TEHNICI FUNDAMENTALE DE PROGRAMARE

Simulator de cozi

*-JavaSE15, Java Swing-*

Realizat de:

Tanul Gabriel-Ștefan

an 2 grupa 30222

1. Problemă și soluție
2. Obiective
3. Analiză
   1. Thread-uri
   2. Pachete
      1. Model
      2. View
      3. Controller
4. Dezvoltare grafică
5. Testare
6. Posibilități de dezvoltare ulterioară

1.Problemă și soluție

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

Cozile sunt utilizate în mod obișnuit pentru modelarea domeniilor din lumea reală. Obiectivul principal al unei cozi este să oferiți un loc unde un „client” să aștepte înainte de a primi un „serviciu”. Scopul este să se minimizeze timpul pe care „clienții” îl așteaptă în cozi înainte de a fi serviți. O modalitate de a minimiza timpul de așteptare este de a adăuga mai multe servere, adică mai multe cozi în sistem (fiecare coadă este considerată ca având un procesor asociat), dar această abordare crește costurile furnizorului de servicii.

Aplicația ar trebui să simuleze (prin definirea unui timp de simulare 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛) o serie de N clienți sosind pentru serviciu, intrând în cozi Q, așteptând, fiind apoi serviți și în final părăsind cozile. Toți clienții sunt generați la pornirea simulării și sunt caracterizați prin trei parametri: ID (un număr între 1 și N), 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 (timpul de simulare când sunt gata să meargă la coadă; adică timpul când clientul a terminat cumpărăturile) și 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 (intervalul de timp sau durata necesară pentru a servi serviciul client; adică timpul de așteptare când clientul se află în fața cozii). Aplicația urmărește totalul timpului petrecut de fiecare client în cozi și calculează timpul mediu de așteptare. Fiecare client este adăugat la coadă cu timp minim de așteptare când timpul său 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 este mai mare sau egal cu timpul de simulare (𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 ≥ 𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛).

Următoarele date trebuie considerate ca date de intrare pentru aplicația care ar trebui inserată de către utilizator în interfața de utilizare a aplicației:

- Numărul de clienți (N);

- Numărul de cozi (Q);

- Interval de simulare (𝑡𝑠𝑖𝑚𝑢𝑙𝑎𝑡𝑖𝑜𝑛𝑀𝐴𝑋);

- Ora minimă și maximă de sosire (𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐼𝑁 ≤ 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 ≤ 𝑡𝑎𝑟𝑟𝑖𝑣𝑎𝑙 𝑀𝐴𝑋);

- Timp minim și maxim de service (𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒

𝑀𝐼𝑁 ≤ 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 ≤ 𝑡𝑠𝑒𝑟𝑣𝑖𝑐𝑒 𝑀𝐴𝑋);

Este costisitor din punct de vedere al timpului să se execute operații de adunare, scădere, înmulțire, împarțire, integrare sau derivare pe foaie, de mână, iar o soluție bună la această problemă ar fi implementarea unui calculator cu interfață grafică într-un limbaj de programare cunoscut. În cazul de față am folosit limbajul de programare Java și interfață grafică din Java Swing folosind mediul de dezvoltare **IntelliJ IDEA.**

2.Obiective

Scopul principal al acestui proiect a fost de a dezvolta și implementa un simulator de cozi de așteptare cu o interfață grafică unde utilizatorul poate introduce niste parametri ca text și poate alege să simuleze asteptarea la coadă a unor clienți vizualizând momentele de timp la care persoanele intră și ies din coadă. Pe lângă această noțiune de “simulator de cozi” voi prezenta cum am analizat eu problema, concluziile la care am ajuns și modul în care am ales să implementez și pun la dispoziție în interfață grafică toate funcționalitățile acestui simulator.

**Cum să implementez soluția?**



**Utilizare**: adunare, scădere, înmulțire, împărțire, derivare, integrare

**Principalul actor: utilizatorul**

**Scenariul cel mai probabil:**

1. Utilizatorul introduce de la tastatură parametrii corespunzători simulării sub formă de text.
2. Actorul principal confirmă introducerea datelor.
3. Calculatorul începe simularea și afișează într-o fereastră evoluția timpilor și a persoanelor care sunt generate aleator.

**Scenariul mai puțin probabil:** Date introduse incorect

* Utilizatorul introduce datele greșit (format incorect, litere, caractere speciale etc.)

Atenție!!! Formatul parametrilor trebuie sa fie numeric.

* În acest caz, se revine la pasul 1.

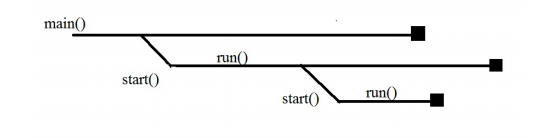
3.Analiză

3.1 Thread-uri

Un sistem care prelucrează date trebuie sa aibă niște intrări, o unitate de execuție care să realizeze anumite operații matematice și niște ieșiri care reflectă funcționalitatea dorită de utilizator. Cu alte cuvinte un simulator nu este altceva decât un sistem de prelucrare a datelor.

