DOCUMENTATIE

TEMA *2*

NUME STUDENT: HIRTESCU CIPRIAN-GABRIEL
GRUPA: 30225

CUPRINS

1. Obiectivul temei

Obiectivul principal al temei este de a analiza sistemelor bazate pe cozi e asteptare , prin simularea unei serii de n client care sosesc pentru a fi serviti , cozile sunt in numar de q , iar fiecare client asteapta sa fie servit ,iar in final paraseste coada ; calcularea timpului mediu de asteptare , a timpului mediu de servire si a orei de varf toate acestea fiind vizibile in cadrul unui panel in mod real-time astfel incat sa fie vizibila orice modificare de la nivelul cozilor si mai departe sa se poata face o oarecare verificare.

Obiectivele secundare sunt urmatoarele: analiza problemei si identificarea cerintelor functionale si a celor non-functionale , descrierea unei aplicatii de simulare , implementarea aplicatiei de simulare si testarea acesteia.

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

Caz de utilizarea al aplicatiei ar putea fi urmatorul : avand la dispozitie o aplicatie ce simuleaza modul de servire al unor clienti care pot astepta la cozi in vederea efectuarii unor operatiuni , utilizatorul care este principalul actor al folosirii aplicatiei are la dispozitie mai multe locatii de inserare a informatiilor ce vor fi folosite in interiorul aplicatiei astfel incat aceasta sa simuleze cat mai aproape de realitate o astfel de situatie posibila.
Astfel incat , user-ul insereaza date precum : numar de client ce vor fi asezati la cozi, numarul de cozi disponibile, apoi doua intervale descrise prin valoare minima si maxima , facand referire la timpul in care un client ar putea ajunge la coada , dar si la timpul in care acesta este servit.
Daca datele introduce sunt valide , in urmatorul moment incepe simulare propriu-zisa fiind vizibila in fereastra de interactiune cu utilizatorul astfel ca orice intrare , iesire sau modificare a unei cozi este observata. In cazul in care datele introduce nu sunt valide , va fi afisat un mesaj de informare , caz in care simularea nu va avea loc pana cand utilizatorul nu va introduce date valide , caz in care scenariul revine la cel anterior.

Printre cerintele functionale se pot enumera urmatoarele:
-> aplicatia de simulare ar trebui sa permita utilizatorilor sa introduca datele
-> aplicatia de simulare ar trebui sa permita utilizatorului sa porneasca simularea
-> aplicatia de simulare ar trebui sa permita vizualizarea pas cu pas a evolutiei starii cozilor.
-> aplicatia de simulare ar trebui sa permita rularea testului suport unu astfel incat sa fie vizibila o verificare necesara si consistenta de la nivelul structurii proiectului,
-> aplicatia de simulare ar trebui sa permita rularea testului suport doi astfel incat sa fie vizibila parcurgerea setului de informatii de validare in vederea rularii acestuia.
-> aplicatia de simulare ar trebui sa permita rularea testului suport trei astfel incat pe un caz mai larg de utilizare sa se garanteze succesul aplicatiei in cazul in care ar fi de plasat 1000 de clienti , lucru ce ingreuneaza intregul proces si poate conduce in unele cazuri la rezultate neasteptate sau chiar eronate .

Printre cerintele non-functionale se pot enumera urmatoarele:
-> aplicatia ar trebui sa fie intuitiva si usor de folosit
-> aplicatia ar trebui sa afiseze rezultatele corect
-> aplicatia ar trebui sa afiseze rezultatele intr-un interval de timp relativ mic

-> aplicatia ar trebui sa aleaga task-ul urmator de introdus in coada in functie de arrival time in comparatie cu timpul curent de simulare si mai departe in caz de egalitate sa se adauge la coada cea mai scurta abordarea mai este cunoscuta sub numele de concretTimeStrategy de la nivelul aplicatiei .

3. Proiectare

*Se va prezenta proiectarea OOP a aplicatiei, diagramele UML de clase si de pachete, structurile de date folosite, interfetele definite si algoritmii folositi (*daca e cazul)

Proiectarea OOP a plicatiei cuprinde o abordare bazata pe clase dupa cum urmeaza:

1. Clasa Task : este responsabila pentru identificarea unui client , principalele sale atribute cu care se remarca sunt ID-ul , un numar intre [1,n] , arrivalTime si serviceTime cele doua din final fiind generate aleator in vederea obtinerii unor rezultate diferite de la o simulare la alta , tocmai din acest motiv valorile primite din partea utilizatorului sunt intrunesc un interval generic , din care sa se poata extraga elemente diverse.

2. Clasa Server: tine evidenta clientilor care urmeaza sa intre la coada fiind o proiectie a cozii sub forma unui obiect , printre atributele sale se numara : waitingPerioad ce reprezinta perioada de asteptare de la nivelul unei cozi , alaturi de o lista de task-uri definita sub forma unui BlockingQueue in vederea mentinerii sincronizarii de la nivelul operatiunilor ce vor avea loc simultan.

3. Clasa Scheduler: va urmari serverele si va aplica una dintre strategiile : cel mai scurt timp sau cea mai scurta coada , astfel incat plasarea unui client la coada sa fie cat mai eficienta. Printre atributele sale se observa lista de servere/cozi alaturi de strategia respectiva ce va dicta modul in care vor evolua cozile.

4. Clasa SimulationFrame: construieste interfata cu utilizatorul , ascunzand intreaga logica din spate astfel incat utilizatorul sa aiba parte de o aplicatie usor de folosit si intuitiva spre a afla solutia la problema supusa a fi rezolvata. Aceasta este constituita din blocuri in care utilizatorul sa poata insera datele de intrare care ulterior vor fi procesate , iar in caz de validare , o data ce butonul „Start” este apasat , poate incepe executia aplicatiei , rezultatele intermediare fiind vizibile in blocul din dreapta. In caz de nevalidare, utilizatorul va fi atentionat ca datele intrare sunt nepotrivite , iar simularea nu va avea loc.

5. Clasa SimulationManager: este responsabila pentru intrunirea tuturor claselor vizibile mai sus astfel incat sa se ajunga la rezultatul asteptat. O particularitate care se remarca este fatpul ca in cadrul rezolvarii s-a propus utilizarea thread-urilor astfel incat la o executie sa se distribuie in mai multe actiuni simultane corespunzatoare fiecarei cozi.

4. Implementare

Se va descrie fiecare clasa cu campuri si metodele importante. Se va descrie implemantarea interfetei utilizator.

