverse cazuri.

**Caz de utilizare:** simulare de configurare

**Actor principal:** utilizator

**Scenariu de succes:**

1. Utilizatorul introduce valorile pentru: numărul de clienți,număr de cozi, interval de simulare, minim și ora maximă de sosire și minim și maxim timp de serviciu.

2. Utilizatorul face clic pe butonul de validare a datelor introduse.

3. Aplicația validează datele și afișează un mesaj informând utilizatorul să înceapă simularea.

**Scenariu alternativ:** valori nevalide pentru parametrii de configurare

- Utilizatorul introduce valori nevalide pentru configurarea aplicației

parametrii

- Aplicația afișează un mesaj de eroare și solicită utilizator să introducă valori valide

- Scenariul revine la pasul 1.

1. **Proiectare**
   1. **Diagrama UML de clase**

Între clasele implementate se poate observa prezența unor diferite relații. **O relaţie** modelează o conexiune semantică sau o interacţiune între elementele pe care le conectează.

* **Relația de compoziție** între: clasa Controller și SimulationFrame, clasa SimulationManager și SimulationFrame, clasa SimulationManager și Scheduler, clasa Scheduler și Strategy etc.;
* **Relația de dependență** între clasa Controller și SimulationManager;

**UML Diagram


Description automatically generated**

* 1. **Structuri de date folosite**

În implementarea proiectului meu, am folosit structuri de date pentru stocarea

și manipularea clienților din cozi și a serverelor:

* **BlockingQueue** este o implementare sincronizată a unei cozi. Acest lucru înseamnă că BlockingQueue poate fi utilizată pentru a sincroniza și coordona activitățile între procese sau thread-uri, prin blocarea thread-urilor care încearcă să adauge sau să elimine elemente din coadă atunci când coada este goală sau plină. Acest tip de coadă este numită „blocking” deoarece operațiile de adăugare sau eliminare din coadă sunt blocate atunci când coada este plină sau goală.De aceea, am folosit această structură de date pentru stocarea clienților.

Text

Description automatically generated

Clasa **AtomicInteger** oferă o variabilă int care poate fi citită și scrisă atomic.

Operațiile atomice sunt efectuate într-o singură unitate de sarcină fără interferențe din partea altor operațiuni. Operațiunile atomice sunt necesare în mediul cu mai multe fire de execuție pentru a evita inconsecvența datelor.

* **List** extinde interfața Collection<> și permite lucrul cu o colecție ordonată. O implementare comună a interfeței List<> este clasa ArrayList<>, care utilizează un vector dinamic pentru a stoca elementele.



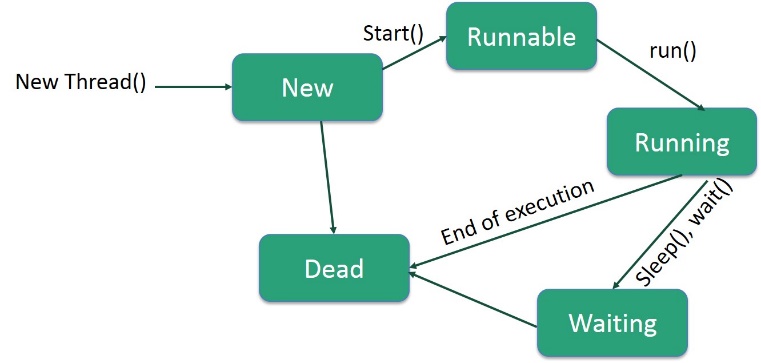


1. **Implementare**

În implementarea acestei aplicații, utilizarea thread-urilor are un rol important,

pentru procesarea cererilor de către servere. Fiecare server este implementat printr-o instanță a clasei Server, care implementează interfața Runnable și este rulată pe un thread separat. Această abordare permite procesarea simultană a mai multor cereri, în timp ce fiecare server funcționează independent de celelalte.

Un **Thread** trece prin diferite etape în ciclul său de viață. De exemplu, un thread se naște, începe, rulează și apoi moare.



**Graphical user interface, application

Description automatically generated**

**Clasa Task** reprezintă un obiect de tip Task (client). Acesta este utilizat pentru a modela sarcinile care sunt procesate de către aplicația de gestionare a cozilor.

Membrii clasei includ câmpuri pentru id-ul sarcinii, timpul de sosire, timpul de procesare. De asemenea, clasa are o variabilă statică order, care se utilizează pentru a genera un id unic pentru fiecare sarcină creată.

**Metode:**

1. Constructorul Task(), care initializează valorile câmpurilor obiectului Task.
2. Metodele setServiceTime(int serviceTime) și setArrivalTime(int arrivalTime), care sunt utilizate pentru a seta valorile timpului de procesare și al timpului de sosire al sarcinii.
3. Metoda getServiceTime(), care este utilizată pentru a obține timpul de procesare al sarcinii.
4. Metoda getArrivalTime(), care este utilizată pentru a obține timpul de sosire al sarcinii.
5. Metoda toString(), care este utilizată pentru a afișa informațiile despre sarcină, inclusiv id-ul, timpul de sosire și timpul de procesare.

