

Gestiunea Clienților

Thread-uri și execuția concurentă



Cuprins

[**1.** **Introducere** 2](#_Toc448328790)

[**2.** **Analiza problemei și modelarea acesteia** 3](#_Toc448328791)

[**2.1.** **Cazuri de utilizare** 4](#_Toc448328792)

[**2.2.** **Scenarii** 4](#_Toc448328793)

[**3.** **Proiectare** 5](#_Toc448328794)

[**3.1.** **Clasa GUI** 6](#_Toc448328795)

[**3.2.** **Clasa Client** 7](#_Toc448328796)

[**3.3.** **Clasa Controller** 8](#_Toc448328797)

[**3.4.** **Clasele MultimeClienti, MyPanel, Casa** 9](#_Toc448328798)

[**3.1.** **Rezultate** 10](#_Toc448328799)

[**4.** **Concluzii** 11](#_Toc448328800)

[**5.** **Bibliografie** 12](#_Toc448328801)

1. **Introducere**

Sistemele dinamice, cu componente ce interacționează se regăsesc peste tot în lume. În realitate, ele sunt atât de comune, încât experiența de zi cu zi ne face să le trecem într-u totul cu vederea. Fiind vorba de un sistem, componentele constituente vor interacționa continuu, și cel mai important, concurent. Concurența este un proces vital în orice sistem natural.

Acest proiect își propune să pună în lumină tocmai un exemplu de un astfel de sistem și anume exemplul trivial al cozilor la magazine. Cum poate fi simulată o coadă/cozi multiple și care sunt timpii de acces la diferitele componente, toate acestea sunt întrebările la care voi încerca să găsesc o soluție generică care va putea fi reaplicată pe diferite date în funcție de parametrii de intrare.

Simularea concurenței pune totuși o serie de probleme pentru calculatoarele cu arhitecturi von Neumann, tocmai din liniaritatea modului lor de funcționare. În absența unei arhitecturi multiprocesor, există totuși metode prin care se pot păcăli privitorii, dând impresia unei execuții concurente.

Java folosește pentru execuția concurentă thread-uri. Aceastea sunt de fapt mini procese care se execută în interiorul unui proces principal ce sunt gestionate la un nivel logic inferior de către un manager de procese.

1. **Analiza problemei și modelarea acesteia**

Prin analiza problemei, ne referim la un prim set abstract de operații și proprietăți prin care încercăm să depistăm eventualele însușiri și comportamente ale proceselor necunoscute. Programarea orientată ne oferă aici un avantaj clar, tocmai fiindcă ea permite să taclăm problema de la un nivel superiror, fără a mai fi constrâși, într-o așa măsură, de caracteristicile tehnice.

Această strategie de conceptualizare, mai poartă numele și de bottom-up design. Este foarte avantajoasă din prisma găsirii componentelor constituente, deoarece pot fi găsite, relativ ușor, structuri cu o legătură directă în lumea reală( obiecte, acțiuni etc.). Din păcate această versatilitate vine cu prețul complexității, ea crescând spre măsură ce se avansează pe nivelele inferioare.

De cele mai multe ori se pornește de la specificația proiectului, căutându-se:

* Substantive, care devin eventuale clase candidat
* Verbe ce ar putea juca rolul metodelor din clasă.

Odată realizat pasul de mai sus, ar trebui să avem o idee foarte generală asupra problemei. Pasul următor constă în descrierea funcțională a acesteia. Ce trebuie să facă aplicația la intrările X?

Programul va putea fi accesat de un număr ridicat de persoane, de aceea interfața cu utilizatorul devine punctul de pornire al proiectului. Ea trebuie să permită, într-o manieră cât mai convenabilă, comunicarea utilizatorului cu aplicația.

În cazul “Gestionarea Clienților”, se cunoaște că aplicația trebuie să implementeze o serie de operații elementare, dar concurente: ne vom dorii ca toți clienții să se poată mișca pe ecran, să se oprească când vor ajunge la case, să simuleze un timp de așteptare, iar apoi să își continue drumul. De asemenea, această mișcare va presupune existența unui alt proces concurent prin care să redesenăm fereastra de drawing. Clienții vor trebui să aibă o oarecare dependență, in sensul că va trebui să existe o modalitate prin care toți să *“vadă”* că una din case este prea plină și să mearga la alta.