Implementarea presupune folosirea paradigmelor OOP astfel incat sa se ajunga la afisarea in real-time a modului in care pot fi asezati anumiti clienti identificati prin ID , arrivalTime si serviceTime la mai multe cozi , la anumite intervale de timp. Cunoscandu-se faptul ca un client poate intra la o coada daca arrivalTime-ul sau este mai mare sau egal decat timpul curent de simulare , el va parasi locatia de asteptare si se va aseza la coada disponibila , conform strategiei propuse . Mai departe referitor la thread-uri implementarea presupune folosire unui thread pentru cate o coada , si inca unul pentru main , ceea ce conduce la un numar de q+1 thread-uri , fiecare fiind responsabil de propria sa actiune asupra cozii de care apartine . Conform strategiei abordate, clientul se va aseza ori la coada cea mai scurta ori la coada care are timpul de service al clientilor aflati inaintea sa minim. Fiecare thread va executa acelasi tip de operatiune , iar in finalul executiei ajunge sa modifice cu succes datele aflate in scheduler astfel incat sa fie vizibila modificarea cozilor , al timpului de asteptare care scade la fiecare pas din simulare.
Toate modificarile efectuate de thread-uri la nivelul intregii structuri ierarhice a proiectului sunt vizibile atat in consola , cat si in fisierul out.txt , printre aceste aspecte se numara si faptul ca adaugarea sau stergerea unui task din lista va fi considerata ca fiind o modificare ce trebuie recunoscuta si tocmai din cauza acestui fapt in fisier apar insiruite toate aceste informatii ,intr-un mod sugestiv. Se cunoaste faptul ca un thread o data ce a fost creat nu isi incepe executia instant , ci ajunge intr-o stare New in care doar este pregatit sa execute operatiunile care ii sunt destinate de programator; wait , suspend , caz in care executia thread-ului curent este incetinita , oprita sau suspendata pentru o periada de timp determinata astfel incat un alt thread care proceseaza informatiile in paralel sa poata efectua de asemenea aceleasi operatiuni. In final, dupa finalizarea tuturor instructiunilor destinate din metoda, iar in caz de validare , o data ce butonul „Start” este apasat , poate incepe executia aplicatiei .Pentru a trece de la starea aceasta la starea Runnable in care poate executa operatiile de verificare a timpului curent si scadere a perioadei de servire total sau respectiv incrementare in cazul in care o persoana este adaugata la coada in vederea servirii. Etapa de Runnable este urmata de cea de Running in care se executa efectiv instructiunile aflate in metoda run() a clasei ce implementeaza interfata Runnable , aceasta fiind clasa Server , dupa care thread-ul poate intra intr-o stare de sleep , wait , suspend , caz in care executia thread-ului curent este incetinita , oprita sau suspendata pentru o periada de timp determinata astfel incat un alt thread care proceseaza informatiile in paralel sa poata efectua de asemenea aceleasi operatiuni. In final, dupa finalizarea tuturor instructiunilor destinate din metoda run() , thread-ul ajunge in starea Terminated ce anunta finalizarea cu succes a operatiilor.
Din punct de vederea al sincronizarii in cadrul implementarii s-a optat pentru folosirea obiectelor imutabile , adica obiecte ce o data ce au fost instantiate si au preluat valori in cadrul constructorului clasei , valoarea nu mai poate fi modificata ulterior, din acest motiv metodele de tipul set() asupra unor astfel de atribute nu au efect si nici sens. Pe langa aceasta abordare s-au folosit „colectii sigure” pentru thread de tipul blocking queue sau liste sincronizate ce permit unui singur thread sa poate actiona asupra unei astfel de colectii , aceast lucru fiind echivalent cu punerea unui „lacat” asupra colectiei la care un thread va efectua diverse operatiuni. O alta solutie de sincronizare a fost folosirea de metode si statement-uri sincronizate de exemplu pentru alegerea cozii in care se va pozitiona un client in cazul in care acesta are arrivalTime-ul mai mare sau egal decat timpul de simulare curent asta in cazul alegerii unei politici de selectie bazate pe timpul de service minim al tuturor cozilor. In cazul celeilalte tipuri de selectie bazate

pe lungimea cozii analog s-a optat ca ea sa fie sincronizata ,intrucat ea este responsabila cu adaugarea unui client la coada cea mai scurta de la un moment dat.
In final , pentru eliminarea inconsistentelor datorate scrierii simultane a unei valori in cadrul aceleiasi variabile s-a optat pentru folosirea variabilelor atomice de exe,plu AtomicInteger.
Fluxul de evenimente este vizibil real time atat in consola cat si in fereastra disponibila utilizatorului , iar rezultatele finale apar insiruite cronologic in cadrul fisierului out.txt . Pe langa evenimentele prezente in cadrul simularii , in final vor aparea si cei trei indici care specifica timpul mediu de asteptare al clientilor , timpul mediu pentru servire si timpul de varf in care au fost cei mai multi pozitionati la coada.

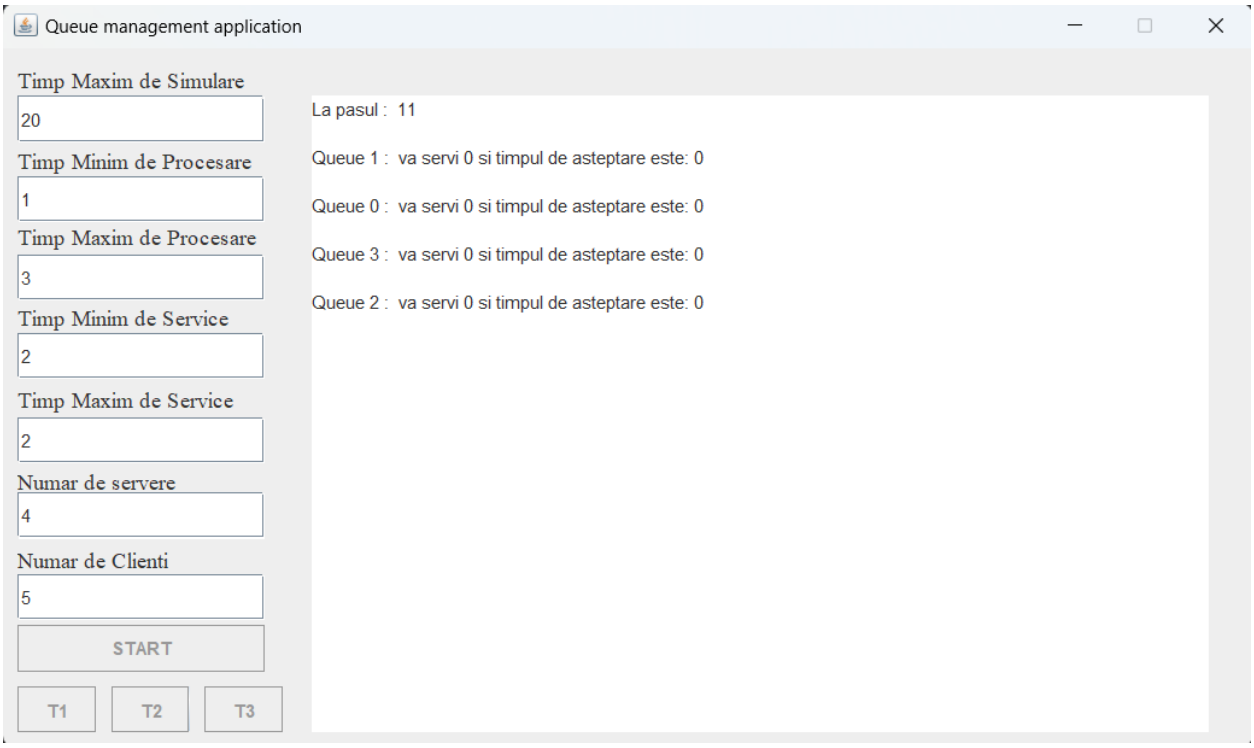
5. Rezultate

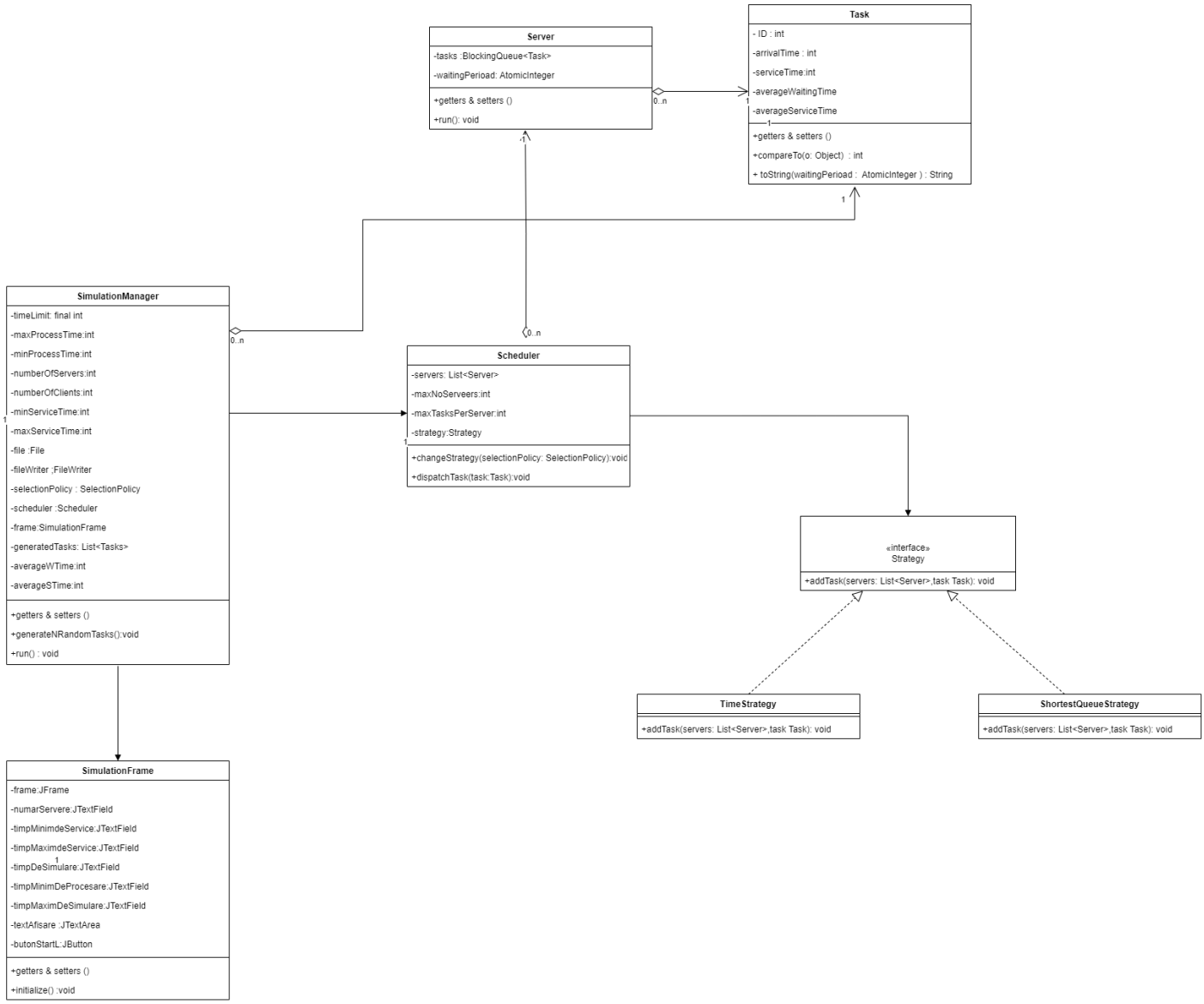
Pentru faza de testare s-a propus verificarea metodelor de interes ca de exemplu cele doua metode care adauga clienti la coada conform unei politici de selectii stabilite , dar si a metodei de simulare a rezultatelor in vederea verificarii ca totul decurge normal. Desi in cazul ultimei avand in vedere faptul ca valorile sunt generate aleator , iar la fiecare simulare este o probabilitate foarte redusa ca sa fie generate aceleasi valori , s-a optat ca variabilele sa fie hard-codate astfel incat sa se cunoasca arrivalTime-ul si serviceTime-ul fiecarui client ce va fi asezat ulerior la coada , astfel incat rezultatul sa poata fi verificat cu usurinta relativ la o abordare clasica pe foaie. In concluzie se poate spune ca aplicatia verifica toate conditiile de utilizare astfel incat sa poata sa afiseze modul in care pot fi plasati clienti in numar de n la cozi in numar de q.

6. Concluzii

In urma finalizarii proiectului s-a constatat ca o abordate de tipul multi-threading este foarte utila in vederea executarii in paralel a aceluiasi set de instructiuni.

7. Bibliografie




Server
-tasks :BlockingQueue<Task>
-waitingPeriod: AtomicInteger
+getters & setters ()
+run(): void
Task
- ID : int
-arrivalTime : int
-serviceTime:int
-averageWaitingTime
-averageServiceTime
+getters & setters ()
+compareTo(o: Object) : int
+ toString(waitingPeriod : AtomicInteger) : String
SimulationManager
-timeLimit: final int
-maxProcessTime:int
-minProcessTime:int
-numberOfServers:int
-numberOfClients:int
-minServiceTime:int
-maxServiceTime:int
-file :File
-fileWriter ;FileWriter
-selectionPolicy : SelectionPolicy
-scheduler :Scheduler
-frame:SimulationFrame
-generatedTasks: List<Tasks>
-averageWTime:int
-averageSTime:int
+getters & setters ()
+generateNRandomTasks():void
+run() : void
Scheduler
-servers: List<Server>
-maxNoServeers:int
-maxTasksPerServer:int
-strategy:Strategy
+changeStrategy(selectionPolicy: SelectionPolicy):void
+dispatchTask(task:Task):void
«interface»
Strategy
+addTask(servers: List<Server>,task Task): void
TimeStrategy
+addTask(servers: List<Server>,task Task): void
ShortestQueueStrategy
+addTask(servers: List<Server>,task Task): void
SimulationFrame
-frame:JFrame
-numarServere:JTextField
-timpMinimdeService:JTextField
-timpMaximdeService:JTextField
-timpDeSimulare:JTextField
-timpMinimDeProcesare:JTextField
-timpMaximDeSimulare:JTextField
-textAfisare :JTextArea
-butonStartL:JButton
+getters & setters ()
+initialize() :void
0..n
1