DOCUMENTATIE

TEMA 2

NUME STUDENT: Dragotoniu Corina Mădălina

GRUPA: 30224

# CUPRINS

[1. Obiectivul temei 3](file:///C:\Users\40770\Downloads\PT2021-2022_Documentation_Template%20(1).doc#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 4](file:///C:\Users\40770\Downloads\PT2021-2022_Documentation_Template%20(1).doc#_Toc95297886)

[3. Proiectare 5](file:///C:\Users\40770\Downloads\PT2021-2022_Documentation_Template%20(1).doc#_Toc95297887)

[4. Implementare 7](file:///C:\Users\40770\Downloads\PT2021-2022_Documentation_Template%20(1).doc#_Toc95297888)

[5. Rezultate 9](file:///C:\Users\40770\Downloads\PT2021-2022_Documentation_Template%20(1).doc#_Toc95297889)

[6. Concluzii 9](file:///C:\Users\40770\Downloads\PT2021-2022_Documentation_Template%20(1).doc#_Toc95297890)

[7. Bibliografie 9](file:///C:\Users\40770\Downloads\PT2021-2022_Documentation_Template%20(1).doc#_Toc95297891)

# Obiectivul temei

Obiectivul principal al temei este de a implementa o aplicație care are ca scop analiza sistemelor bazate pe cozi de așteptare. Astfel, se va simula o serie de N clienți care sosesc pentru servire, intra in Q cozi , așteaptă, sunt serviți si in cele din urma părăsesc cozile. De asemenea, se va calcula timpul mediu de așteptare, timpul mediu de serviciu si a orelor de vârf.

Sub-obiective:

* Analiza problemei si identificarea cerințelor
* Crearea unei aplicații de simulare
* Implementarea aplicației de simulare
* Testarea aplicației

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

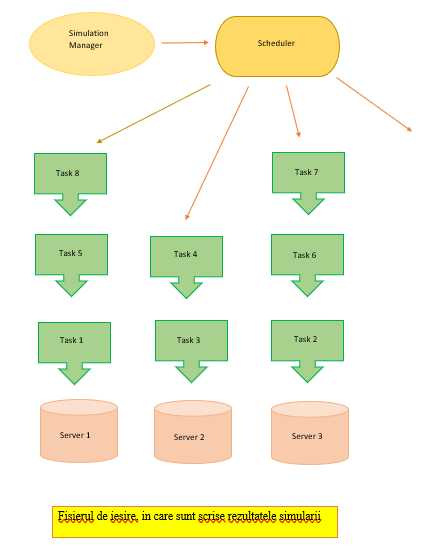
Conform cerinței, aplicația trebuie sa simuleze mai multe cozi de așteptare si mai multe task-uri care, in funcție de strategia de repartizare, sunt distribuite in cozi. In funcție de timpul de sosire si cel de așteptare, care sunt generate aleator, intre doua valori limita introduse de utilizator.

La o analiza mai atenta a problemei, avem ca si date de intrare:

* Numărul de task-uri
* Numărul de servere(cozi)
* Timpul de sosire minim, respectiv maxim
* Timpul de servire minim, respectiv maxim

Ca si date de ieșire, vom obține:

* N task-uri care au generat aleator timpul de sosire si cel de servire
* Vom obține, intr-un fișier de ieșire, simularea, la fiecare pas, a gestionarii task-urilor si a serverelor



# 3. Proiectare

În proiectarea acestei aplicații am folosit structura pusa la dispoziție in prezentarea temei.

