Simulator de cozi

Facultatea de Automatică și Calculatoare

Student: *Petrea Irina-Alexandra*

An 2, grupa 30226

. . . . . . . . . . . . . .

Cuprins:

1. **Obiectivul temei**
   1. **Obiectivul principal**
   2. **Obiective secundare**
2. **Analiza problemei**
3. **Proiectarea soluției**
4. **Implementare**
5. **Rezultate**
6. **Concluzii**
7. **Bibliografie**

. . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

1. Obiectivul temei

**Obiectivul principal** al temei este cel de a realiza o aplicație desktop pentru simularea unui sistem de cozi și distribuirea unui set de clienți generat aleatoriu către aceste cozi, așa încât timpul de așteptare al fiecăruia este minim. Aplicația este proiectată conform paradigmelor programării orientate pe obiecte folosind limbajul de programare Java. Punem în evidență mai multe obiective secundare, părți constituente ale obiectivului principal sus-menționat:

* **Divizarea responsabilităților pe clase și metode**, ca bună practică în proiectarea aplicației, prin structurarea pe pachete de clase, pentru a facilita înțelegerea programului și dezvoltarea ulterioară – va fi detaliată în capitolele 3 și 4
* **Realizarea diagramelor UML** (diagrame de clase, de pachete, etc), pentru a facilita proiectarea ulterioară și transpunerea funcționalităților în cod
* Realizarea unei aplicații ce rulează pe mai multe fire de execuție (**multi-threadded application**), corespunzătoare fiecărei cozi (**server**), respectiv distribuitorului de clienți pentru fiecare coadă (**scheduler**)
* **Lucrul cu fișiere și realizarea unui executabil .jar**
* **Simularea realistă a cozilor** prin închiderea și deschiderea lor dinamică în funcție de prezența sau absența clienților, prin anumite mijloace de sincronizare.

1. Analiza problemei

Se vor prezenta funcționalitățile oferite de aplicație, precum și detalierea cazurilor de utilizare a acesteia.

**Aplicația** realizează simularea unui sistem de cozi după urmează:

* Se definește și se păstrează un timp de simulare (simulation time); acest timp este unui discret (definit în cuante de timp de câte o secundă); acesta începe cu valoarea 0 și incrementarea lui marchează trecerea timpului; succesiunea pașilor simulării se face în acord cu acest timp de simulare;
* task-urile (clienții) sunt generate în mod automat de către aplicație, fiecare client fiind caracterizat de:
  + ID – Client ID
  + Momentul intrării în coadă – raportat la momentul inițial al simulării – Arrival Time
  + Timpul necesar procesării de către capătul cozii (casa de marcat, de exemplu – server) – Processing Time
* Server-ii (capetele de coadă) procesează fiecare client în ordinea în care a fost adăugat în coadă, un număr de cuante de timp egal cu timpul necesar procesării (definit mai sus). Când nu sunt clienți care stau la coadă, serverul își încetează temporar execuția, așteptând venirea unui nou client. După ce se scurge timpul de procesare al clientului curent, acesta este scos din coadă.
* Distribuția clienților către cozi este realizată de către un distribuitor specializat (scheduler), care ia clienții pe rând (ordonați crescător după momentul intrării în coadă) și îi asignează pe fiecare unei cozi, conform unei strategii de distribuire.
* Aplicația se sfârșește atunci când nu mai sunt clienți de distribuit și toate cozile s-au golit (s-a procesat și ultimul client)

Această aplicație nu prezintă interfață grafică, ea lucrând cu argumente date la execuție. Aceste argumente sunt:

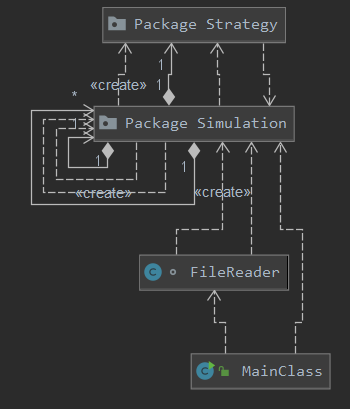
* Fișierul de intrare, ce are următoarea structură, pe fiecare linie:
  + Numărul de clienți ce va fi procesat
  + Numărul de cozi ce vor fi puse la dispoziție
  + Timpul maxim de simulare (timeout - după care simularea se oprește, indiferent de numărul de clienți nedistribuiți sau rămași în cozi
  + Timpul minim de intrare în coadă și timpul maxim de intrare în coadă, separate prin virgulă – acești timpi sunt relativi la timpul de simulare menținut de thread-ul principal de simulare
  + Timpul minim de procesare și timpul maxim de procesare, separate prin virgulă, specificați în cuante de timp, identice cu cele din timpul menținut de thread-ul principal de simulare; timpul de procesare reprezintă numărul de cuante de timp cât stă clientul respectiv la coadă