În cadrul GC(Gestiunea Clientilor), majoritatea feedback-ului se va face vizual prin intermediul diverselor animații, dar și prin intermediul unui logger, ce reprezintă un JtextArea unde sunt adăugate diferetele informații, în timp real.

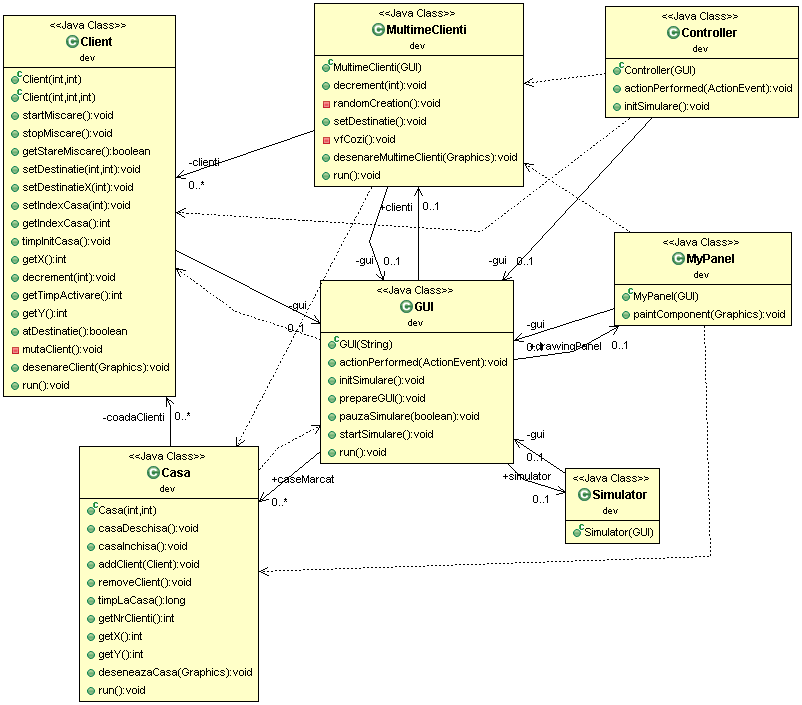
* 1. **Cazuri de utilizare**

Aplicația permite, prin intermediul parametrilor de intrare o gamă foarte largă de rezultate. O mare atenție am acordat la găsirea unui set de parametrii cât mai redus, dar cu un impact cât mai semnificativ asupra întregii simulări:

* Primul parametru, Durata Simulare permite setarea unui timp în care să se desfășoare toată aplicația. Odată terminat timpul simulării, întreaga aplicație va trebui să se oprească.
* Frecvența de apariție a clienților este setată cu ajutorul celui de-al doilea câmp. Ea ține locul gradului de aglomerare a unui magazin. Setând o valoare mare, puțini clienți vor fi generați și implicit, șansa pentru a vedea o coadă scade vertiginos.
* Următorul câmp, Timpul de Activare, ține locul timpului de ședere a unui client în magazin. O valoare mare și ne vom aștepta, în medie ca toți clienții să vină după o valoare proporțională la case.
* Ultimul câmp reprezintă o greutate generică a unui coș. Toți clienții vor avea aceeași valoare generică, diferite valori simulându-se doar la clasă. Deși nu este o abordare cu totul bună, este foarte ușor de implementat fiind vorba de o serie de proprietăți și metode statice.
  1. **Scenarii**

Utilizarea normală a programului presupune setarea unor parametrii consistenți cu un scenariu normal de utilizare. La această aplicație interacțiunea între utilizator și GUI este redusă, programul fiind autosustenabil. Am observat că la anumite valori mari, când trecem de un număr semnificativ de thread-uri, apar probleme de sincronizare la afișarea corectă a cozilor.

