# 

**DOCUMENTATIE TEMA 2**

**Queue Simulator using threads**

**Varvara Raluca Ana-Maria**

**Grupa 30222**

**Profesor Laborator: Oana Andreea Marin**

Contents

[1](#_Toc68552938)

[1. Obiectivul temei 3](#_Toc68552939)

[2. Analiza 3](#_Toc68552940)

[3. Proiectarea șistemului 4](#_Toc68552941)

[3.1 Proiectarea generală a șistemului 4](#_Toc68552945)

[3.2 Decizii de proiectare 5](#_Toc68552946)

[3.3 Divizarea în subșisteme/pachete 5](#_Toc68552947)

[3.4 Împărțirea în clase 6](#_Toc68552948)

[3.5 Diagrama de clase 8](#_Toc68552949)

[3.6 Structuri de date 9](#_Toc68552950)

[3.7 Interfețe folosite/definite 10](#_Toc68552951)

[3.8 Algoritmi 10](#_Toc68552952)

[3.9 Interfața 10](#_Toc68552953)

[4. Implementare 12](#_Toc68552954)

[5. Rezultate 17](#_Toc68552955)

[7. Concluzii 17](#_Toc68552956)

[8. Bibliografie 18](#_Toc68552957)

# **Obiectivul temei**

Obiectivul principal al proiectului este de a creea o aplicație care să pună la dispoziție utilizatorului un șistem de șimulare minimizat care calculează timpul mediu de așteptare al unor client la cozi și prezintă evoluția lor în timp. Cozile sunt prezente peste tot în viața reală, iar în contextual programării ele sunt o structură de date de tip FIFO. Proiectul va prezenta o interfață grafică user-friendly prin care utilizatorul poate introduce datele pe care vrea să le șimuleze pentru un șistem de cozi/servere, adică: numărul de client, numărul de cozi, timpul minim și maxim de ajungere la coadă, timpul minim și maxim de servire, strategia de așezare la cozi și timpul de șimulare.

Sub-obiectivele principale ale temei reprezintă pașii care au fost urmați pentru atingerea obiectivului final:

* Proiectarea unui client;
* Proiectarea unei cozi;
* Crearea threadurilor;
* Proiectarea șistemului ce manipulează cozile și clieții;
* Crearea strategiilor pentru servirea clienților;
* Implementarea interfeței ce arată în timp real evoluția cozilor.

# **Analiza**

**Cerințele funcționale** vor include următoarele:

* introducerea datelor pentru simularea sistemului de cozi, date care include: numărul de client, numărul de cozi, timpul minim și maxim de ajungere la coadă, timpul minim și maxim de servire și timpul de simulare;
* alegerea strategiei de așezare la cozi
* vizualizare simulare;

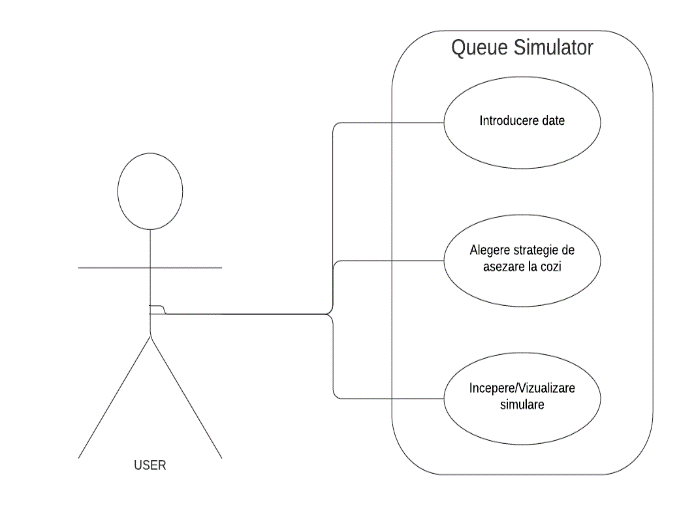


Fig.1.

**Cerințele non-funcționale** vor fi următoarele:

* Simulatorul de cozi ar trebui să fie intuitiv și ușor de folosit de către user;
* Incapsularea datelor;
* Afișarea coerentă și în timp real a evoluției cozilor cu prezentarea listei de clienți inițiali.

**Use case:** începerea simulării

**Actorul principal:** user-ul

**Scenariul principal de success:**

1. User-ul introduce datele necesare în interfața grafică;
2. User-ul selectează strategia de așezare la cozi;
3. User-ul apasă pe butonul de începere a simulării;
4. Simularea începe și așezarea la cozi a clienților se poate vedea în timp real în interfața grafică;
5. Dacă se vrea să se revadă evoluția cozilor, se poate verifica fișierul txt care prezintă “log of events”.

**Secvența alternativă:** date introduce incorect

Userul introduce în unul dintre câmpurile de date ceva ce nu este un număr, sau lasă gol un câmp. Acesta va fi intenționat cu mesajul necesar, poate să reactualizeze datele și să înceapă din nou simularea.

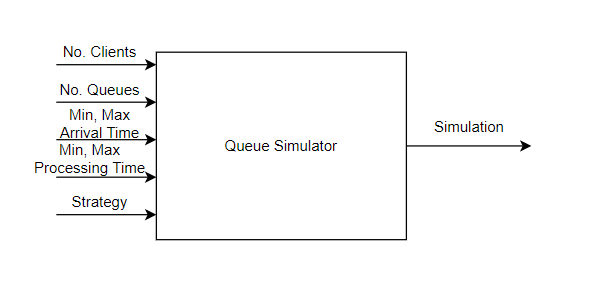
**Secvența alternativă:** timp de simularea prea mic

Userul introduce un timp de simulare mai mic decât este necesar pentru servirea tuturor clienților, iar aceștia nu vor fi procesați în totalitate.

