DOCUMENTAȚIE

TEMA_2

Student: Pop Raul-George

Grupa: 30225

Cuprins

1.Obiectivul temei

Obiectivul temei este reprezentat de implementarea și proiectarea unui sistem de gestionare a unui sistem de cozi cu o interfață grafică dedicată menită să ușureze și faciliteze introducerea de date de către utilizator cât și alegerea de distribuție in care pot fi împărțite datele introduse și vizualizarea rezultatului final. Sistemul este capabil să distribuie un număr dat de clienți într-un număr dat de cozi fie în funcție de dimensiunea cozilor sau în funcție de timpul de astepatare. Sistemul de gestiune trebuie să fie ușor de folosit și ușor de înțeles de orice utilizator indiferent de nivelul de pregătire în domeniul tehnologiei.

Obiectivele secundare considerate în cadrul implementării obiectivului principal sunt:

1. Separarea problemei în structuri ușor de implementat: realizarea de clase și pachete care vor lua parte la formarea sistemului de calcul;
2. Realizarea interfeței grafice: crearea unei interfețe care va fi folosită de către utilizator pentru introducerea și prelucrarea datelor;
3. Implementarea unei metode de conversie și stocare a datelor introduse de utilizator;
4. Sincronizarea thread-urilor reprezentate prin cozi;
5. Testarea sistemului pentru anumite situații.

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

"Multithreading" înseamnă capacitatea unui program de a executa mai multe secvențe de cod în acelaşi timp. O astfel de secvență de cod se numește fir de execuție sau thread. Limbajul Java suportă multithreading prin clase disponibile în pachetul java.lang. În acest pachet există 2 clase Thread şi ThreadGroup, şi interfaţa Runnable. Clasa Thread şi interfaţa Runnable oferă suport pentru lucrul cu thread-uri ca entităţi separate, iar clasa ThreadGroup pentru crearea unor grupuri de thread-uri în vederea tratării acestora într-un mod unitar.

Există 2 metode pentru crearea unui fir de execuţie: se creează o clasă derivată din clasa Thread, sau se creează o clasă care implementeză interfaţa Runnable.

Crearea unui fir de execuţie prin extinderea clasei Thread

Se urmează etapele:

- se creează o clasă derivată din clasa Thread
- se suprascrie metoda public void run() moştenită din clasa Thread
- se instanţiază un obiect thread folosind new
- se porneşte thread-ul instanţiat, prin apelul metodei start() moştenită din clasa Thread.

Apelul acestei metode face ca maşina virtuală Java să creeze contextul de program necesar unui thread după care să apeleze metoda run().

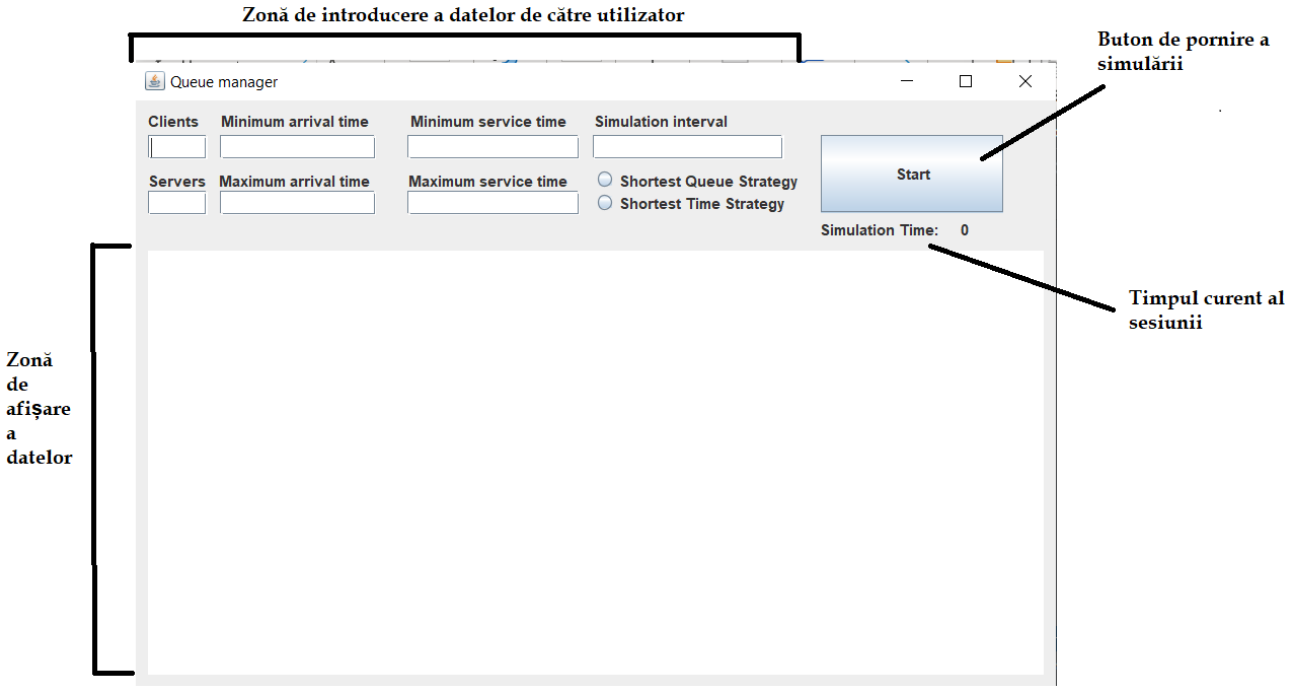
Sistemul de gestiune primește ca date de intrare numărul de cozi, intervalul de simulare, numărul de clienți, timpul minim de sosire al clienților, timpul maxim de sosire al clienților, timpul minim de așteptare al clienților, timpul maxim de așteptare al clienților și posibilitatea de a alege în funcție de ce parametru (timp de așteptare sau dimensiunea cozii) dorim să implementăm distribuția clienților în cadrul cozilor.

