

## **Tehnici de programare fundamentale – Tema 2**

### **Simulator al unor cozi de clienți**

#### **1. Obiectivul temei**

Principalul obiectiv al acestei teme este de a proiecta și implementa un program care să simuleze gestiunea clienților într-un supermarket, mai exact felul în care aceștia aleg o casă de marcat la care să aștepte servirea (scanarea produselor) după ce și-au încheiat cumpărăturile. Programul va avea interfață grafică dedicată, concepută într-o manieră intuitivă și atractivă pentru utilizator, atât pentru introducerea datelor de intrare, cât și pentru afișarea în timp real a simulării evoluției cozilor de clienți. De asemenea, un fișier text va documenta situația cozilor de clienți în fiecare moment al simulării, pentru o analiză ulterioară precisă.

Proiectul își propune să îi permită utilizatorului să introducă:

- numărul de clienți ce să fie generați aleatoriu
- numărul de cozi în care clienții vor aștepta să fie serviți
- intervalul de timp pentru care să se desfășoare simularea
- intervalul de timp minim și maxim în care un client a încheiat cumpărăturile și este pregătit să se așeze la coadă
- intervalul de timp minim și maxim în care un client va fi servit la casă
- strategia după care un client va fi asignat unei cozi:
  - cea mai scurtă coadă
  - cel mai scurt timp de așteptare
- numele fișierului în care se va documenta simularea, cu rezultatele sale

Obiectivele secundare pe care dezvoltatorul și le propune se regăsesc în tabelul de mai jos, alături de secțiunile din prezenta lucrare în care vor fi adresate:

	<b>Obiectiv secundar</b>	<b>Scurtă descriere</b>	<b>Secțiunea dedicată</b>
<b>1.</b>	Analiza problemei	- stabilirea cerințelor la care programul trebuie să răspundă	2. Analiza problemei, modelare, scenarii, cazuri de utilizare
<b>2.</b>	Modelare	- imaginarea modelului conceptual pentru programul ce urmează a fi dezvoltat	2. Analiza problemei, modelare, scenarii, cazuri de utilizare
<b>3.</b>	Scenarii posibile	- analiza exhaustivă a situațiilor în care programul poate fi pus de către utilizator (input valid, șirul nul ca nume de fișier, valoare minimă introdusă ca fiind mai mare decât valoarea maximă, numere negative ca valori de timp, etc.)	2. Analiza problemei, modelare, scenarii, cazuri de utilizare

4.	Cazuri de utilizare	- determinarea cazurilor pentru care utilizatorul dorește să folosească programul creat	2. Analiza problemei, modelare, scenarii, cazuri de utilizare
5.	Proiectare	- definirea structurilor de date necesare, a claselor și interfețelor ce urmează a fi implementate	3. Proiectare
6.	Implementare	- descrierea claselor și a metodelor definite, explicarea funcționării și a utilității acestora; descrierea interfețelor grafice	4. Implementare
7.	Testare	- generarea fișierelor text ce documentează evoluția simulării în 3 cazuri prestabilite	5. Rezultate

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

### 2.1. Analiza problemei

S-au stabilit cerințele funcționale și non-funcționale la care programul, simulatorul de cozi de clienți, trebuie să răspundă:

#### a) Cerințe funcționale:

- a) simulatorul trebuie să permită utilizatorului să introducă datele de simulare referitoare la numărul de cozi și de clienți, precum și la datele ce vor caracteriza clienții <sup>[1]</sup>
- b) simulatorul trebuie să permită utilizatorului să aleagă intervalul de timp pentru care face analiza datelor <sup>[2]</sup>
- c) simulatorul trebuie să permită utilizatorului să introducă numele fișierului output <sup>[3]</sup>
- d) simulatorul trebuie să informeze utilizatorul dacă inputul său nu este valid
- e) simulatorul trebuie să redea în timp real evoluția cozilor de clienți
- f) simulatorul trebuie să furnizeze un fișier la finalul simulării, în care să redea la fiecare secundă starea cozilor de clienți și a potențialilor clienți care nu au fost încă asigurați unei cozi, precum și calculul timpului mediu de așteptare, a timpului mediu de servire și a orei de vârf <sup>[4]</sup>

#### b) Cerințe non-funcționale:

- simulatorul trebuie să fie ușor de folosit și intuitiv
- simulatorul trebuie să aibă o interfață atractivă, atât pentru introducerea datelor de intrare, dar mai ales pentru redarea simulării în timp real <sup>[5]</sup>

## Note:

[1] – Clienții vor fi generați aleatoriu, cu un ID unic și parametrii lor caracteristici într-o plajă de valori configurată de utilizator prin interfață.

[2] – Intervalul de analiză va fi considerat în secunde. Momentul inițial de timp este  $T = 0$ . Momentul final de timp pentru o valoare  $x$  introdusă din interfață este  $x-1$ . Este important de menționat că acest interval reprezintă intervalul maxim de timp pentru care se desfășoară analiza, dar simularea poate fi terminată și mai devreme, în cazul în care nu mai există clienți care să aștepte (nici la coadă, nici „în interiorul magazinului”)

