**DOCUMENTATIE**

**TEMA 1**

**Queues Management Aplication**

**Nume: Ilovan Mara Gabriela**

**Grupa: 30222**

**CUPRINS**

1. Obiectivul temei ............................................................................................. 3

2. Analiza problemei, modelare, scenarii, cazuri de utilizare ............................ 3

3. Proiectare ........................................................................................................ 5

4. Implementare .................................................................................................. 7

5. Rezultate ......................................................................................................... 8

6. Concluzii ......................................................................................................... 9

7. Bibliografie .................................................................................................... 10

1. **Obiectivul temei:**

La aceasta tema a trebuit sa implementam o aplicație care gestionează cozile. Cu toții știm mecanismul de funcționare a unei cozi, acestea având utilizări in viața de zi cu zi. Un exemplu de utilizare al cozilor ar fi la magazin, unde avem mai multe case si clienții decid la ce coada se așază, alegând casa la care acestea cred ca așteaptă cel mai puțin. Astfel, aceasta tema a presupus proiectarea unei aplicații care administrează niște cozi, distribuie clienții la cozi astfel încât timpul de așteptare sa fie minimizat. Exista doua strategii posibile: clienții se așază la cea mai scurta coada, sau clienții se uita cate produse au persoanele din fata si se așază la coada la care timpul de așteptare este minim, indiferent de numărul de clienți de la coada.

Obiectivul principal al acestei teme a fost simularea a unei serii de N clienți care sosesc la anumite momente de timp la „magazin”, o coada de așteptare. Aceștia intra in cozi, așteaptă sa le vina rândul, sunt serviți, iar la final ies din coada. Managementul sistemelor ce au la baza cozi are ca scop minimizarea timpului, pe care clienții îl petrec la rând.

Un alt obiectiv al temei este învățarea si înțelegerea thread-urilor, scopul utilizării acestora. Astfel, aplicația are la baza thread-urile. Pentru fiecare coada deschidem cate un thread care rulează simultan, fiecare coada având un procesor asociat. Cu ajutorul thread- urilor reușim sa minimizam timpul. Alte obiective ar putea fi: reușirea implementării unei interfețe ușor de utilizat si posibilitatea unor dezvoltări ulterioare ale aplicației.

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

Cerințele funcționale sunt acelea care descriu funcționalitățile pe care programul trebuie sa lev îndeplinească, in timp ce cerințele non funcționale sunt cerințele care descriu programul.

Cerința problemei este: crearea unei aplicații care gestionează cozile: generarea unei serii de n clienți care ajung pentru a primii servicii la un „magazin” sau alt sistem care presupune cozi, un client este adăugat la coada in funcție de o strategie. Doua strategii sunt posibile: după numărul de persoane așezate la cozi, clientul se așază la coada cea mai scurta, sau după timpul de procesare al unui client. Timpul petrecut la coada de așteptare de către fiecare client, fiind monitorizat.

Alte aspecte care descriu aplicația sunt: aceasta trebuie sa ofere o performanta buna si sa răspundă la interacțiunile utilizatorului, aceasta trebuie sa fie, totodată, ușor de utilizat, aceasta ar trebui sa execute comenzile rapid.

Aceasta aplicație are 6 intrări: numărul de clienți, numărul de cozi, minimul timpului in care clientul ajunge la coada, maximul timpului in care clientul ajunge la coada, minimul timpului de procesare si maximul timpului de procesare. Daca apăsam pe buton ar trebui sa vedem răspunsul programului la datele introduse de utilizator. Răspunsul va apărea pe ecran in timp real.

Funcționalitatea aplicației e descrisa de diferite cazuri de utilizare a acestuia. In ceea ce privește aplicația de gestiune a cozilor funcționalitatea e redata de: introducerea datelor si apăsarea unui buton, care va pornii simularea. Daca datele sunt introduse greșit de către utilizator aplicația va trebui sa fie capabila sa ofere posibilitatea utilizatorului sa mai încerce pana reușește sa introducă toate datele corect, de tipul atributelor din interiorul claselor.

Scenariul principal este:

* Utilizatorul introduce valorile pentru: numărul de clinti, numărul de cozi, intervalele de simulare si sosire
* Utilizatorul apasă pe butonul de validare a datelor
* Aplicația începe sa ruleze conform datelor introduse de către utilizator. Sunt afișate datele pe ecran.