Fiecare coadă în care se află un client reprezintă un server de gestionat eficient al clienților. Având în vedere numărul de cozi ca intrare, putem translata numărul de servere, iar coada este folosită doar ca zonă de așteptare pentru clienții care așteaptă să fie serviți. Fiecare server va funcționa individual, astfel încât să poată fi procesați un număr de clienți mai mare sau egal cu numărul de cozi. Utilizatorul poate urmări și în timp real starea fiecărei cozi și jurnalul de activitate al aplicației (*log.txt*). Cu aceste informații putem deduce că avem nevoie de o clasă care să formeze clientul ca unitate, o clasă care modelează serverul și conține o coadă de obiecte client, o clasă

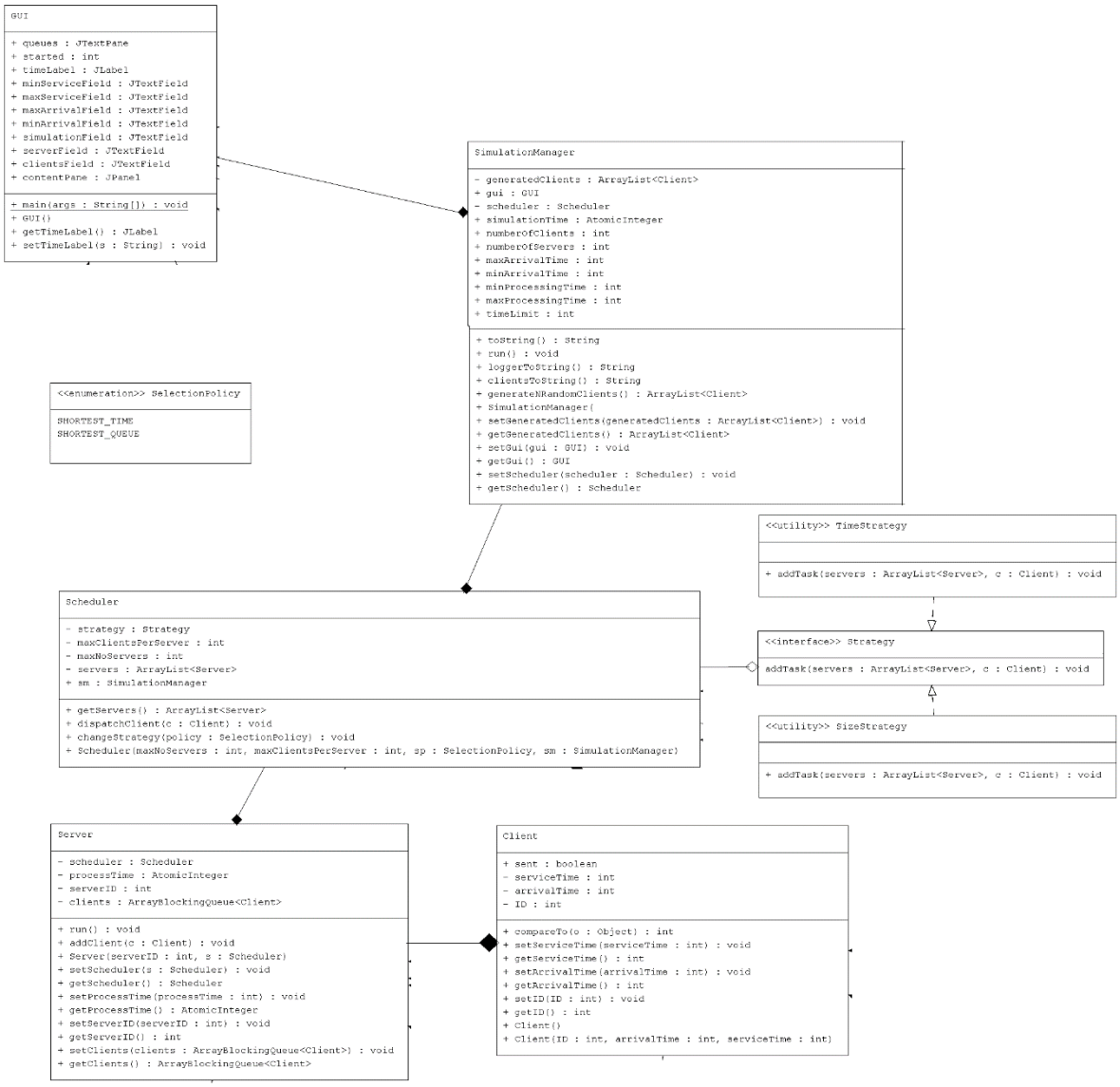
care va procesa datele și gestiona cozile cât și o clasă în care va fi implementată interfața grafică a aplicației.

