Documentatie – Tema 2

Iurieț Denis – Ionuț

Grupa: 30223

Profesor Laborator : Moldovan Sorin

Contents

1.Obiectivul temei

Sistemele dinamice se regăsesc peste tot în lume.. Fiind vorba de un sistem, componentele constituente vor interacționa continuu, și cel mai important, concurent.

Acest proiect redă într-o formă simplistă, un exemplu de astfel de sistem și anume o simulare a cozilor din magazine. În următoarele capitole, voi explica cum se poate simula o coadă, cum s-au calculat timpii medii de așteptare și eficientizarea sistemului, adăugând fiecare client nou venit la cea mai scurtă coadă, dar înainte trebuie sa înțelegem conceptul de multithreading.

Java este un limbaj de programare multi-threaded, ceea ce înseamnă că putem dezvolta un program multi-threaded folosind Java. Un program multi-threaded conține două sau mai multe componente care pot rula simultan și fiecare parte poate să se ocupe de o altă sarcină, făcând în același timp o utilizare optimă a resurselor disponibile, mai ales atunci când calculatorul are mai multe CPU-uri.

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

2.1 Analiza problemei

Dacă trebuie să ne gândim la o situație reală și aici mă refer la casele dintr-un supermarket, realizăm ca toate acțiunile (așezarea clienților la o anumită casa, așteptarea până ce ajung în capăt pentru a le fi scanate produsele și pentru a efectua plata, precum și ieșirea din coadă) sunt acțiuni ce se efectuează în paralel. Adică, să presupunem că avem doi clienți (‘x’ și ‘y’), clientul “x” de la casa 1, nu trebuie să aștepte după clientul “y” ce se află la casa 2.

În limbajul de programare Java, pentru a exprima cât mai concret o asemenea situație, avem posibilitatea de a folosi Threads, lucru pe care l-am detaliat în capitolul anterior.

2.2 Modelare

Modelarea aplicației se bazează pe conceptele programării orientate pe obiect, și anume:

Obiect = componentă software care incorporează atât atributele cât şi operaţiile care se pot efectua asupra atributelor şi care suportă moştenirea;

Clase - o clasă este o descriere generalizată a unui obiect. Un obiect este o instanță a unei clase. Clasa definește toate atributele pe care un obiect le poate avea și metodele care definesc funcționalitatea obiectului;

Încapsulare – unește informațiile importante ale unui obiect, dar restricționează și accesul la date și metode din exterior;

Moștenirea - permite claselor similare să se suprapună în mod ierarhic, unde clasele inferioare sau subclasele pot importa, pune în aplicare și reutiliza variabilele și metodele permise din clasele superioare immediate;

Polimorfismul – abilitatea de a lua mai multe forme, aceeași metodă folosită într-o superclasă poate fi suprascrisă în subclase pentru a da o funcționalitate diferită.

O modalitate bună de modelare a problemei este folosirea de cozi multiple asupra cărora am creat thread-uri, implementate ca o structură de date FIFO(First In, First Out), astfel încât să se poată exemplifica într-un mod cât mai real – primul client venit, primul servit.

2.3 Cazuri de utilizare

Aplicația prin intermediul interfeței permite introducerea unor parametri de intrare, o varietate largă de situații de simulare.