# **Proiectarea sistemului**



## Proiectarea generală a sistemului



## Decizii de proiectare

Din punct de vedere al proiectării am luat următoarele decizii:

* Implementarea interfeței Comparable pentru clasa Client pentru a putea sorta mai eficient o lista de Clienti;
* Adăugarea câmpului finishTime în clasa Client pentru a calcula mai ușor timpul de așteptare mediu;
* Extinderea clasei SwingWorker de view ul ce se ocupa cu afisarea datelor in timp real.

## Divizarea in subșisteme/pachete

Am decis să împart problema astfel: un pachet in care sa retin datele, obiectele “reale”: client, coada și Selection Policy, un pachet care sa contina functionalitatile și implementarea șimularii, și un pachet care sa se ocupe de interfata grafica cu utilizatorul.

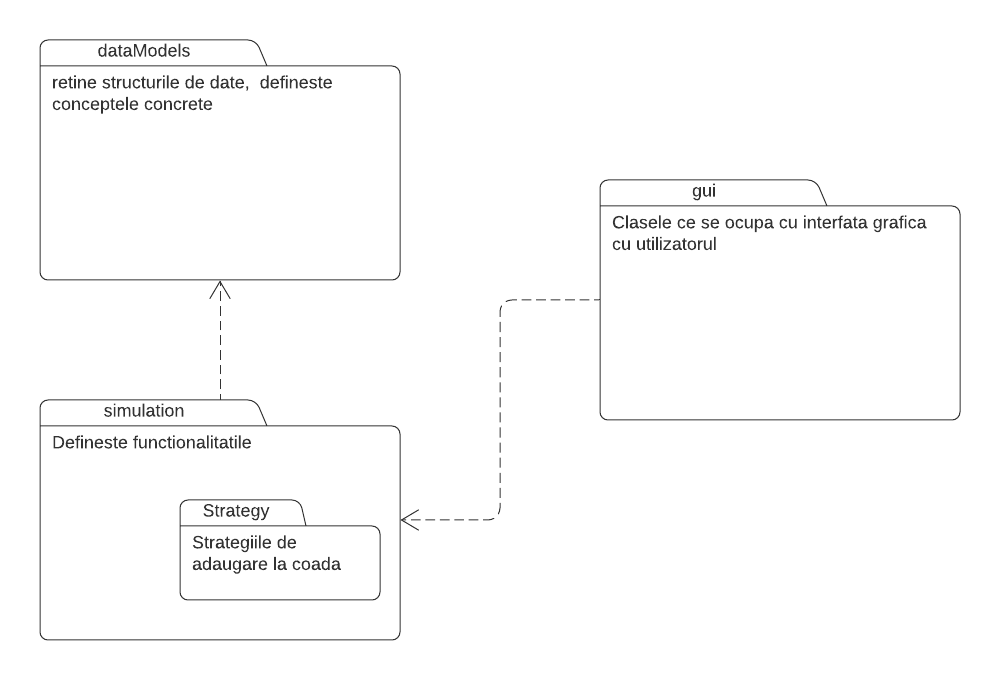


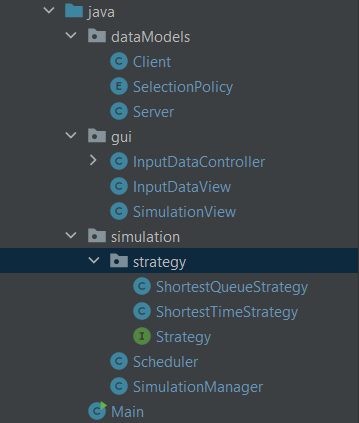
Fig. 2

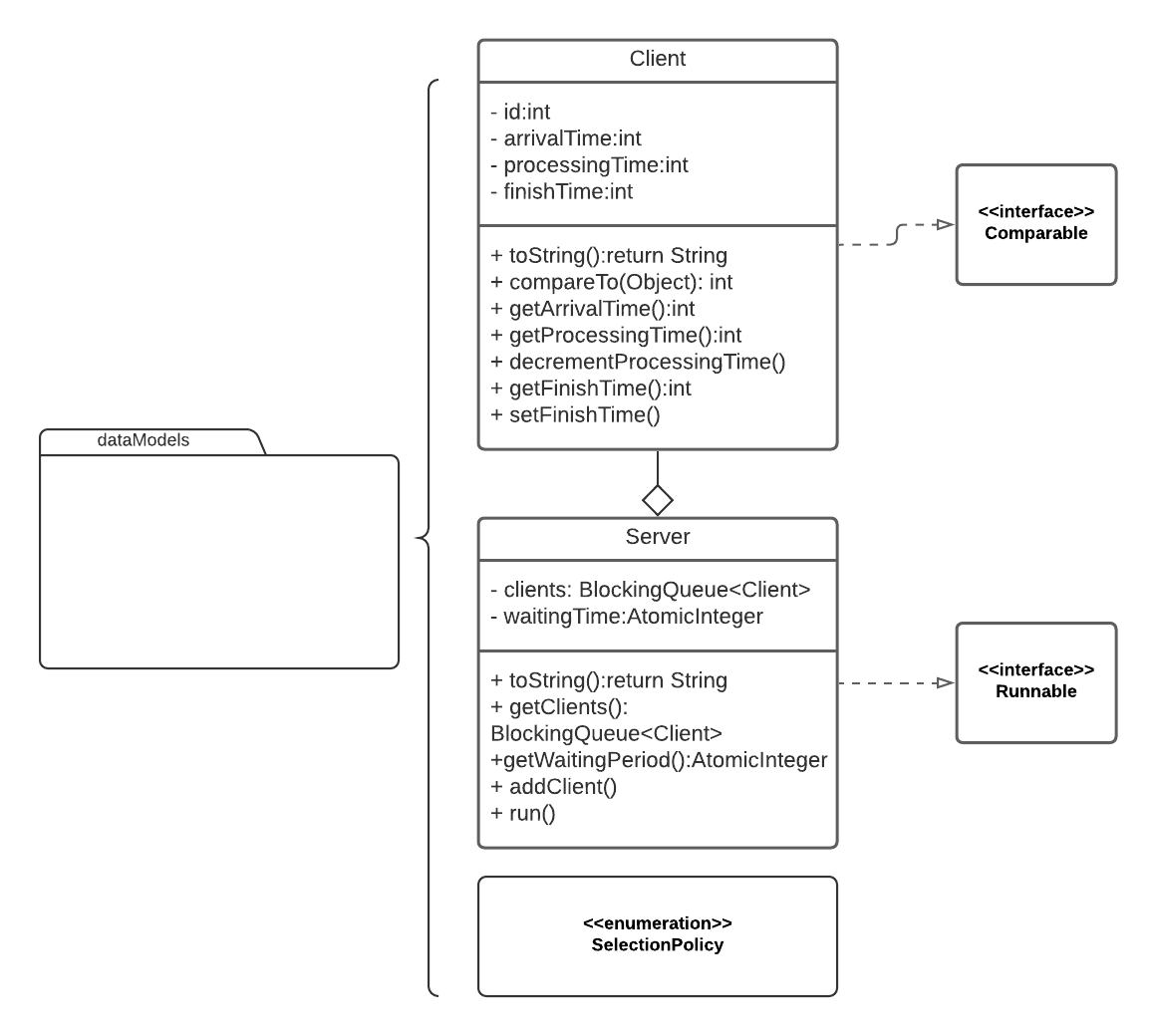
Am ales un pattern architectural de tip Layered deoarece împarte problema în subprobleme asemănătoare pentru fiecare pachet, după cum am descris mai sus:

* Subproblema de stocare
* Subproblema de funcționaliți
* Subproblema de interacționare cu utilizatorul.

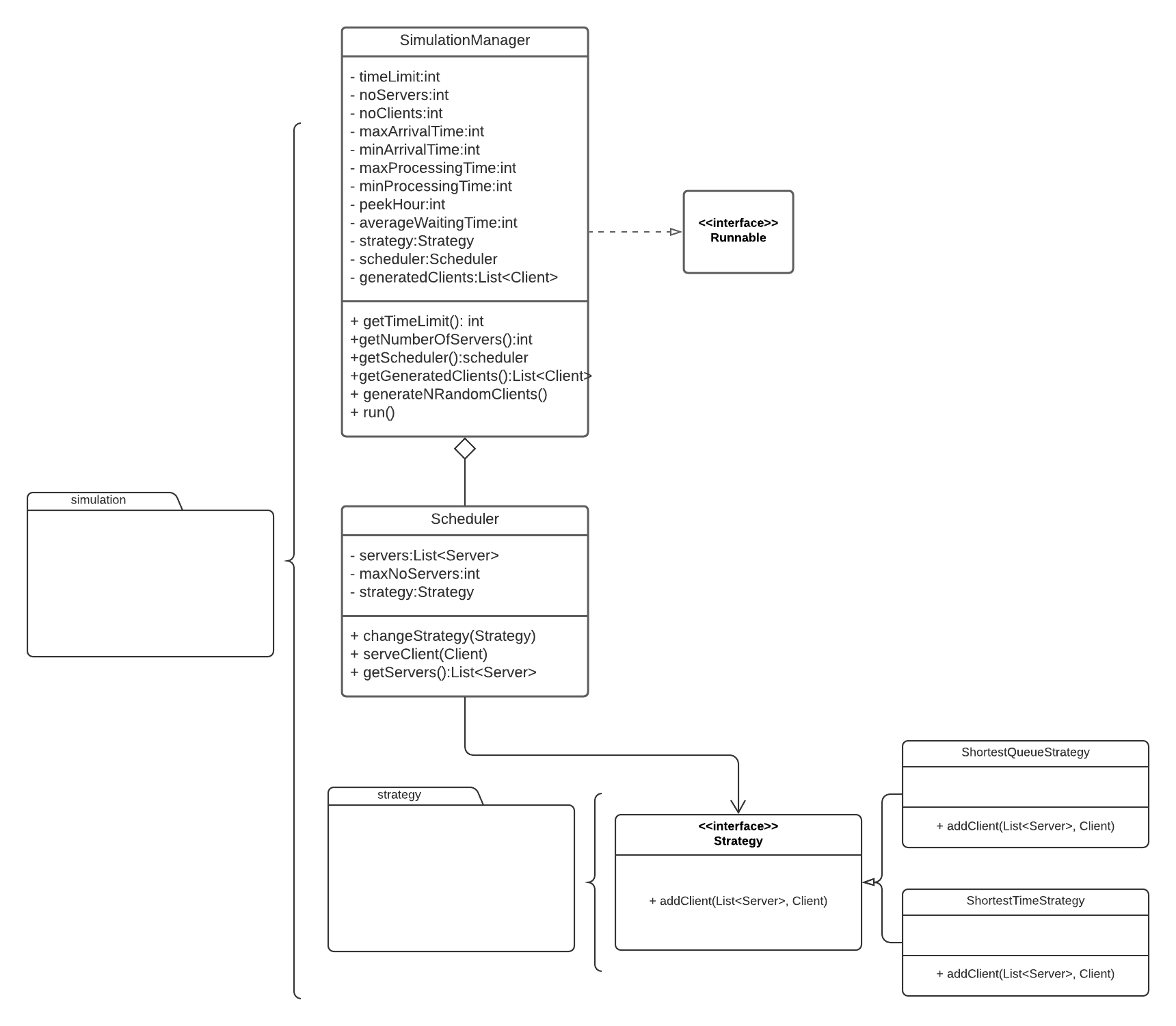
Este un mod ușor și intuitiv de implementare și împărtire în pachete, și nu există confuzii în legătură cu interacționarea dintre pachete.