Există mai multe scenarii de utilizare a aplicației:

- Intervalul de venire al clienților este mai mic decât timpul mediu de procesare al fiecăruia, caz in care cozile se vor supra-aglomera din cauza faptului ca serverul nu poate procesa clienții in ritmul in care ei ajung in coada. Acest scenariu este dependent de numărul de cozi din sistem. Este un scenariu mai apropiat de realitate;
- Intervalul de venire al clienților este mai mare decât timpul mediu de procesare al fiecăruia, caz in care cozile vor fi in majoritatea timpului goale, în funcție de diferența dintre cele doua date de intrare . Acest scenariu ar putea fi apropiat de scenariul unor servere care gestionează diferite încercări de citire sau trimitere a datelor;
- Distribuirea clienților este făcută în funcție de timpul de așteptare;
- Distribuția clienților este făcută în funcție de dimensiunea cozii.



3. Proiectare



Aplicația conține 7 clase principale și o interfață, care va fi implementată de alte 2 clase. În fiecare clasă se respectă principiul încapsulării. Pentru a simula activitatea sistemului descris am folosit o lista de servere. Clasa Server care are în componența sa o coada de clienți. Fiecare server reprezintă un thread separat care operează pe o coadă de clienți proprie și diferită. Serverul se ocupă de operația de adăugare și eliminare a clienților din coadă. Modelarea cozii este realizată folosind colecția *ArrayBlockingQueue* care oferă suport automat pentru sincronizarea operațiile de introducere si eliminare în mediul concurent al thread-urilor. Distribuirea clienților este realizată în funcție de timpul de așteptare acumulat al fiecărei cozi sau în funcție de dimensiunea

cozii. Timpul de așteptare se calculează și stochează în fiecare instanță a obiectului Server înainte de fiecare introducere în coada. În ceea ce privește interfața grafica, aplicația prezintă în partea superioară o serie de căsuțe de text în cadrul cărora utilizatorul are posibilitatea de a introduce datele de simulare, cât și un buton de *START* care va porni simularea propriu-zisă, bazându-se pe datele introduse de utilizator în căsuțele menționate anterior. Sub butonul de start poate fi urmărit și timpul curent al simulării.

Pe lângă acestea evoluția sistemului de gestiune se poate observa și în fișierul (*log.txt)* de activitate care afișează fiecare operație de introducere si procesare din sistem. Acest logger salvează toate afișările astfel că poate fi urmărita evoluția sistemului de la început pana la sfârșitul sesiunii.