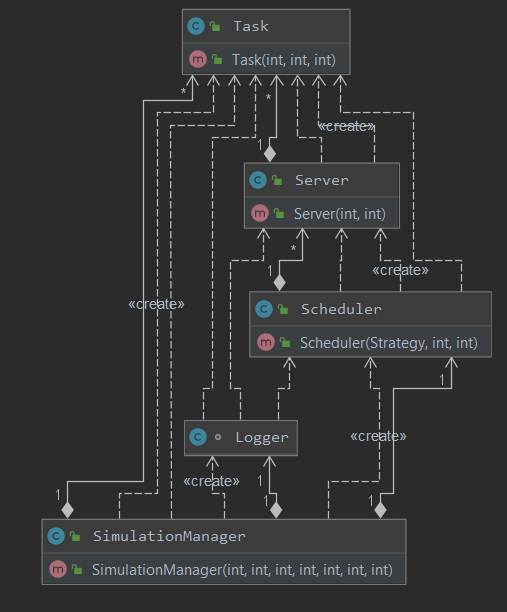
Pe baza acestor date, vor fi generate un număr de servere cu thread-uri aferente, precum și un număr de task-uri (clienți) având timp de procesare și moment de intrare în coadă aleatoare (între valorile minime și maxime citite din fișier)

* Fișierul de ieșire, care va fi generat ca o succesiune de pași corespunzători cuantelor de timp ce se scurg în timpul simulării. Numărul de pași nu este cunoscut la fiecare rulare de program, dar este sigur că nu va depăși valoarea de timeout citită din fișierul de intrare. Fiecare secțiune din fișierul de ieșire, corespunzătoare trecerii unei cuante de timp, va avea următoarea structură:
  + Time (momentul curent al simulării, un număr întreg între 0 și valoarea de timeout)
  + Waiting clients: (succesiune de clienți, conform structurii (id client, timp intrare, timp procesare), clienți care așteaptă să fie alocați unei cozi, de către scheduler)
  + Queue (numărul cozii) : (clienții care se află momentan la coadă, reprezentați conform structurii (id client, timp intrare, timp procesare))
  + Queue (numărul cozii) : (clienții care se află momentan la coadă)
  + ... (alte structuri similare cu cea de mai sus)
  + Average waiting time: (valoarea timpului mediu de așteptare, calculat ca fiind suma tuturor timpilor de așteptare ai tuturor clienților împărțită la numărul total de clienți; se consideră ca timp de așteptare al unui client numărul de cuante de timp scurse din momentul în care acesta a fost alocat unei cozi și până o părăsește, adică inclusiv timpul propriu de procesare).

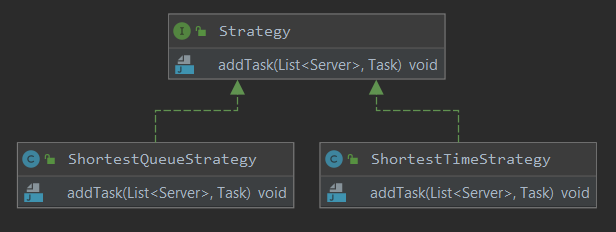
1. Proiectarea soluției

Se vor prezenta diagramele de pachete și de clase ce pun în evidență aspectele de proiectare OOP ale aplicației, precum și justificări legate de alegerile făcute în contextul proiectării.