## Împărțirea în clase

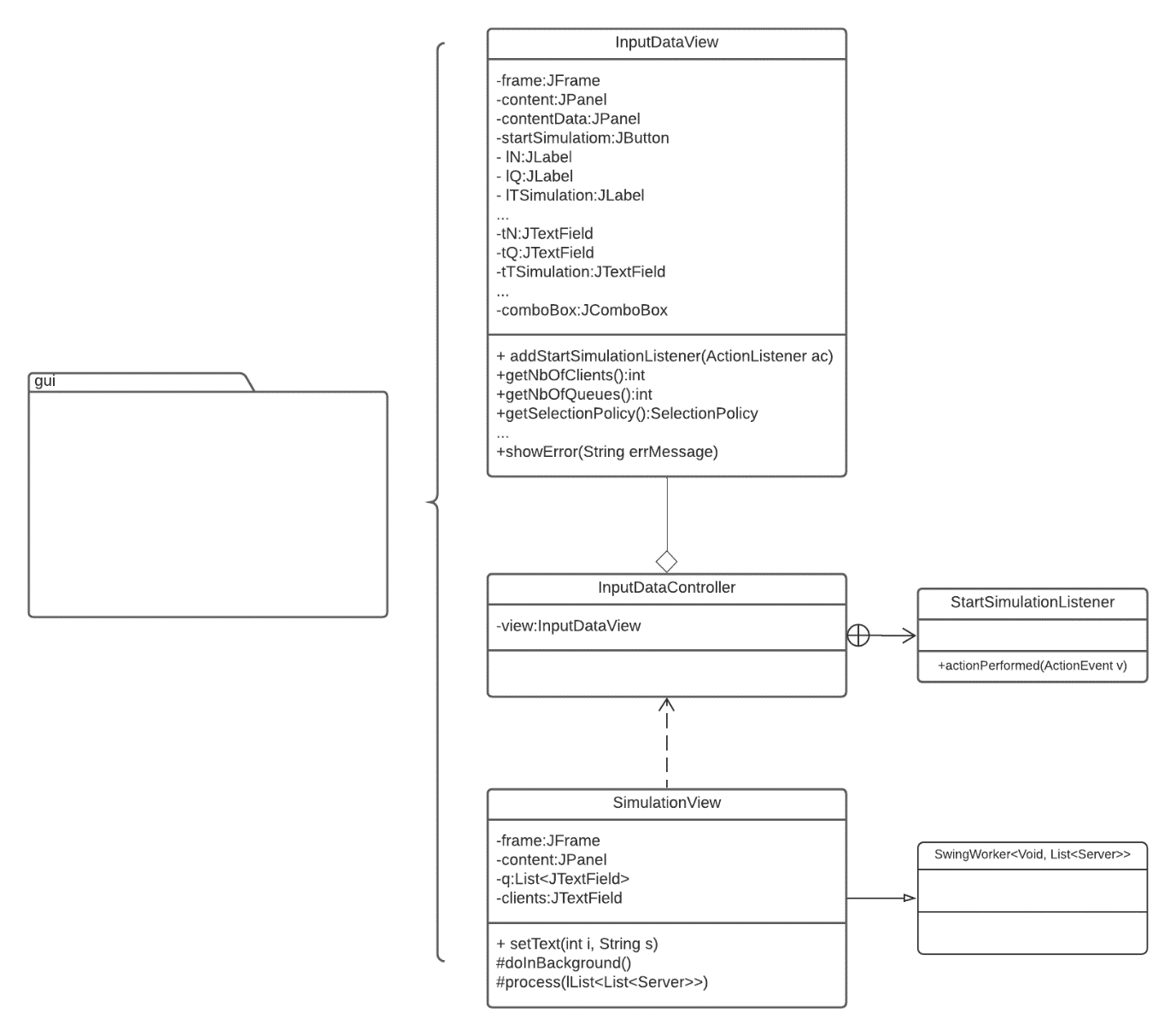




Pachetul dataModel va reține clasele care definesc modelele de date și care le vor putea stoca perșistent astfel am împărțit problema in client și servere. Clienti vor retine datele importante despre ei pentru procesare, iar serverele vor retine liste de clienti și waitingTime-ul lor care se va schimba in timp real, la fiecare secunda. De asemenea am mai introdus in acest pachet și enumeratia SelectionPolicy care se va foloși pentru strategia de adaugare la cozi.



