Documentație TEMA 2

THREADS

Nume: Câmpean Adelina Ioana

Grupa: 30227

Profesor Laborator: Pop Cristina

Cuprins

1. Cerințe Funcționale
2. Obiectivul temei
3. Informații despre THREADS
4. Analiza Problemei modelare, scenarii, cazuri de utilizare
   1. Diagrama USE CASE
5. Proiectare
   1. Diagrama de clase UML
   2. Diagrama de pachete
   3. Implementare
6. Rezultate
7. Concluzii
8. Bibliografie
9. Cerințe funcționale

Proiectați și implementați o aplicație de simulare care vizează analiza sistemelor bazate pe așteptare pentru determinarea și minimizarea timpului de așteptare al clienților.

1. Obiectivul temei

Obiectivul temei este de a realiza o aplicație de simulare, care va afișa în timp real, cum sosesc clienții (respectiv Task-urile) la cozi (respectiv Servere) și modul în care fiecare client are un anumit timp de sosire, generat aleatoriu, un anumit timp de procesare, de asemenea aleatoriu și totul se va afișa într-una (sau mai multe) interfețe grafice. Aplicația este realizată astfel încât pentru fiecare client, se va afișa într-un textArea, un mesaj care ne va spune la ce timp a ajuns clientul respectiv, cât a stat în coadă și respectiv la ce timp a părăsit coada.

Obiectivele secundare sunt realizarea unei interfețe grafice, realizată în clasa ViewI, care va afișa rezultatele, crearea claselor SimulationManager, care va prelua datele din interfață, Sever, care creează un server, Scheduler, care generează serverele, Task, care creează un task, Simulator, care rulează aplicația.

Obiectivele secundare ale temei sunt:

* Setarea intervalelor de sosire, respectiv procesare, a numărului de clienți și timpul de simulare, din interfața grafică
* Să ruleze câte un thread (fir de execuție) pe fiecare coadă
* Să afișam în interfața grafică, un timp mediu de așteptare la coadă

Obiectivele secundare ale temei vor fi prezentate în capitolul de Analiză a problemei, în care se va explica funcționalitatea fiecărei clase și se vor prezenta de asemenea, metodele importante, în care au fost îndeplinite cerințele.

1. Informații despre Threads

Un thread reprezintă un fir de execuție, care este folosit în momentul în care dorim să executăm mai multe lucruri simultan. Fiecare fir de lucru (execuție) își începe execuția la o locație stabilită, totul se execută ordonat, iar fiecare fir funcționează independent față de restul firelor de execuție din restul programului.

Într-un program se execută câte o singură linie de cod odată, se așteaptă finalizarea acesteia, răspunsul, apoi se trece la următoarea linie de cod. Programul nu poate răspunde acțiunii unui utilizator în acest timp, datorită programării single-thread.

Un fir de execuție este un program secvențial care poate fi executat concurent cu alte fire. Fiecare fir își execută codul, independent față de celelalte, astfel putându-se realiza mai multe comenzi în același timp. Un fir de lucru, este o unitate într-un proces. Un proces poate avea mai multe fire de execuție, care se numesc multithread.

Pentru a face ca o aplicație să îndeplinească o sarcină și să poată primi și altele în același timp, sunt folosite firele de execuție multiple (*multiple threads*).

Cu alte cuvinte, firele de execuție fac trecerea de la programarea secvențială la programarea concurentă. Un fir de execuție este o succesiune secvențială de instrucțiuni care se execută în cadrul unui proces. Diferența dintre un fir de execuție și un proces constă în faptul că un fir de execuție nu poate rula independent ci trebuie să ruleze în cadrul unui proces.

1. Analiza Problemei, modelare, scenarii, cazuri de utilizare
   1. Diagrama USE CASE

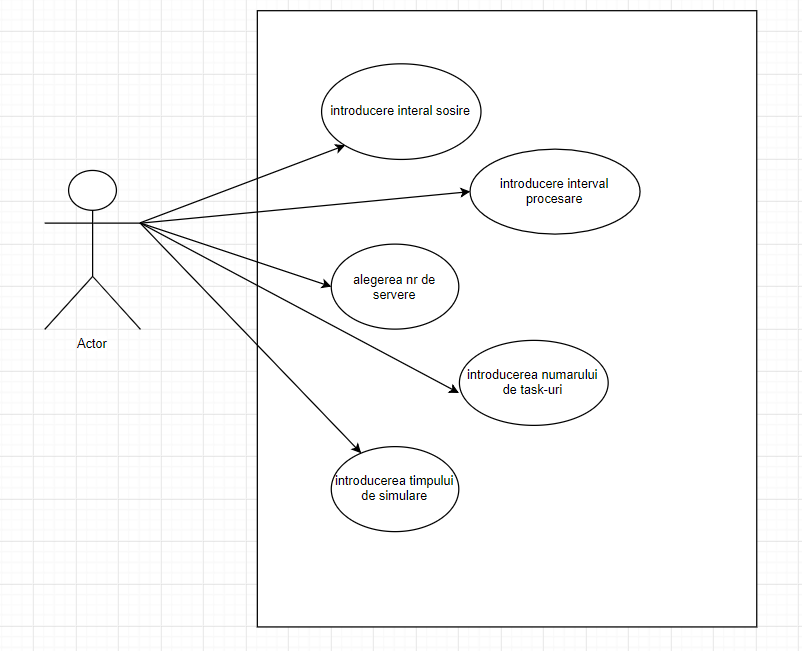


Diagrama USE – CASE este formată pentru a fi mai ușor de înțeles care este rolul aplicației.

Actorul este utilizatorul care introduce intervalele de timp în care să se genereze automat timpii de sosire si timpii de procesare pentru fiecare task și de asemenea apasă pe butonul start, pentru a începe rularea programului.

Scenariu de utilizare:

* Autorul introduce datele de la tastatură în interfața grafică, pentru a stabili în ce intervale pot ajunge task-urile și cât pot să stea la un server, adică cât vor avea timpul de procesare
* Apasă pe butonul start, moment în care se prelucrează datele și se execută programul, astfel încât în interfață, se va deschide o nouă fereastră, care va afișa cele 4 servere și se va observa în timp real cum sosesc task-urile și cum apar în textArea, toate informațiile necesare.
* Afișarea rezultatului in a doua fereastră

1. Proiectare
   1. Diagrama de clase UML

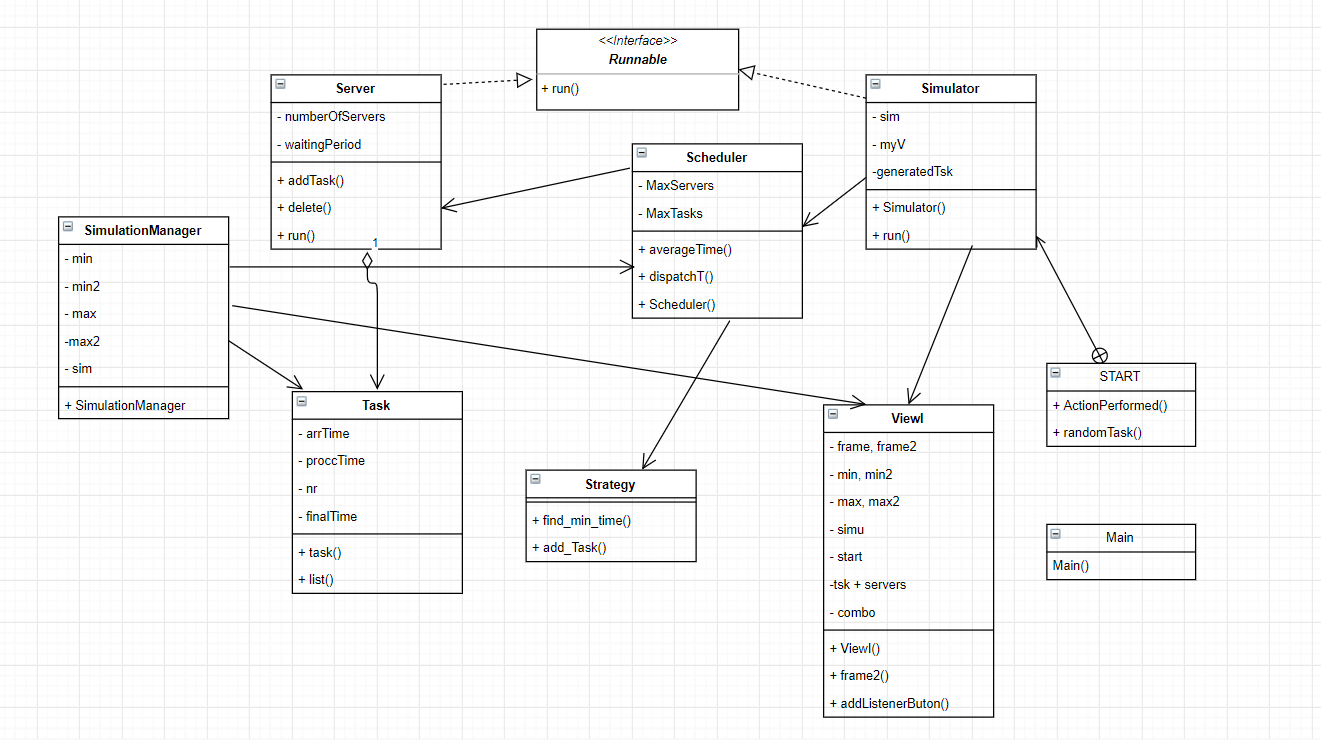


Diagrama UML este alcătuită din 8 clase: Server, Task, Scheduler, SimulationManager, ViewI, Simulator, Strategy si Simulator.

Clasele Server, Task, Scheduler și Strategy sunt implementate în interiorul pachetului Actions.

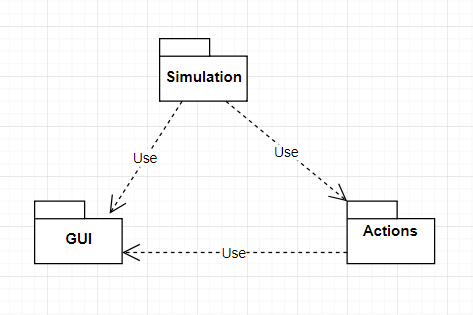
SimulationManager, Simulator și Main sunt realizate în interiorul pachetului Simulation.

Iar clasa ViewI aparține de pachetul GUI.

Fiecare clasă, are in componență un constructor, care este folosit pentru crearea unui nou obiect. În momentul în care instanțiem un nou obiect, acela va fi de forma constructorului declarat in clasa respectivă.

* 1. Diagrama de pachete

Aplicația a fost realizată în pachete, Simulation, Actions si GUI. Fiecare conținând clasele corespunzătoare.



* 1. Implementare

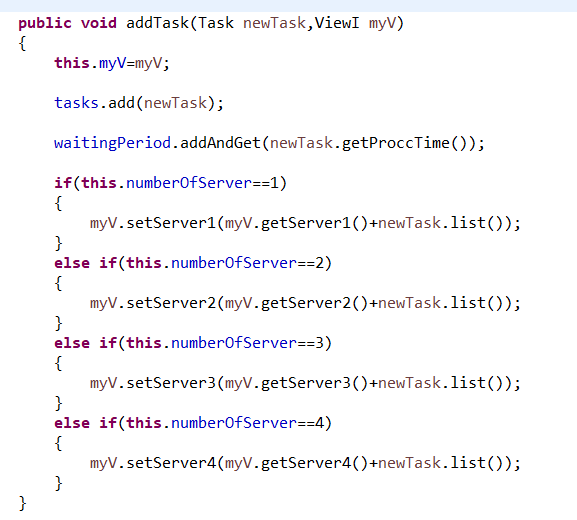
Clasa Task

În clasa task, am definit care sunt proprietățile fiecărui task (client), și anume, fiecare task are un timp de sosire (arrTime), timp de procesare sau servire (proccTime), un număr, prin care diferențiem task-urile în momentul simulării și un timp final (finalTime). De asemenea au fost realizate și mesajele care se vor afișa în interfața grafică.

Clasa Server

Clasa Server implementează interfața Runnable, deoarece fiecare server creat are câte un fir de execuție, care permite realizarea mai multor operații în același timp. Clasa conține proprietățile fiecărui server, și anume, fiecare server are un număr, pentru a fi diferențiat, un ArrayBlockingQueue de task-uri și un AtomicInteger care reprezintă timpul de așteptare pentru fiecare server. În cadrul acestei clase, se realizează o metoda addTask(), prin intermediul căreia se populează acel ArrayBlockingQueue și în același timp, se verifică numărul server-ului în care se adaugă task-urile respective pentru a fi introduse, respectiv setate și în interfața grafică, pentru a putea fi observabil, fiecare task la ce server este. Fiecare task stă la un server, atât cât îi este timpul de procesare, astfel încât, după terminarea acelui timp, task-ul respectiv va fi eliminat din server. Acest lucru este realizat in metoda delete() în clasa Server, în care se verifică din nou, numărul server-ului din interiorul căruia dorim să eliminăm task-ul, creăm un string auxiliar în care vom salva rezultatul. Apelăm getter-ul respectivului server și ștergem primul task din textField-ul respectiv, deoarece server-ul este de tip FIFO (First In First Out), astfel primul task din server va fi și primul care va fi eliminat.

Tot în cadrul acestei clase, este implementată și metoda run(), care pornește firele de execuție de pe servere. Într-un bloc try catch, thread-ul este setat sa doarmă (thread.sleep()) atât cât este timpul de procesare al task-ului activ, acela fiind timpul de servire al respectivului task. După scurgerea acelui timp, se actualizează timpul de așteptare al serverului și se elimină task-ul prin apelul metodei delete().

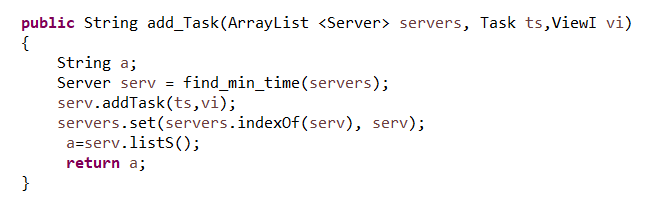
 Se creează de asemenea și mesajele care se vor afișa în interfața grafică.

Clasa Strategy

Este utilizată pentru a realiza împărțirea task-urilor în servere. Pentru acest lucru, am ales să împart task-urile în funcție de timpul minim de așteptare al serverului. Astfel, am realizat o metodă de gasire a minimului, find min time, care primea ca și parametru un ArrayList de Servere și la începutul metodei am declarat un minim, am parcurs arraylist-ul de servere și dacă timpul de așteptare al serverului (s.getWaitingPeriod(). intValue()) este mai mic decât minimul, atunci minim ia valoarea timpului de așteptare. Și totodată, se memorează într-o variabilă index-ul serverului cu timpul minim. Se verifică pentru fiecare server, iar la final se returnează serverul cu timpul minim, pentru a i se adăuga task-uri.

Se realizează și o metodă de add Task, în interiorul căreia se apelează metoda find min time, prezentată anterior. Astfel, in metoda add, se vor transmite ca parametri un arraylist de servere, un task și view-ul care se va actualiza. În interior, se apelează metoda fiind min time, pentru a alege în ce server se va introduce task-ul, apoi se apelează metoda addTask din clasa Server, pentru a se realiza adăugarea propriu zisă. Se returnează un string pentru a realiza afișarea în interfața grafică.

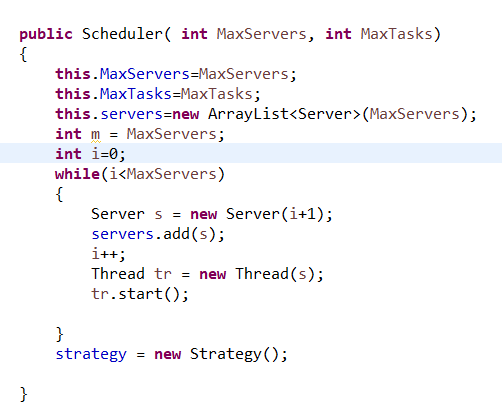
Se returnează un string pentru a realiza afișarea în interfața grafică.



Clasa Scheduler

Conține un arraylist de servere, o variabilă în care se salvează numărul maxim de servere, o variabilă cu numărul maxim de task-uri și se realizează un obiect de tip Strategy.

În constructorul acestei clase, se primesc ca și parametri nr maxim de servere și nr maxim de task-uri, care se inițializează. Se inițializează arraylist-ul de servere, apoi cât timp nu depășim numărul maxim de servere, creăm serverele, le adăugăm in arraylist și pornim firele de execuție cu metoda start(). Tot în această clasă se realizează metoda dispatchT(), care primește ca și parametru un task și apelează în interiorul metodei, prin intermediul obiectului strategy, metoda add task din clasa strategy în care trimit ca și parametri arraylist-ul de servere si task-ul. În această clasă se realizează și calculul timpului mediu de așteptare pentru servere, în metoda averageTime(), se aduna suma tuturor valorilor de așteptare, iar când se apelează se împarte la numărul de task-uri.



Clasa Simulator

Este o clasă care implementează interfața Runnable, astfel încât prin intermediul acestei clase, se realizează simularea propriu zisă a programului. Constructorul sincronizează view-ul, timpul de simulare si obiectul de tipul schedular cu cele din Simulation Manager, de unde sunt preluate datele. Apoi se implementează metoda run(), care verifică, dacă timpul curent este mai mic decât timpul de simulare, adică nu s-a terminat simularea, se parcurge array listul de task-uri si daca task-ul are timpul de sosire egal cu timpul curent, atunci se apelează metoda dispatchT, din clasa Scheduler, prin intermediul obiectului instantiat. Se setează și textArea, se afișează mesajele declarate anterior, pentru a observa evoluția simulării. Și se afișează și valoarea medie de așteptare, apelând metoda averageTime. Dupa fiecare task, se incrementează timpul curent.



Clasa SimulationManager

Este realizată pentru a prelua datele din interfața grafică. În interiorul acestei clase se implementează și o clasă internă, care implementează interfața ActionListener, astfel încât în momentul în care se apasă buton START se preiau toate datele, se generează aleatoriu intervalele de sosire a task-urilor si de procesare prin intermediul metodei random task care este apelată în interiorul constructorului. Apoi se apelează constructorul clasei Simulator, care primește ca și parametri arraylist-ul de task-uri, timpul de simulare și un obiect de tip schedular. Apoi se creează un nou fir de execuție, și se pornește simularea.

Clasa ViewI

Reprezintă interfața grafică propriu zisă, care e formată din 2 frame-uri. Primul frame se pornește la rularea programului și este format din mai multe textField-uri, care reprezintă intervalele pentru timpii de sosire si procesare și diferite label-uri sugestive. Sunt de asemenea și textfield-uri pentru timpul de simulare, un comboBox, pentru a alege pe cate servere se va realiza execuția, un textfield pentru timpul de simulare și un buton de START.

Al doilea frame, se deschide în momentul apăsării butonului de START, și constă în cele 4 servere generate (se puteau realiza și mai multe) și un textArea, în care se vor afișa toate mesajele, despre activitatea task-urilor.

1. Rezultate

Rezultatele sunt vizibile atât în textField-urile Server, acolo realizându-se evoluția în timp real, cât și în textArea, unde se poate observa, modul în care sunt introduse task-urile, în ce moment au sosit, cu ce timp de procesare și se specifică și în ce server au fost adăugați și de asemenea se afișează un mesaj și în momentul în care au părăsit serverul, specificându-se din ce server au plecat și care este timpul de finalizare al task-ului respectiv.

1. Concluzii

Prin intermediul acestei teme, am reușit să înțeleg mult mai bine modul în care funcționează firele de execuție și cât sunt de folositoare, la aplicațiile mai complexe, deoarece nu limitează programul să execute o singură instrucțiune la un anumit moment de timp.

Aplicația ar mai putea fi îmbunătățită prin adăugarea numărului de cozi, de către utilizator, dar fără a avea o limită, iar în momentul execuției, în frame, să apară doar atâtea textField-uri câte au fost alese de către utilizator, sa se genereze automat.

În concluzie, această aplicație simulează modul în care se realizează task-urile in cadrul unor servere, cum se distribuie task-urile în funcție de timpul minim de așteptare la fiecare server și modul în care se eliberează fiecare coadă la final de execuție, deoarece au fost procesate toate task-urile cu succes.

1. Bibliografie

* <http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/LabDescription.pdf>
* <http://users.utcluj.ro/~igiosan/Resources/POO/Curs/POO11.pdf>
* <https://stackoverflow.com/questions/363681/how-do-i-generate-random-integers-within-a-specific-range-in-java>