DOCUMENTAȚIE

TEMA 2

Pașcan Daniela Maria

Grupa 9

Cuprins

1. Obiectivul temei....................................................................................................................3

2. Analiza problemei, modelare, cazuri de utilizare..................................................................3

3. Proiectare...............................................................................................................................5

4. Implementare.........................................................................................................................8

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

6. Concluzii.............................................................................................................................14

7. Bibliografie.........................................................................................................................14

**1.Obiectivul temei**

Dorim să implementăm o aplicație care are ca scop analiza sistemelor bazate pe cozi prin simularea unui număr de N clienți care sosesc într-un magazin, sunt așezați la Q cozi pe baza unei strategii, sunt serviți după care părăsesc magazinul. Vom vrea să calculăm și timpul mediu de așteptare, timpul mediu de servire și ora de vârf.

1. *Analiza problemei, modelare, scenariu, cazuri de utilizare-* aici se vor reprezenta cerițele funcționale și cele non-funcționale, precum și use-case-urile

2*. Proiectare-* în această secțiune se va discuta despre diagrama UML a proiectului și despre organizarea acestuia în pachete și clase

1. *Implementare-* aici se va descrie în detalui fiecare clasă și metodele importante ale acesteia
2. *Rezultate-* la final vom discuta despre datele obținute ăn urma simulării

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

**2.1. Analiza problemei**

*Cerințe funcționale*

* Aplicația de simulare trebuie să permit utilizatorului să introducă datele pe care vrea să aplice simularea
* Aplicația trebuie să permit utilizatorului să pornească simularea
* Aplicatia trebuie să permit utilizatorului să vadă în timp real cum se trimit și procesează clienții la coadă

*Cerințe non-funcționale*

* Aplicația de simulare a cozilor trebuie să fie intuitivă și simplu de utilizat de către utilizator
* Aplicația trebuie să aibă o bună performanță ( timpul de răspuns să fie minim, utilizarea sa să meargă linear)
* Aplicația de simulare trebuie sa aibă scalabilitate ( să poată lucra cu un volum mai mare de informație și să permită mărirea sau extinderea sa)
* Aplicația de simulare trebuie să țină cont de integritatea datelor, să afișeze rezultate corecte, complete și consistente și să păstreze datele neschimbate indiferent de cat timp sunt stocate sau cat de des sunt accesate
* Aplicația trebuie să fie ușor de reparat și de întreținut
* Trebuie să se țină cont de integritatea datelor în aplicație

**2.2.Cazuri de utilizare**

Diagram

Description automatically generated

**Caz de utilizare**: Introduce datele pentru simulare

Actor primar: Utilizatorul

Scenariu de succes:

1. Utilizatorul inserează valorile pentru numărul de clienți, numărul de cozi, valoarea minimă și maximă a timpului de sosire, valoarea minimă și maximă a timpului de servire și intervalul simulării
2. Utilizatorul apasă butonul de start pentru a începe simularea
3. Aplicația de simulare validează datele introduse și pornește simularea

Secvență alternativă:

* Utilizatorul inserează valori invalide pentru parametrii de simulare
* Se va afișa un mesaj de eroare iar utilizatorul va fi rugat să introducă din nou date corecte
* Scenariul se întoarce în pasul 1

**Caz de utilizare**: Pornește simularea

Actor primar: Utilizatorul

Scenariu de succes:

1. Utilizatorul a introdus datele și a apăsat butonul de pornire al simulării
2. Utilizatorul urmărește desfășurarea simulării
3. Simularea se termină și sunt afișate în interfață cozile și clienții din acestea la fiecare timp iar la final avem timpul mediu de servire, timpul mediu de așteptare și ora de vârf

Secvență alternativă:

* Simularea nu se desfășoară corect din cauza valorilor introduse
* Scenariul se întoarce în pasul 1

**3.Proiectare**

Programarea orientată pe obiecte este o paradigma de programare care se concentrează pe obiecte ca elemente principale de construire a aplicațiilor. În POO, obiectele sunt instanțe ale claselor, care conțin date și metode asociate acestor date. POO are mai multe beneficii, cum ar fi encapsularea datelor, reutilizarea codului și abstracția.