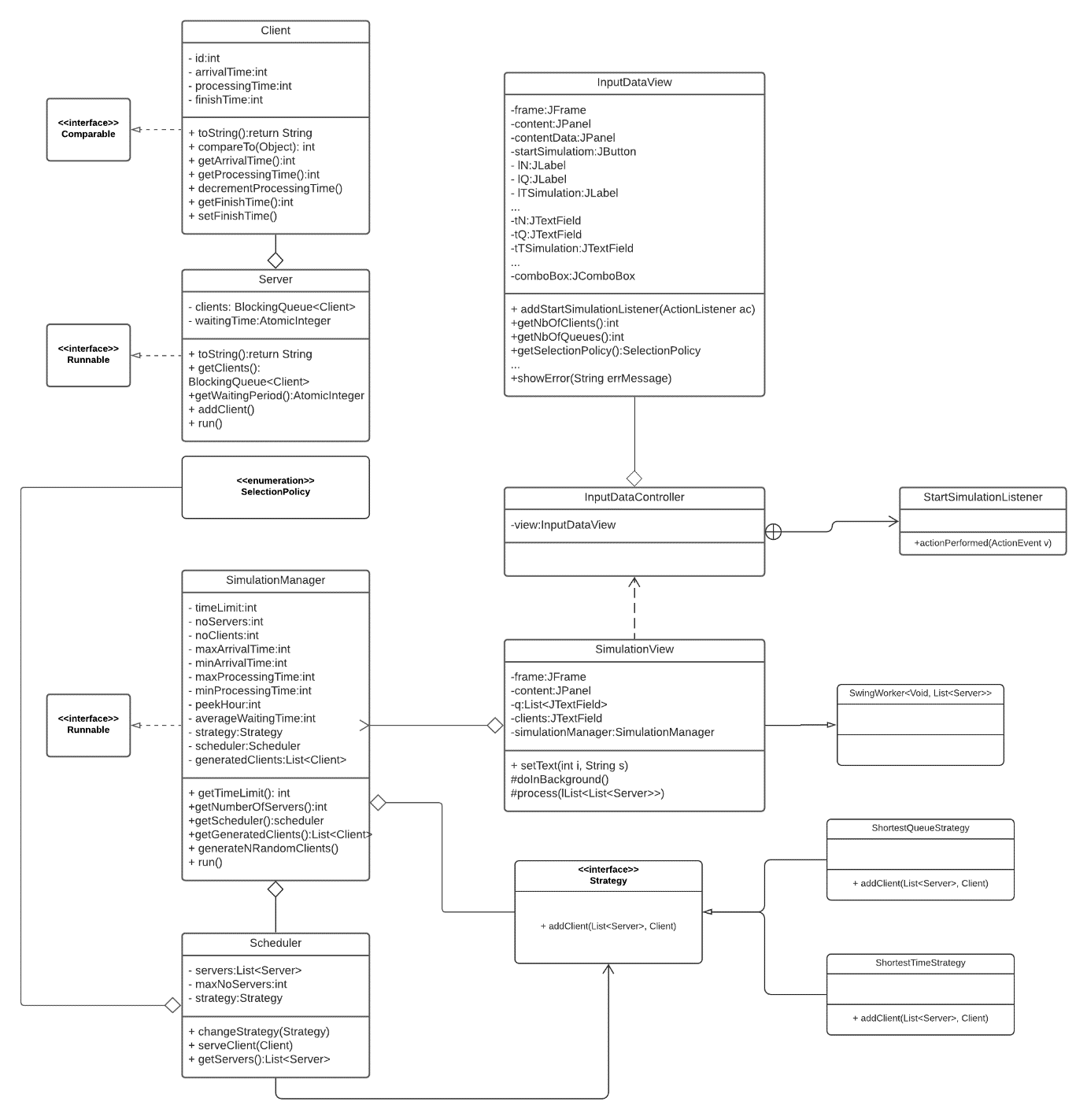
Pachetul șimulation va defini clasele care se vor ocupa cu funcționalitățile, adică șimularea și strategiile de adaugare la cozi. Astfel mai avem un subpachet, strategy, care cuprinde numai clasele ce implementeaza strategiile: interfata strategy și cele 2 clase de strategii concrete, ShortestTime și ShortestQueue. In rest mai avem clasa Scheduler, care se ocupa cu trimiterea clientilor la servere și retine lista de servere, și clasa ȘimulationManager care genereaza clientii random in functie de constrangeri și detine bucla principala de șimulare.



Pachetul gui detine 2 view-uri și un controller. Un view este pentru introducerea datelor, iar celalt este pentru afisarea in timp real al evolutiei cozilor. Contollerul este al view-ului de introducere a datelor deoarece este șingurul view care are o componente interactive, campurile de introducere a textului și butonul de pornire a șimularii. Cand acest buton este apasat se creaza un view nou de șimulare. Intre controller și clasele interne este o **relatie de compozitie**, intrucat acestea nu exista in afara controlerului.

## Diagrama de clase

Unified Modeling Language (prescurtat UML) este un limbaj standard pentru descrierea de modele și specificații software. Diagrama de clase UML este folosită pentru reprezentarea vizuală a claselor și a interdependențelor, taxonomiei și a relațiilor de multiplicitate dintre ele. Diagramele de clasă sunt folosite și pentru reprezentarea concretă a unor instanțe de clasă, așadar obiecte și a legăturilor concrete dintre acestea.



## Structuri de date

Structurile de date predefinite pe care le-am folosit sunt: ArrayList, LinkedBlockingQueue și AtomicInteger. ArrayList a fost folosit pentru lista de servere și Scheduler și pentru generarea de Clienti random. LinkedBlockingQueue a fost folosit pentru coada de client dintr-un server, aceasta structura de date fiind conceputa pentru a fi folosita in cadrul threadurilor, fiind o structura de date care are implementate operatii atomice, care nu pot fi intrerupte de un alt thread. AtomicInteger este o alta structura de date ce prezinta operatii atomice pentru a defini perioada de așteptare dintr-un server.

De asemenea, clasa Client poate fi văzută ca o structură de date ce definește atributele importante ale unui client pentru simularea evoluției unei cozi.

## Interfețe folosite/definite

Interfețele gata definite pe care le-am folosit sunt List, și BlockingQueue care se pot găsi în biblioteca java util, și de asemenea interfața comparable pentru client pentru a putea sorta o listă de clienți.

Am definit o interfață pentru strategia de adăugare la cozi a clienților, întrucât voi folosi 2 strategii concrete, era bine să le definesc cu ajutorul unei interfețe. Interfața definește o singură metodă : **public void addClient(List<Server> servers, Client c)**; care va adăuga clientul la unul dintre serverele din listă, în funcție de o strategie (cea mai scurtă coadă sau coada cu cel mai mic timp de așteptare).

## Algoritmi

Algoritmii folositi sunt cei pentru implementarea funcționalităților.