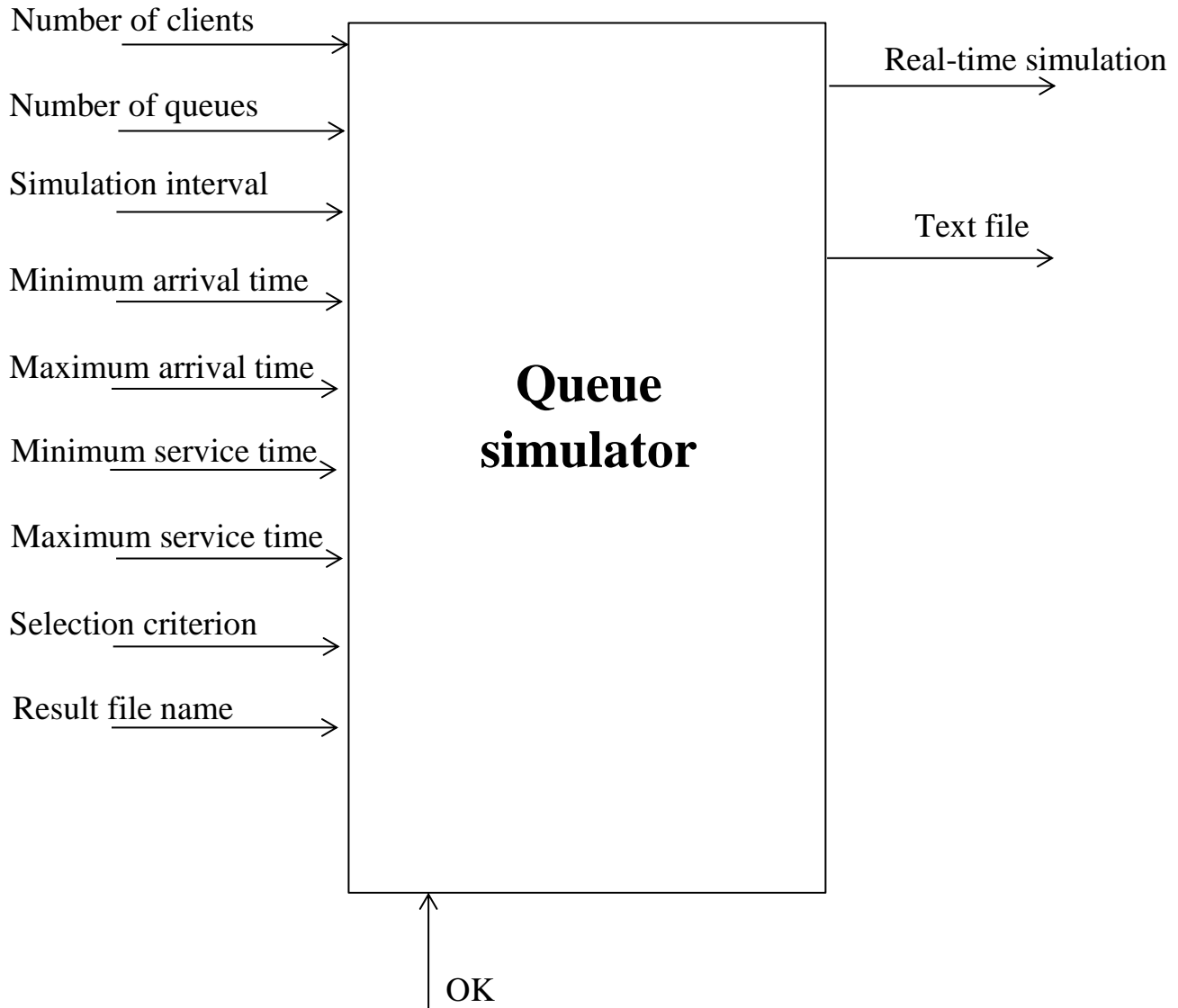
[3] – Pentru maximizarea interacțiunii cu user-ul s-a decis includerea unui câmp în care utilizatorul să specifice numele fișierului în care să se facă scrierea. Acest câmp nu poate fi lăsat necompletat.

[4] – În fișierul de ieșire se va specifica, pe primul rând, strategia de alegere a coziilor (*SHORTEST\_QUEUE* sau *SHORTEST\_TIME*). Apoi, la fiecare moment de timp se vor afișa: timpul curent, clienții care încă nu au fost asignați unei cozi (cei care încă fac cumpărături), toate cozile, cu numele lor, urmate de clienții care așteaptă la respectiva coadă. Pentru fiecare client se va specifica *id*, *arrivalTime* (momentul de timp la care a terminat cumpărăturile și dorește să se pună la rând pentru una dintre casele de marcat), *serviceTime* (intervalul de timp pe care clientul este nevoit să îl (mai) petreacă la casă chiar dacă este primul, adică atât cât (mai) durează ca produsele sale să fie scanate și plătite; după ce clientul a ajuns la casa de marcat, adică este primul în coadă, timpul de servire este succesiv decrementat la fiecare secundă). La finalul simulării se vor afișa datele concluzie ale analizei: timpul mediu de așteptare, timpul mediu de servire și ora de vârf. Considerând că acest simulator ar putea să fie folosit pentru a analiza necesarul de case de marcat pentru un supermarket care ar deservei un număr estimativ de clienți (pe baza unor analize demografice), s-a ales ca timpii medii să fie calculați în raport cu clienții, în totalitatea lor, și nu cu intervalul de timp impus pentru stoparea simulării; dacă în intervalul de timp dat nu se reușește epuizarea listei de clienți de la toate casele, calculele vor analiza în perspectivă simularea, pentru a oferi rezultate mai extinse. Timpul mediu de așteptare este intervalul de timp pe care un client îl petrece, în medie, la casă până să fie el cel servit (până să fie primul), cu alte cuvinte, intervalul de timp în care acesta stă „degeaba”. Timpul mediu de servire este intervalul de timp în care îi sunt scanate, în medie, produsele unui client (cât stă un client, în medie, în fruntea cozii). Ora de vârf este momentul de timp la care magazinul este cel mai aglomerat la casele de marcat (când așteaptă cei mai mulți clienți la coadă)

[5] – Pentru o interfață de simulare cât mai atractivă s-a ales folosirea de imagini pentru a ilustra casele de marcat și clienții.

## 2.2. Modelare

S-a definit modelul de lucru al simulatorului de cozi de clienți, sub formă de schemă bloc:



### 2.3. Scenarii posibile

Scenariile în care se poate găsi simulatorul de cozi de clienți pot fi diferențiate în două mari categorii: scenarii uzuale de utilizare și scenarii limită. În cele ce urmează se va prezenta felul în care programul va răspunde la fiecare dintre aceste tipologii de scenarii:

#### I. Scenariu uzual:

- utilizatorul introduce date valide despre clienți și cozi, intervalul de simulare
- utilizatorul alege strategia de adăugare a clienților
- utilizatorul introduce numele fișierului în care se va documenta simularea
- programul scrie la fiecare secundă datele în fișier și le afișează în mod dinamic în interfața de simulare
- programul calculează timpul mediu de așteptare, timpul mediu de servire și ora de vârf și le scrie la finalul fișierului de ieșire

#### II. Scenarii limită:

##### a. Șir de caractere în loc de numere:

- utilizatorul introduce un șir de caractere pe post de valoare întreagă (pentru timp, număr de clienți sau număr de cozi)
- utilizatorul acționează butonul *OK* care pornește simularea
- programul îl notifică despre faptul că datele pot fi doar numere întregi, și îi dă posibilitatea de a corecta eroarea => utilizatorul are posibilitatea să intre într-un scenariu uzual

##### b. Număr negativ:

- utilizatorul introduce un număr negativ în unul dintre câmpurile pentru date de simulare
- utilizatorul acționează butonul *OK*
- programul îl notifică despre faptul că datele nu pot fi numere negative; utilizatorul are posibilitatea să intre într-un scenariu uzual