Pachetul Simulation Pachetul Default, relatie cu celelalte pachete



Pachetul Strategy

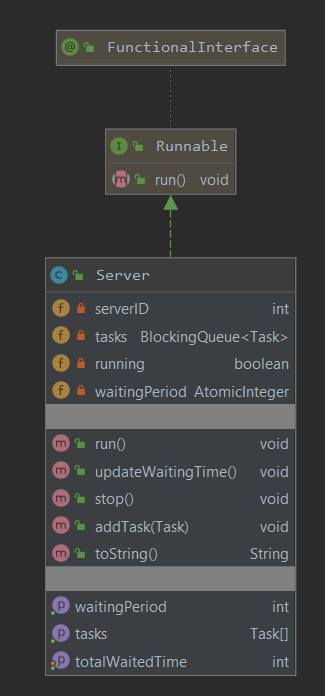
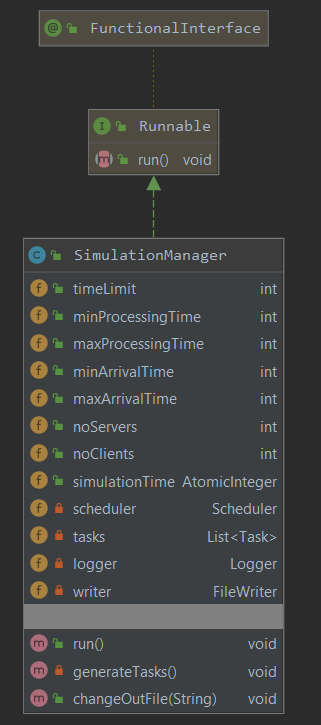
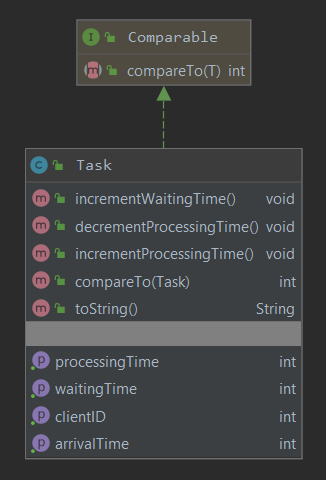
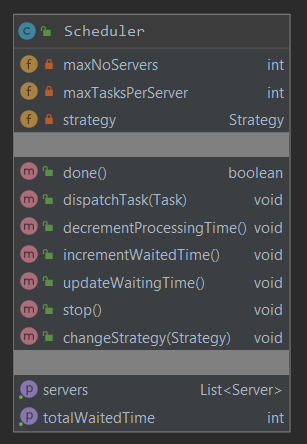
Diagramele de pachete de mai sus înfățișează structura pachetelor importante din implementarea aplicației. Chiar dacă numărul de clase este relativ mic, am ales o implementare pe pachete, distribuind responsabilitățile claselor din fiecare pachet după cum urmează:

* **Pachetul Simulation** – clasele din acest pachet modelează obiectele principale ale acestei simulări: clienții, prin clasa Task, cozile/casele de marcat prin clasa Server, distribuitorul de task-uri prin clasa Scheduler și managerul responsabil cu simularea propriu-zisă prin clasa SimulationManager; tot în acest pachet are loc și scrierea în fișier
* **Pachetul Strategy** – este un pachet ce include clase care implementează interfața Strategy și anume sunt clase care definesc o metodă de alegere a cozii optime la care se poate alătura un alt client. Acest mod de implementare a strategiei de distribuție a fost ales pentru facilitarea unei viitoare schimbări de strategie, fără a schimba codul în mod direct, lucru care se va putea face chiar și în real-time.
* **Pachetul default –** aici se procesează fișierul de intrare pentru a extrage parametrii simulării și se creează obiectul corespunzător simulatorului.

1. Implementare

Se va analiza fiecare clasă în parte, argumentând alegerile de implementare acolo unde este cazul.