Algoritmii/Strategiile pentru adăugare la coadă

**Shortest time**

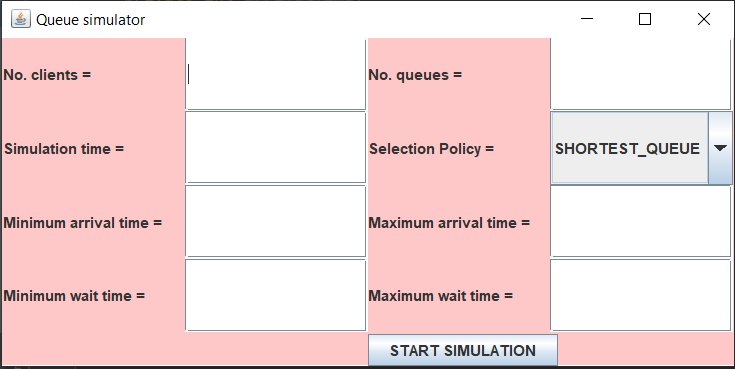
Se iterează prin lista de servere din Scheduler și se caută cel mai mic timp de așteptare, apoi se caută iar primul server care are acel timp de așteptare și se adaugă la final Clientul.

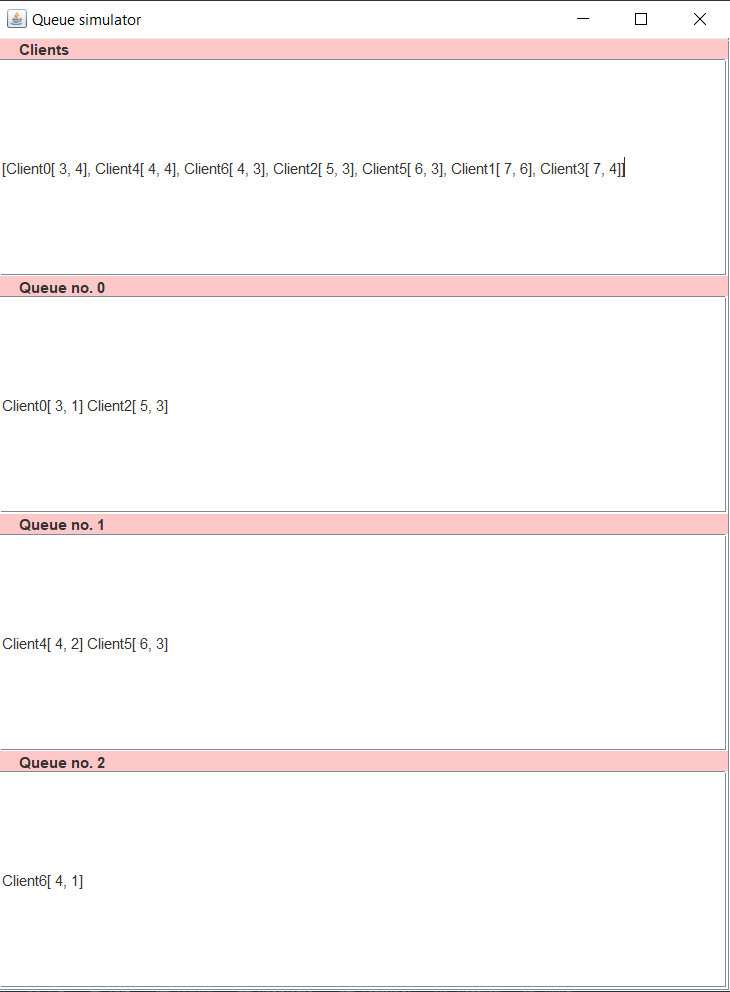
**Shortest queue**

Se iterează prin lista de servere din Scheduler și se caută cel mai mic număr de client la o coadă, apoi se caută iar primul server care are acel size și se adaugă la final Clientul.

## Interfața

Am ales să implementez o interfata simpla și intuitivă, ușor de folosit de către user. Are 7 câmpuri în care se pot introduce datele (nr de clienti, de cozi, etc.) și un combo box pentru alegerea strategiei. De asemenea mai există un buton pentru începerea simulării, care va deschide un nou view. Acesta va arăta evoluția live a cozilor, fiecare coadă fiind reprezentată de un text field sim și are un text field care prezintă clienții generați.





Am ales să implementez interfața cu Java Swing. Swing este un set de instrumente widget GUI pentru Java. Face parte din Oracle Java Foundation Classes (JFC) - un API pentru furnizarea unei interfețe grafice de utilizator (GUI) pentru programele Java. De asemenea pentru a nu încarca threadul principal cu printarea în text fielduri am folosit SwingWorker care permite crearea unui thread worker pentru interfața care să o updateze la intervalul necesar.

# Implementare

**Pachetul dataStructures**

**Clasa Client**

Clasa Client este folosită pentru a descrie clienții care stau la cozi, având atributele importante pentru procesare

**Atribute:**

* **private int id; -** folosit pentru a identifica mai bine clienții
* **private int arrivalTime; -** timpul la care ajung la coadă
* **private int procesșingTime; -** timpul de procesare, cât durează să fie serviți
* **private int finishTime; -** timpul la care se termină procesarea lor, ne va ajuta să calculăm timpul de așteptare mediu

Toate atributele sunt **private**, folosind conceptul specific progrămarii orientate pe obiect de incapsulare a datelor.

**Metode:**

* **public int getArrivalTime()**
* **public int getProcesșingTime**
* **public void decrementProcesșingTime() –** se decrementează timpul de procesare la fiecare secundă care a trecut în threadul principal
* **public int getFinishTime()**
* **public void setFinishTime(int finalTime)**
* **@Override public String toString() –** suprascrierea metodei toString pentru o afișare mai ușoară și mai eficientă a serverelor, tehnica specifică poo de împărțire a problemei în subprobleme;
* **@Override public int compareTo(Object o) ­–** este metoda definită în interfața Comparable implementată de această clasă, metoda care ne va ajuta pentru sortarea listei de client generate random.