Pentru acest proiect vom avea nevoie de cozi. Cozile sunt o structură de date folosită pentru stocarea datelor în ordinea în care au fost adăugate, urmând principiul "primul intrat, primul ieșit" (FIFO). În Java, există o clasă predefinită numită "Queue" care implementează această structură de date și care poate fi folosită pentru a gestiona cozi în aplicații. BlockingQueue este o interfață din Java care extinde interfața Queue și oferă metode suplimentare pentru gestionarea coziilor în aplicații cu mai multe fire de execuție (multi-threading). Într-o BlockingQueue, operațiile de adăugare și eliminare a elementelor pot fi blocate dacă coada este plină sau goală, respectiv. În loc să arunce o excepție, aceste metode blochează firul de execuție până când o operațiune de adăugare sau eliminare poate fi efectuată.

Vom folosi de asemenea și metoda "run" este o metodă definită în clasa Thread din Java, care conține codul care va fi executat atunci când un fir de execuție este pornit. Această metodă este utilizată pentru a defini comportamentul unui fir de execuție și trebuie suprascrisă de către programator. Atunci când metoda "run" este apelată, codul din această metodă este executat într-un fir de execuție separat, care poate fi utilizat pentru a efectua sarcini diferite simultan în aplicație.

**3.1 Modelare**

**Designul sistemului**

Text

Description automatically generated

**3.2 Organizarea în pachete**

Diagram

Description automatically generated

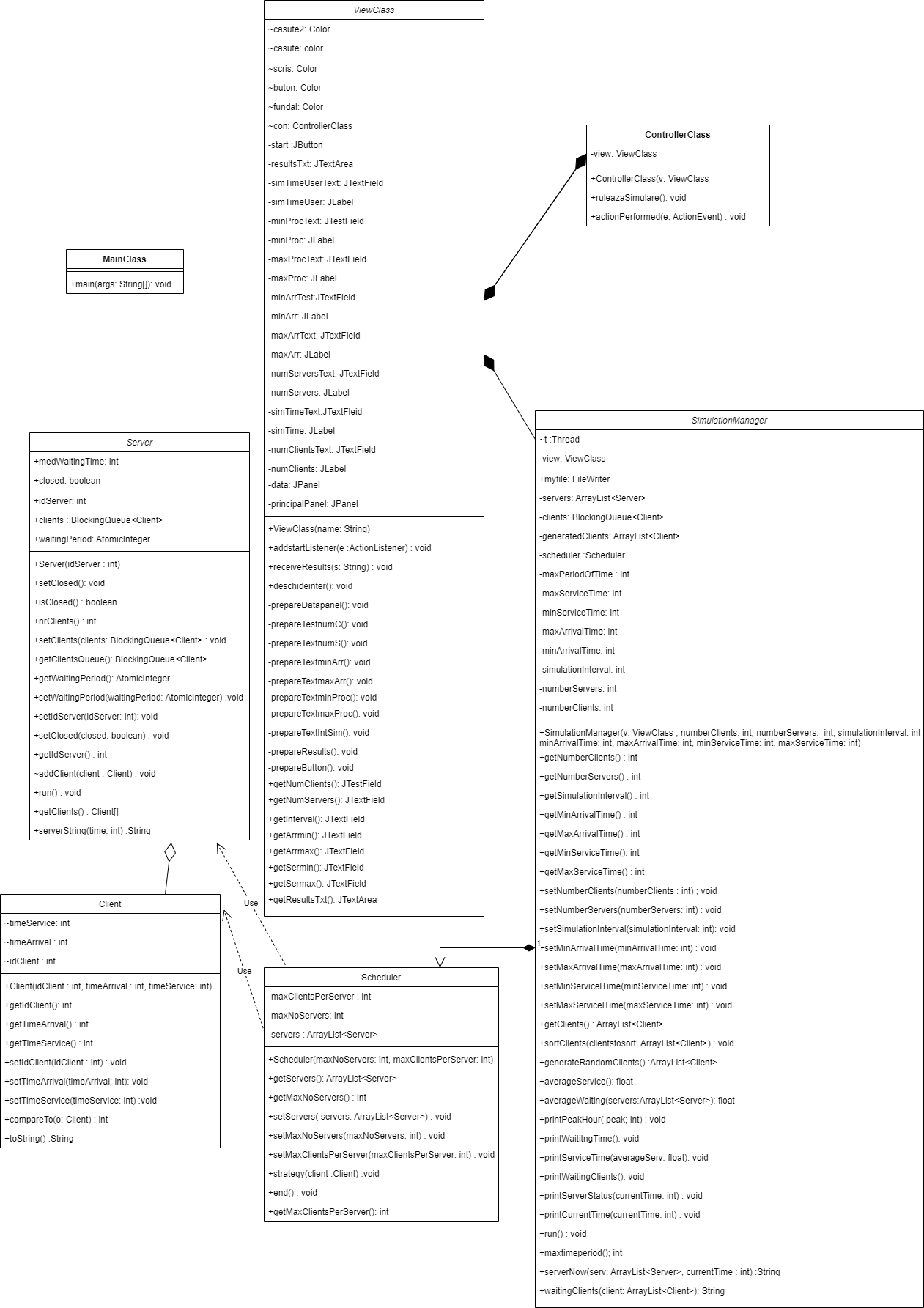
Pachetul pentru Model conține toate clasele ce modelează aplicația de simulare. Pachetul pentru View conține codul pentru interfața cu utilizatorul iar pachetul Controller leagă Modelul cu View-ul.