Pentru a realiza funcționalitățile simulatorului m-am folosit de conceptul de “Thread”.

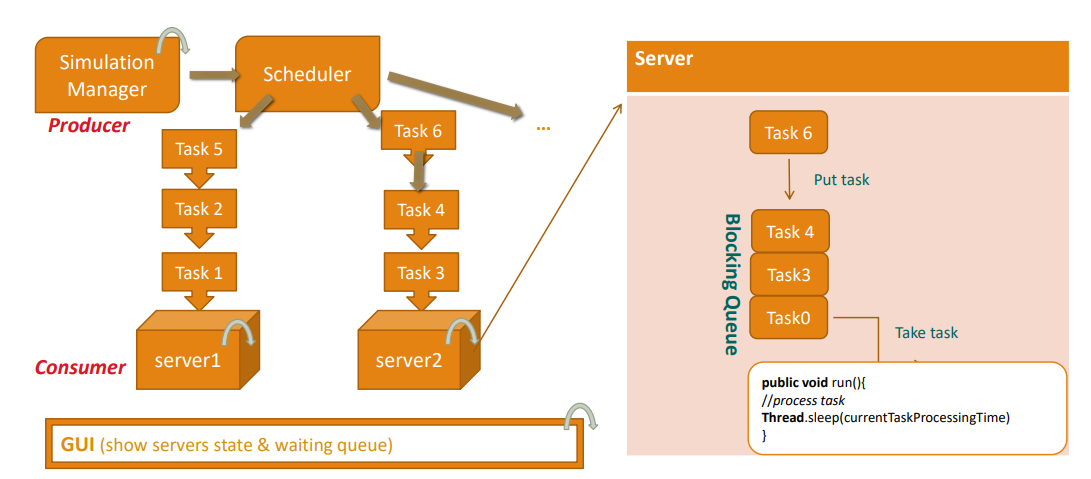
În Java, **thread-urile permit execuția concurentă a mai multor metode sau operații**.



De exemplu, mai multe case de marcat la un magazin trebuie sa ruleze concurent pentru a a nu exista asteptări foarte mari în cazul unui eșantion mare de persoane. Această problemă este rezolvată de acest simulator care permite rularea concurentă a uneia sau mai multe cozi care va executa o anumita operație ( ex.: scanare produse, elibarare chitanță etc.).

În cazul de față se simulează doar timpul petrecut de persoană la coadă, nu și alte facilități deoarece se dorește observarea și evidențierea execuției concurente.

Am ales să implementez proiectul pe modelul atașat mai jos.

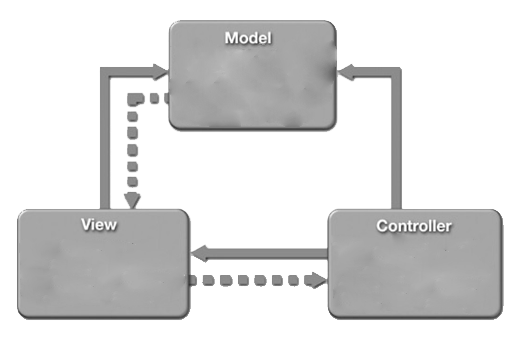


**Simulation Manager**-ul este cel care va trimite parametrii de simulare și va instanția scheduler-ul. Când **scheduler**-ul este instanțiat acesta începe și execută anumite task-uri. **Scheduler**-ul conține mai multe servere care execută concurent **task**-urile. Ca analogie, putem referi scheduer-ul la un magazin care are mai multe case de marcat (**server**) și care oferă suport mai multor clinți (**task**-uri).

3.2. Pachete

Modularitatea este foarte importantă în astfel de proiecte și cea mai bună soluție este de a structura în anumite părți utilitățile calculatorului, iar ca arhitectură am ales să folosesc **Model, View, Controller.**

În acest fel am reușit să separ partea efectivă de date care vor fi folosite în prelucrare într-un pachet numit **Model**, am implementat toate detaliile grafice în pachetul **View**, iar din **Controller**, după cum îi spune și numele am comandat toate operațiile și schimbările grafice care sunt necesare să apară și să actualizeze informațiile afișate pe ecran. Pentru o întelegere mai bună a acestui concept am atașat o imagine corespunzătoare.



**Server, Task, Strategy.**

**Partea grafică, frame-uri, excepții, afișare erori.**

**Comandarea tuturor schimbărilor interioare și grafice.**

Toate aceste 3 pachete au fost create în Java, iar în interiorul fiecărui pachet am introdus clasele corespunzătoare descrierii pachetului.