**Clasa Server**

Este clasa care definește structura unei cozi întregi, care e definită de lista de clienți care așteaptă și de timpul de așteptare total al cozii. Între Client și Server este o relație de agregare, un server are mai mulți clienți. Clasa server **implementează interfața Runnable** deoarece se va folosi câte un thread pentru fiecare server pentru a simula concurența cozilor din viața reală.

**Atribute:**

* **private BlockingQueue<Client> clients; -** coada de client, clasa BlockingQueue este o clasa specializată pentru concurența. Este definită astfel încat toate metodele ei se execută atomic, pentru a nu crea probleme legate de concurenta și date în threaduri.
* **private AtomicInteger waitingPeriod;**.**–** reprezinăa timpul de așteptare la coadă, de asemenea clasa AtomicInteger este o clasă specializată pentru concurența, având definite numai metode ce se execută atomic.

**Metode:**

* **public BlockingQueue<Client> getClients()** - este necesar pentru a accesa variabila instantă privată a listei de clienți;
* **public AtomicInteger getWaitingPeriod();**
* **public void addClient(Client c)** – adaugă clientul la lista și mărește timpul de așteptare;
* **@Override public String toString()** – este suprascrierea metodei toString din clasa mama Object, și se parcurge cu o buclă foreach fiecare client din lista de clienți;
* **public void run() –** este suprascrierea metodei definite de Runnable. Cât timp lista de clienți nu e goală se ia primul client de la coasa și I se decrementează timpul de așteptare, iar dacă timpul de așteptare s-a terminat, îl scoate din coadă și se așteaptă o secundă.

**Enumerația SelectionPolicy**

Conține cele două politici de așezare la coadă care vor determina ce strategie se folosește: SHORTEST\_QUEUE, SHORTEST\_TIME.

**Pachetul simulation**

**Clasa Scheduler**

Este clasa ce reține lista de servere și trimite fiecare client la o coadă în funcție de politica folosită.

**Constructor:**

Constructorul

**Atribute:**

* **private List<Server> servers;**
* **private int maxNoServers;**
* **private Strategy strategy; -** strategia ce se folosșste la adăugarea în cozi

**Metode:**

* **public void changeStrategy(SelectionPolicy policy) –** se implementează o strategie în funcție de politica de selecție folosită SHORTEST\_TIME => ShortestTimeStrategy**;**
* **public void serveClient(Client c)**– servește un client cu ajutorul strategiei definite;
* **public List<Server> getServers()**

**Clasa SimulationManager**

Este clasa care se ocupă de firul de principal de lucru (implementeaza Runnable),gestionează toate serverele și calculează date specific simularii (peekTime, average waiting time, average serving time.

**Atribute:**

* **private int timeLimit**– următoarele date determină parametrii simulării;
* **private int maxProcesșingTime;**
* **private int minProcesșingTime;**
* **private int maxArrivalTime;**
* **private int minArrivalTime;**
* **private int numberOfServers;**
* **private int numberOfClients;**
* **private SelectionPolicy selectionPolicy;**
* **private Scheduler scheduler;**
* **private List<Client> generatedClients = new ArrayList<Client>()**
* **File file = new File("eventLog.txt") –** definește fișierul în care scriem cursul evenimentelor.
* **FileWriter writer** – cu ajutorul acestuă writer putem scrie în fișier

**Metode:**

* **public int getTimeLimit()**
* **public int getNumberOfServers**
* **public Scheduler getScheduler()**
* **public List<Client> getGeneratedClients()**
* **private void generateNRandomClients()** – generează un număr de client specificat de numberOfClients. Aceștia au id-uri de la 1 la câți clienți sunt, dar timpul la care ajung la magazine și timpul de servire sunt generate random cu ajutorul metodei Math.rand(), între parametrii specificați de atribute. Apoi clienții sunt sortați în funcție de timpul la care au ajuns.
* **@Overridepublic void run()** – se ocupă de firul principal. “Trece prin” fiecare secundă, de la secunda 0 până la timeLimit, și trece prin lista de client random până când arrivalTime-ul clienților nu mai este egal cu secunda curentă sau până când lista e goală și cu ajutorul scheduler-ului servește clientul. La fiecare client procesat se incrementează numărul de client procesati, I se adaugă waiting timeul la variabile ce calculează media timpului de așteptare și la fel pt media timpului de servire. Apoi iterez printer liste, le scriu în log-ul evenimentelor și calculez câți clieți sunt la coadă la secunda curentă. Dacă sunt mai mulți decât maximul calculate anterior se va actualiza, la fel și ora de vârf. Apoi se așteaptă o secundă până se repeată tot.

**Subpachetul strategy**

**Interfata Strategy**

Este o interfață care descrie contractual pentru clasele care implementează strategiile. Singura funcționalitate a claselor ce implementează va fi de a adăuga un client indiferent de strategia folosită.

**Metode:**

* **public void addClient(List<Server> servers, Client c); -** adaugă un client într-un server dintr-o listă de servere.

**Clasa ShortestQueueStrategy**

Este clasa ce implementează strategia de adăugare a clientului în funcție de care server are cei mai puțini clienți.

**Metode:**

* **public void addClient(List<Server> servers, Client c); -** adaugă un client într-un server dintr-o listă de servere și se folosește algoritmul shortest queue prezentat anterior

**Clasa ShortestTimeStrategy**

Este clasa ce implementează strategia de adăugare a clientului în funcție de care server are cel mai mic timp de așteptare.

**Metode:**

* **public void addClient(List<Server> servers, Client c); -** adaugă un client într-un server dintr-o listă de servere și se folosește algoritmul shortest time prezentat anterior.

**Pachetul gui**

A fost implementat cu o arhitectura MVC.

**Clasa InputDataView**

Este clasa care se ocupă cu prelucrarea datelor și stocarea lor din partea de interfață, tipic unei clase model dintr-o arhitectură MVC. Reține datele cu care va lucra calculatorul și prelucrează șirurile de caractere introduse de utilizator astfel încât să rezulte polinoame, și dacă sunt date introduse greșit va arunca o excepție.