De asemenea in cazul in care sunt introduse date invalide este afișat un mesaj de eroare, iar aplicația revine la pasul 1. Scopul principal al acestui scenariu este de a permite unui utilizator sa configureze parametrii de intrare pentru a începe simularea.

Aceasta este diagrama de use case-uri:

O imagine care conține text, scrisoare

Descriere generată automat

**3. Proiectare**

Programarea orientata pe obiecte este o paradigma de programare ce are la baza conceptul obiectelor- unități de programare care conțin funcționalități asociate. Astfel, este permisa manipularea de obiecte pentru crearea mai multor sarcini in cadrul unui proiect. Acest tip de programare folosește conceptul de încapsulare al datelor, care permite ascunderea detaliilor unui obiect, fapt ce duce la protejarea datelor. Pentru a construi un program eficient, ușor de întreținut, cu o structura clasica si o organizare logica a datelor la prima tema vom folosi principii OOP.

Astfel, pentru acest proiect am folosit doua clase principale, care reprezintă modelul aplicației. Aceste clase conturează perfect ideea temei fiind specifice tehnicii de programare oop întrucât stochează elementele in clase cu nume sugestive care ne fac munca mai ușoara.

1. Clasa Task: aceasta clasa are ca atribute un id, arrivalTime si serviceTime. Aceasta clasa este corespondenta pentru un client. Acesta are un timp de sosire la coada de așteptare si un timp de procesare (cat durează ca acesta sa fie servit din momentul in care ajunge la coada).
2. Clasa Server: aceasta clasa are ca atribute o coada de task-uri si o perioada de așteptare. Aceasta este clasa in care intervin thread-urile, astfel avem nevoie de metoda run, metoda care va face posibila luarea unui client din coada.

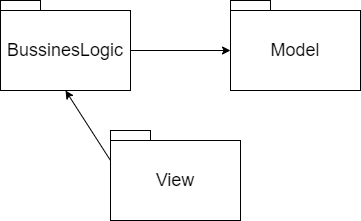
Proiectul va fi împărțit in mai mute pachete:

BusinessLogic - in care avem doua clase pentru strategii de așezare la cozi: ConcreteStrategyQueue, ConcreteStrategyTime, o clasa Scheduler si una SiumulationManager. De asemenea, in cadrul acestui pachet avem si o interfața Strategy, clasele ConcreteStrategyQueue si ConcreteStrategyTime implementad-o. Acest pachet reprezintă logica proiectului, aici se afla toate metodele ce stau la baza funcționarii proiectului.

