**DOCUMENTATIE TEMA 2**

**SIMULARE COZI**

**Nume prenume Man Silvana**

**Grupa 30224**

**Profesor Laborator Assist Pop Claudia**

Contents

[1. Cerinte Functionale. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .2](#_Toc479301366)

[2. Obiective. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .3](#_Toc479301367)

[3. Analiza Problemei. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. 3](#_Toc479301368)

[4. Proiectare. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5](#_Toc479301369)

[4.1. Structuri de date. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .5](#_Toc479301370)

[4.2. Diagrama de clase. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6](#_Toc479301371)

[4.3. Algoritmi. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .7](#_Toc479301372)

[5. Implementare. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7](#_Toc479301373)

[6. Concluzii si Dezvoltari Ulterioare. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8](#_Toc479301374)

[7. Bibliografie. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .9](#_Toc479301375)

# Cerinte Functionale

Proiectati si implementati o aplicatie de simulare care sa analizeze sistemele bazate pe cozi pentru determinarea si reducerea timpului de asteptare al clientilor.

# Obiective

**Obiectivul** temei este de a implementa o aplicatie cu interfata grafica pentru a fi folosita ca simulator de cozi. Acest simulator este implementat folosind limbajul de programare Java si conceptele de programare orientata obiect. Mediul de dezvoltare este Eclipse.

Pentru o mai buna intelegere a problemei, voi prezenta definitia cozii ca si concept in lumea informaticii. Coada este un tablou de date cu proprietatea FIFO (First In First Out) avand ca operatii specifice adaugarea (se realizeaza intodeauna ca ultim element al cozii) si scoaterea din coada (intotdeauna va fi servit primul client din coada). In esenta, primul client care se aseaza la coada va fi primul client servit. De asemenea clientii se pot aseza doar la sfarsitul cozii.

Cozile sunt frecvent intalnite atat in lumea reala cat si in lumea informaticii. Obiectivul pricipal al unei cozi este de a oferi un loc pentru client in asteptarea unui anumit serviciu. Aplicatia urmareste minimizarea timpului de asteptare. Cea mai evidenta solutie este adaugarea mai multor case, dar acesta este un proces costisitor pentru intreprinzator. De aceea se cauta alte solutii, precum gestionarea clientilor in asa fel incat fiecare client adaugat sa aiba timpul de asteptare cat mai mic.

# Analiza Problemei

Functionarea unei cozi se bazeaza pe principiul FIFO (First In First Out), care permite introducerea unui element exclusiv la finalul cozi, in timp ce deservirea si scoaterea unui element din coada se realizeaza pe primul element al cozii.

Problema consta in realizarea unui algoritm de impartire a clientilor la casele disponibile. Pentru aceasta, la sosirea unui nou client se calculeaza timpul de servire a clientilor distribuiti in fiecare casa, iar apoi se adauga clientul la coada cu timpul cel mai mic.

Ca *date de intrare* se considerA:

* Intervalul de sosire dintre clienti - un nou client va fi generat dupa un numar de minute din acest interval fata de clientul generat inaintea acestuia
* Intervalul de servire pentru clienti - clientii generati vor lua ca valoare pentru timpul de servire o valoare din intervalul specificat.
* Numarul caselor de marcat deschise – reprezinta numarul de cozi la care se poate aseza oricare dintre client.
* Intervalul de simulare – intervalul in care pot fi generati clienti. Simularea se va incheia doar in momentul in care ultimul client generat este deservit.

*Datele de iesire* constau in:

* Determinarea timpului mediu de asteptare si de servire - timpii medii sunt calculati pentru fiecare coada in parte, iar apoi afisati la finalul simularii
* Istoricul evenimentelor – date referitoare la timpul in care un client intra sau iese din coada
* Evolutia cozilor – date referitoare la situatia unei cozi la un anumit moment din timpul simularii ce consta in descrierea tuturor clientilor ce sunt in momentul repectiv la coada. Datele se refera la timpul la care clientul a intrat in coada, timpul de asteptare si timpul de servire al acestuia, cat si un identificator pentru acesta ce consta intr-un ID unic.

Pentru efectuarea propriu-zisa a cozilor avem nevoie sa cunoastem ce inseamna conceptul de thread (fir de executie). Un fir de executie ***(thread)*** este cea mai mica entitate sau secventa de instructiuni care poate fi identificata si gestionata de sine statator de catre planificatorul unitatii centrale din fiecare sistem de operare si executata de catre proces. Spre deosebire de procese, un thread poate exista doar in cadrul unui proces. Din acest motiv, un proces contine cel putin un fir de executie care foloseste resursele procesului gazda.