**3.3. Organizarea pe clase**

Diagram

Description automatically generated

**3.4. Diagrama UML de clase**



**4.Implementare**

***Clasa Client***

Această clasă reprezintă obiectul "Client" din sistemul de cozi pe care dorim să îl simulăm. În mod specific, această clasă are trei variabile de instanță: idClient, timeArrival și timeService, care reprezintă identificatorul clientului, momentul de timp la care acesta ajunge în coadă și timpul necesar pentru a fi servit.

Clasa implementează interfața Comparable, ceea ce înseamnă că obiectele de tip Client pot fi comparate între ele. Această funcționalitate este utilă atunci când dorim să sortăm obiectele de tip Client în ordinea în care au ajuns în coadă, adică după momentul de timp la care au sosit.

Clasa are, de asemenea, metode get și set pentru cele trei variabile de instanță, ceea ce permite accesul și modificarea valorilor acestor variabile din alte clase din program.

Metoda toString este o metodă de utilizare generală din Java, care este utilizată pentru a transforma obiectul într-un șir de caractere. În cazul acestei clase, metoda toString este suprascrisă pentru a returna un șir de caractere care conține valorile variabilelor de instanță ale obiectului Client.

***Clasa Server***

Această clasă reprezintă un server care este capabil să servească clienți. Aceasta implementează interfața Runnable, ceea ce înseamnă că obiectul poate fi executat într-un thread separat.

Clasa conține un idServer, o coadă de clienți (utilizând BlockingQueue din Java), un indicator closed pentru a indica dacă serverul este deschis sau nu, un waitingPeriod care reprezintă timpul total de așteptare a clienților în coadă și o variabilă medWaitingTime care reprezintă media timpului de așteptare pentru clienții care au fost deserviți.

Metoda addClient() adaugă un client nou în coada serverului și actualizează timpul de așteptare. Metoda run() reprezintă bucla principală a thread-ului, care încearcă să servească clienții din coada serverului. În timp ce există clienți în coadă, serverul încetinește timpul de procesare al fiecărui client cu o secundă și actualizează timpul de așteptare pentru fiecare secundă de procesare. După ce timpul de procesare al unui client ajunge la zero, serverul elimină clientul din coadă și actualizează media timpului de așteptare pentru toți clienții.