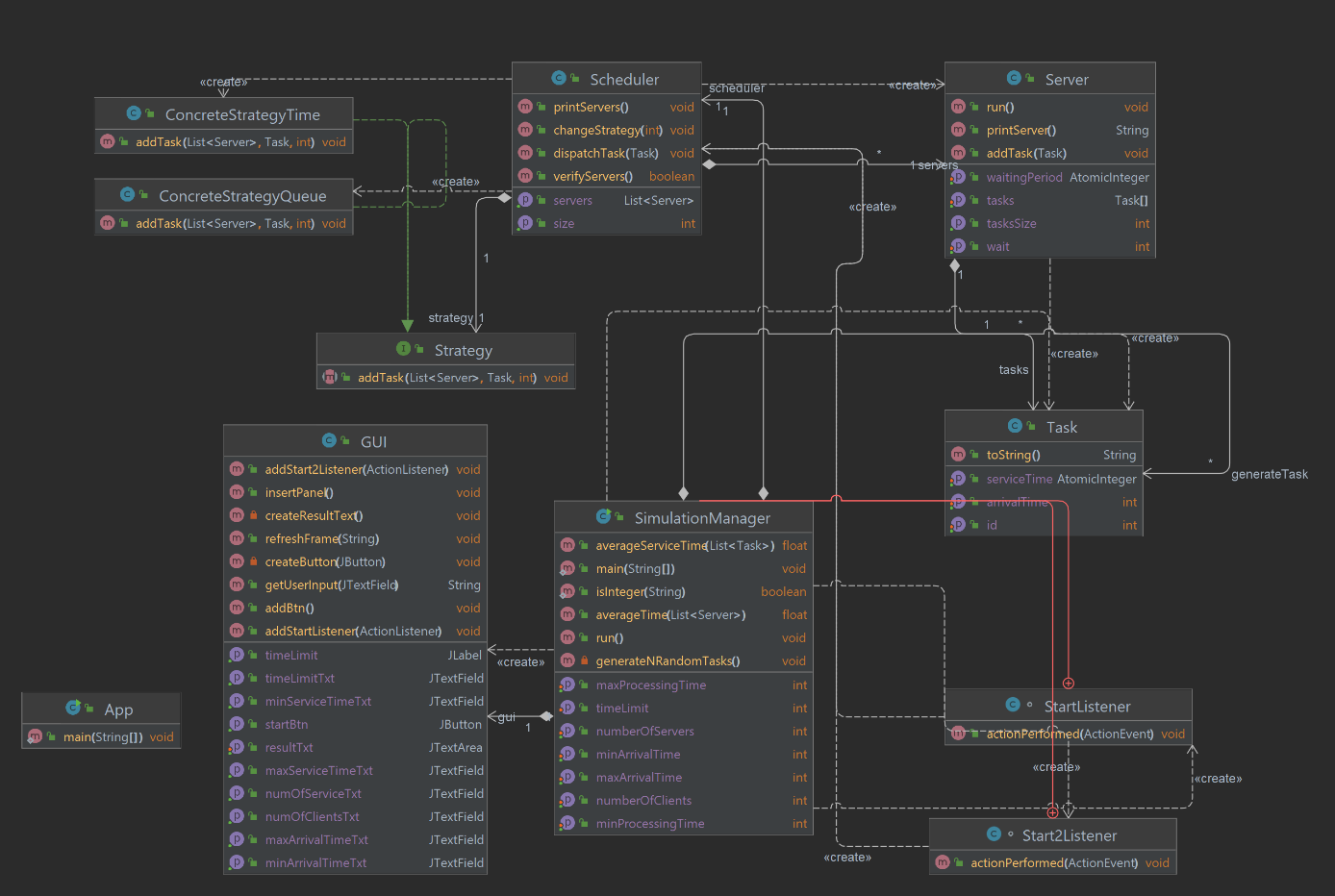
Model – in care avem Task si Server. Acest pachet reprezintă partea de gândire OOP, care ne determina sa împărțim proiectul in clase sugestive, in funcție de necesități. Astfel, principalele clase de care avem nevoie pentru întocmirea aplicației sunt: Task si Server, task fiind clasa de client, iar server reprezentând o coada la care clienții se așază.

View – in care avem clasa GUI, pentru interfața. Acest pachet reprezintă partea vizuala, văzută de utilizator, astfel, cu ajutorul acesteia utilizatorul vede funcționalitățile aplicației redate de celelalte pachete. Conexiunea dintre aceasta parte si partea de logica e făcută într-o clasa din BusinessLogic – SimulationManager, care este, totodată, si clasa principala a proiectului, aici pornim thread-ul principal, iar astfel pornește execuția programului.

Așa arata diagrama pe pachete:



Aceasta este diagrama pe clase:



**4.Implementare**

**1. Clasa Task:** aceasta clasa reprezintă un client si are doua atribute principale: timpul de sosire (arrivalTime) si timpul de procesare (serviceTime), de tipul int. Primul reprezintă momentul in care clientul va fi pus la o coada, in timp ce al doilea reprezintă cat va sta la coada. Totodata, aceasta clasa mai are si un atribut id, unic calculat automat la declararea unui nou client.

**2. Clasa Server**: aceasta clasa reprezintă o coada, aceasta extinde clasa Runnable, fapt ce o obliga sa folosească metoda run, metoda care va permite luarea unui client din coada. In metoda run thread-ul va face sleep pentru un număr de secunde corespunzător cu seviceTime-ul. Clasa are ca atribute un BlockindQueue Tasks (clienti) si un AtomicInteger waitingPeriod. Când serviceTime-ul va fi 0 vom șterge clientul din coada. Astfel, aici ne ocupam de fiecare coada in parte.

**3. Clasa Scheduler:** aceasta clasa creează un număr dorit de cozi, organzanizandu-le. Avem ca atribute o lista de servere, un număr maxim de servere si un număr maxim de clinti pe server. In constructorul acesteia am inițializat atributele si am pornit thread-urile, creând cate un Server pentru fiecare Thread. Aici vom apela metodele din strategii pentru adăugarea unui client la o coada.