4. Implementare

- Clasa *Client*:

  Are 3 variabile instanță care descriu ID-ul, timpul de sosire și timpul de procesare a fiecărui client din polinomul din care face parte. Clasa Client implementează interfața *Comparable* pentru a fi ușor de ordonat în funcție de timpul de sosire a fiecărui client, prin intermediul metodei *compareTo(Object o).*

- Clasa *Server*:

  Are 3 variabile instanță. Variabila *clients* va memora coada de clienți pe care urmează să-i servească; variabila *serverID:* reprezintă un ID unic al fiecărui server; variabila *processTime*: va reprezenta o instanță a unui obiect de tip *AtomicInteger* de care ne vom folosi în cadrul implementării unor metode și este folosit pentru a contoriza timpul curent al simulării în mod concurent. Clasa extinde clasa Thread deoarece fiecare server reprezintă un alt fir de execuție. Astfel, în cadrul clasei vom avea metoda *run()* care va implementa practic ”servirea” clienților din coada fiecărui server.

- Clasa *SimulationManager*:

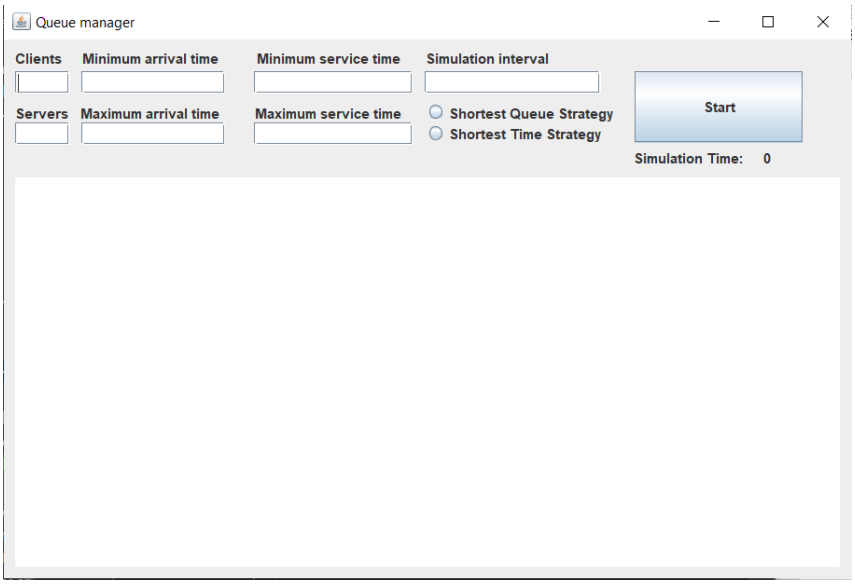
  Această clasă are 11 variabile instanță fiecare având un rol crucial în implementarea clasei, astfel că variabilele: *timeLimit, maxProcessingTime, minProcessingTime, minArrivalTime, maxArrivalTime, numberofServers,* și *numberOfClients* vor memora datele de intrare și inițializare a simulării. Acestea vor fi citite din interfața grafică și introduse de către utilizator. Variabila *simulationTime* este de tipul *AtomicInteger* și are rolul de a contoriza timpul curent al sesiunii în mod concurent. Variabila *scheduler* va avea rolul de a distribui clienții spre cozile potrivite. Variabila *GUI* reprezintă o instanță a interfeței grafice. Variabila *generatedClients* va memora clienții realizați aleator.

  - Metoda *generateNRandomClients* este metoda care va genera un număr *numberOfClients* de clienți pe care ulterior îi va atribui spre câte o coadă.
  - Metoda *loggerToString* reprezintă metoda prin care vom genera datele care trebuie introduse în fișierul *log.txt*, fișierul de activitate al aplicației.
  - Metoda *toString* reprezintă o metodă prin care vom genera datele care vor fi afișate ulterior în interfața grafică.

  Fiind o clasă care moștenește clasa *Thread*, clasa *SimulationManager* va avea și o metoda *run()* care va realiza și dirija majoritatea acțiunilor necesare în implementarea aplicației.

- Clasa *GUI*

  Implementează interfața grafică și face legătura dintre utilizator și simularea efectivă a cozilor, legătură implementată prin intermediul căsuțelor de text și a butoanelor.



- Clasa *Scheduler*

  Realizează distribuirea clienților spre cozi în funcție de strategia aleasă, conținând și o metodă de schimbare a strategiei de distribuție.

- Interfața *Strategy*

  Prezintă o singură metodă abstractă, *addTask* care va adăuga un client la o listă. Astfel că, prin folosirea principiului moștenirii, clasele *TimeStrategy* și *SizeStrategy* implementează această interfață și distribuie clienții în funcție de strategia aleasă.