Metoda serverString() creează o reprezentare sub formă de șir de caractere a stării curente a cozi, care poate fi utilizată pentru a afișa starea serverului la un moment dat.

În metoda "run", serverul așteaptă să ia următorul client din coadă prin apelul metodei "peek()" a cozi. Dacă există un client disponibil, thread-ul doarme pentru o perioadă de timp egală cu timpul de procesare al clientului prin apelul metodei "Thread.sleep()". Apoi, timpul de servire al clientului este actualizat prin scăderea cu 1, iar perioada de așteptare a serverului este decrementată prin apelul metodei "decrementAndGet()" a obiectului "waitingPeriod". Dacă timpul de servire al clientului devine 0, acesta este eliminat din coadă și perioada medie de așteptare a serverului este actualizată prin adăugarea perioadei de așteptare curente a serverului (waitingPeriod) la medWaitingTime.

Prin utilizarea unei BlockingQueue, putem asigura că accesul la coadă este sincronizat între thread-uri și că nu există posibilitatea ca doi sau mai mulți clienți să fie serviți în același timp de același server. Acest lucru face ca simularea să fie mai precisă și mai sigură în ceea ce privește concurența.

Metoda Run()

Text

Description automatically generated

***Clasa Scheduler***

Această clasă denumită "Scheduler" este o componentă a unui sistem care are rolul de a gestiona serverele și clienții acestora. Aceasta are câteva variabile de stare și metode pentru a efectua diverse operațiuni.

Variabilele de stare includ:

- "servers", un ArrayList care conține toate serverele disponibile

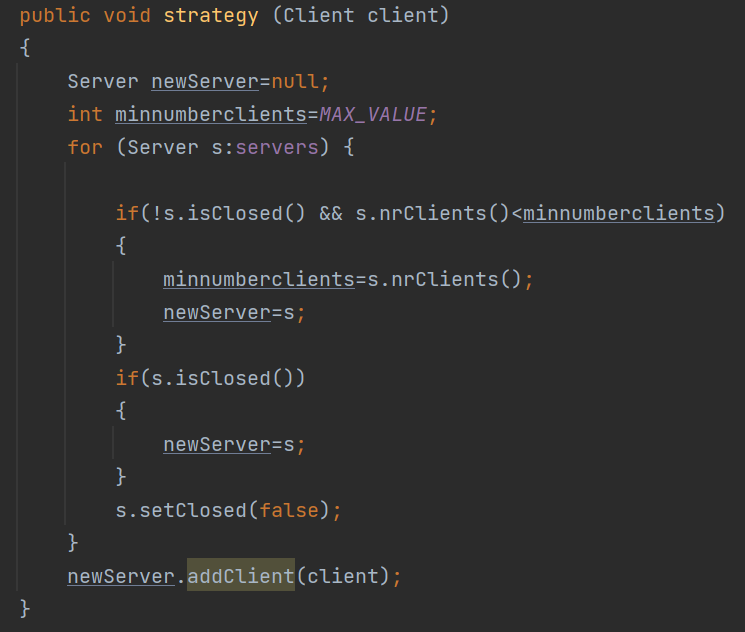
- "maxNoServers", care reprezintă numărul maxim de servere care pot fi utilizate

- "maxClientsPerServer", care reprezintă numărul maxim de clienți care pot fi serviti de un singur server

Constructorul clasei initializează lista serverelor și creează un obiect Server pentru fiecare server din listă. De asemenea, creează și pornește un fir de execuție pentru fiecare server.

Metoda "strategy()" primește un obiect Client și îl atribuie unui server disponibil în funcție de o strategie simplă, care alege serverul cu cel mai mic număr de clienți. Această metodă este utilizată în cadrul sistemului pentru a distribui clienții către servere.