**4. Clasa SimulationManager:** aceasta clasa este clasa principala, in care vom pornii thread-ul principal. Aici avem ca atribute: time limit, limitele timpului de procesare, limitele timpului de sosire, numărul de clienți, numărul de cozi, o lista de task-uri, care o vom genera random si un scheduler. Si aceasta clasa extinde clasa Runnable, fiind obligata sa implementeze metoda run. Aici vom afișa datele de ieșire in fișier, dar si modificarea conținutului din interfața grafica, in timp curent. Datele vor fi afișate după introducerea elementelor in interfața si după apăsarea butonului de start. In metoda run vom adaugă task-urile in cozi in funcție de strategia aleasa, strategie apelata prin intermediul dispatchTask, si care pate fi schimbata prin intermediul metodei changeStrategy, ambele metode fiind implementate in clasa Scheduler.

**5. GUI** – aceasta clasa este pentru formarea interfeței grafice: adaug label-uri, text field-uri, buton, dar si un text area, pentru scrierea datelor. Datorita acestei clase utilizatorul poate sa vadă funcționalitățile aplicației: poate introduce numărul de clienți, numărul de cozi, dar si intervalele timpilor in care clienții ajung la coada, sau timpii in care aceștia sunt serviți.

**6. ConcreteStrategyTime si ContreteStrategyQueue:** in aceste clase am implementat cele doua strategii de adăugare a clienților in cozi. O strategie se bazează pe timpul de așteptare la coada, iar cealaltă se bazează pe numărul clienților așezați la fiecare coada in parte, alegând in fiecare caz minimul.

**6. Rezultate**

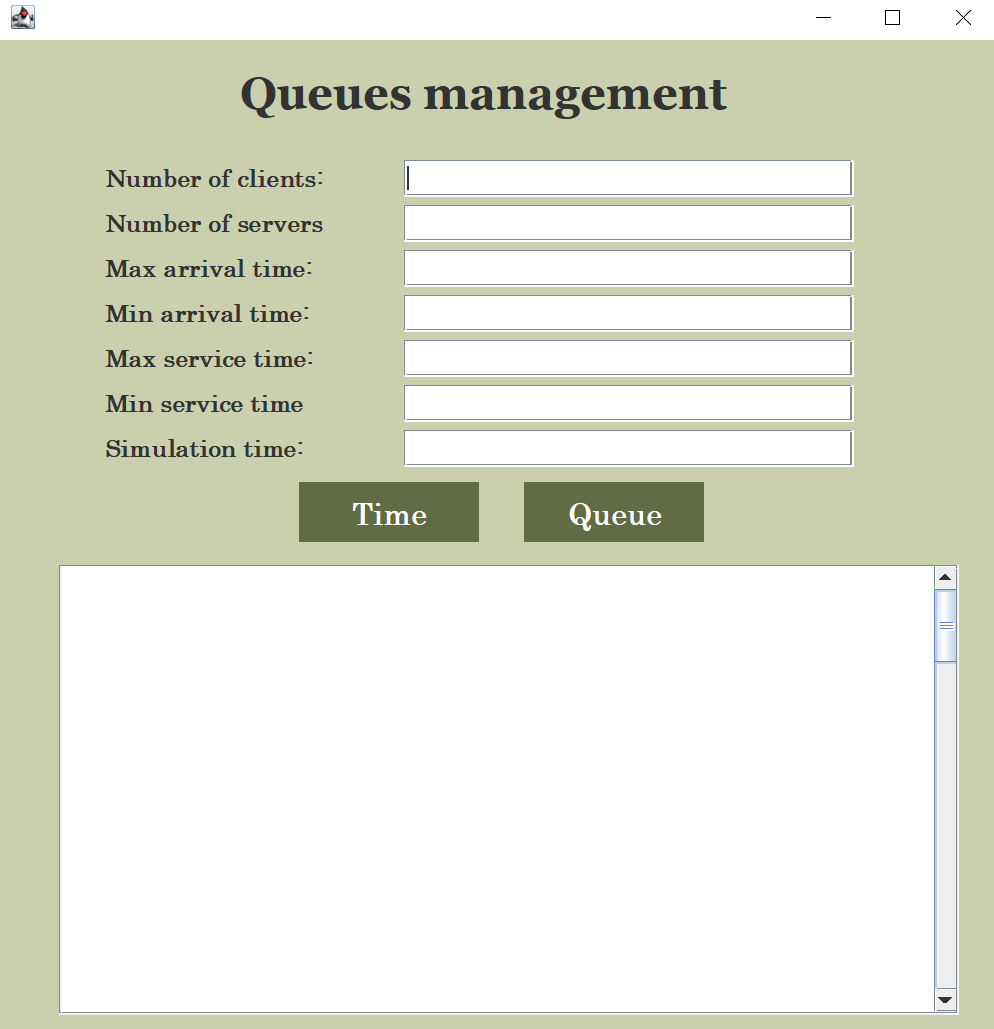
Pentru verificarea corectitudinii codului am folosit cele 3 exemple de inputuri date in prezentarea temei. De exemplu in primul test primeam patru clienți si doua cozi, un time limit de 60 de secunde, intervalul timpului de sosire [2, 30] si intervalul timpului de servire [2, 4]. După rularea programului cu fiecare dintre aceste date, observam faptul ca aplicația funcționează corect, conform așteptărilor. Totodată, aceste exemple m-au ajutat sa îmi dau seama daca media timpului de servire, ora de vârf si media timpului de așteptare la coada sunt aproximativ corecte.