1. **Proiectare**



Calculatorul de Polinoame conține 5 clase de bază și o clasă main ce le instanțiază.

1. Clasa **GUI** – Reprezintă clasa principală a aplicației și este locul unde se fac asocierile cu celelalte clase. Aici se află thread-ul principal al programului cu care se redesenează toate elementele Swing ale programului în conjuncție cu clasa MyPanel care suprascrie metoda paintComponent(). În această manieră se asigură redesenarea corespunzătoare a tutoror componentelor participante la animație.
2. Clasa **Client** – această clasă conține setul de date ce caracterizează un client(poziția pe ecran, starea de mișcare, dacă s-a ajuns la destinație etc.) și o serie de date adiționale în care se stochează diferiții timpi de acces la celelalte componente.
3. Clasa **Controller** – este o clasă de mici dimensiuni în care se definesc metodele care trebuie inițializate la activarea butonului de OK. Aici se pornește întreaga aplicație și setarea prin intermediul unor metode statice a unor constante din care se pot deriva valori apropiate. În esență, aici se definesc expectanțele valorilor de simulat.
4. Clasa **Casa** – cea mai importantă proprietate este vectorul de Clienti și metodele de adăugare și de scoatare a clienților din vector. La inițializarea programului se apelează metoda wait() pentru nu începe metoda run ce ar da de altfel eroare deoarece ar fi vorba de un empty vector.
5. Clasa **MultimeClienti** – inițial s-a dorit a fi o simplă colecție concurentă, dar s-a dovedit a fi nevoie de mai multe metode sincrone fiindcă s-a dorit un anumit grad de interdepență între elementele colecției.
6. Clasa **MyPanel** – extinde Jpanel și suprascrie metoda paintComponent(). Clasa e mai degrabă un Helper class pentru GUI și ajută la redesenare.
7. Clasa **Simulator** – generează facturi în format .pdf folosind librăria Itext, cea mai populară librărie de la această dată în ceea ce privește gestiunea documentelor .pdf

Pe lângă tipurile de date primitive existente în clasele specificate mai sus, se folosesc cu preponderență și colecții. Una din cele mai importante colecții se află în clasa Polinom, ce conține o listă de monoame. Desigur, există o de compoziție între un polinom și elementele sale fundamentale, monoamele – în absența lor, o instanță Polinom nu există.

În ansamblu, există o similitudine între modul de proiectare a claselor și MVC pattern. Panel joacă rolul unui View, PolyInterpreter este Controller-ul și în final Polinom, Monom și Cartezian reprezintă clase model.

* 1. **Clasa GUI**

Conține un Jframe pe care se construiește întreaga interfață.

mainFrame = **new** JFrame("Tabbed Pane");

mainFrame.setSize(1000, 600);

mainFrame.setLayout(**new** GridLayout(1,1));

mainFrame.setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

În partea de control, există câteva segmente de cod de aceeași natură, care se repetă cu mici variații. Această repetiție, care se abate de la conceptul DRY, este o consecință a faptului că folosim un JTabbedPane, cu trei tab-uri distincte, dar care păstrează același antet și aceeași manieră de așezare în pagină.

Încă de la început, m-am decis să folosesc GridBagLayout, pentru a face întreaga aplicație responsive. Spre deosebire de primul proiect, această aplicație permite redimensionarea la orice valori a ferestrei, fără a se pierde poziția relativă între elemente.

GridBagLayout layout = **new** GridBagLayout();

jp4.setLayout(layout);

GridBagConstraints gbc = **new** GridBagConstraints();

gbc.fill = GridBagConstraints.***HORIZONTAL***;

gbc.gridx = 0;

gbc.gridy = 0;

gbc.weightx = 0.5;

JLabel label1 = **new** JLabel();

label1.setText("<html><p style='font-size:20px;font-weight:200'>Produse</p></html> - adaugare, editare si stergere"); jp4.add(label1, gbc);

În snippet-ul de mai sus se poate remarca structura de bază prin care s-au adăugat elementele. Se poate observa că înainte de fiecare apel la add, se reinițializează un set de constrângeri prin care putem să controlăm modul de așezare în pagină și afișarea elementelor.