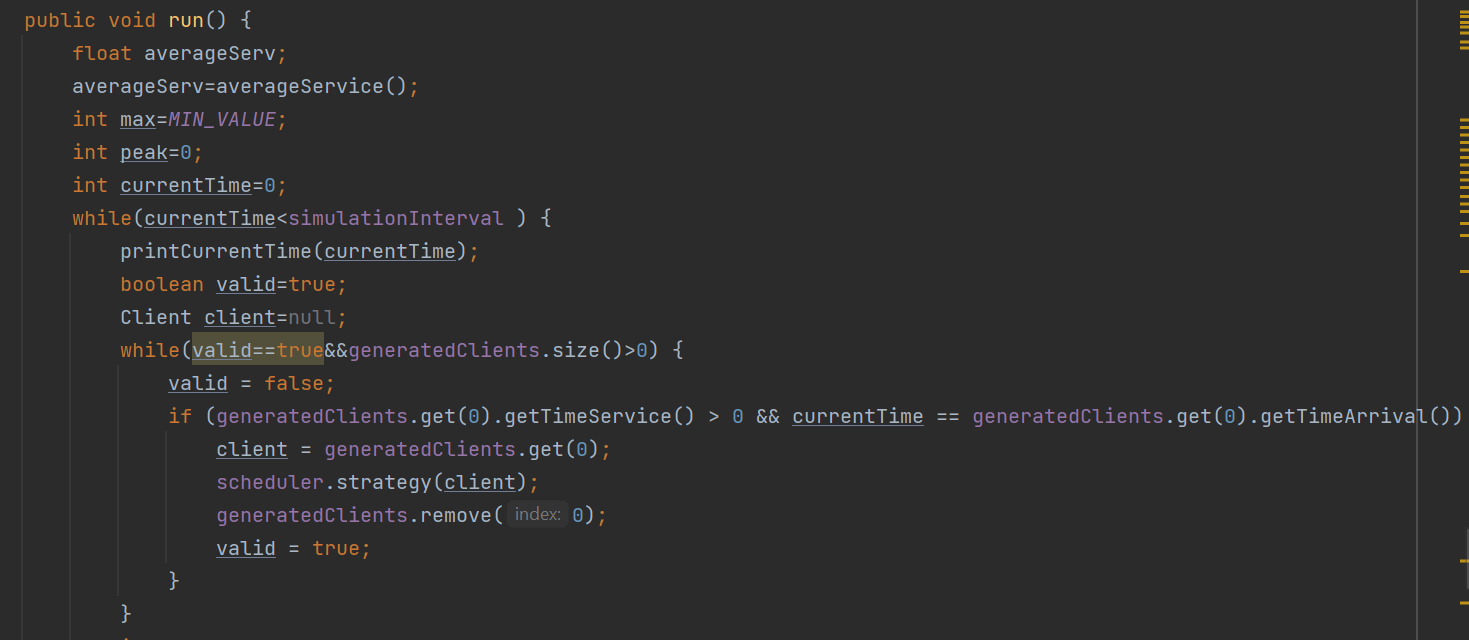
- "end()", care marchează toate serverele drept disponibile, pentru a putea fi utilizate din nou în viitor.

În general, această clasă oferă o interfață pentru a gestiona serverele și clienții, și implementează o strategie simplă pentru a distribui clienții către servere disponibile.******