De asemenea, am verificat funcționalitatea folosind si alte exemple date de mine pentru a vedea corectitudinea ambelor strategii (time si queue). Pentru a testa o aplicație cat mai bine trebuie sa ne gândim la toate cazurile posibile. De exemplu in cazul in care un utilizator începe sa folosească aplicația, sa ne gândim la faptul ca acesta poate introduce datele greșit, rezultatul fiind automat unul greșit. Astfel, daca acesta nu introduce datele de intrare ca fiind numere întregi, in caseta text rezultat va apărea un mesaj de eroare, iar utilizatorul va putea introduce datele încă o data.

La final aplicația va fi gata de utilizare si va arata așa:

In casetele text de la început datele de intrare pot fi introduse. Daca datele sunt greșite in caseta text de jos va apărea mesajul de eroare. Daca datele sunt introduse corect atunci vor fi afișate in timp real cozile si clienții din acestea, dar si clienții care urmează sa fie introduși in cozi.

De asemenea, aceste rezultate vor fi afișate si in fișierul text de ieșire.



**6.Concluzii**

Aceasta tema a avut ca scop principal înțelegerea thread-urilor. Acest fapt m-a obligat sa mă documentez înainte si sa învăț cum se folosesc acestea, faptul ca clasele care le folosesc trebuie sa extindă clasa Runnable si sa suprascrie metoda run, cum se scriu metodele run, când trebuie sa le dam sleep, si nu in ultimul rand când le dam start. Prin intermediul acestui assignment am reușit sa înțeleg thread-urile si sa îmi aprofundez cunoștințele de java, învățând noi structuri de date precum AtomicInteger.

Pentru realizarea temei a trebuit sa mă documentez si in privința afișării in fier, cum creez un fișier si cum îl închid.

Si aceasta tema ne-a arătat importanta structurării unui proiect in mai multe pachete. Astfel, codul devine mai lizibil si mult mai organizat, oricine se uita peste proiect poate, astfel sa înțeleagă ce am vrut sa fac. De asemenea, denumirile sugestive pentru clase si atribute, fiind si ele la fel de importante.

Totodată, tema ne-a învățat încă o data importanta citirii cu atenție a cerinței, sa nu ne apucam de întocmirea proiectului înainte de înțelegerea in totalitate a acesteia întrucât daca ne apucam prea repede am putea pierde timp pe chestii inutile. După citirea cerinței trebuie sa ne facem un plan de realizare a proiectului cu structurarea acestuia pe clase si pachete si legăturile dintre acestea.

**7.Bibliografie**

1. https://www.w3schools.com/java/java\_threads.asp - Java Threads
2. <https://stackoverflow.com/>
3. <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html> - Class AtomicInteger
4. <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html> - Interaface BlockingQueue
5. <https://www.javatpoint.com/> - Java Threads