Având în vedere că în cadrul implementării programului s-a folosit tehnica de multithreading, în cadrul claselor a fost necesară folosirea anumitor tehnici de sincronizare a firelor de execuție concurente. Una dintre aceste metode este folosirea clasei *ArrayBlockingQueue*, care este o este o coadă de blocare delimitată susținută de o matrice. Prin delimitat, referim că dimensiunea cozii este fixă. Odată creată, capacitatea nu poate fi modificată. Încercările de a pune un element într-o coadă plină vor duce la blocarea operației. În mod similar, încercările de a prelua un element dintr-o coadă goală vor fi, de asemenea, blocate. Limitarea ArrayBlockingQueue poate fi atinsă inițial ocolind capacitatea ca parametru din constructorul ArrayBlockingQueue. Această coadă comandă elementele FIFO (primul-intrat-primul-out). Adică capul acestei cozi este cel mai vechi element dintre elementele prezente în această coadă. Finalul acestei cozi este cel mai nou

element dintre elementele acestei cozi. Elementele nou introduse sunt întotdeauna inserate la coada cozii, iar operațiunile de recuperare a cozii obțin elemente din capul cozii.

O altă metodă este folosirea cuvântului cheie *syncronized*. Când folosim un bloc sincronizat, Java folosește intern un monitor, cunoscut și sub denumirea de blocare a monitorului sau blocare intrinsecă, pentru a asigura sincronizarea. Aceste monitoare sunt legate de un obiect; prin urmare, toate blocurile sincronizate ale aceluiași obiect pot avea un singur fir care le execută în același timp, lucru care duce la folosirea in mod serial a resurselor comune dintre thread-uri (de exemplu: afișarea în panoul cu rezultate din cadrul interfeței grafice etc).

De asemenea, pentru a asigura accesarea concurentă și corectă a variabilelor comune tuturor thread-urilor, vom folosi clasa *AtomicInteger*. Aceasta este utilizată în aplicațiile cu mai multe fire de execuție în care acestea trebuie să acceseze un număr întreg partajat. Este o parte a API concurente în Java și, prin urmare, aparține pachetului concurent. *AtomicInteger* conține o valoare întreagă care este actualizată atomic (o vom folosi de exemplu în urmărirea, incrementarea cât și decrementarea timpurilor de simulare din cadrul proiectului).

5. Rezultate

Execuția acestui program reprezintă o analiză în detaliu a evoluției și dezvoltării unui sistem format dintr-o serie de clienți și o serie de cozi capabile să susțină un număr de clienți și să-i proceseze. Această analiză a sistemului este posibilă atât datorită interfeței grafice, în panoul căreia sunt afișate în timp real datele corespunzătoare simulării curente, cât și datorită implementării fișierului de activitate (*log.txt)* in interiorul căruia este salvată în timp real evoluția și dezvoltarea procesului de gestionare a cozilor și a clienților. Mai mult, la finalul simulării, atât în cadrul interfeței grafice cât și a fișierului de activitate, va fi afișată o medie a timpului de așteptare, o medie a timpului de procesare cât și ”ora” de vârf din cadrul simulării. Ca rezultat final, aplicația reprezintă o modalitate ușoară de folosit și interpretat care oferă informații în timp real despre sistemul pe care îl gestionează.

6. Concluzii

În concluzie, obținem un sistem de gestiune ușor de folosit și bine organizat care se poate dovedi foarte folositor în cadrul unei situații din viața reală, cozile de așteptare fiind un fenomen întâlnit la orice pas în viața cotidiană, fiind cea mai ușoară metodă de ordonare a clienților și activităților în general. În urma implementării proiectului au fost aprofundate cunoștințele de implementare a unei interfețe grafice, de înțelegere de funcționare și folosire a thread-urilor și sincronizarea acestora cât și cunoștințe de organizare a claselor și de logică generală. Ca și dezvoltări ulterioare putem aminti: legarea unei baze de date la întreg proiectul astfel fiind posibilă memorarea datelor și a activității într-un mod mult mai organizat, implementarea unui buton de resetare a simulării sau alte strategii de distribuire a clienților.

7. Bibliografie

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

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

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

http://inf.ucv.ro/documents/tudori/laborator8_53.pdf

https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#:~:text=A%20thread%20is%20a%20thread,to%20threads%20with%20lower%20priority.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html#:~:text=An%20AtomicInteger%20is%20used%20in,deal%20with%20numerically%2Dbased%20classes.