##### c. Minim mai mare decât maxim:

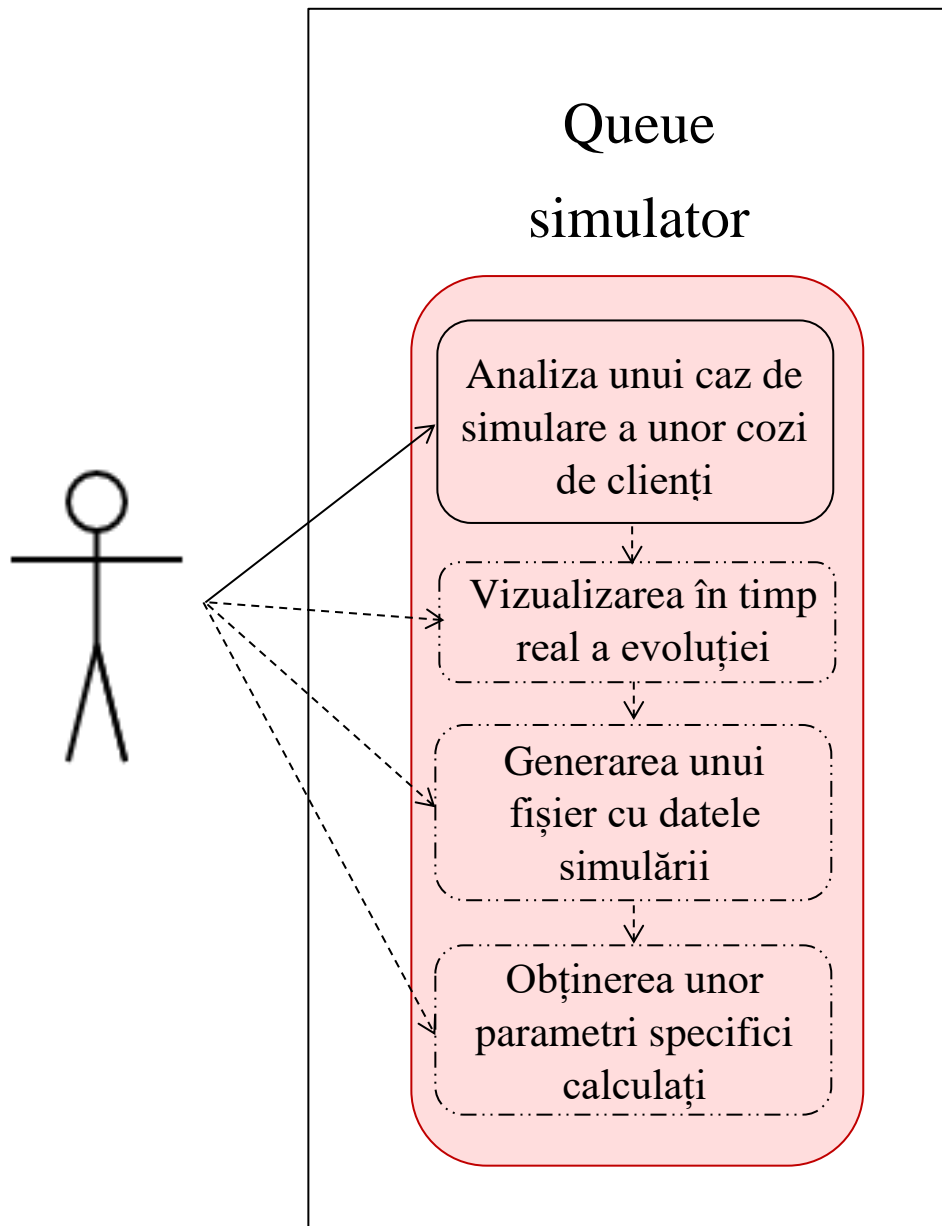
- utilizatorul introduce o valoare de timp minimă mai mare decât valoarea de timp maximă pentru timpul de sosire la casă, ori pentru timpul de servire
- utilizatorul acționează butonul *OK*
- utilizatorul selectează operația de împărțire
- programul îl notifică despre faptul că un minim trebuie să fie mai mic sau egal decât un maxim, și îi dă posibilitatea de a corecta eroarea

##### d. Șirul vid ca nume de fișier

- utilizatorul lasă necompletat câmpul pentru numele fișierului de ieșire
- utilizatorul acționează butonul *OK*
- programul îl notifică despre faptul că numele fișierului trebuie obligatoriu completat, și îi dă posibilitatea să facă această adăugare

#### 2.4. Cazuri de utilizare

Se vor prezenta aici rațiunile pentru care utilizatorul poate dori să folosească acest simulator de cozi de clienți:



### Descrierea utilizării:

Folosirea simulatorului de cozi de clienți este facilă și intuitivă. Datele de intrare cerute nu conțin niciun fel de termeni de specialitate care să deruteze utilizatorul. Mai mult decât atât, comunicarea cu user-ul în caz de eroare la introducerea datelor este una complexă și explicită.

Utilizatorul introduce datele de intrare în câmpurile specifice folosind tastatura propriului computer. Strategia de selecție se va alege prin click dintre cele două strategii disponibile. La apăsarea butonului *OK*, dacă există date incorecte se va afișa pe ecran o casetă de dialog ce va semnaliza eroarea. Dacă toate datele sunt valide, se deschide fereastra de simulare în timp real și începe scrierea în fișierul text. La terminarea simulării fereastra rămâne în continuare vizibilă pentru a putea analiza starea finală.

### Descrierea celor 3 cerințe pe care programul le satisface:

#### i. Vizualizarea în timp real a evoluției:

După introducerea unui set complet și valid de date de intrare, și după acționare butonului *OK*, se va deschide o nouă fereastră în care se va reda pas cu pas evoluția simulării. Momentul de timp  $T = 0$  corespunde afișării tuturor cozilor închise (fără clienți), și a listei complete de clienți generați aleatoriu de program (ordonați crescător în funcție de timpul de sosire la casă). Pe măsură ce timpul avansează, clienții sunt mutați din zona de listei de așteptare în cea a cozilor de la casele de marcat, în funcție de parametrul *arrivalTime* al fiecărui client și de strategia de asignare aleasă din fereastra de introducere a datelor. Simularea se oprește fie la scurgerea timpului de simulare, fie la epuizarea tuturor clienților din cozi și din lista de așteptare. Simularea în timp real este o bună metodă de prezentare cu impact vizual a unei situații date, potrivită pentru a capta atenția oricărei audienței.

#### ii. Generarea unui fișier cu datele simulării:

Programul generează un fișier detaliat cu toate datele înregistrate în fiecare secundă a simulării. Pentru fiecare moment se scriu în fișier: secunda curentă, clienții din lista de așteptare, toate cozile cu toți clienții ce așteaptă la fiecare dintre ele. Fișierul poate fi folosit pentru o analiză exactă a fiecărui moment de pe axa timpului în cadrul simulării.

#### iii. Obținerea unor parametri specifici calculați

La finalul fișierului se scriu datele calculate: *averageWaitingTime*, *averageServiceTime*, *peakHour*. Acestea pot fi utilizate, spre exemplu, în analizele estimative pe care un supermarket le face cu scopul îmbunătățirii serviciilor prestate pentru clienți.