3.2.1. Model

În pachetul **model** am introdus clasele cu numele **ConcreteStrategyQueue, ConcreteStrategyTime, SelectionPolicy, Server, Strategy, Task.**

**Server** este clasa responsabilă de execuția task-urilor care primește un thread ce va permite o execuție concurentă. Are ca atribute o listă de task-uri și un waitingPeriod care depinde de task-urile arondate server-ului.

**Task** reține datele despre un anumit task care urmează a fi prelucrat.

**Strategy** este o interfață care conține o metodă de **addTask**() care practic decide unde se plasează un task în coadă în funcție de lungimea cozilor existente sau de timpul de asteptare al fiecarei cozi. (**ConcreteStrategyTime sau ConcreteStrategyQueue).**

**SelectionPolicy** este o enum și este regula după care se schimbă strategia.

3.2.2. View

Pachetul **View** conține toate elementele grafice ale acestei aplicații. Pentru a realiza o interfață grafică am folosi Java Swing. Acesta conține clasa **SimulationFrame.**

În clasa **SimulationFrame** am creat un thread separat,am adăugat un buton de simulare și mai multe căsuțe text și etichete specifice unde se introduc datele pentru simulare. **(Intervalul timpilor de asteptare/procesare, timpul total de simulare, numărul de cozi, numărul de clienți).**

Atenție!!! Formatul parametrilor trebuie sa fie numeric de tip **întreg**.

3.2.3. Controller

Ultimul pachet care este unitatea de execuție și de comandă a calculatorului este **Controller-ul** care conține clasele **Scheduler, SimulationManager și Main.**

**SimulationManager** este una dintre clasele cheie ale acestui proiect deoarece aici se controleaza și se manipulează datele date înspre execuție. Aici se instanțiază **scheduler**-ul care creează mai multe **servere** ( cu thread-urile corespunzătoare fiecărui server) și o strategie de adăugare într-un anumit server a task-ului. Se generează toate task-urile în mod aleator și se salvează într-o listă.

Extragerea datelor se face printr-un actionListener al butonului **simulate.**

*Am folosit funcția Integer.parseInt() care primește un text și returnează un int.*

**Main** este clasa ce conține metoda statică **main** și care face posibilă rularea intregului algoritm.

*Am folosit getters și setters pentru a prelua detalii sau a seta valori ale anumitor date din program, deci toate atributele claselor au fost declarate de tip* ***private.***

4.Dezvoltarea grafică

* SWING API
  1. Parte din clasele fundamentale java (JFC)
  2. Facilitează scrierea grafică a aplicațiilor java
  3. Include 17 pachete standard printre care cel mai folosit:
* **javax.swing** - Is the most important package from Swing

|  |  |
| --- | --- |
| **Tipul componentei** | **Examples** |
| Componente atomice | JLabel, JButton, JCheckBox, JRadioButton, JToggleButton, JScrollBar, JSlider |
| Componente complexe | JTable, JTree, JComboBox, JList, JFileChooser, JColorChooser, JOptionPane |
| Componente de tip text | JTextField, JPasswordField, JTextArea, JEditorPane, JTextPane |
| Meniuri | JMenuBar, JMenu, JPopupMenu, JMenuItem, CheckboxMenuItem , JRadioButtonMenuItem |
| Cointainere intermediare | JPanel, JTabbedPane, JDesktopPane |
| Containere principale | JFrame, JDialog |

Am folosit un container principal de tip JFrame pentru a ilustra fereastra principală a calculatorului iar apoi am creat un fel de ferestre de tip pop-up care apar în cazul excepțiilor și anume a erorilor prezentate mai sus.

Managerul de aspect este folosit pentru aranjarea într-o anumită structură a containerelor sau a componentelor.

a) BorderLayout – plasează componentele în cinci zone ale containerului principal: sus, jos stânga, dreapta și centru.