**Atribute:**

* private final JFrame frame
* private final JPanel content
* private final JPanel contentData – pentru aranjarea frumoasă a componentelor
* private final JLabel lN
* private final JLabel lQ
* private final JLabel lTSimulation
* private final JLabel lMinArrival
* private final JLabel lMaxArrival
* private final JLabel lMinService
* private final JLabel lMaxService
* private final JLabel lSelection
* private final JComboBox comboBox
* private final JTextField tN
* private final JTextField tQ
* private final JTextField tTSimulation
* private final JTextField tMinArrival
* private final JTextField tMaxArrival
* private final JTextField tMinService
* private final JTextField tMaxService
* private final JButton startSimulation

**Metode:**

* **public int getNbOfClients()**
* **public int getNbOfQueues()**
* **public int getTSimulation()**
* **SelectionPolicy getSelectionPolicy()**
* **public int getMaxArrival()**
* **public int getMinArrival()**
* **public int getMaxService()**
* **public int getMinService()**
* **void addStartSimulationListener(ActionListener ac)** – adaugă listenerul pentru butonul de începere a simulării
* **void showError(String errMessage) –** acestă metodă afișează un pop up în cazul în care este o eroare la introducerea datelor. Se va folosi dacă atunci când se vor folosi metodele declarate mai sus pentru constructorul clasei SimulationManager, metoda parseInt va arunca un NumberFormatException, va fi prinsă și se va afișa pop-up-ul.

**Clasa InputDataControler**

Este clasa care se ocupă cu interacțiunea dintre utilizator și view, adaugă ascultatorul și începe simularea

**Atribute:**

* private InputDataView view;

**Constructor: -** el va inițializa view ul și va apela metoda de adăugare de ascultător definite pentru view cu clasa interioare corespunzătoare .

**Clasa internă:**

* **class StartSimulationListener implements ActionListener** – implementează operația ActionPerformed care la apăsarea butonului de start simulare va inițializa un SimualtionManager cu datele date în interfața de input, dacă nu se vor da datele corecte va deschide un pop up, apoi va inițializa un view pentru simulare și va apela metoda execute pentru a porni threadul worker.

**Clasa SimulationView**

Este clasa care se ocupă în timp real cu afișarea datelor în interfața și are un thread worker care lucrează în background. Astfel am decis să extindă clasa. SwingWorker<Void,List<Servers>>, void fiind tipul returnat de metoda doInBackground iar List<Server> este tipul care se va pasa la metoda process pentru a se afișa.

**Atribute:**

* **private final JFrame frame = new JFrame("Queue simulator");**
* **private final JPanel content = new JPanel();**
* **private final List<JTextField> q = new ArrayList<JTextField>() –** câte un textField pentru fiecare coadă
* **private final JTextField clients = new JTextField() –** text field care va afișa clienții inițiali generați.
* **SimulationManager simulationManager –** pentru a putea afișa datele din simulationManager

**Constructor:**

Se vor adauga în lista q atâtea textFielduri câte cozi sunt reținute în simulationManager. Înaintea adăugarii fiecărui textField adăugăm un label care va spune a câta coadă este.

**Metode:**

* **public void setText(int i, String s)** – setează al i-ilea text field cu un anumit String;
* **@Overrideprotected Void doInBackground() throws Exception** – este metoda ce descrie firul de lucru worker, la fiecare secundă până se ajunge la timeLimit-ul din SimulationManager se va **apela metoda publish** cu lista de servere din Scheduler pentru a putea fi “publicate” în interfața de metoda process.
* **@Overrideprotected void process(List<List<Server>> servers)** – parametrul este o listă de liste de servere deoarece datele pot veni în “chunks”. Se vor afișa numai ultimele date trimise din doInBackground de metoda public, cu ajutorul metodei setText**.**Se va itera peste ultima listă din parametru și se va afișa fiecare server în text fieldul corespunzător.

# Rezultate

Rezultatele la testele specificate în cerință au fost adăugate în directorul încărcat pe Git.

În fișierul eventlog.txt se pot observa rezultatele pentru fiecare rulare în cazul în care procesul nu a fost terminat forțat prin apăsarea butonului X (fișierul nu se închide și nu se va putea vedea nimic în el). La fiecare secundă apar timpul curent, și fiecare coadă cu ce client are, dacă are, și waiting time-ul pentru fiecare coadă iar la finalul fișierului vor apărea datele specifice simularii calculate: peek hour, average waiting time, average service time.

# Concluzii

În concluzie realizarea acestui proiect a fost foarte utilă , amintind de o mare parte din principalele concepte ale programării orientate pe obiecte învățate semestrul trecut.

Am învățat să lucrez mai bine cu threaduri și cu clase speciale pentru concurență, clase predefinite ce au funcționalitati atomice. De asemenea am învățat să lucrez cu clasa abstractă swing worker pentru a nu încarca un thread și cu afișarea în interfața și am învățat diferite tipuri de threaduri : current thread și worker thread.

O dezvoltare ulterioară ar fi atenționarea mult mai clară a utilizatorului în cazul în care datele sunt introduse greșit, adică programul să îi spună utilizatorului exact care dintre câmpuri nu a fost completat corespunzător. De asemenea s-ar putea adăuga la view-ul de afișare în timp real și un câmp care arată secunda curentă, iar la final să se deschida un pop up pentru a afișa Peak Time-ul și Average Waiting period -ul.

# Bibliografie

* <https://lucid.app/documents>
* https://app.diagrams.net/
* <https://www.informit.com/articles/article.aspx?p=336264&seqNum=3>
* <https://www.javacodegeeks.com/2012/12/multi-threading-in-java-swing-with-swingworker.html>
* <https://www.w3schools.com/java/java_files_create.asp>
* ASSIGNMENT\_2\_SUPPORT\_PRESENTATION