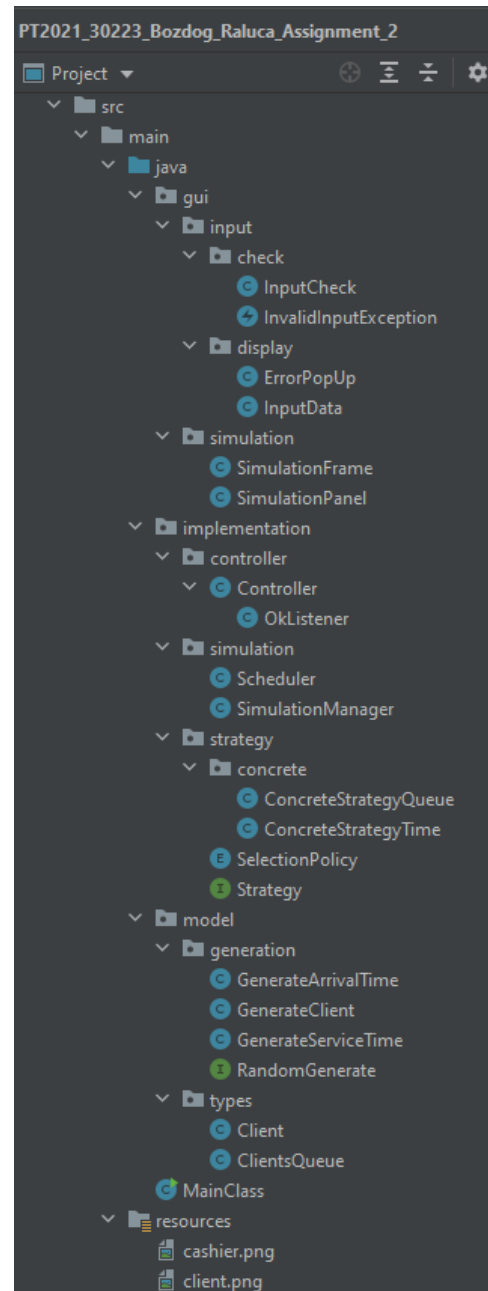
### 3. Proiectare

Pentru a respecta conceptele programării orientate pe obiect, fiecare clasă implementată este cât mai specifică, având cât mai puține funcționalități (ex: clasă separată pentru fiecare operație de generare de parametri ai unui client).

#### 3.1. Pachete

Tiparul arhitectural MVC (Model – View – Controller) este adaptat pentru proiectul de față, fiind implementat diferit față de standard. Clasa Controller a sub-pachetului cu același nume este cea care definește și adaugă *ActionListener* pentru butonul *OK*. Pachetul *gui* conține clasele pentru interfețele grafice (*InputData*, *ErrorPopUp*, *SimulationFrame* și *SimulationPanel*), dar și clasa de verificare a input-ului (*InputCheck*) și cea de excepție la introducerea de date invalide (*InvalidInputException*). Pachetul *model* gestionează definirea claselor *Client* și *ClientsQueue*, dar și generarea aleatorie a variabilelor instanță pentru un client

Proiectul este structurat pe unități funcționale cât mai specifice, clasele care se ocupă de o anumită ramură a programului fiind grupate în pachete și sub-pachete, după cum se poate vedea alături în lista claselor așa cum apare ea în IDE-ul IntelliJ, dar și mai jos în diagrama de pachete:



*Structura proiectului.*

*Organizarea pe pachete*





Diagrama de pachete

### 3.2. Clase

Diagrama de clase a întregului proiect arată relațiile dintre cele 30 de clase și interfețe definite, și este atașată pe următoarea pagină. Clasele vor fi descrise detaliat în secțiunea 4.Implementare.

### 3.3.Interfețe definite

Interfețele definite în proiect sunt *Strategy* și *RandomGenerate*, fiecare dintre acestea având câte o singură metodă abstractă.

În *Strategy*, metoda *addClient* adaugă un *Client* transmis ca parametru într-una din cozile de clienți ale obiectelor *ClientsQueue* din lista de cozi de clienți transmisă ca parametru. Clasele ce implementează această interfață sunt:

- *ConcreteStrategyQueue*
- *ConcreteStrategyTime*

Fiecare dintre acestea oferă o implementare nouă pentru metoda *addClient*, în conformitate cu strategia de selecție sugerată de numele clasei.

În *RandomGenerate*, metoda default *generate* primește ca parametri două valori întregi și returnează un întreg generat aleatoriu în intervalul definit de cele două valori. Clasele ce implementează această interfață sunt:

- *GenerateArrivalTime*
- *GenerateServiceTime*

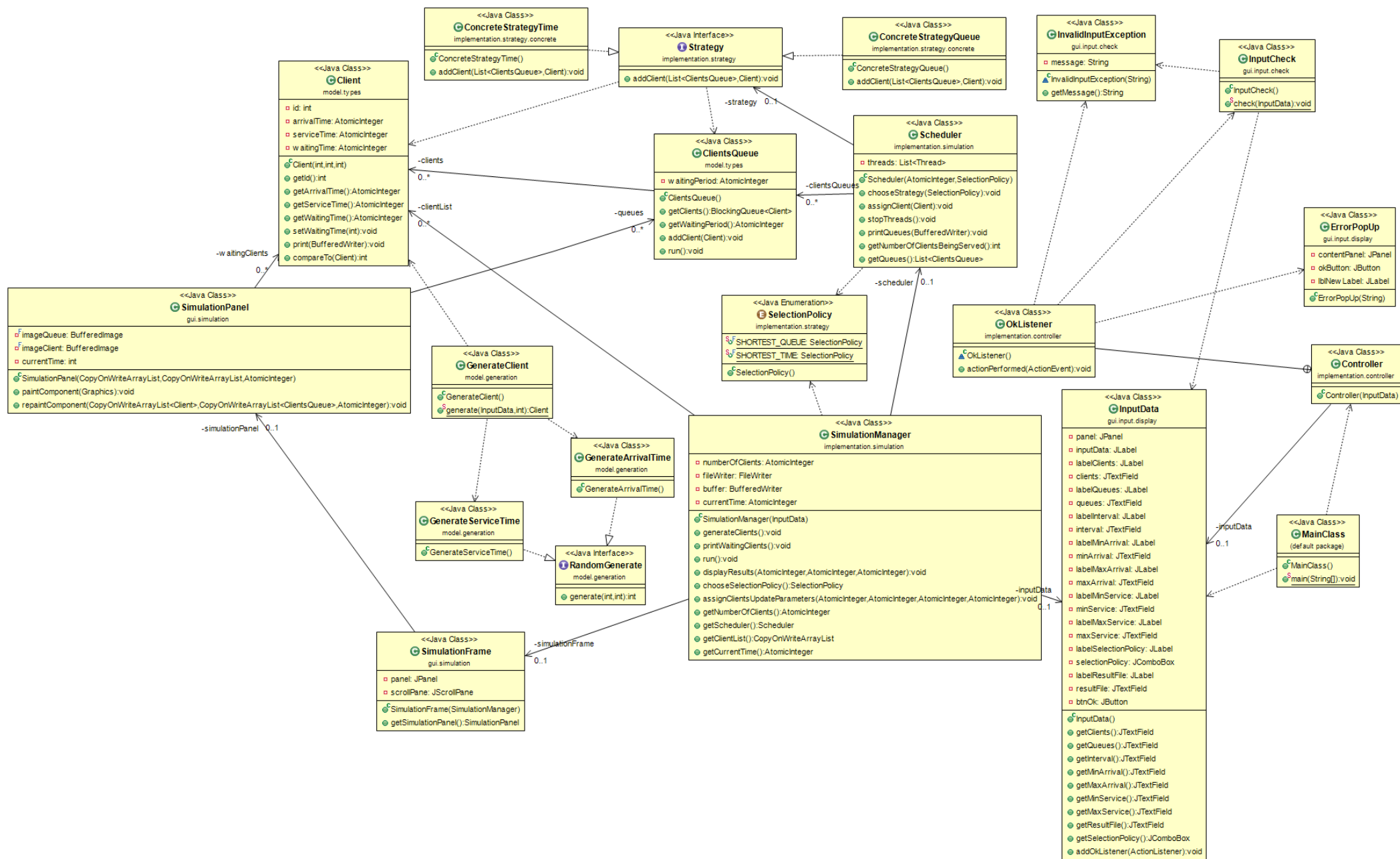
Fiecare clasă oferă o implementare particulară a metodei din interfață, în concordanță cu operația denumită de clasă.