***Clasa SimulationManager***

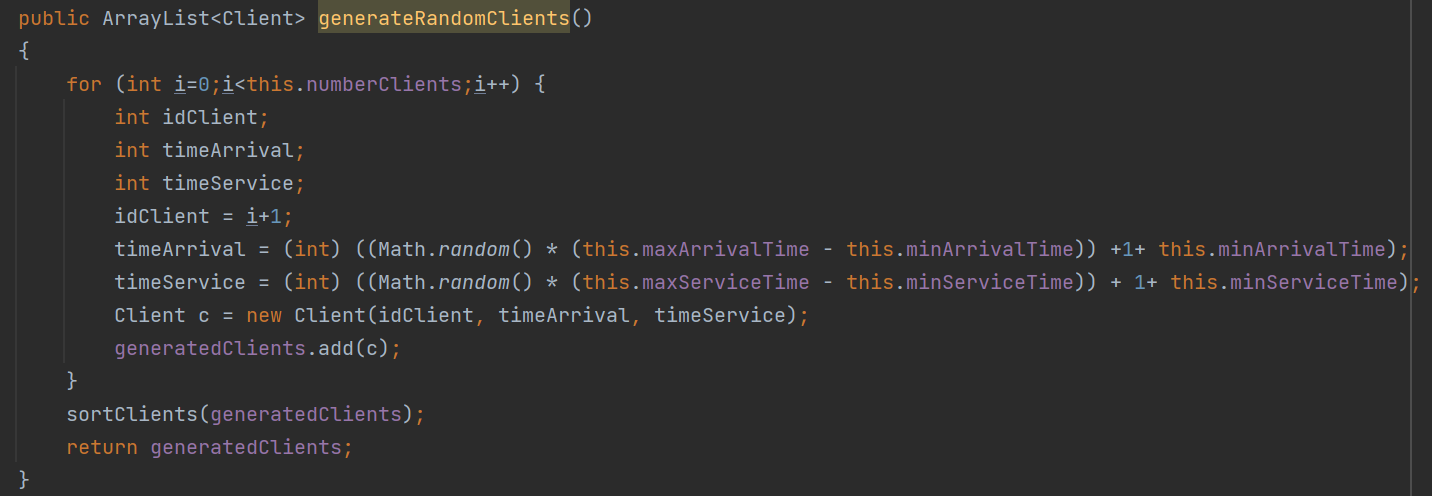
Această clasă este cea mai importantă deoarece aici se generează clienții în mod aleator, după care prin intermediul metodei Run() aceștia sunt trimiși în cozi. Aici se petrece și scrierea în fișier. Lucrul cu fișiere în Java poate fi realizat folosind clasa "File" din biblioteca standard Java, care oferă un set de metode pentru a gestiona fișiere și directoare. Aceste metode includ crearea, citirea, scrierea, redenumirea și ștergerea fișierelor.

Metoda Run() reprezintă o simulare a unui sistem de coadă. În interiorul metodei sunt inițializate variabilele pentru a înregistra informații despre timpul mediu de servire, ora de vârf a sistemului, timpul curent al simulării și altele. Apoi, în timp ce timpul curent este mai mic decât intervalul de simulare, metoda simulează adăugarea clienților la coadă, alocarea lor către un server și procesarea lor. De asemenea, în timpul simulării, se înregistrează informații despre numărul maxim de clienți aflați în sistem, timpul de așteptare al fiecărui client și starea fiecărui server. După simulare, metoda afișează informații cum ar fi timpul mediu de servire, timpul de așteptare mediu, ora de vârf a sistemului, iar apoi închide fișierul de ieșire și încheie simularea.



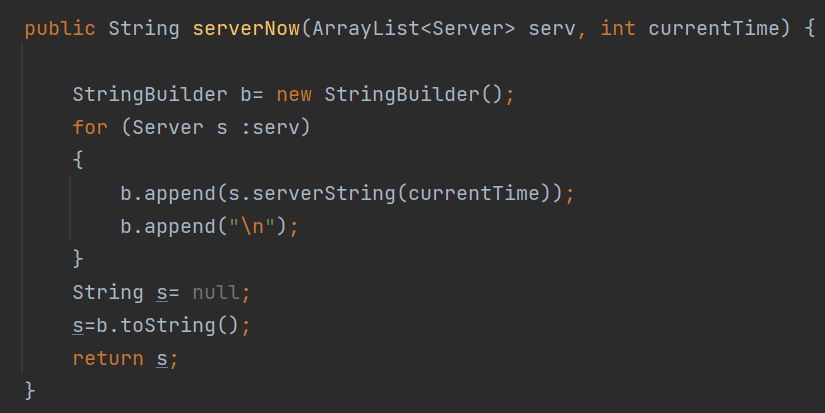


Metoda generateRandomClients() generează o listă de clienți aleatori într-un anumit interval de timp și cu un anumit timp de servire. Mai precis, metoda folosește valorile minime și maxime specificate pentru intervalele de timp de sosire și servire și apoi generează aleatoriu un timp de sosire și un timp de servire pentru fiecare client dintr-un număr total specificat anterior de clienți. Fiecare client este creat cu un ID unic și apoi adăugat la lista de clienți generați. La final, lista este sortată după timpul de sosire al clienților.



Metoda serverNow() primește o listă de obiecte de tip Server și un timp curent și returnează un șir de caractere care conține statusul fiecărui server la momentul dat.

În detaliu, metoda parcurge fiecare obiect Server din lista primită ca argument și apelează metoda serverString(currentTime) a fiecărui obiect pentru a obține un șir de caractere care reprezintă statusul serverului la timpul dat. Aceste șiruri de caractere sunt concatenate într-un șir mai mare, separate prin caracterul de linie nouă (\n) și apoi returnate.

În acest fel, această metodă poate fi folosită pentru a afișa statusul fiecărui server la un anumit moment din timpul simulării.

Metodele averageWaiting() și averageServing() sunt folosite pentru a calcula timpul mediu de așteptare și timpul mediu de servire.

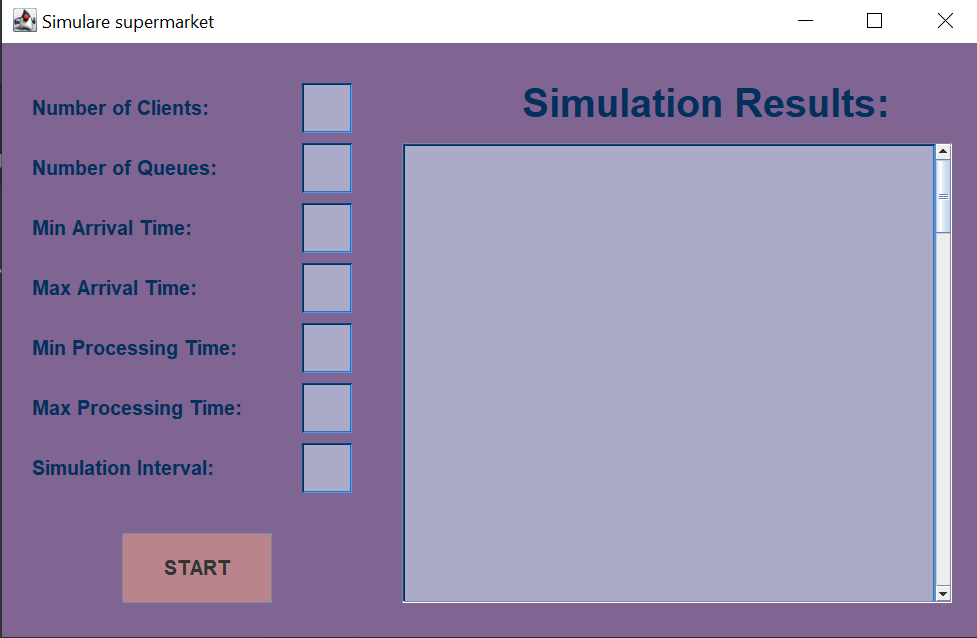
De asemenea, pentru a face codul mai lizibil am folosit mai multe metode pentru afișare în fișier și în interfață.

***Clasa ViewClass***

Interfața este construită folosind biblioteca Swing din Java, care oferă elemente vizuale cum ar fi butoane, casete de text, panouri etc. pentru a crea o interfață grafică de utilizator.

Interfața conține un panou principal împărțit în două, unde partea stângă conține casete de text și etichete pentru introducerea datelor de intrare ale utilizatorului, cum ar fi numărul de clienți, numărul de cozi, timpii de procesare și așa mai departe.

Partea din dreapta este destinată afișării rezultatelor simulării. Există, de asemenea, un buton de pornire a simulării, care declanșează simularea după ce utilizatorul a introdus toate datele necesare.



***Clasa ControllerClass***

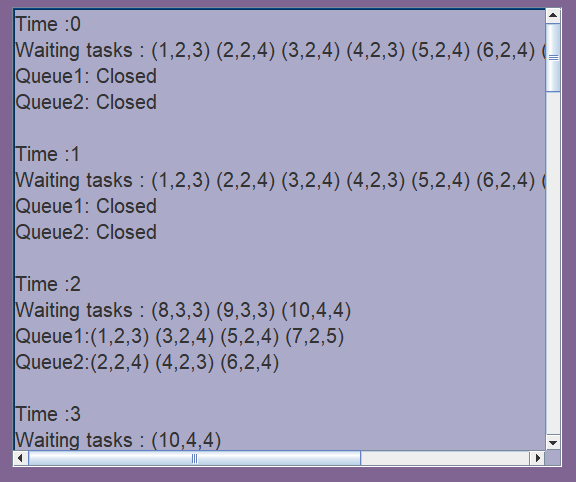
Această clasă este un controler pentru o aplicație de simulare. Implementează interfața ActionListener și are o metodă ruleazaSimulare() care este apelată atunci când utilizatorul apasă butonul "START" în interfața grafică.

În metoda ruleazaSimulare(), se extrage valoarea introdusă de utilizator pentru numărul de clienți, numărul de cozi, intervalul de simulare și intervalele de sosire și servire a clienților. Aceste valori sunt apoi convertite în numere întregi și sunt folosite pentru a crea un nou SimulationManager, care este un obiect care gestionează întreaga simulare. SimulationManager este apoi plasat pe un fir de execuție și începe simularea.

În metoda actionPerformed(ActionEvent e), controlerul ascultă evenimente de acțiune, cum ar fi apăsarea butonului "START". Dacă evenimentul este declanșat de butonul "START", se încearcă pornirea simulării prin apelarea metodei ruleazaSimulare(). Dacă există o excepție în timpul simulării, un mesaj de eroare este afișat în interfața grafică.

**5.Rezultate**

La finalul rulării aplicației trebuie să avem acces la rezultatele obținute pentru fiecare timp al simulării. Trebuie să vedem în cozi clienții care așteaptă la momentul dat. Trebuie să vedem cum li se decrementează clienților timpul de servire cu 1 de fiecare data când trece o secundă. De asemenea, trebuie să vedem că atunci când un client este trimis la coadă, acesta este scos din lista cu clienții care așteaptă. La final, trebuie să vedem și timpul mediu de așteptare, timpul mediu de servire și ora de vârf ale simulării.

****

**6. Concluzii**

În concluzie, proiectul de simulare a unui manager de cozi cu threaduri și scrierea rezultatului într-un fișier m-a ajutat să înțeleg mai bine cum se lucrează cu threaduri. Am reușit să dezvolt o aplicație Java eficientă care poate gestiona multiple cozi simultan și să le proceseze în paralel, utilizând threaduri separate pentru fiecare coadă.

Utilizarea threadurilor a permis procesarea mai rapidă a coziilor și evitarea blocajelor în timpul așteptării pentru obținerea resurselor. De asemenea, am putut să scriu rezultatele într-un fișier într-un mod eficient, utilizând mecanismele de scriere a fișierelor oferite de Java.

În concluzie, proiectul demonstrează cum tehnologia Java poate fi folosită pentru a crea soluții eficiente și scalabile pentru probleme complexe de simulare și procesare de date.

**7.Bibliografie**

<https://www.w3schools.com/colors/colors_rgb.asp>

<https://stackoverflow.com/questions/10177183/add-scroll-into-text-area>

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

<https://stackoverflow.com/questions/47753528/how-to-compare-and-increment-an-atomic-variable>

<https://www.digitalocean.com/community/tutorials/multithreading-in-java>