DOCUMENTAȚIE
TEMA 2

Nume student: Cotei Ruxandra-Maria
Grupa: 8

Cuprins

1. Obiectivul temei

Obiectivul temei este proiectarea și să implementarea unei aplicații de gestionare a cozilor folosind fire de execuție (threads) și mecanisme de sincronizare în Java. Această aplicație trebuie să simuleze un sistem de gestionare a cozilor, în care clienții sunt atribuiți la cozi astfel încât timpul de așteptare să fie minimizat.

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

Mecanisme de sincronizare:

Codul furnizat utilizează două mecanisme importante de sincronizare din Java: `Lock` și `Condition`, pentru a gestiona accesul concurent la resursele comune și pentru a asigura sincronizarea corectă între firele de execuție. Iată o explicație detaliată a acestor metode de sincronizare și de ce sunt acestea bune pentru gestionarea cozilor:

1. **Utilizarea `Lock` pentru sincronizare:**
   - În codul furnizat, folosim un obiect `ReentrantLock` pentru a asigura accesul exclusiv la resursele critice. Astfel, fiecare server (coadă) are propriul său lacăt (`Lock`), ceea ce permite doar un fir de execuție să acceseze resursele comune la un moment dat.
   - `Lock`-urile oferă o flexibilitate mai mare decât mecanismele de sincronizare tradiționale, cum ar fi blocurile `synchronized`, deoarece putem controla manual momentul blocării și eliberării acestora.
   - De asemenea, `ReentrantLock` este reentrant, ceea ce înseamnă că același fir de execuție poate dobândi lacătul de mai multe ori fără a bloca.
2. **Utilizarea `Condition` pentru a aștepta și a notifica:**
   - În codul furnizat, folosim un obiect `Condition` asociat cu fiecare lacăt pentru a gestiona așteptarea și notificarea altor fire de execuție.
   - Folosind metoda `await()` a obiectului `Condition`, firul de execuție poate aștepta până când o anumită condiție devine adevărată, ceea ce ne ajută să evităm poluarea și să economisim resurse.
   - Metodele `signal()` și `signalAll()` sunt utilizate pentru a notifica unul sau mai multe fire de execuție că o anumită condiție s-a schimbat și că acestea ar trebui să verifice din nou condiția.
3. **Evitarea blocării nedorite și a întârzierilor:**
   - Prin utilizarea explicită a mecanismelor `Lock` și `Condition`, putem evita blocarea accidentală sau blocarea nedorită care poate apărea în cazul utilizării blocurilor `synchronized`.
   - De asemenea, aceste mecanisme ne permit să gestionăm mai bine momentul eliberării lacătelor, reducând astfel întârzierile și creând o mai mare eficiență în gestionarea cozilor.

Prin utilizarea `Lock` și `Condition` în codul nostru, asigurăm o sincronizare corectă și eficientă între multiplele fire de execuție care manipulează cozile, evitând problemele legate de concurență, cum ar fi supraîncărcarea, blocarea și coruperea datelor.

Am optat pentru sincronizarea cu lock deoarece este o metodă mai familiară pentru mine, având în vedere experiența anterioară cu astfel de metode în cadrul materiei de Sisteme de Operare. Un avantaj semnificativ constă în faptul că oferă controlul asupra momentelor de blocare și deblocare.

3. Proiectare

În ceea ce privește structurile de date, această aplicație folosește câteva tipuri principale pentru a gestiona clienții și cozile de așteptare. Acestea sunt:

Lista de servere (`List`):

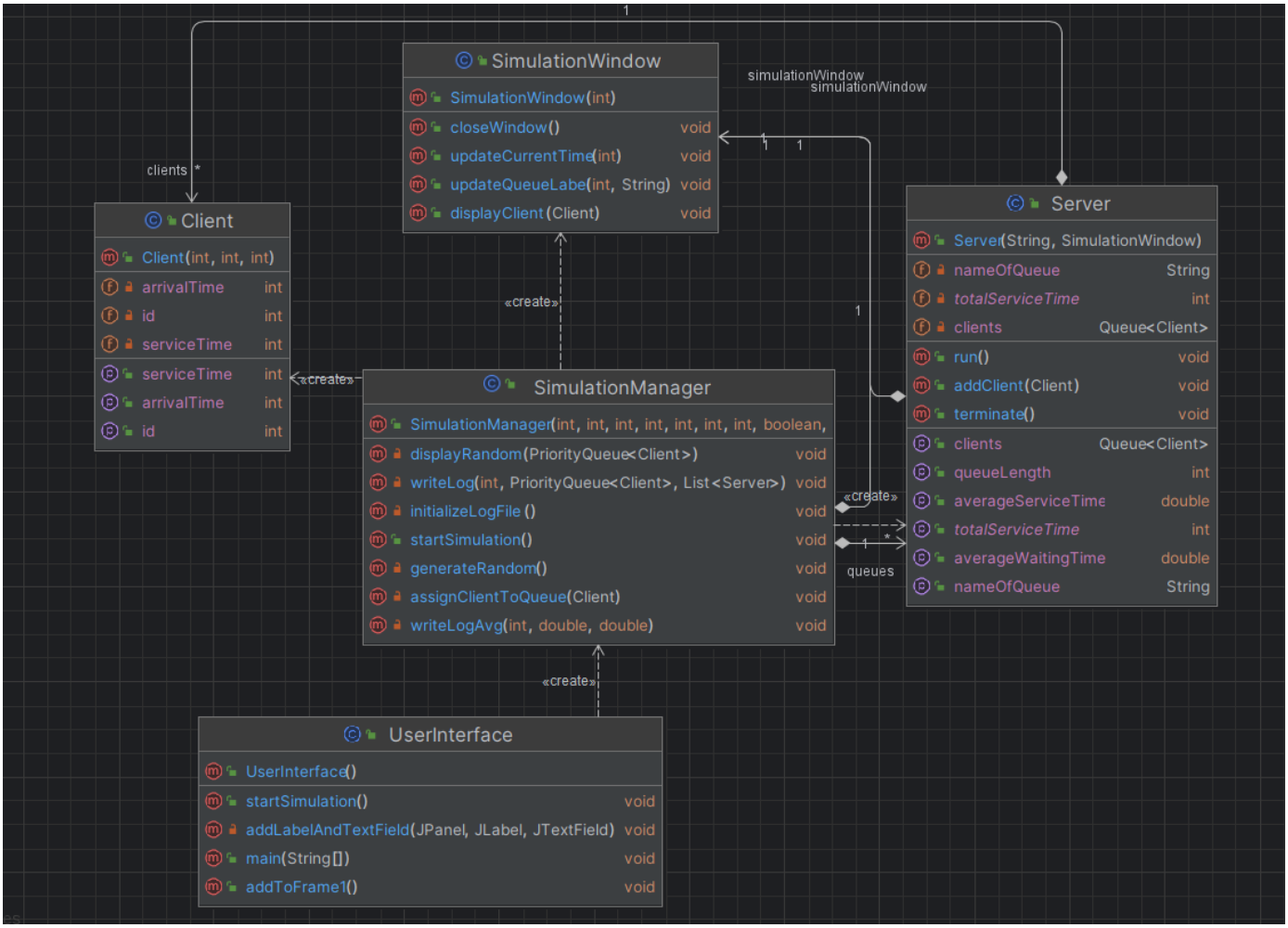
- **Descriere:** Lista de servere este implementată folosind interfața `List` din Java, care permite stocarea și gestionarea unui număr variabil de obiecte de tip `Server`.
- **Rol:** Această structură de date este esențială pentru gestionarea serverelor disponibile în simulare și permite accesul rapid la fiecare server în parte.
- **Utilizare:** În cadrul simulării, lista de servere este utilizată pentru a stoca și gestiona fiecare server disponibil. Fiecare server este adăugat la această listă la începutul simulării și este accesat și actualizat pe parcursul execuției pentru a procesa clienții și a colecta date statistice.
- **Beneficii:**
    1. **Acces rapid:** Lista permite accesul rapid la fiecare server în parte, ceea ce este crucial pentru gestionarea eficientă a clienților.
    2. **Flexibilitate:** Lista poate gestiona un număr variabil de servere, permițând extinderea sau restrângerea numărului acestora în funcție de nevoile simulării.

Coada de priorități (`PriorityQueue`):

- **Descriere:** Coada de priorități este implementată folosind clasa `PriorityQueued` in Java, care organizează și gestionează elementele în funcție de o anumită ordine definită de priorități.
- **Rol:** Coada de priorități este folosită pentru a organiza clienții în ordinea sosirii acestora în sistem, astfel încât cel mai devreme client sosind este mereu accesat primul.
- **Utilizare:** În cadrul simulării, coada de priorități este utilizată pentru a stoca clienții în funcție de timpul lor de sosire. Pe măsură ce clienții sosesc în sistem, aceștia sunt adăugați în coadă, iar apoi sunt procesați în ordinea sosirii lor.
- **Beneficii:**
    1. **Ordine de procesare clară:** Coada de priorități asigură că clienții sunt procesați în ordinea sosirii lor, ceea ce este esențial pentru simularea corectă a unui sistem de cozi.
    2. **Eficiență:** Implementarea subiacentă a colei de priorități asigură o performanță eficientă în gestionarea clienților, chiar și în cazul unui număr mare de clienți și sosiri concurente.

4. Implementare

Proiectul este construit după diagrama prezentată mai jos.



Pentru a facilita interacțiunea utilizatorului cu programul, am definit o interfață numită `UserInterface`. Aceasta facilitează introducerea datelor pe care se va modela simularea, fiind în strânsă legatură cu clasa `SimulationManager`, ce are rolul de a afișa într-o manieră inteligibilă simularea.

- **Clasa** `Client`**:**
  - Această clasă reprezintă un client care sosește la sistemul de simulare. O instanță a acestei clase stochează informații despre un client specific, cum ar fi id-ul, timpul de sosire și timpul de serviciu.
  - Este utilă pentru a gestiona și organiza informațiile despre clienți în cadrul sistemului de simulare, permitând manipularea acestora în mod corespunzător în timpul procesului de simulare.
- **Clasa** `Server`**:**
  - Această clasă reprezintă un server din cadrul sistemului de simulare. Un server este responsabil pentru gestionarea clienților care așteaptă să fie serviți.
  - **Campuri importante:**
    - `clients`: Reprezintă coada de clienți care așteaptă să fie serviți de către acest server.
    - `nameOfQueue`: Stochează numele corespunzător al cozii asociate acestui server.
    - `running`: Un indicator care indică dacă serverul este în continuare activ sau nu.
    - `lock`: Un obiect de tipul `Lock` folosit pentru sincronizarea accesului la resurse comune între firele de execuție.
    - `notEmpty`: O condiție asociată cu `lock`, utilizată pentru a aștepta când coada de clienți este goală.
    - `totalServiceTime`: Totalul timpului de servire pentru toți clienții.
    - `totalWaitingTime`: Totalul timpului de așteptare al clienților.
    - `numberOfClients`: Numărul total de clienți serviți.
    - `currentTime`: Timpul curent al serverului în timpul simulării.
    - `simulationWindow`: Fereastra de simulare asociată cu acest server, utilizată pentru actualizarea etichetelor cozii.
  - **Metode importante:**
    - `run()`: Metoda care rulează în firele de execuție ale serverului și gestionează procesul de servire a clienților.
    - `terminate()`: Metoda utilizată pentru a opri serverul și pentru a elibera resursele asociate.
    - `addClient(Client client)`: Metoda pentru adăugarea unui client nou în coada serverului și actualizarea statisticilor asociate.
    - `getTotalServiceTime()`: Returnează totalul timpului de servire pentru toți clienții.
    - `getAverageWaitingTime()`: Calculează și returnează timpul mediu de așteptare al clienților.
    - `getAverageServiceTime()`: Calculează și returnează timpul mediu de servire al clienților.
    - `getNameOfQueue()`: Returnează numele cozii asociate serverului.
    - `getClients()`: Returnează o copie a cozii de clienți a serverului pentru a evita modificările neașteptate ale acesteia din exterior.

- **Clasa** `SimulationManager`:
  - `startSimulation()`: Inițiază simularea, creând fire de execuție pentru servere, generând clienți aleatori și coordonând funcționarea întregii simulări.
  - `generateRandom()`: Generează clienți cu parametri aleatori într-o coadă prioritară.
  - `displayRandom(PriorityQueue<Client> clientQueue)`: Afisează clienții din coadă în fereastra de simulare.
  - `assignClientToQueue(Client client)`: Asignează un client la unul din serverele disponibile, urmărind o strategie specificată (timp sau coadă).
  - `writeLog(int currentTime, PriorityQueue<Client> clientQueue, List
  - `writeLog(int currentTime, PriorityQueue<Client> clientQueue, List`: Scrie loguri pentru fiecare moment de timp în timpul simulării, înregistrând starea cozi de clienți și a serverelor.
  - `initializeLogFile()`: Inițializează fișierul de jurnalizare pentru înregistrarea datelor.
  - `writeLogAvg(int peakHour, double averageWaitingTime, double averageServiceTime)`: Scrie loguri pentru orele de vârf și timpii medii de așteptare și de servire în fișierul de jurnalizare.

5. Rezultate

Logurile rezultatelor sunt atașate pentru a oferi o imagine detaliată și completă a desfășurării simulării. Aceste loguri conțin informații esențiale despre fiecare etapă a procesului, precum și rezultatele obținute la finalul simulării. Prin intermediul lor, se pot urmări înregistrările temporale, precum și evoluția stării cozi de clienți și a serverelor în timpul rulării. De asemenea, logurile includ informații importante despre orele de vârf și timpii medii de așteptare și de servire, oferind o perspectivă amplă asupra performanței sistemului simulat. Astfel, atașarea acestor loguri asigură transparență și coerență în evaluarea rezultatelor obținute în cadrul simulării.