### 3.4.Structuri de date folosite

Din categoria structurilor de date uzuale s-au folosit *List* și *ArrayList*. Spre exemplu, un planificator *Scheduler* conține o listă de cozi de clienți *ClientsQueue* și o listă de thread-uri. Pentru securitatea datelor din punctul de vedere al concurenței și pentru consistența acestora s-au folosit și tipuri de date thread-safe cum ar fi: *AtomicInteger*, *BlockingQueue*, *CopyOnWriteArrayList*.

Clasele *Client* și *ClientsQueue* pot fi considerate și ele structuri de date, întrucât acționează ca un tot-unitar în cadrul programului de față.

*Diagrama de clase*



### 3.5. Algoritmi folosiți

Cu titlatura de algoritmi se pot considera metodele de calcul pentru cei trei parametrii ce vor fi scriși la finalul documentului text al fiecărei simulări. Având în vedere faptul că simularea se poate adresa pentru găsirea numărului optim de case de marcat pentru un supermarket ce estimează un anumit flux de clienți, s-a adoptat opțiunea unui calcul de timpi medii mai extins decât strict pe durata simulării. Un client adăugat într-o coadă va contribui la calculul timpilor medii. Cei trei parametri menționați mai sus sunt:

a. *averageWaitingTime*:

Este timpul mediu de așteptare al unui client la o casă de marcat (din momentul în care a ajuns la coada de la casă și până când a ajuns primul la coadă). Se calculează mai întâi timpul total de așteptare al tuturor clienților: de fiecare dată când un client este scos din lista clienților în așteptare și este introdus în una din cozile de la casă, se adaugă timpul de așteptare al clientului la timpul total. La final se împarte această valoare la numărul de clienți. Timpul de așteptare al unui client este setat la introducerea unui client într-o coadă, și este egal cu timpul de așteptare specific cozii.

b. *averageServiceTime*:

Este timpul mediu de servire al unui client (intervalul de timp în care clientul este primul în coadă). Se calculează timpul total de așteptare al tuturor clienților: de fiecare dată când un client este scos din lista clienților în așteptare și este introdus în una din cozile de la casă, se adaugă timpul său de așteptare la timpul total. La final se împarte această valoare la numărul de clienți.

c. *peakHour*:

Este momentul de timp la care magazinul este cel mai aglomerat (când sunt cei mai mulți clienți simultan la toate cozile). Se setează inițial la 0 valoarea ce va reține numărul maxim de clienți înregistrat simultan la toate casele de marcat. La fiecare moment de timp se calculează câți clienți sunt în total la toate cozile. Dacă această valoare este mai mare decât maximum de până în acest moment, momentul de timp curent devine noul *peakHour*.

## 4. Implementare

S-a folosit conceptul încapsulării din paradigma programării orientate pe obiect, atributele claselor fiind declarate *private* și accesibile claselor exterioare prin metode *getter* și *setter*.

### 4.1. Clasele pachetului *model.types*

#### 4.1.1. *Client*:

Clasă care implementează interfața *Comparable <Client>*, pentru a putea compara doi clienți după *arrivalTime*.

Variabilele instanță ale clasei sunt: *int id*, *AtomicInteger arrivalTime*, *AtomicInteger serviceTime*, *AtomicInteger waitingTime*.

Constructorul clasei primește ca parametri *id*, *arrivalTime* și *serviceTime* și creează un client cu aceste atribute. Trebuie menționat că timpul de așteptare al unui client poate fi determinat doar în momentul în care clientul este asignat unei cozi (după cum este și logic, nu pot ști cât am de așteptat la coadă decât după ce am ales coada la care voi aștepta, și este clar câți clienți mai așteaptă deja înaintea mea).

S-au definit metodele mutatoare *setWaitingTime*, și cele accesoare *getId*, *getArrivalTime*, *getServiceTime* și *getWaitingTime*, cu rolul de a permite accesul la variabilele instanță private ale clasei.

Metoda *public int compareTo (Client o)* este folosită pentru a compara clientul curent și cel transmis ca parametru după timpul de sosire. Ea returnează 1 dacă clientul curent are timpul de sosire mai mare decât clientul *o*, -1 dacă clientul curent are timpul de sosire mai mic decât *o*, și 0 dacă timpul de sosire al clientului curent este egal cu timpul de sosire al clientului *o*. Este important de remarcat faptul că 0 ca rezultat returnat de metoda *compareTo* din *Client* nu înseamnă că acest client și cel transmis ca parametru sunt identici; comparația se referă strict la atributul *arrivalTime* al obiectelor *Client*. Această metodă este folosită la sortarea clienților imediat după generare, înainte de a-i pune într-o coasă.

Metoda *public void print(BufferedWriter buffer)* scrie datele clientului în buffer-ul transmis ca parametru. Scrierea se va face sub forma șablonului impus: „(*id*, *arrivalTime*, *serviceTime*);”. Metoda aruncă o excepție de tipul *IOException* în cazul unor probleme cu buffer-ul de scriere.

#### 4.1.2. ClientsQueue

Clasă care implementează interfața *Runnable*, pentru a putea crea thread-uri care să apeleze metoda *run()* a interfeței, implementată în această clasă.

O coadă de clienți este caracterizată de coada efectivă a clienților, *BlockingQueue<Client> clients*, și de perioada de așteptare, *AtomicInteger waitingPeriod*, timpul pe care un client intrat la un moment dat în coadă este nevoit să îl aștepte până să îi vină rândul

Constructorul clasei alocă spațiu în memorie pentru coada de clienți și setează perioada de așteptare a cozii în momentul inițial la 0.

Metodele *getClients* și *getWaitingPeriod* au scopul de a permite accesarea variabilelor instanță.

Metoda *addClient(Client c)* adaugă clientul transmis ca parametru la coada curentă, realizând și modificările necesare pentru timpii de așteptare.

Metoda *run()* este metoda cu rol de *main()* a thread-ului descris de coada curentă. Instrucțiunile din *run()* se vor executa la apelul *t.start()*, unde *t* este thread-ul corespunzător acestei cozi. Metoda descrie comportare normală a unei cozi de

cumpărături: caut primul client din coadă; dacă timpul lui d serve este mai mare ca 0, deci dacă încă mai are de așteptat la coadă, decrementez *serviceTime*, fiindcă a mai trecut o secundă. Dacă este 0, clientul a fost servit și poate fi scos din coadă. Execuția este sistată pentru o secundă.

#### 4.2. Clasele pachetului *model.generation*

Acesta este pachetul în care este definită interfața *RandomGenerate*, descrisă anterior în secțiunea 3.3. Clasele *GenerateArrivalTime* și *GenerateServiceTime* implementează metoda default *int generate(int min, int max)*.

##### 4.2.1. *GenerateArrivalTime*

Implementează interfața *RandomGenerate*, și, deci, metoda default *int generate(int min, int max)*. Este folosită pentru generarea timpului de sosire a clienților ce vor fi generați aleatoriu.

##### 4.2.2. *GenerateServiceTime*

Implementează interfața *RandomGenerate*, și, deci, metoda default *int generate(int min, int max)*. Este folosită pentru generarea timpului de servire a clienților ce vor fi generați aleatoriu.

##### 4.2.3. *GenerateClient*

Clasă ce conține metoda de generare aleatorie a unui client cu caracteristicile transmise de interfața de intrare, și cu id-ul transmis ca parametru.

#### 4.3. Clasele pachetului *implementation.strategy.concrete*

În pachetul *implementation.strategy* este definită interfața *Strategy*, implementată de *ConcreteStrategyQueue* și *ConcreteStrategyTime*, dar și enumerarea *SelectionPolicy*, folosită pentru a aduce la un loc cele două strategii de asignare a clienților în coadă.

##### 4.3.1. *ConcreteStrategyQueue*

Clasă care oferă implementarea proprie metodei *addClient(List<ClientsQueue> clientsQueues, Client client)* din interfața pe care o implementează. Metoda adaugă clientul transmis ca parametru la cea mai scurtă coadă.

##### 4.3.2. *ConcreteStrategyTime*

Clasă care oferă implementarea proprie metodei *addClient(List<ClientsQueue> clientsQueues, Client client)* din interfața pe care o implementează. Metoda adaugă clientul transmis ca parametru la coada cu cel mai scurt timp de așteptare.

#### 4.4. Clasele pachetului *implementation.simulation*

##### 4.4.1. *Scheduler*

Clasă care gestionează planificarea operațiilor executate de *SimulationManager*.

Atributele sale sunt: *List<ClientsQueue> clientsQueues* (lista tuturor cozilor de clienți), *List<Thread> threads* (lista tuturor thread-urilor corespunzătoare acestor cozi), *Strategy strategy* (strategia de asignare a clienților în coadă).

Constructorul primește ca parametru numărul de cozi ce trebuie să fie create și politica de selecție, stabilește strategia pe baza politicii de selecție, creează cozile și thread-urile corespunzătoare, și denumește sugestiv thread-urile.

Metoda *chooseStrategy(SelectionPolicy policy)* stabilește strategia de selecție.

Metoda *assignClient(Client client)* apelează metoda de adăugare a unui client într-o coadă, corespunzătoare strategiei alese.

Metoda *stopThreads()* întrerupe execuția tuturor thread-urilor corespunzătoare cozilor.

Metoda *printQueues(BufferedWriter writer)* face scrierea în fișier a cozilor și a clienților corespunzători.

Metoda *getNumberOfClientsBeingServed()* returnează numărul de clienți care așteaptă, în total, la toate cozile, la momentul curent de timp.

Metoda *getQueues()* este definită pentru a putea accesa lista cozilor de clienți și din alte clase.

#### 4.4.2. *SimulationManager*

Clasă ce extinde *Runnable*, folosită pentru a gestiona simularea efectivă.

Constructorul inițializează variabilele instanță, deschide fișierul de scriere și buffer-ul corespunzător, generează aleatoriu un număr de N clienți (unde N este extras din interfața cu utilizatorul), și face vizibil JFrame-ul simulării în timp real.

Metoda *generateClients()* generează aleatoriu și sortează clienții crescător după timpul de sosire.

Metoda *printWaitingClients()* scrie în fișier lista clienților care încă nu sunt asigurați niciunei cozi.

Metoda *run()* este cea care se va apela atunci când se va executa *t.start()*, unde *t* este thread-ul corespunzător clasei de față. Este, de fapt, metoda ce descrie funcționarea programului. Dacă încă nu s-a terminat simularea, scriu în fișier ce moment de timp este, asignez clienții ce și-au terminat în acest moment cumpărăturile și sunt gata să intre într-o coadă, actualizez parametrii ce se vor afișa la finalul simulării, redesenez panel-ul simulării în timp real după noile modificări, scriu în fișier care este lista clienților care nu sunt încă în coadă, dar și lista clienților care așteaptă la fiecare coadă, incrementez timpul curent și opresc execuția pentru o secundă. Dacă simularea s-a încheiat pentru că nu mai există clienți rămași nici în lista de așteptare și nici în vreuna dintre cozile de clienți, las încă un pas de simulare să se execute pentru a putea vizualiza cozile goale și lista vidă de clienți în

așteptare. După ce simularea s-a încheiat cu siguranță, opresc toate thread-urile create, scriu în fișierul de ieșire rezultatele calculate, închid buffer-ul și fișierul și întrerup thread-ul curent.

Metoda *displayResults(AtomicInteger totalWaitingTime, AtomicInteger totalServiceTime, AtomicInteger peakHour)* scrie în fișier rezultatele de final de simulare.

Metoda *chooseSelectionPolicy()* alege politica de selecție a cozilor la care să fie asigurați clienții, pe baza datelor introduse de utilizator din interfața grafică.

Metoda *assignClientsUpdateParameters(AtomicInteger totalWaitingTime, AtomicInteger totalServiceTime, AtomicInteger peakHour, AtomicInteger maxNumber)* verifică dintre toți clienții care sunt cei care trebuie asigurați unei cozi la momentul curent de timp, apelează metoda de asignare a variabilei instanță *scheduler*, șterge din lista de așteptare clientul parametru și actualizează datele ce se folosesc în calculul parametrilor concluzie a simulării.

Metodele getter *getNumberOfClients()*, *getScheduler()*, *CopyOnWriteArrayList getClientList()*, *AtomicInteger getCurrentTime()* au rolul de a permite accesul la unele dintre variabilele instanță ale clasei.

#### 4.5. Clasele pachetului *implementation.controller*

##### 4.5.1. Controller

Clasă pentru definirea *ActionListener*-ului pentru interfata grafica de introduce a datelor de intrare.

În constructor se adaugă *ActionListener*-ul corespunzător.

Clasa internă *OkListener* specifică acțiunile ce trebuie să fie executate la apăsarea butonului *OK* de introducere a datelor de intrare: se verifică validitatea datelor de intrare; dacă se generează vreo excepție, atunci o casetă de dialog cu un mesaj corespunzător va fi afișată pe ecran; dacă datele sunt corecte, se creează un *SimulationManager* folosind datele interfeței de intrare, se creează și se începe thread-ul corespunzător acestuia.

#### 4.6. Clasele pachetului *gui*

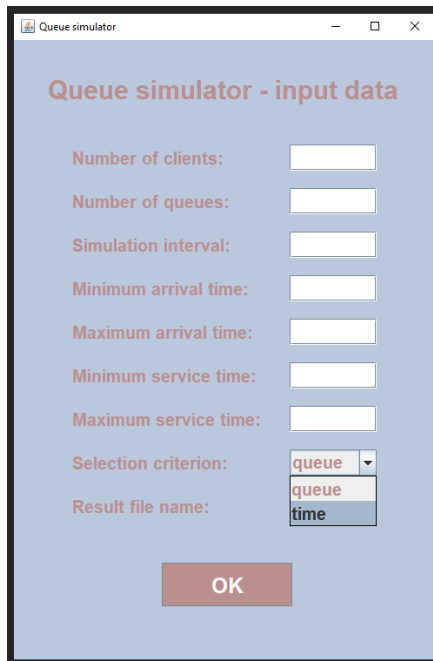
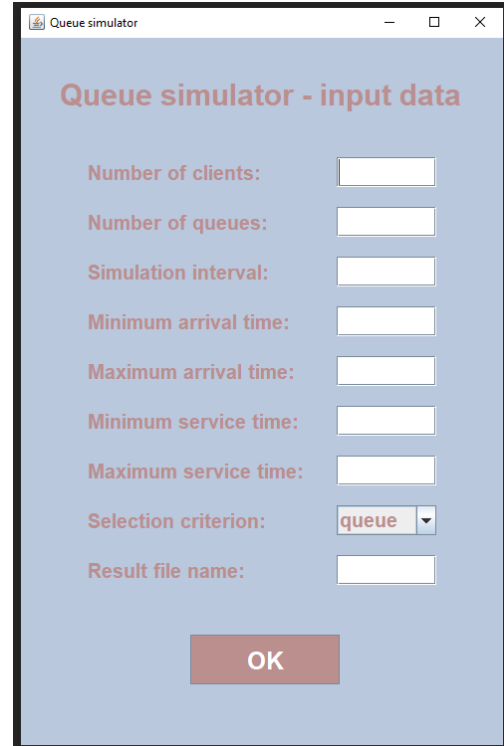
##### 4.6.1. InputData



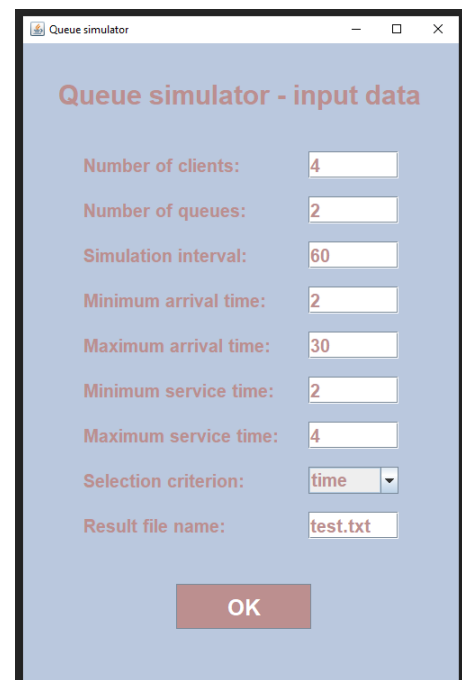
Clasa care creează interfața grafică de introducere a datelor de intrare. Această clasă extinde *JFrame*, ceea ce înseamnă că, în fapt, orice obiect *InputData* este o fereastră ce poate fi configurată grafic.

Toate elementele specifice Java Swing (*JPanel*, *JLabel*, *TextField*, *Button*, *ComboBox*) sunt variabile instanță de tip *private* ale obiectelor *InputData*. Se definesc doar metodele *getter* și *setter* care sunt necesare pentru transmiterea de informații de la utilizator la program și vice-versa.

Pentru butonul *OK* s-a definit o metodă care să îi adauge un *ActionListener*, pentru a putea capta informația că acest buton a fost acționat.



Fereastra de interfață are titlul *Queue simulator* (ca titlu efectiv al ferestrei, eticheta cu rol de titlu în fereastră fiind *Queue simulator – input data*). Etichete text (*JLabel*) îi indică utilizatorului care sunt câmpurile (*TextField*) dedicate introducerii fiecărui tip de dată de intrare, dar și care este câmpul pentru strategia de selecție (*ComboBox*).