Scopul folosirii threadurilor este pentru a eficientiza executia programelor, executand portiuni distincte de cod in parallel, in interiorul aceluiasi proces. Totusi, cateodata aceste portiuni de cod care constituie corpul threadurilor, nu sunt complet independente si in anumite momente ale executiei, se poate intampla ca un thread sa trebuiasca sa astepte executia unor instructiuni din alt thread, pentru a putea continua executia propriilor instructiuni. Aceasta tehnica prin care un thread asteapta executia altor threaduri inainte de a-si continua propria executie, se numeste sincronizarea threadurilor.

Un fir de execuţie se poate afla în Java în mai multe stări, în funcţie de ce se întâmplă cu el la un moment dat. Atunci când este creat, dar înainte de apelul metodei start, singurele metode care se pot apela pentru firul de execuţie sunt metodele start şi stop. Metoda start lansează firul în execuţie prin apelul metodei run. Metoda stop omoară firul de execuţie încă înainte de a fi lansat. Orice altă metodă apelată în această stare provoacă terminarea firului de execuţie prin generarea unei excepţii de tip *IllegalThreadStateException*.

Dacă este apelata metoda start pentru firul de execuţie va trece în starea *running*. În această stare, instrucţiunile din corpul metodei run se execută una după alta. Execuţia poate fi oprită temporar prin apelul metodei *sleep* care primeşte ca argument un număr de milisecunde care reprezintă intervalul de timp în care firul trebuie să fie oprit. După trecerea intervalului, firul de execuţie va porni din nou.

În timpul în care se scurge intervalul specificat de sleep, obiectul nu poate fi repornit prin metode obişnuite. Singura cale de a ieşi din această stare este aceea de a apela metoda *interrupt*. Această metodă aruncă o excepţie de tip *InterruptedException* care nu este prinsă de *sleep,* dar care trebuie prinsă obligatoriu de metoda care a apelat metoda sleep.

Dacă dorim oprirea firului de execuţie pe timp nedefinit, putem apela metoda *suspend*. Aceeaşi stare este folosită şi pentru oprirea temporară cu sleep. În cazul apelului suspend însă, execuţia nu va putea fi reluată decât printr-un apel al metodei resume. După acest apel, firul va intra din nou în starea *running*.

Pe timpul în care firul de execuţie nu rulează (prin suspend sau sleep), acesta nu este planificat niciodată la controlul unităţii centrale, aceasta fiind cedată celorlalte fire de execuţie din aplicaţie.

Firul de execuţie poate intra în starea *not running* şi din alte motive. De exemplu se poate întâmpla ca firul să aştepte pentru terminarea unei operaţii de intrare/ieşire de lungă durată caz în care firul va intra din nou în starea *running* doar după terminarea operaţiei.

O altă cale de a ajunge în starea *not running* este aceea de a apela o metodă sau o secvenţă de instrucţiuni sincronizată după un obiect. În acest caz, dacă obiectul este deja blocat, firul de execuţie va fi oprit până în clipa în care obiectul cu pricina apelează metoda *notify* sau *notifyAll* .

# Proiectare

## Structuri de date

Pentru realizarea proiectului am folosit mediul de dezvoltare Eclipse si limbajul de programare Java. Problema a fost divizata in clase ce vor fi prezentate in cele ce urmeaza. Pasii de rezolvare a problemei au fost realizarea diagramei UML care sta la baza proiectului, iar apoi realizarea claselor si a metodelor necesare.

O listǎ este o colecţie **ordonatǎ**. Listele pot conţine elemente **duplicate**. Pe langǎ operaţiile moştenite de la Collection, interfaţa List defineşte urmǎtoarele operaţii:

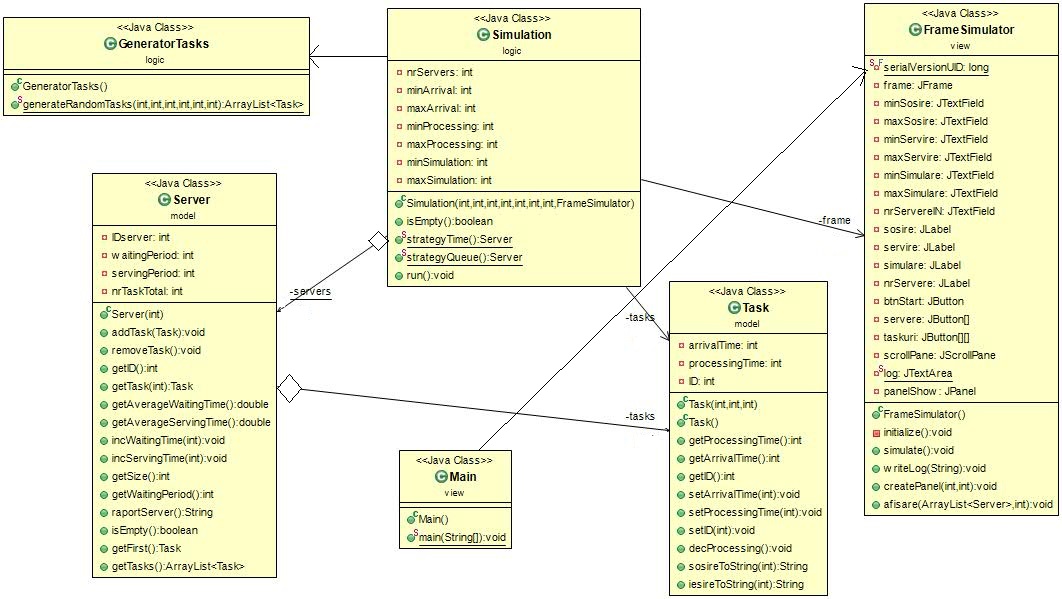
* T get(int index) - întoarce elementul de la poziţia index
* T set(int index, T element) - modificǎ elementul de la poziţia index
* void add(int index, T element) - adaugǎ un element la poziţia index
* T remove(int index) - şterge elementul de la poziţia index

Alǎturi de List, este definitǎ interfaţa ListIterator, ce extinde interfaţa Iterator cu metode de parcurgere în ordine inversǎ.  
List posedǎ implementarea ArrayList - implementare sub formǎ de vector. Accesul la elemente se face în timp constant: O(1).

Am ales ca structura de date ArrayList, deoarece valorile sunt accesate folosind o pozitie întreagă.

Serverul este este un ArrayList de taskuri, iar in proiectul meu va exista si un ArrayList de Servere, deci vom ajunge la o matrice de taskuri.

## Diagrama de clase



## Algoritmi

Algoritmul consta in distribuirea clientilor la coada cu timpul de asteptare minim. Aceasta coada este determinata in metoda *selectareCoada()* din clasa SimulareCoada. Coada minima este determinata prin insumarea timpurilor de servire a tuturor clientilor si selectarea acelei cozi care are timpul minim.

# Implementare

Proiectul este structurat dupa design pattern-ul MVC (Model- View- Controller). Avem trei pachete: logic, model si view. In pachetul “model” gasim clasele “Server” si “Task”.

Clasa “Task” reprezinta de fapt, clientul din supermarket. Acesta este identificat cu ajutorul unui ID. Lui ii mai sunt atribuite un *arrivalTime*- timp de sosire, si un *processingTime*- timpul de procesare. Aici gasim functii de get si set pentru variabilele folosite, si metode care sunt de ajutor la afisarea informatiilor despre clientul care ajunge si pleaca din coada.

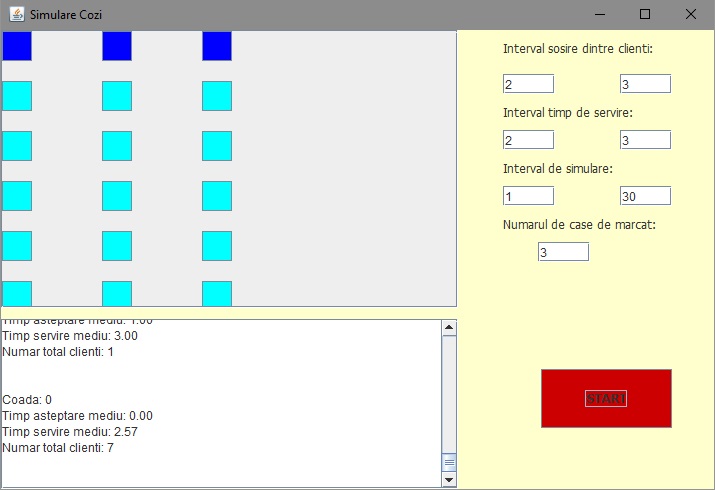
Clasa “Server” reprezinta casa de marcat din supermarket. Este de fapt o coada (lista) de taskuri, repectiv client. Un server este identificat de un *IDserver* si contine o perioada de asteptare *waitingPeriod,* o perioada de servire *servingPeriod* si un numar de taskuri total *nrTaskTotal.* Pe langa metodele de get si set, aici gasim metoda de adaugare, stergere, luare a unui task. Metodele *removeTask()* si *getTask()* sunt *synchronized*. Tot in aceasta clasa calculam *Average Waiting Time* si *Average Serving Time.*

In pachetul “logic” gasim clasa “GeneratorTasks” si “Simulation”. Clasa “GeneratorTasks” genereaza o lista de taskuri cu ajutorul functiei Random(). Astfel vor fi generate random timpul de sosire si cel de procesare pentru fiecare task.