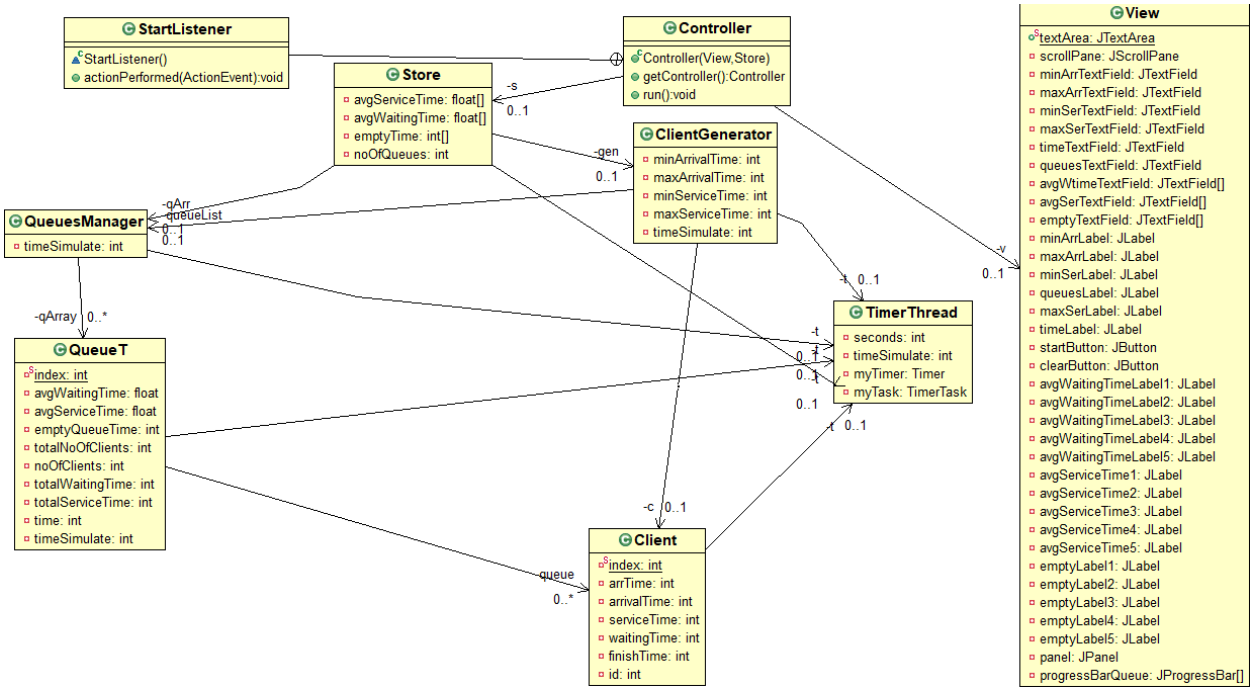
În rândurile următoare voi prezenta parametrii de intrare:

- Primii doi parametrii (minArrivalTime și maxArrivalTime) permit setarea unui interval de timp, din interiorul căruia se va selecta o valoare aleatorie ce reprezintă frecvența de apariție a clienților.
- Următorii doi parametrii(minServiceTime și maxServiceTime) au un rol asemănător cu cel al parametrilor de mai sus, cu deosebirea că valoarea aleatorie rezultată reprezintă timpul de așteptare a unui client, atunci când se află în capătul cozii.
- “Time to Simulate” – în acest text field, se va întroduce durata simulării în secunde.
- “NoOfQueues” – reprezintă numărul de cozi ce vor fi activate/deschise.

3.Proiectare

Aplicația “Store Simulator” are la baza 8 clase și o clasă main, fiind organizată conform șablonului Model-View-Controller (MVC). Acest tip de organizare reprezintă separarea interfeței utilizator într-o veder(View) care interacționează cu Modelul după nevoi, și un Controller care răspunde la cererile utilizatorului, interacționând atât cu View-ul, cât și cu Modelul.

3.1 Diagrama UML



3.2 Clasa Client

Această clasă conține atribute destinate clienților și aici ma refer la timpi de așteptare, de servire, timpul la care a ajuns clientul și la care a plecat, dar și un id pentru a-i putea identifica. Tot în acestă clasă s-au implementat si metodele de determinare a valorilor aleatorii pentru timpul sosirii și de servire.

3.3 Clasa QueueT

În cadrul acestei clase, s-a creat o colecție de tipul LinkedList în care s-au stocat clienții, simplificând operațiile de adăugare sau de ștergere, având implementate deja metodele (.add() și .remove()). Tot aici s-au declarat variabilele pentru stocarea timpilor medii de așteptare, servire și a duratei în care coada este goală. Am implementat un Thread pentru a putea șterge din mai multe cozi deodată, atunci când este cazul.

