DOCUMENTATIE

TEMA *NUMARUL\_2*

NUME STUDENT: Poponet Tiberiu Sergiu

GRUPA: 30221

# CUPRINS

[1. Obiectivul temei 3](#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](#_Toc95297886)

[3. Proiectare 5](#_Toc95297887)

[4. Implementare 8](#_Toc95297888)

[5. Rezultate 9](#_Toc95297889)

[6. Concluzii 9](#_Toc95297890)

[7. Bibliografie 10](#_Toc95297891)

# Obiectivul temei

Obiectiv principal:

Proiectarea și implementarea unei aplicații care are ca scop analiza sistemelor bazate pe cozi de așteptare, simulând a serie de N clienți care sosesc pentru a plati, care intră în cozile Q, așteaptă, sunt serviți și în cele din urmă părăsesc cozile, cât și calcularea timpului mediu de așteptare, a timpului mediu de serviciu și a orelor de vârf.

Subobiective:

• Analiza problemei și identificarea cerințelor

• Proiectarea aplicației simulare de magazin

• Implementarea aplicației simulare de magazin

• Testarea aplicației simulare de magazin

# Analiza problemei, modelare, scenarii, cazuri de utilizare

Caz de utilizare: simulare a configurării cozilor din magazin

Actor principal: utilizator

Principalul scenariu de succes:

1. Utilizatorul introduce valorile pentru:

* numărul de clienți
* număr de cozi
* interval de simulare minim
* ora maximă de sosire
* timp de serviciu minim
* timp de serviciu maxim

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

3. Aplicația validează datele și afișează:

- timpul la care se află, informând utilizatorul să înceapă simularea

- Clienții generați random care urmează să fie repartizați la cozi

- Numărul cozii și clienții care se află la cozile respective

Secvență alternativă: valori nevalide pentru parametrii de configurare, cum ar fi situația ân care utilizatorul introduce valori nevalide pentru configurarea aplicației, spre exemplu timpul minim de serviciu este mai mare decat timpul maxim de serviciu, timpul minim de sosire este mai mare decat timpul maxim de sosire

Exemplu de utilizare (scenariu 1):

* Userul insereaza in interfata grafica: timpul minim de sosire, timpul maxim de sosire, timpul minim de serviciu, timpul maxim de serviciu, numarul de clienți, numarul de cozi și timpul maxim de simulare
* Userul apasă butonul Press
* Aplicația afișează rezultatele, pas cu pas, incepând cu Timpul 0, când primim în câmpul Waiting Clients clienții care așteaptă repartizarea la coada cea mai potrivită

Press

(Inceputul simulării)

Introducere

Date

User

# Proiectare

Simulare

Magazin

2

5

1

**Așezare la cozi**

6

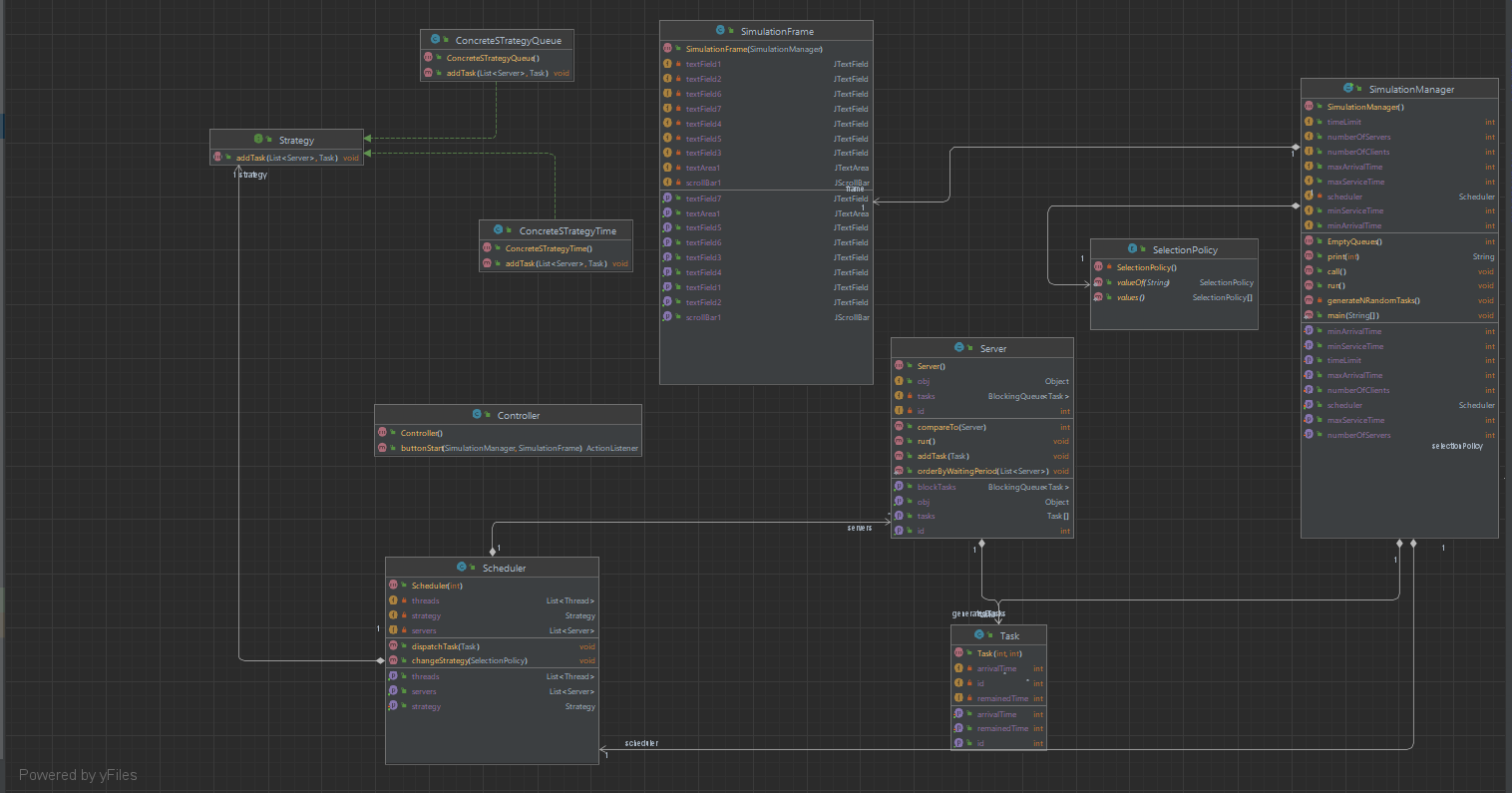
10

6

20

Press

Diagrama UML:



# Implementare

Pentru aplicatia mea,am un pacheorg.example, în care am creat 4 pachete : BusinessLogic, Controler, GUI si Model.

In BusinessLogic, avem:

1. O clasa ConcreteSTrategyQueue, care implementează interfața Strategy și definește o strategie specifică pentru organizarea clienților într-o coadă în cadrul unui magazin, în funcție de lungimea cozii. Metoda "addTask" este responsabilă pentru adăugarea unui element(Task) la coada asociată unuia dintre servere (Server). Această metodă primește o listă de servere și un task și găsește serverul cu cea mai mică coadă de sarcini, apoi adaugă sarcina în coada serverului respectiv. În cazul în care există mai multe servere cu cozi de lungime egală, se alege primul server din listă. Această abordare de organizare a coziilor se numește "First-Come, First-Served" (primul venit, primul deservit).
2. Clasa ConcreteSTrategyTime, care implementează interfața Strategy și definește o strategie specifică pentru organizarea clienților într-o coadă în cadrul unui magazin, în funcție de timpul de așteptare al cozii. Metoda "addTask" primește o listă de servere și o sarcină și adaugă sarcina în coada serverului cu cel mai mic timp de așteptare (waiting period). Acest lucru se realizează prin apelul metodei "orderByWaitingPeriod" definită în clasa Server, care sortează lista de servere în ordinea crescătoare a timpilor de așteptare. După sortare, metoda "addTask" adaugă sarcina în coada primului server din lista sortată, deoarece acesta are cel mai mic timp de așteptare. Această abordare de organizare a coziilor se numește "Shortest Job First" (primul servit cel mai scurt). Această clasă poate fi utilizată în cadrul unei aplicații pentru a implementa o strategie de organizare a clienților în cozi, într-un mod eficient și care optimizează timpul de așteptare al clienților.
3. Clasa Scheduler, care reprezintă un planificator (Scheduler) pentru organizarea sarcinilor (Tasks) primite în cadrul unui magazin, pe baza unor strategii specifice. Planificatorul conține o listă de servere (servers) și o listă de fire de execuție (threads) asociate fiecărui server. În plus, planificatorul are o strategie (strategy) și o metodă (changeStrategy) pentru a schimba strategia în timpul funcționării, pe baza unui tip de selecție dat (SelectionPolicy). Metoda dispatchTask primește o sarcină (Task) și folosește strategia curentă pentru a adăuga sarcina la coada corectă a serverului ales. Această metodă utilizează clasa ConcreteStrategyQueue sau ConcreteStrategyTime în funcție de tipul de selecție ales. În final, planificatorul oferă și metode de acces pentru lista de servere (getServers), strategia curentă (getStrategy), lista de fire de execuție (getThreads), precum și o metodă pentru a seta o strategie nouă (setStrategy). Această clasă poate fi utilizată în cadrul unei aplicații pentru a gestiona și planifica sarcinile primite într-un mod eficient, pe baza unor strategii specifice.
4. Enumerația SelectionPolicy, care care definește două politici de selecție posibile: "SHORTEST\_QUEUE" și "SHORTEST\_TIME". Aceste politici sunt utilizate în cadrul aplicației pentru a determina modul în care sarcinile sunt planificate și organizate în cozi în funcție de criterii precum lungimea cozii sau timpul de așteptare.
5. Interfața Strategy, pe care o implementează clasele ConcreteSTrategyQueue si ConcreteSTrategyTime. Această clasă definește o interfață pentru strategia utilizată în cadrul aplicației pentru a organiza sarcinile în cozi în funcție de politica de selecție specificată. Mai precis, această interfață conține o metodă "addTask" care primește o listă de servere și o sarcină și adaugă sarcina la un server corespunzător, în funcție de politica de selecție specificată. Utilizarea unei interfețe pentru a defini strategiile de organizare a sarcinilor în cozi face ca aplicația să fie mai flexibilă și mai ușor de întreținut, deoarece poate fi schimbată în mod dinamic la timpul de execuție, fără a necesita modificări ale codului sursă. De asemenea, această abordare încurajează utilizarea principiilor de programare orientate pe obiecte și permite utilizarea polimorfismului pentru a implementa diferite strategii de organizare a sarcinilor.
6. Clasa SimulationManager, Clasa SimulationManager este clasa principală a simulatorului și conține logica de simulare. Aceasta implementează interfața Runnable pentru a putea fi rulată într-un fir de execuție separat. Variabilele membre ale clasei sunt utilizate pentru a stoca parametrii de simulare precum timpul limită, numărul maxim și minim de clienți, timpul maxim și minim de servire, politica de selecție a cozii și obiecte precum Scheduler, SimulationFrame și lista de sarcini generate. Constructorul clasei inițializează obiectele necesare pentru simulare și afișează fereastra de simulare. Metoda generateNRandomTasks() generează un număr specificat de sarcini cu parametrii aleatorii. Metoda call() este apelată de la apăsarea butonului "Start" din interfața grafică și generează sarcinile. Metoda print() este folosită pentru afișarea stării actuale a simulării. Metoda run() conține logica principală a simulării și rulează într-un fir de execuție separat. Aceasta realizează următoarele acțiuni:
   1. Verifică dacă mai există sarcini în așteptare sau în cozi
   2. Așteaptă o secundă
   3. Adaugă sarcinile care au ajuns la momentul curent în cozi
   4. Actualizează interfața grafică
   5. Notifică serverele că au sarcini noi de executat

Metoda EmptyQueues() este folosită pentru a verifica dacă mai există sarcini în așteptare sau în cozi. Metodele set...() sunt folosite pentru a seta valorile parametrilor de simulare. Metoda main() este folosită pentru a testa clasa.

În pachetul Controller avem:

Clasa Controller, care folosește pentru simularea sistemului de cozi intr-o interfață grafică (GUI). Clasa "Controller" conține o metodă "buttonStart" care primește două obiecte ca parametri: "SimulationManager" (un obiect care gestionează simularea) și "SimulationFrame" (un obiect care reprezintă interfața grafică). Când butonul "Start" din interfața grafică este apăsat, acțiunea este captată de un "ActionListener" definit în cadrul metodei "buttonStart". Acesta setează parametrii necesari pentru simulare (timpul minim și maxim de sosire a clienților, timpul minim și maxim de servire a clienților, numărul de clienți, numărul de servere și limita de timp pentru simulare). Apoi, se creează un obiect "Scheduler" și se pornește câte un fir de execuție pentru fiecare server. În funcție de politica de selecție a serverului (politica "SHORTEST\_TIME" sau politica "QUEUE"), se setează strategia corespunzătoare. În final, se pornește simularea și se pornesc firele de execuție pentru fiecare server și pentru "SimulationManager".

În pachetul GUI avem:

Clasa SimulationFame, care reprezintă interfața grafică a aplicației de organizare a unui sistem de cozi. Clasa conține câmpuri private pentru componente GUI precum JTextField, JButton, JTextArea, JPanel și JScrollBar. Acestea sunt utilizate pentru a permite utilizatorilor să introducă informații și să vizualizeze rezultatele simulării sistemului de cozi. Constructorul clasei primește un obiect SimulationManager, care este utilizat pentru a inițializa valorile din câmpurile JTextField. De asemenea, în constructor se inițializează interfața grafică prin crearea unei instanțe JFrame și configurarea ei în consecință. În cele din urmă, se adaugă un ActionListener pe butonul pressButton, care declanșează simularea sistemului de cozi atunci când este apăsat. Metodele getter sunt utilizate pentru a obține referințe la componentele GUI din clasa Controller, care este responsabilă pentru a procesa acțiunile utilizatorului și a actualiza starea sistemului de cozi în consecință. Mai avem si Jlabel-urile interfeței, create cu ajutorul Java Swing.

În pachetul Model, avem:

1. Clasa Server. Clasa Server este folosită pentru a reprezenta un server care prelucrează sarcini (Task). Atributul tasks este un obiect BlockingQueue care stochează sarcinile care trebuie procesate de către server. Atributul waitingPeriod reprezintă timpul total de așteptare al tuturor sarcinilor din coadă. Constructorul clasei Server inițializează tasks cu o coadă blocantă cu o capacitate maximă de 1000 și waitingPeriod cu valoarea zero. De asemenea, atributul id este inițializat cu o valoare incrementată static, astfel încât fiecare server are un ID unic. Metoda addTask adaugă o sarcină nouă în coada serverului. Această metodă adaugă sarcina la coada tasks și actualizează waitingPeriod cu timpul rămas din noua sarcină. Metoda run este o buclă infinită care rulează pe server și procesează sarcinile. Mai întâi, metoda așteaptă să primească semnalul notify() pentru a continua execuția, după care verifică dacă există sarcini în coada tasks. Dacă există, preia prima sarcină din coadă și verifică dacă aceasta trebuie să fie eliminată sau actualizată. Dacă sarcina are un timp rămas de 1, atunci este eliminată din coadă, în caz contrar se actualizează timpul rămas și waitingPeriod. Metoda getTasks returnează toate sarcinile din coada tasks sub formă de array. Metoda getBlockTasks returnează coada blocantă de sarcini a serverului. Metoda compareTo este suprascrisă pentru a permite compararea a doi servere în funcție de valoarea waitingPeriod. Metoda orderByWaitingPeriod sortază lista de servere în ordinea creșterii timpului de așteptare. Acest cod implementează un sistem de gestionare a cozilor pentru servere, cu posibilitatea de a adăuga noi sarcini, a procesa sarcini existente și de a gestiona cozi multiple.
2. Clasa Tasks. Atributul arrivalTime reprezintă momentul în care sarcina a fost adăugată în coadă, remainedTime reprezintă timpul rămas pentru a finaliza sarcina și id reprezintă ID-ul unic al sarcinii. Constructorul clasei Task primește două parametri: arrivalTime și remainedTime și inițializează acești doi parametri, precum și atributul id cu o valoare incrementată static. Metodele setRemainedTime și setArrivalTime actualizează valorile atributelor remainedTime și arrivalTime. Metodele getArrivalTime, getRemainedTime și getId returnează valorile atributelor corespunzătoare. Această clasă este folosită pentru a reprezenta sarcinile care sunt adăugate în coada serverului și care trebuie procesate.

# Rezultate

Rezultatele pentru cele 3 teste cerute au fost incarcate în repo-ul de pe GIT.

# Concluzii

Din tema cu gestionarea cozilor, putem trage mai multe concluzii:

* Utilizarea coadelor și a blocurilor de sincronizare este utilă în contextul unui sistem cu mai multe firme de execuție care trebuie să proceseze sarcini venite din exterior.
* Este important să avem o bună gestionare a resurselor și să avem în vedere sincronizarea lor pentru a evita probleme precum blocarea sau dead-lock-ul.
* O structură de date precum coada este utilă în contextul adăugării de sarcini în sistem, deoarece acestea se adaugă în ordinea în care au fost primite.
* Blocurile de sincronizare pot fi folosite pentru a evita conflictele dintre thread-uri, de exemplu pentru a evita accesul simultan la aceeași resursă.
* Pentru a optimiza performanța sistemului, este important să avem o bună gestionare a timpului de așteptare și a priorităților sarcinilor, de exemplu prin ordonarea serverelor în funcție de timpul de așteptare și alocarea sarcinilor în consecință.

# Bibliografie

1. [*www.tutorialspoint.com*](http://www.tutorialspoint.com)
2. [*https://dsrl.eu/courses/pt/materials/PT2023\_Laboratory\_Resources.pdf*](https://dsrl.eu/courses/pt/materials/PT2023_Laboratory_Resources.pdf)
3. *code-it.ro*
4. [*www.jetbrains.com*](http://www.jetbrains.com)
5. *inf.ucv.ro*
6. [*www.scritub.com*](http://www.scritub.com)
7. *dsrl.eu*
8. <http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
9. <http://www.tutorialspoint.com/java/util/timer_schedule_period.htm>
10. <http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-andthreadpoolexecutor.html>