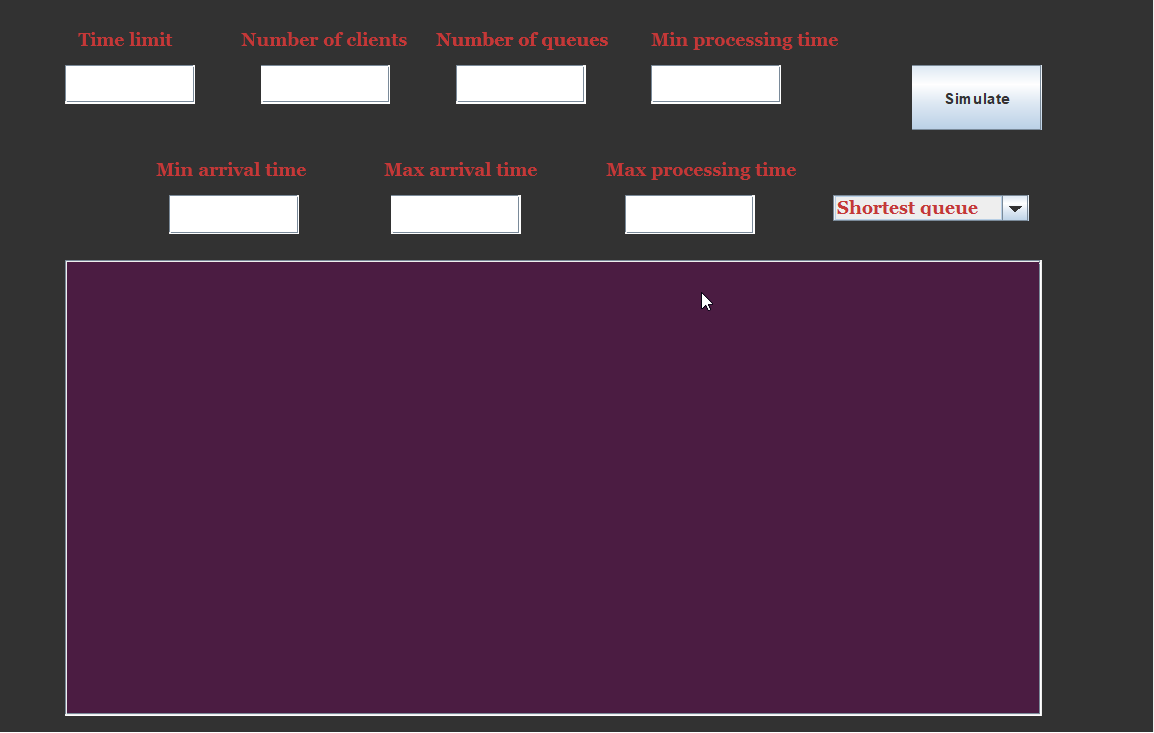
b) BoxLayout – plasează componentele pe o linie sau pe o coloană

c) FlowLayout – plasează componentele într-o singură linie dreaptă

d) GridLayout – plasează componentele într-un aspect matricial în mai multe celule

Am ales un GridLayout însă nu l-am folosit pe cel implicit ci am preferat să aleg eu coordonatele pentru componente deoarece a fost mai simplu să le așez exact în locul dorit.

Așa arată partea de grafică a calculatorului. Au fost implementate butoane pentru a alege operația dorită, pentru a confirma introducerea polinoamelor și etichete pentru a ilustra rezultatul sau restul polinomului final.



a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a

Alegere strategie

Eticheta parametri

Buton simulare

Fereastra de afișare

În cazul în care nu se respecta convenția introducerii datelor vor apărea mesajele de eroare corespunzătoare.

5.Testare

**Pentru N=4; Q=2; [tMinArrival,tMaxArrival]=[2, 30];**

**[tMinProcessing,tMaxProcessing]=[2,4];**

Așa ar arăta simularea pentru 5 secunde.

Time: 0

Waiting clients : (2,8,3) (1,10,3) (3,11,3) (0,25,3)

Queue-0 is closed

Queue-1 is closed

Time: 1

Waiting clients : (2,8,3) (1,10,3) (3,11,3) (0,25,3)

Queue-0 is closed

Queue-1 is closed

Time: 2

Waiting clients : (2,8,3) (1,10,3) (3,11,3) (0,25,3)

Queue-0 is closed

Queue-1 is closed

Time: 3

Waiting clients : (2,8,3) (1,10,3) (3,11,3) (0,25,3)

Queue-0 is closed

Queue-1 is closed

Time: 4

Waiting clients : (2,8,3) (1,10,3) (3,11,3) (0,25,3)

Queue-0 is closed

Queue-1 is closed

Time: 5

Waiting clients : (2,8,3) (1,10,3) (3,11,3) (0,25,3)

Queue-0 is closed

Queue-1 is closed

...

6.Posibilități de dezvoltare ulterioară

Ulterior se pot alege anumite elemente grafice care simulează o animație grafică 2D. Se pot face mutări de pătrate care semnifica intrarea și ieșirea din coadă și se poate semnala fiecare coadă cu niște culoare care conțin la începutul fiecăruia un pătrat verde corespunzător cozii libere și un pătrat roșu corenspuzător cozii închise.

Se poate de asemenea adăuga mai multe atribute task-ului ( produse de adăugat în coș, numele clientului etc.).

Bibliografie

[1] https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

[2] https://docs.oracle.com/javase/8/docs/api/java/util/Timer.html

[3] http://www.tutorialspoint.com/java/util/timer\_schedule\_period.html

[4] http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-andthreadpoolexecutor.html

[5] B. Goetz et al., Java Concurrency in Practice, Addison-Wesley Professional; 1 edition (May 19, 2006)