3.4 Clasa ClientGenerator

S-a implementat un thread și cu ajutorul variabilelor (minArrTime, maxArrTime, minServiceTime, maxServiceTime) s-au stocat valorile introduse în parametrii de intrare. Cu aceste variabile și cu thread-

ul implementat s-a realizat adăugarea clienților in cozi, făcându-se o verificare a cozilor determinând-o pe cea mai scurtă din punct de vedere a numărului de clienți.

3.5 Clasa QueuesManager

În această clasă avem un array de cozi pentru o utilizare mai ușoară a acestora și o metodă de determinare a celei mai scurte cozi, precum și o metodă de inițializare în funcție valoarea introdusă în parametrul de intrare “NoOfQueues”.

3.6 Clasa TimerThread

O clasă unde pur și simplu am creat un timer.

3.7 Clasa Store

Aici am adăugat un obiect de tipul ClientGenerator, unul de tipul QueuesManager și unul de tipul TimerThread pentru o organizare mai bună, de unde am pornit thread-ul de adăugare clienți și de creare a cozilor în funcție de numărul introdus. După cum spune și numele, în această clasă am dorit să fac o referire la un magazin propriu-zis (un magazin având un anumit număr de cozi, clienți etc.).

3.8 View

Aici se află implementarea interfeței, având cinci progress bar-uri pentru reprezentarea cozilor în cadrul simulării, un scrollPane cu un JTextArea introdus în care se afișează operațiile făcute la fiecare secundă, mai pe scurt aici am realizat un log., text field-uri pentru introducerea parametrilor de intrare și pentru afișarea parametrilor de ieșire si un buton “START” care pornește simularea.

3.9 Controller

Această clasă extinde de asemenea clasa “Thread” . Am declarat un obiect de tipul Store și unul de tipul View pentru a le putea face să colaboreze.

Am creat o clasă interioară cu un ascultător pentru butonul “START” creat in clasa View, unde am citit frecvența de apariție a clienților, timpii de servire aleatorii, timpul de simulare exprimat în secunde și numărul de cozi introduse în cadrul interfeței. Tot în cadrul acestui ascultător am pornit Thread-ul realizat în clasa Shop care pornește Thread-urile de generare a clienților și a cozilor, dar și Thread-ul din această clasă cu ajutorul unei metode.

Firul de lucru din clasa Controller, are rolul de a modifica text field-urile destinate timpilor medii de așteptare, de servire și a duratelor în care cozile sunt goale, dar și de a modifica progress bar-urile la fiecare adăugare sau ștergere a unui client dintr-o coadă.

4.Rezultate

Rezultatele acestui proiect pot fi văzute doar prin pornirea aplicației, pentru că numai în acest fel putem vedea cum se așează clienții la coadă(unde se alege cea mai scurtă coadă din punct de vedere a numărului prezent de clienți), așteptând să ajungă în capătul cozii, unde va aștepta un timp de servire aleatoriu și apoi va fi scos din coadă .
După ce am creat toate clasele pe care le-am descris în capitolul de mai sus și toate relațiile dintre ele, pot spune că aplicația pe care am dezvoltat-o simulează foarte bine un sistem bazat pe cozi. Acesta este construit pe baza scheletului primit la laborator și are o interfață simplă și ușor de utilizat.

5.Concluzii

Din această temă, am putut să văd concret, avantajul abstractizărilor încă din primele stagii ale dezvoltării aplicației. Aș putea afirma că am învățat importanța organizării timpului acordat unui proiect, structurarea și analizarea cu atenție a problemei înainte de începerea implementării acesteia în vederea găsirii unei rezolvări eficiente.

Posibile dezvoltări ulterioare pot fi considerate: îmbunătățirea interfeței din punct de vedere al aspectului, folosind imagini în loc de acele progress bar-uri pentru a reprezenta cozile, stabilirea unor excepții în ceea ce privește introducerea parametrilor de intrare și afișarea unor mesaje în cazul unor introduceri eronate.

În concluzie, cred că am respectat condițiile prestabilite și că acest proiect simulează o situație reală cât într-un mod cât mai concret. În ciuda afișării simpliste, aplicația descrie corect și inteligibil o situație reală.

6.Bibliografie

- http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- http://www.tutorialspoint.com/java/util/timer_schedule_period.html