**Clasa Server** reprezintă un server care procesează task-uri. Are o coadă de task-uri de tip BlockingQueue și un AtomicInteger waitingPeriod care reține timpul total de așteptare al task-urilor din coadă.

Metode:

1. Metoda addTask() adaugă un nou task în coadă și actualizează waitingPeriod cu durata de servire a noului task.
2. Metoda run() este metoda care rulează în thread-ul serverului. În interiorul acesteia, serverul ia primul task din coadă și îl procesează prin așteptarea pentru durata de procesare a task-ului și decrementarea waitingPeriod cu aceeași valoare. După ce task-ul este procesat, acesta este eliminat din coadă și se ia următorul task din coadă, dacă există.
3. Metoda toString() afișează informații despre server.

**Clasa SimulationManager** este o clasă ce coordonează simularea și interacțiunea între obiectele Task și Server. Această clasă are rolul de a iniția simularea și de a interacționa cu utilizatorul prin intermediul interfeței de tip consolă.

Metode:

1. Metoda run() - implementează bucla principală a simulării și coordonează interacțiunea între cele două obiecte menționate anterior.
2. Metoda generateNRandomTasks(int n), care generează un număr dat de obiecte Task cu proprietăți aleatoare și le adaugă în coada de așteptare.

Această clasă este un element important în arhitectura simulării și are rolul de a asigura o interacțiune eficientă și coerentă între diferitele componente ale sistemului simulat.

**Clasa SimulationFrame** reprezintă o interfață grafică de utilizator (GUI) pentru simularea desfășurată în cadrul aplicației. Interfața grafică generată conține mai multe câmpuri de text, unde utilizatorul introduce datele dorite.

Clasa conține componente precum JTextArea pentru afișarea rezultatelor. Clasa JTextArea oferă o componentă care afișează mai multe linii de text și, opțional, permite utilizatorului să editeze textul. Dacă trebuie să obțineți o singură linie de intrare de la utilizator, ar trebui să utilizați un JTextField.

Graphical user interface, application

Description automatically generated

**Clasa Scheduler** este responsabilă de gestionarea serverelor și a strategiei de

distribuire a sarcinilor la servere. Constructorul clasei primește ca argumente maxNoServers și maxTasksPerServer și creează un număr de servere egale cu maxNoServers, apoi pornește un fir de execuție pentru fiecare server. Clasa are o metodă **changeStrategy** care permite schimbarea strategiei de distribuire a sarcinilor și o metodă **dispatchTask** care utilizează strategia pentru a distribui sarcinile la servere.

**Clasa Controller** este o clasă care acționează ca intermediar între interfața grafică și logica aplicației. Scopul principal al clasei Controller este de a separa interfața utilizator de logica aplicației și de a face ca aplicația să fie ușor de întreținut și de extins.

**Strategy** definește o interfață pentru implementarea unor strategii de alocare a task-urilor la servere. Clasa conține două clase încorporate, ConcreteStrategyTime și ShortestQueueStrategy, care implementează metoda addTask în funcție de politica de selecție specificată.

Metoda **addTask()** este utilizată pentru a aloca un task la un server specific, în funcție de strategia specificată. Clasa conține și o enumerație SelectionPolicy pentru a specifica strategia dorită, care poate fi **SHORTEST\_QUEUE** sau **SHORTEST\_TIME. Clasa ConcreteStrategyTime** implementează strategia SHORTEST\_TIME și selectează serverul cu cel mai mic timp de așteptare pentru a atribui task-ul.

**5. Rezultate**

În vederea obținerii unor rezultate , utilizatorul introduce datele aferente

numărului de clienți, numărului de cozi, timpului minim/maxim de sosire, timpului minim/maxim de servire și intervalului de simulare.

La apăsarea butonului „Validate Data Input”, în componenta JTextArea este

generată lista de clienți si conținutul cozilor la fiecare moment de timp.

De asemenea, în cadrul metodei run() din clasa SimulationManager, outputul din console este redirecționat către fișierul „log.txt”.

Text

Description automatically generated

Toate afișările standard făcute cu System.out.println() vor fi redirecționate către

acest fișier de acum încolo.

Graphical user interface, text

Description automatically generated

1. **Concluzii**

În concluzie, proiectul a avut ca scop realizarea unei aplicații care să simuleze

un sistem de manageriere a cozilor, în funcție de datele introduse de către utilizator prin intermediul interfeței grafice. În implementarea acestei aplicații, am avut ocazia să înțeleg mai bine funcționalitatea thread-urilor și importanța acestora în cadrul problemelor associate cu sincronizarea.

**7. Bibliografie**

<https://www.geeksforgeeks.org/arrayblockingqueue-class-in-java/>

<https://www.tutorialspoint.com/java/java_multithreading.htm>

https://www.digitalocean.com/community/tutorials/atomicinteger-java

Top of Form