Clasa “Simulation” este cea care foloseste thread-urile. Ea implementeaza clasa “Runnable”. Aici avem o lista la servere, fiecare server continand o lista de task-uri, astfel vom ajunge la o matrice de task-uri. Aceasta clasa contine datele de intrare din proiect, respective numarul de servere, intervalul de sosire dintre clienti, intervalul de procesare si intervalul de simulare al programului. Tot aici se face si legatura cu interfata grafica. Avem implementate 3 metode. Doua dintre ele sunt strategiile folosite pentru plasarea unui task intr-o coada – *StrategyTime()* si *StrategyQueue()*. Prima distribuie task-urile in cozi in functie de timpul de asteptare minim, iar a doua strategie distribuie task-urile dupa lungimea cozii. A treia metoda pe care o gasim in clasa “Simulation” este cea de *run().* Practic, aici este descris intregul proces. Se creeaza lista de severe, in care se pun task-urile generate random, se calculeaza timpii doriti, se adauga si se sterg din coada task-urile, in functie de strategia aleasa si apoi se da sleep thread-ului pentru 1000 milisecunde.

In pachetul “view” avem doua clase importante. Una este cea de “Main”, capul proiectului, iar cealalata este “FrameSimulator”, adica interfata grafica a proiectului. Aceasta contine toate componentele interfetei grafice si metodele folosite la afisarea datelor. In principal, avem JTextField-uri pentru datele de intrare cerute, un buton de start, care porneste intregul proces, un JPanel in care vom arata modul in care se aseaza clientii la coada si un JTextArea in care se la afla log-ul; adica toate informatiile necesare despre ce se intampla in process. Cand ajunge un client si la ce coada se aseaza, cand pleaca de la coada si timpul mediu de asteptare, respective de servire.

Butoanele albastru inchis reperezinta casele de marcat, respective serverele, iar cele albastru deschis cunt clientii sau task-urile.



# Concluzii si Dezvoltari Ulterioare

Ca o concluzie, acest proiect m-a ajutat sa imi consolidez cunostintele de programare orientata pe obiect dobandite in primul semestru si sa imi organizez munca in baza paradigmelor POO. Totodata, felul in care am structurat proiectul si am creat clasele si metodele m-a ajutat sa eficientizez codul din punct de vedere al lungimii si al usururintei intelegerii acestuia, lucruri vitale in cazul programelor complexe care necesita mai multe clase sau care presupun un numar mai mare de programatori.

In cadrul dezvoltarilor ulterioare se pot aminti urmatoarele:

-imbunatatirea interfetei grafice,

-o vizionare animata a procesului,

-inchiderea si deschiderea caselor in functie de numarul de clienti,

-crearea unui algorit mai eficient pentru distribuirea clientilor la case, precum si aducerea de noi posibilitati pentru utilizator, cum ar fi selectarea timpului de rulare a simularii (1 secunda = 0.5 secunde -> simulare de 2 ori mai rapida pentru aceleasi date)

-Adaugarea unor informatii suplimentare despre clienti , cum ar fi suma de bani de care dispune. In cazul in care nu are destui bani sa plateasca cumparaturile, i se mareste timpul de servire deoarece trebuie sa se astepte sa renunte la anumite produse;

-Afisarea datelor de iesire intr-un fisier .txt;

- Datele de intrare pot fi actualizate pentru a avea posibilitatea unor setări mai detaliate de timp (cum ar fi ora, minut, zi – poate )

- S-ar putea adăuga o metodă cu scopul de a face clientii să lase o coadă și să meargă la alta în cazul în care există cozi de lungime mai scurtă.

- Crearea unui program "tip buclă", ceea ce înseamnă că, după ce un rezultat este dat, utilizatorul ar trebui să aibă posibilitatea de a re-folosi programul fără a fi nevoie să iasă din el.

# Bibliografie

* [www.stackoverflow.com](http://www.stackoverflow.com)
* <http://docs.oracle.com/javase/tutorial/collections/algorithms/index.html>
* <http://docs.oracle.com/javase/1.5.0/docs/api/java/util/List.html>
* <https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html>
* <http://www.tutorialspoint.com/java/util/timer_schedule_period.htm>