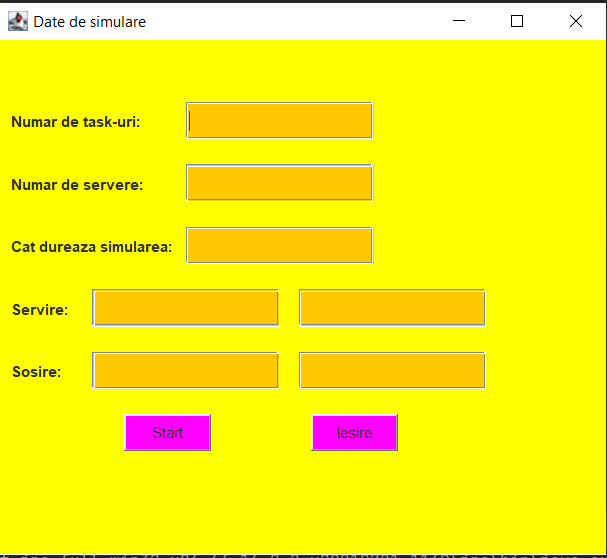
Astfel, proiectul cuprinde trei mari pachete: BusinessLogic, Model si GUI.]n pachetul Model am creat clasele Task si Server, modelul de date folosit in rezolvarea cerinței.Pachetul BusinessLogic cuprinde clasele Scheduler, SimulationManager, interfața Strategy, cu clasele care implementează Strategy, ConcreteStrategyTime si ConcreteStrategyQueue si enum-ul StartegyPolicy. Ultimul pachet, GUI, conține clasele View si Controller, care se ocupa de interactiunea cu utilizatorul, respectiv de acțiunea ce se petrece prin apăsarea butoanelor din interfata. De asemenea, mai am o clasa Main in pachetul BussinesLogic, cea care creeaza un nou View si un nou Controller.

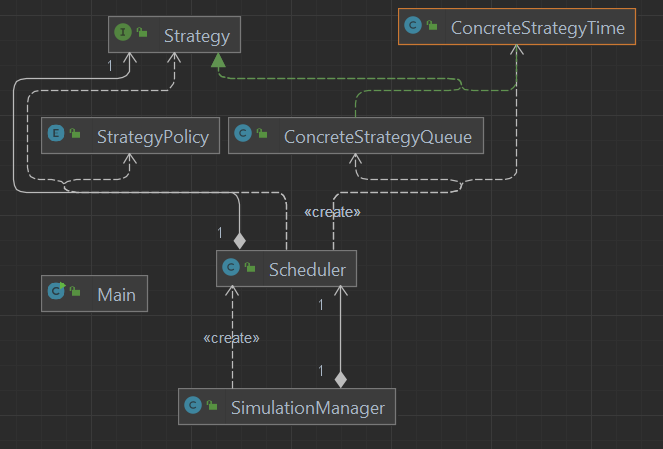
SimulationManagerul este centrul întregii aplicații, și acesta cuprinde Scheduler-ul, care asigneaza Task-urile unui Server. Fiecare server corespunde unui Thread de execuție. Scheduler-ul alege serverul în care un Task va merge ținând cont de strategia aleasă. Strategiile puse la dispoziție sunt ConcreteStrategyTime respectiv ConcreteStrategyQueue. Prima alege coada care are cel mai mic timp de așteptare. Cea din urmă, alege coada care are cele mai puține Task-uri aflate deja în coadă.

Tot SimulationManagerul genereaza Task-urile, asignându-le un id, un arrivalTime și un serviceTime aleator, dar care se încadrează intre doua limite date. În cadrul acestuia, se scrie în fișier simularea intregului proces de servire.

Pentru folosirea firelor de execuție, clasele Server respectiv SimulationManager implementează interfața Runnable, și este creat pentru fiecare un Thread care este lansat în execuție.

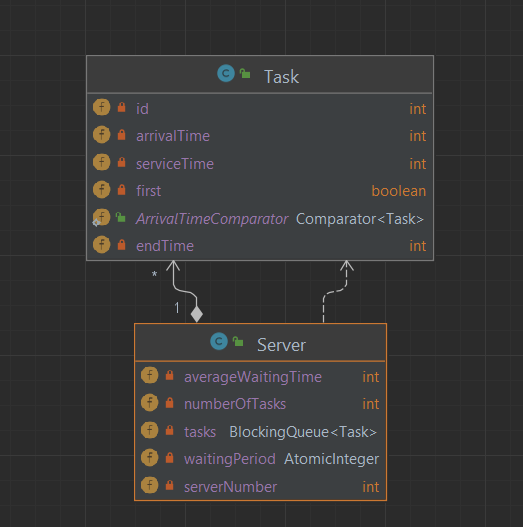
Mai jos, se poate vedea interfața in care urmează a fi introduse de către utilizator datele de intrare pentru simulare.





O imagine care conține text

Descriere generată automat



Mai sus sunt prezentate diagramele UML pentru fiecare pachet din cadrul proiectului, alături de legăturile create intre aceste clase.

# 4. Implementare

* Clasa **Task** reprezintă sarcinile ce așteaptă ce așteaptă în coadă. Clasa cuprinde un constructor care instanțiază un Task, care este caracterizat de un ID, un arrivalTime, un serviceTime care sunt primiți ca parametru, dar si un endTime, care se calculează ca suma dintre arrivalTime si serviceTime si un first de tip boolean, care ne arata daca task-ul va fi primul in noua coada sau nu . Clasa cuprinde Settere și Gettere pentru câmpuri, care sunt private. Pentru a respecta ordinea firească a sarcinilor din punctul de vedere al timpului în care acestea ajung în coadă, trebuie să poată fi realizată o comparație. Astfel, clasa implementează un Comparator, care suprascrie metoda compare, returnând un rezultat in funcție de arrivalTime al celor doua Task-uri comparate.
* Clasa **Server** simbolizează coada din lumea reală, în care sunt distribuiți și așteaptă sarcinile. Clasa cuprinde 4 atribute, un BlockingQueue de elemente de tipul Task, structură de date sigur de folosit și corespunzătoare lucrului cu fire de execuție, un AtomicInteger ce reprezintă timpul de așteptare al cozii, adică suma tuturor timpilor de procesare ale Task-urilor din coadă. Pentru a fi mai ușor de afișat, am creat un câmp numit serverNumber, care reprezintă numărul serverului si, de asemenea, clasa noastră mai are si un câmp numberOfTasks, care reprezintă numărul de task-uri din coada. Clasa cuprinde settere respectiv gettere pentru toate câmpurile de care este nevoie a fi folosite in alte clase. Pentru a adăuga elemente de tip Task în coadă, folosim metoda addTask, care va adăuga un Task primit ca parametru în coada de tip BlockingQueue menționată mai sus. Deoarece Server-ului îi este atribuit un Thread, clasa implementează interfața Runnable respectiv suprascrie metoda run() specifică clasei.
* În **StrategyPolicy** sunt declarate tipurile de politica de adăugare în coadă, care vor fi descrise fiecare în parte: SHORTEST\_QUEUE, SHORTEST\_TIME.
* Interfața **Strategy** conține antetul metodei addTask, care va trebui implementată de clasele ce vor implementa această interfață, ConcreteStrategyTime respectiv ConcreteStrategyQueue, corespunzătoare elementelor din SelectionPolicy.
* Clasa **ConcreteStrategyQueue** implementează interfața Strategy și în metoda addTask, primește ca parametru o listă de servere, un Task si un ArrayList newServers, și adaugă Task-ul în serverul cu cele mai puține Task-uri.
* Clasa **ConcreteStrategyTime** implementează interfața Strategy și în metoda addTask, primește ca parametru o listă de servere și un Task, și adaugă Task-ul în serverul cu cel mai scurt waitingPeriod.
* Clasa **Scheduler** conține o listă de servere,una de Thread-uri, un timeCounter de tip AtomicInteger si un strategy de tip Strategy. Rolul acestuia este de a adăuga în servere task-uri, conform strategiei alese. Clasa cuprinde un constructor în care sunt inițializate și lansate serverele, settere și gettere pentru atribute, și o metodă de schimbare a strategiei de adăugare a Task-urilor.
* Clasa **SimulationManager** reprezintă centrul întregii aplicații. Aceasta cuprinde atributele corespunzătoare datelor de intrare, care sunt primite ca parametru și luate din interfața grafică de introducere a datelor. Aceasta mai cuprinde un obiect de tip Scheduler, un String care reprezintă fișierul de scriere al simulării, un FileWriter, respectiv lista de Task-uri generate aleator. În constructor sunt date valorile atributelor, este instantiat schedulerul, este aleasă strategia de repartizare și sunt generate task-urile aleator, prin metoda generateNRandomTasks. În această metodă, ID-urile task-urilor sunt cuprinse între 1 și numărul total de Task-uri, iar serviceTime-ul și arrivalTime-ul sunt generate conform timpilor minimi și maximi. Task-ul cu aceste atribute este instanțiat și adăugat listei generatedTasks. La final, aceasta listă este ordonată folosind metoda sort() a clasei Collections. SimulationManagerul este și el lansat, astfel implementează interfața Runnable respectiv suprascrie metoda run(). În această metodă are loc repartizarea task-urilor de către scheduler, prin metoda sa dispatchTask(). Se pornește de la un currentTime = 0, și totul se execută cât timp currentTime este mai mic decât timpul de simulare. La fiecare pas, Task-urile cu arrivalTime egal cu timpul curent sunt scoase din lista generatedTasks, pe care o putem vedea ca o listă de așteptare. Apoi, este repartizat de către scheduler. După aceasta, se face afișarea ce reprezintă cheia acestei aplicații. Este afișat timpul curent, task-urile în așteptare, și fiecare coadă cu task-urile pe care le conține. Odata cu incrementarea timpului curent, timpul de procesare al task-ului din capul cozii este decrementat, și când ajunge la 0 capul cozii este eliberat. Dacă coada rămâne goală, este considerată închisă. Se calculează timpul mediu de procesare, adunându-se timpii de la fiecare task care a fost repartizat într-o coadă și împărțit la numărul de task-uri. La finalul simulării, sunt afișate timpul mediu de așteptare si timpul mediu de servire.Toate valorile care trebuie afisate vor fi trecute intr-un fișier de ieșire, care va conține detaliat situația in fiecare moment al simulării.
* Clasa **View** cuprinde numeroase label-uri și textfield-uri pentru fiecare dată de intrare si, de asemenea, mai cuprinde doua butoane, unul care se ocupa de startul simulării si unul de Exit, care odată apăsat ne închide aplicația.
* Clasa **Controller** se ocupa de acțiunea din spatele celor doua butoane din Interfața. In aceasta clasa avem implementata o noua clasa, startActionListener, care implementeaza ActionListener si suprascrie metoda actionPerformed(). In aceasta metoda se instantiaza un obiect de tip SimulationManafer si, de asemenea, are loc scrierea in fișier daca datele de intrare au fost introduse corect. In mod similar, clasa endActionListener are suprascrisa metoda actionPerformed(), in care are loc închiderea aplicației.

# 5. Rezultate

Pentru testarea aplicației s-au folosit atât date de intrare aleatorii, cât și cele propuse în cerința temei.

În cadrul primului test, care presupunea ca date de intrare un număr de 4 task-uri, 2 servere, un timp de simulare de 60 de secunde, arrivalTime-ul cuprins între 2 și 30 de secunde și serviceTime de la 2 până la 4 secunde, pentru fiecare rulare a aplicației, nu s-a atins timpul de simulare maxim, simularea oprindu-se mai repede deoarece numărul mic de clienți respectiv timpii de procesare mici au permis ca fiecare să fie servit în limitele timpului de simulare.

În cadrul celui de-al doilea test, datele de intrare presupuneau un număr de 50 de task-uri ce trebuiau repartizate într-un număr de 5 servere, în timpul de simulare de 60 de secunde, cu arrivalTime-ul cuprins intre 2 și 40 și cu serviceTime intre 1 si 7. Depinzând de task-urile generate și de arrivalTime-urile lor, sunt posibile două scenarii. Primul, în care toate task-urile sunt finalizate în timp util, adică în timpul de simulare. Al doilea scenariu presupune terminarea simulării în condițiile în care încă există taskuri în execuție, în servere.

În cadrul celui de-al treilea test, datele de intrare puse la dispoziție sunt un număr de 1000 de Task-uri, 20 de servere în care vor fi repartizate, un timp de simulare de 200 de secunde, arrivalTime intre 10 și 100 și serviceTime între 3 si 9. La fiecare rulare a aplicației pentru aceste date de intrare, la finalul simulării rămâne un număr semnificativ de task-uri în servere, care nu sunt finalizate.

# 6. Concluzii

Prin rezolvarea acestei teme, consider că am aprofundat noțiunea de fir de execuție, noțiune pe care o cunoșteam mai mult teoretic, nu si din punctul de vedere al lucrului cu aceasta. Partea cea mai dificila a fost înțelegerea modului de funcționare al Threadurilor, aceeași parte de cod fiind rulată în același timp dar cu date diferite. O dezvoltare ulterioară imediată ar fi afișarea intr-o noua fereastra a datelor obținute in urma simulării in timp real, nemaifiind necesar ca utilizatorul sa deschidă fișierul de ieșire pentru a vedea rezultatul obținut.

# 7. Bibliografie

<https://dsrl.eu/courses/pt/materials/PT2021-2022_Assignment_2.pdf>

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

<https://dsrl.eu/courses/pt/materials/A2_Support_Presentation.pdf>

<https://www.tutorialcup.com/ro/java/multithreading-java.html>