Pentru a transmite anumiți parametrii clasei Controller se foloșește o mnemonică similară cu cea de mai jos.

<component>.setActionCommand("example");

<component>.addActionListener(**new** Produs(**this**));

* 1. **Clasa Client**

Este clasa care gestionează toate acțiunile venite din partea utilizatorilor și le corelează cu funcționalitățile preexistente. Tocmai de aceea, această clasă implementează interfața **ActionListener** și implicit metoda **actionPerformed()**. Există astfel o singură metodă care gestionează toate evenimentele venite din partea de view și asta fiindcă nu avem nevoie de un paralelism la nivelul tratării acțiunilor – în sensul că utilizatorul nu va executa două sau mai multe acțiuni în mod simutan.

Caracteristica definitorie a clasei Controller este că ea gestionează toate evenimentele asincrone. De aceea, ea nu joacă un rol important decât atunci când utilizatorul dorește să interacționeze cu aplicația. Informația este neschimbată atâta timp cât utilizatorul nu actualizează datele din text field-uri, dar ea poate fi reîmprospătată. Prin reîmprospătare nu înțelegem doar modificări de la nivelul aplicației strict în timpul execuției, ci orice fel de modificări, care au fost făcute în timp ce aplicația rula sau la alt moment de timp. Reîmprospătarea este o acțiune generală de actualizare a unui model de date și abea apoi de redesenare a vreunui tabel sau alt element grafic.

Ne rămâne doar să “selectăm” între diferitele acțiuni folosindu-ne de ActionCommand setate în clasa TabbedPane.

Indiferent de tipul acțiunii, la fiecare nou apel primul pas pe care îl facem este să îi depistăm sursa și mai apoi să îi asignăm o funcție de gestionare. Funcțiile de gestionare s-au dovedit a fi destul de specifice, lucru ce a cauzat o dificultate sporită la implementarea acestora.

**private** **static** **void** mkClient() {

JTextField field1 = **new** JTextField("");

JTextField field2 = **new** JTextField("");

JTextField field3 = **new** JTextField("");

JPanel panel = **new** JPanel(**new** GridLayout(0, 1));

panel.setPreferredSize(**new** Dimension(400, 200));

panel.add(**new** JLabel("Nume: "));

panel.add(field1);

panel.add(**new** JLabel("Email: "));

panel.add(field2);

panel.add(**new** JLabel("Telefon: "));

panel.add(field3);

**int** result = JOptionPane.*showConfirmDialog*(**null**, panel, "Adauga un client",

JOptionPane.***OK\_CANCEL\_OPTION***, JOptionPane.***PLAIN\_MESSAGE***);

**if** (result == JOptionPane.***OK\_OPTION***) {

ArrayList<String> data = **new** ArrayList<String>();

data.add(field1.getText());

data.add(field2.getText());

data.add(field3.getText());

Client nouClient = **new** Client(data);

System.***out***.println("Un nou client a fost adaugat! \n");

} **else** {

System.***out***.println("Cancelled");

}

}

Este de remarcat că print-un singur apel la clasa Client, cu ajutorul clasei de Serializare putem să creăm un nou fișier Client, să actualizăm modelul de date, dar lăsăm pe seama utilizatorului opțiunea de reîmprospătare.

* 1. **Clasa Controller**

Împreună cu clasa Controller și TabbedPane, clasa Serializable este una din clasele de bază, necesară aproape continuu clasei TabbedPane. Principalul rol al acestei clase este să creeze modele de date și să serializeze, respectiv deserializeze fisierele în care se află informația despre Depozit.

Clasa conține doar metode statice, de o generalitate crescută, tocmai pentru a facilita polimorfismul și mai ales pentru a crea o platformă echitabilă între clase și pentru a nu obține disconcordanțe la nivelul mediului de stocare.

Mai jos se pot vedea diferențele între partea care serializează, respectiv deserializează obiectele.

* Partea de serializare

FileOutputStream fileOut = **new** FileOutputStream("tmp/" + fileName + ".ser");

ObjectOutputStream out = **new** ObjectOutputStream(fileOut);

out.writeObject(written);

* Partea de deserializare

FileInputStream fileIn = **new** FileInputStream("tmp/" + fileName);

ObjectInputStream in = **new** ObjectInputStream(fileIn);

obj = in.readObject();

Deși corpurile metodelor de mai sus prezintă succint cum funcționează mecanismul de serializare și deserializare obiectele returnate de către acestea, de cele mai multe ori, nu respectă în vreo manieră un anumit standard și sunt destul de greu de folosit. Tocmai de aceea, a apărut necesitatea unui generator, o metodă care, primind puțini parametrii să facă foarte multe lucruri și astfel să simplifice mai târziu implementarea altor funcționalități.

Ca idee, metoda generator din Serializable funcționează în următorul mod:

* Odată apelată, funcția generator caută după directorul specificat și după fișierele care au în componență un anumit string
* Apoi apelează metoda cu care se deserializează fișierele
* Trece cu listele rezultate printr-un convertor de tip
* Inițializează un model de date
* Mapează datele obținute după transformare în modelul nou creat
* Returnează modelul

În urma acestei înlănțuiri, obținem un model complet cu care putem crea foarte ușor tabele, sau chiar obține informații de scurtă durată. Cu ajutorul acestei metode, se implementează și o parte din metoda generală refresh.

* 1. **Clasele MultimeClienti, MyPanel, Casa**

La nivel funcțional, cele 3 clase sunt identice și tocmai de aceea le voi trata pe larg în grup. Motivul principal pentru care am folosit 3 clase distincte este că această aplicație, dacă ar fi dezvoltată, cu siguranță ar ajunge să tapeze și multe elemente specifice unui anumit tip de Obiect.

**public** Client(ArrayList<String> data) {

id = mkID();

nume = data.get(0);

email = data.get(1);

telefon = data.get(2);

Serializable.*serialize*(**this**, "client"+id);

}

**public** Client(ArrayList<String> data, **int** oldID) {

id = oldID;

nume = data.get(0);

email = data.get(1);

telefon = data.get(2);

Serializable.*serialize*(**this**, "client"+id);

}

**private** **int** mkID() {

**return** (**int**) (Math.*random*()\*10000);

}

Acestă formă generală este păstrată de către cele 3 clase, și reprezintă doar instanțierea unor variabile instanță. Elementul cheie se află la metoda serialize, din clasa Serializable care creează fișiere pornind de la obiectele specificate.

* 1. **Rezultate**

Cu excepția unor mici cazuri când programul este forțat la în zona de input-uri, programul nu va rula corespunzător deoarece nu există parte de parsare pentru eventualele string-uri aleatoare.

În rest, atât rezultatele, cât și valorile se află în grila celor așteptate, programul având un oarecare grad de flexibilitate la nivelului modului de procesare a datelor.

1. **Concluzii**

Din această temă, am putut să văd concret, avantajul abstractizărilor încă din primele stagii ale dezvoltării aplicației. La final, o bună parte din timp s-a irosit pe reparații minore care au apărut totuși în locurile unde la început am considerat că nu va fi necesar să gândesc temeinic eventualele funcționalități. Aceste funcționalități s-au acumulat și au interferat desigur, cu clasele și metodele pe care le consideram lipsite de importanță devenind imediat din părți terță în elemente cu funcții de bază.

Crearea de clase generale, cu parametrii de tip Obiect au fost poate cel mai important lucru pe care l-am învățat. În această abordare, am putut scrie cod care altfel ar fi necesitat o repetiție continuă a unor bucăți existente, eventual cu mici modificări cheie.

Folosirea Java Reflection s-a dovedit a fi un tool deosebit de util, mai ales în generarea automată a tabelelor. În această manieră, am putut iarăși să refolosesc util codul, dar cel mai important a fost că nu am modificat o bună parte din metode ci doar le-am supraîncărcat.

1. **Bibliografie**
2. <http://stackoverflow.com>
3. <https://docs.oracle.com>
4. <http://forum.codecall.net/>