* + - 1. **Pachetul Simulation**



* + - * + **Clasa Task** – este clasa care modelează clientul/task-ul ce este pus în coadă de către scheduler și procesat de către serverul la care este alocat. Un task este caracterizat de câmpurile client ID, arrivalTime (momentul în care se poate alătura la o coadă), processingTime (timpul de procesare, adică durata de timp pe care trebuie să o petreacă task-ul în capătul cozii până să fie scos) și waitingTime, reprezentând timpul petrecut de către acel client/task la coadă.
        + **Clasa Server** – este clasa care modelează provider-ul de servicii, sau procesorul de task-uri. Aceasta este o clasă ce implementează interfața Runnable, deoarece pentru fiecare server va exista un thread separat, toate serverele procesând clienți în mod paralel. Fiecare server este caracterizat de un ID de server, totalWaitedTime ce reprezintă totalul de timp petrecut de toți clienții care au stat la coada respectivă, waitingPeriod, ce reprezintă timpul pe care trebuie să îl petreacă la coadă un client nou pentru a ajunge în capul cozii, flagul boolean running care servește la oprirea threadului atunci când simularea se sfârșește, precum și colecția tasks, care este un ArrayBlockingQueue, colecție concurentă și thread-safe care reprezintă obiectul gestionat la comun de către thread-ul distribuitor de task-uri și serverul ce urmează să proceseze acel task. Dat fiind faptul că nu tot timpul sunt clienți la coadă, fiecare server verifică periodic dacă are clienți noi, iar dacă nu are, se suspendă pe durata unei cuante de timp. Dacă în acea cuantă de timp a fost alocat un client (sau mai mulți) acelei cozi, atunci începe procesarea, cât timp flag-ul de running este setat pe true. Procesarea reprezintă suspendarea pe o perioadă egală cu timpul de procesare al task-ului(pentru a simula procesarea lui), urmată de scoaterea acestuia din coadă. Se folosesc metodele offer și take pentru lucrul pe colecția thread-safe ArrayBlockingQueue.
        + **Clasa Scheduler** – este clasa care se ocupă cu distribuția task-urilor la servere și, de asemenea, clasa prin care SimulationManager comunică cu toate serverele pe care le are în subordonanță. Această clasă se ocupă cu crearea, pornirea și oprirea obiectelor server, precum și adăugarea de task-uri, modificarea waitingTime și waitedTime.
        + **Clasa SimulationManager –** este clasa responsabilă cu realizarea simulării ca întreg. Este o clasă ce implementează interfața Runnable, urmând să servească drept suport unui alt fir de execuție decât cel principal. În această clasă se generează aleatoriu clienții, apoi se ordonează după momentul de intrare în coadă, pentru a putea fi procesați corect. În metoda run a acestei clase se va ține timpul de simulare. Fiecare client va fi trimis către un server (ales de scheduler) doar atunci când timpul de intrare == timpul de simulare. Dacă sunt mai mulți clienți care îndeplinesc această condiție, vor fi cu toții distribuiți la cozi. După adăugare, se scrie în fișierul de ieșire conform structurii prezentate în capitolul precedent. Este important de menționat că timpul de procesare al primului client din fiecare coadă scade cu fiecare iterație și creștere a timpului de simulare. Acest timp de procesare este inițial la valoarea lui din obiectul Task, scade până la 1, urmând ca la următoarea iterație obiectul să fie scos din coada serverului respectiv.
      1. **Pachetul Default**
         * **Clasa MainClass –** de aici începe execuția programului, prin crearea unui nou thread având ca suport obiectul de tip SimulationManager și apelând pe acesta metoda start().
         * **Clasa FileReader –** realizează, prin metoda read, citirea din fișierul de intrare a parametrilor simulării și instanțiază un obiect din clasa SimulationManager cu acești parametri.
      2. **Pachetul Strategy**
         * **Clasa ShortestQueueStrategy –** clasă ce implementează interfața Strategy și care adaugă prin metoda addTask un task nou primit ca parametru la o singură coadă dintr-o listă de cozi primită ca parametru, acea coadă având lungimea ce mai mică (numărul de clienți cel mai mic);
         * **Clasa ShortestTimeStrategy -** clasă ce implementează interfața Strategy și care adaugă prin metoda addTask un task nou primit ca parametru la o singură coadă dintr-o listă de cozi primită ca parametru, acea coadă având timpul de așteptare minim; definim timpul de așteptare al unei cozi ca fiind numărul de cuante pe care trebuie să le petreacă un client la coada respectivă până când ajunge să fie în capul cozii.
         * **Interfața Strategy –** interfață ce obligă pe orice clasă ce o implementează să implementeze metoda addTask, prin care să se adauge unui server un task primit ca parametru, după un anumit algoritm.

1. Concluzii

În urma realizării acestei teme, am dobândit cunoștințe cu privire la utilizarea și sincronizarea thread-urilor, folosirea claselor thread-safe și utilizarea design pattern-ului Strategy. De asemenea, am aprofundat cunoștințe cu privire la citirea și scrierea din fișier, realizarea unui executabil .jar care să fie rulat cu argumente în linia de comandă, realizarea diagramelor UML ca ajutor în proiectarea aplicației, precum și realizarea unei documentații cuprinzătoare.

1. Bibliografie

[1] <http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/Java_Concurrency.pdf>

[2] <http://tutorials.jenkov.com/java-concurrency/index.html>

[3] <https://docs.oracle.com/javase/tutorial/essential/concurrency/>

[4] <https://www.udemy.com/>

[5] <https://www.geeksforgeeks.org/>

[6] <https://www.w3schools.com/>

[7] <https://stackoverflow.com/>

. . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . .