Logo, company name

Description automatically generated

**Documentatie tema 2**

**Nume student: Maracine Stefania Maria**

**Profesor coordonator: Rancea Alexandru**

**Grupa: 30223**

# CUPRINS

[1. Obiectivul temei **.**](#_Toc95297885)

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

[3. Proiectare **.**](#_Toc95297887)

[4. Implementare **.**](#_Toc95297888)

[5. Rezultate **.**](#_Toc95297889)

[6. Concluzii **.**](#_Toc95297890)

[7. Bibliografie **.**](#_Toc95297891)

1. **Obiectivul temei**

Obectivul principal al celei de a doua tema de la laboratorul de tehnici de programare este de a implementa si proiecta o aplicație care urmărește să analizeze sisteme bazate pe cozi de așteptare prin simulare a serii de N clienți care sosesc pentru service, care intră în cozile Q, așteaptă, sunt serviți și în cele din urmă părăsesc cozile și calcularea timpului mediu de așteptare, a timpului mediu de serviciu și a orelor de vârf.

Subobiective

• Analizați problema și identificați cerințele

• Proiectați aplicația de simulare

• Implementarea aplicației de simulare

• Testați aplicația de simulare

1. **Analiza problemei, modelare, scenario, cazuri de utilizare**

Problema este una simpla, utilizatorul introduce valorile pentru: numărul de clienți, număr de cozi, interval de simulare, ora minima si maximă de sosire și timpul maxim si minim de serviciu. Astfel, clientii si task-urile lor vor fii, generate si vor contine un id crescator in functie de ordinea in care au fost generate, un timp de sosire random cuprins in intervalul selectat de utilizator si un timp de servire random cuprins in intervalul selectat de utilizator, apoi task-urile vor fii ordonate crescator dupa timpul de sosire. Dupa ordonare, toate task-urile vor fii adaugate in numarul de cozi disponibile astfel, coada cu cel mai mic timp de serviciu va primi urmatorul task din lista. Dupa ce toate task-urile au fost generate, ordonate si adaugate in cozi, se va incepe executia task-urilor. Executia acestora se va face cu ajutorul mai multor thread-uri, cate cozi avem, atatea thread-uri vor rula, pentru o executie mai rapida a programului. Dupa ce toate task-urile au fost terminate, programult se va opri. In consola din IntelliJ, se va afisa, cu ajutorul log-urilor, generarea, ordonarea, cum au fost impartite task-utile in cozi si apoi executia task-urilor, cand s-a inceput executia si cand s-a terminat.

Graphical user interface, diagram

Description automatically generated

1. **Proiectare**

In acest capitol este prezentata diagrama UML, clasele existente si structurile de date folosite.

Ca si clase avem 6 clase folosite in acest proiect.

Clasa principana este clasa Task. Cea de-a doua clasa este Generator, apoi avem clasele Server, Scheduler. Pentru afisarea cu ajutorul log-urilor avem clasa Logs iar la final pentru testare si verificare avem clasa Test.

Ca si structure de date am folosit listele, BlockingQueue.

1. **Implementare**

In acest capitol se va prezenta mai in detaliu functionalitatea fiecare clase, cu variabilele si metodele lor.

1. Clasa **Task**

Prima clasa despre care vorbim este clasa de baza Task care implementeaza Comparable, o interfata predefinita din Java care ne ajuta sa ordonam prin comparare anumite elemente. Fara aceasta clasa nu ar fii posibila realizarea rpoiectului. In aceasta clasa este definit cum arata un task. Clasa are ca si variabile un id, un arrivalTime si un serviceTime. Id-ul evidentiaza ordinea in care au sosit clientii cu task-urile lor. arrivalTime stocheaza timpul la care a sosit clientul cu task-ul respective, iar serviceTime repezinta timpul de servire specifil task-ului clientului respective.

In aceasta clasa am creat de asemenea gettere si settere pentru fiecare din variabilele explicate mai sus.

Mai avem de asemeanea si metoda suprascrisa toString, modificata dupa nevoile noastre, care ne va ajuta mai tarziu la afisarea generarii task-urilor.

Ca si ultima metoda in aceasta clasa avem metoda compareTo. Aceasta metoda ne ajuta sa comparam timpii de sosire adica arrivalTime pentru a putea ordona task-urile crescator dupa timpul de sosire.

1. Clasa **Generator**

In clasa Generator avem mai multe variabile, care pot fii modificate de catre utilizator pentru a da ce valori doreste aceasta simulatii noastre. Astfel avem timeLimit care va fii timpul limita al simularii. Dupa aceasta avem maxProcessingTime si minProcessingTime care sunt timpii minimi si maximi pentru durata de servire al fiecarui task. Urmatoarele doua variabile sunt maxArrivalTime si minnArrivalTime. Aceste doua variabile stocheaza timpul minim si timpul maxim in care persoanele pot sa soseasca cu task-urile lor. Ultima variabila este variabila numberOfServers, care ne va spune care servere adica cozi sunt folosite in simulare.

Pe langa variabilele declarate si explicate mai sus aveam si declararea unei liste de task-uri.

Ca si metode in aceasta clasa avem contructorul pentru lista de task-uri. Apoi avem definite un getter si un setter pentru lista noastra.

Cea mai importanta metoda din aceasta clasa este metoda generateTasks care primeste ca parametru unn numar n ce reprezinta numarul de clienti utilizati in simularea noastra si care genereaza n task-uri pentru cei n clienti. Dupa generare se face si sortarea task-urilor crescator in functie de arrivalTime.

Metoda functioneaza astfel: se ia un for de la 1 la numarul de clienti primit ca si paramentru. Pentru fiecare i se creaza un task nou si un obiect de tip random. Apoi in doua variabile locale rand\_int1 si rand\_int2 se salveaza 2 numere random intre 0 si maxArrivalTime si respective intre 0 si maxProcessingTime care vor reprezenta timpul de sosire al clientului si timpul de procesare al task-ului.

Dupa alegerea celor doua numere random se verifica conditia ca cele 2 variabile locale sa fie mai mari decat minArrivalTime si minProcessingTime pentru a fii siguri ca introducem in task valori corespunzatoare cu intervalele mentionate mai sus de utilizator. Daca conditia este indeplinita atunci se seteaza id-ul cu i-ul folosit in for, se seteaza arrivalTime si serviceTime cu cele 2 variabile locale rand\_int1 si rand\_int2 definite mai sus, si se adauga task-ul in lista. Daca conditia nu se indeplineste atunci i-ul se decrementeaza si se incepe iar aceeasi operatiie pana cand se indeplineste conditia. Dupa incheiarea for-ului si umplerea listei cu n task-uri pentru cei n clienti doriti, vom sorta lista de task-uri crescator dupa arrivalTime pentru a ne fii mai usor in urmatoarea parte a proiectului cand vom adauga task-urile in cozi. La final returnam lista ordonata obtinuta.

1. Clasa **Scheduler**

Primul si primul lucru in aceasta clasa este importarea librariilor necesare pentru functionarea proiectului nostru.

In cadrul acestei clase se vor pune task-urile in coada, adics server-ul nostru cu cel mai mic serviceTime sau waitingTime. Incepem prin a initializa o lista de servere, o lista de thread-uri si o lista de task-uri. De asemenea avem definit si logger-ul pentru afisarea in consola.

Ca si prima metoda avem contructorul care primeste ca parametrii lista de task-uri si numarul de servere sau de cozi pe care dorim sa le folosim, numar ce este ales de utilizator. In constructor se ia un for de la 0 la numarul de servere maxNoServers, iar pentru fiecare I se creaza un nou obiect de tip server. De asemenea pentru fiecare server create, cu ajutorul unui foreach se creaza si cate un thread pentru fiecare server.

Urmatoarea metoda din aceasta clasa este metoda getServerWithMinimumWaitingPeriod. Aceasta metoda este una sincronizata. Ce inseamna acest lucru? Metoda sincronizată este folosită pentru a bloca un obiect pentru orice resursă partajată. Când un fir de execuție invocă o metodă sincronizată, acesta dobândește automat blocarea pentru acel obiect și o eliberează când firul de execuție își finalizează sarcina. Pe scurt, metoda poate fii apelata doar de cate un thread odata, cand thread-ul care ruleaza deja isi termina treaba, poate incepe urmatorul. Aceasta metoda returneaza serverul cu cea mai mica perioada de asteptare, adica cel mai mic serviceTime. Acest lucru este realizat cu ajutorul unei metode de sortare implementata in clasa server si apoi apelata aici de fiecare data dupa fiecare inserare de task.

Ultima metoda din aceasta clasa este metoda startScheduler. Aceasta este metoda care se ocupa propriu zis cu punerea fiecarui task in coada cu cel mai mic timp se servire. Primul lucru pe care trebuie sa il facem in aceasta metoda este sa pornim thread-urile, astfel cu ajutorul unui foreach este pornit fiecare thread in parte. Al doilea foreach pe care il utilizam este foreach-ul care pune task-urile la locul lor si care afiseaza in consola cu ajutorul log-urilor unde a fost pus fiecare task. Toate acestea sunt realizate cu ajutorul metodei de mai sus getServerWithMinimumWaitingPeriod care ne da serverul cu cel mai mic waitingTime sau serviceTime, stiind apoi coada sau serverul perfect pentru urmatorul task.

1. Clasa **Server**

Aceasta este clasa Server care implementeaza Comparable, o interfata predefinita din Java care ne ajuta sa ordonam prin comparare anumite elemente. Aceasta clasa mai implementeaza si Runnable, o alta interfata predefinita in Java. Pentru a implementa un Runnable, trebuie doar implementat metoda run. În această metodă, avem codul pe care vrem să-l executăm pe un fir concurent, adica pe un thread. În această metodă, putem folosi variabile, instanția clase și efectua o acțiune la fel ca firul principal. Aceasta este clasa in care se executa fiecare task in parte in functie de cum a fost pusa in coada sau server. De asemenea aici sunt definite serverele si atributele lor, pe care le vom discuta mai in detaliu in cele ce urmeaza.

Incepem prin a define variabilele de care avem nevoie. Primul lucru pe care il declaram este coada de liste. Ne luam si un ID pentru a stii numarul fiecarei cozi sau server. Urmatoarea variabila este o variabila te tip AtomicInteger. Clasa AtomicInteger protejează o valoare int de baza prin furnizarea de metode care efectuează operații atomice asupra valorii. Nu trebuie utilizat ca înlocuitor pentru o clasă Integer. Aceast variabila este cea care ne ajuta sa contorizam timpul de asteptare intre task-uri si care ne ajuta sa dam timpul de sleep intre executiile thread-urilor. Avem o variabila latsID care ne ajuta sa contorizam corect id-urile serverelor si pe care o initializam cu 0. Si de asemenea avem logger-ul pentru afisarea datelor de executie in consola.

Prima metoda din aceasta clasa este de fapt contructorul. Constructorul seteaza ID-ul cu lastID si apoi incrementeaza lastID-ul. De asemenea initializeaza si fiecare coada de task-uri.

Urmatorul lucru pe care il facem este sa generam un getter pentru waitingPeriod, ne va fii util sa preluam fiecare waitingPeriod pentru a stii timpul intre executii.

Urmatoarea metoda este un getter petru ID-ul serverului.

Aceastea au fost metode mai de baza. Urmatoarea metoda este o metoda sincronizata addTask. Astfel aceasta metoda adauga in coada task-ul pe care il primeste ca parametru si incrementeaza waitingPeriod cu serviceTime-ul sau timpul de serviciu al taskului adaugat.

A doua metoda sincronizata din aceasta clasa este waitngTime care primeste ca parametru un Task t si care da sleep thread-urilor, adica o mica pauza intre executii, in functie de serviceTime-ul fiecarui task, dupa care decrementeza waitingPeriod.

Avem doua metode suprascrise compareTo si ToString. compareTo este folosita in Scheduler pentru sortarea serverelor dupa timpul de asteptare adica waitingTime, iar toString este utilizata pentru a afisa task-ul si serverul in care a fost adaugat.

Ultima metoda din aceasta clasa este metoda run, pe care o implementeaza clasa. Ne-am luat mai sus o variabila de tip Boolean running pe care o initializam cu valoarea true. Cat tipm aceasta variabila este true, programul nostru va rula. Cat timp ruleaza, intr-un obiect de tip Task scoatem din coada si salvam un task. Daca task-ul este diferit de null, incepe executia acestuia, se apeleaza metoda waitingTime pentru a avea spatiu intre executii exact cat este serviceTime-ul, si afisam cand se incepe executia si cand se termina. Daca task-ul este null inseamna ca s-au terminat toate task-urile din coada, deci ii putem da variabilei running valoarea false pentru a putea opri executia.

1. Clasa **Log**

In aceasta clasa am definit tot ce era necesar pentru afisarea log-urilor intr-un fisier txt.

1. Clasa **Test**

In clasa test avem Log-ul pentru afisarea in consola. Instantiem un obiect de tip Generator si intr-o lista de task-uri generam task-urile pentru valorile date di dorite de utilizator.

Apoi pentru fiecare task in parte se face o afisare a id-ului timpului de sosire si timpului de servire.

La final cream un obiect de tip Scheduler, ii dam lista de task-uri si numarul de servere si pornim apoi simularea.

1. **Rezultate**

Nu s-a folosit nicio metoda speciala de testare.

1. **Concluzii**

Producătorul și Consumatorul sunt două procese separate. Ambele procese au un buffer sau o coadă comună. Producătorul produce continuu anumite date și le împinge în coada, în timp ce consumatorul consumă acele date din coada. La aceasta tema pot aparea multe probleme, daca producatorul produce prea mult nu va mai avea loc sa puna in coada si vas ta intr-o pauza, busy waiting, pana cand vor venii consumatorii sa consume. La fel se poate sa avem si consumatori care vor sa consume dar nu au ce deoarece producatorul nu produce sufficient, intrandu-se iar intr-o pauza, busy waiting.

1. **Bibliografie**

<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>

<https://howtodoinjava.com/java/multi-threading/atomicinteger-example/>

<https://www.baeldung.com/java-producer-consumer-problem>