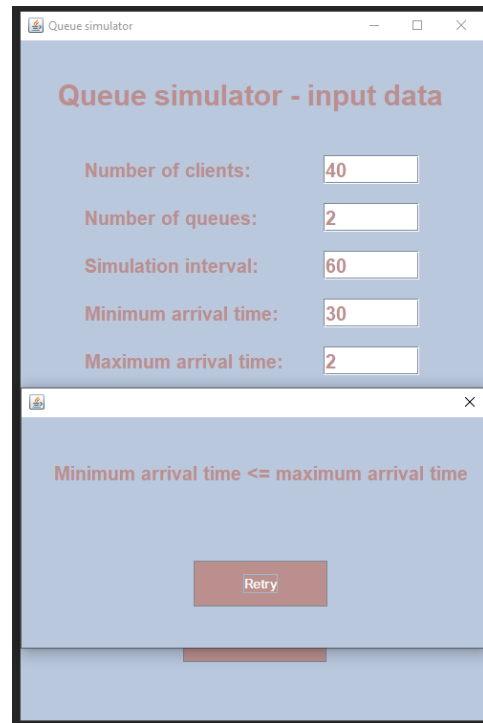
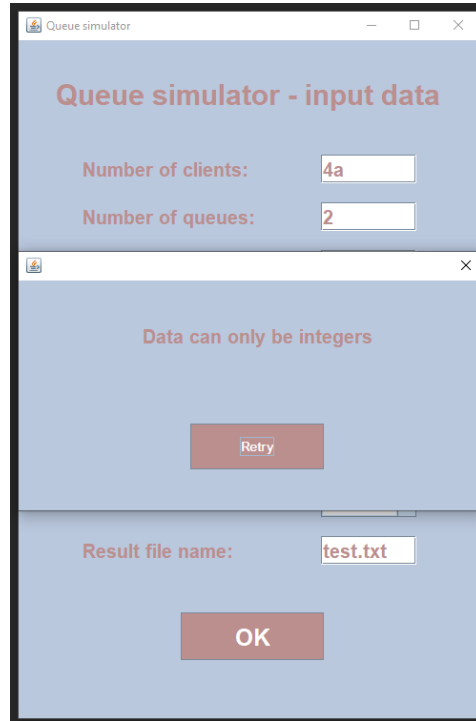
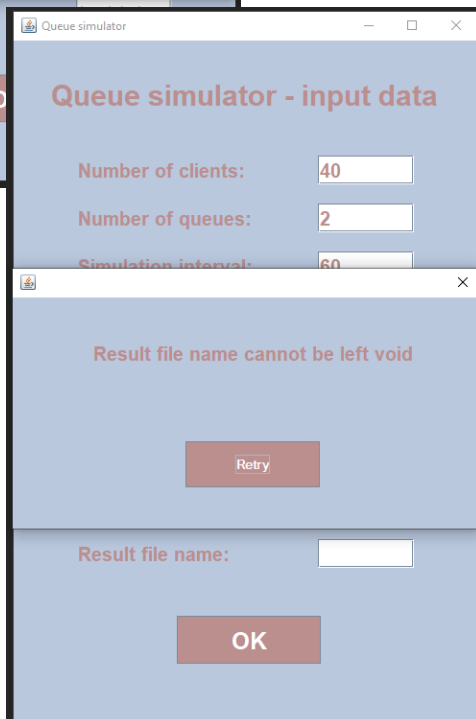
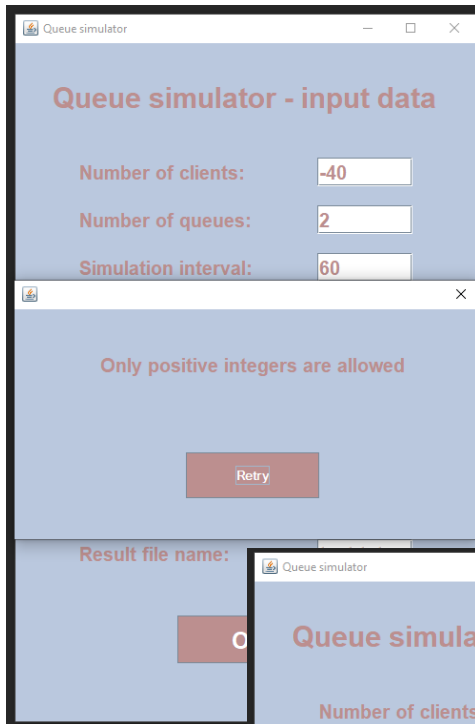


La apăsarea butonului *OK*, programul deschide fereastra simulării în timp real și începe execuția simulării, scriind simultan și în fișierul de ieșire al cărui nume a fost introdus din interfață.

#### 4.6.2. *ErrorPopUp*

Clasă care extinde *JDialog*, reprezentând, de fapt, caseta de dialog prin care utilizatorul este avertizat că inputul său nu este valid.

Aceasta afișează un mesaj sugestiv și îi dă user-ului posibilitatea de a reveni în fereastra principală fie prin închiderea casetei de dialog, fie prin acționarea butonului *Retry* din casetă.



#### 4.6.3. InputCheck

Clasă pentru verificarea validității datelor de intrare introduse de utilizator. Dacă se introduce un șir de caractere pe post de număr întreg, dacă se introduce un întreg negativ, dacă o valoare minimă introdusă este mai mare decât valoarea maximă introdusă pentru același parametru, sau dacă nu se completează câmpul pentru numele fișierului de ieșire, se generează o excepție de tipul *InvalidInputException*.

#### 4.6.4. InvalidInputException

Clasă ce extinde *Exception*, primește în constructor un mesaj corespunzător și îl returnează prin intermediul metodei *getMessage()*.

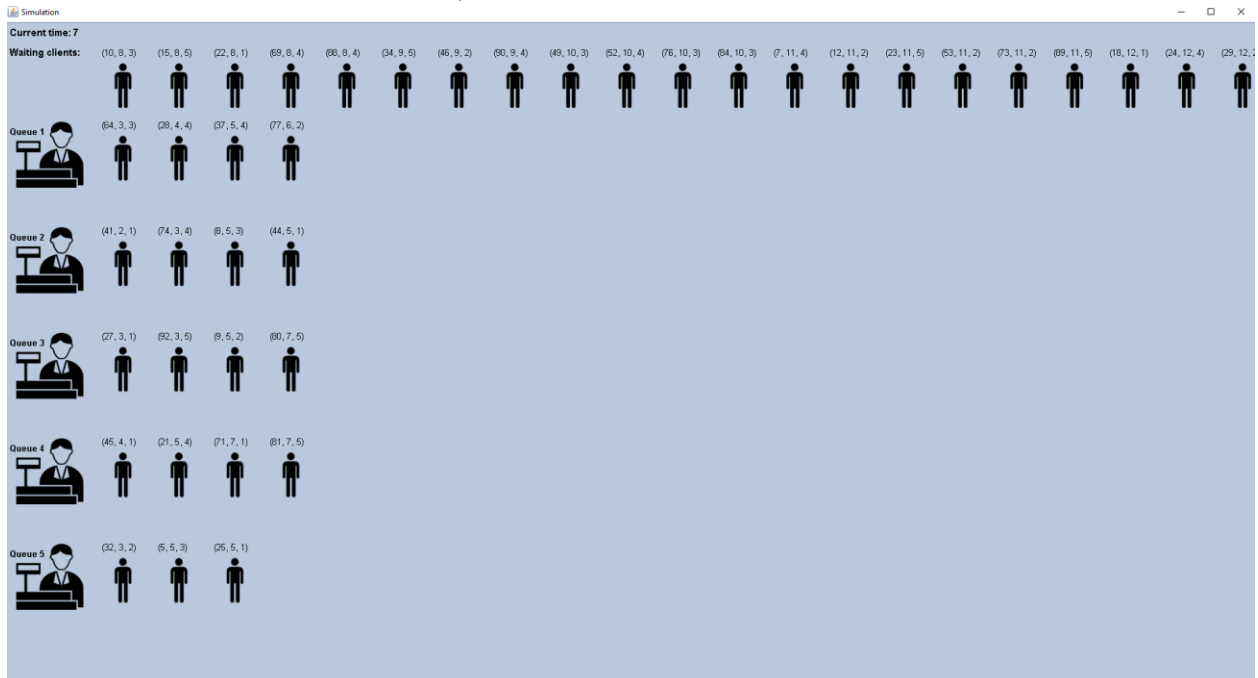
#### 4.6.5. SimulationPanel

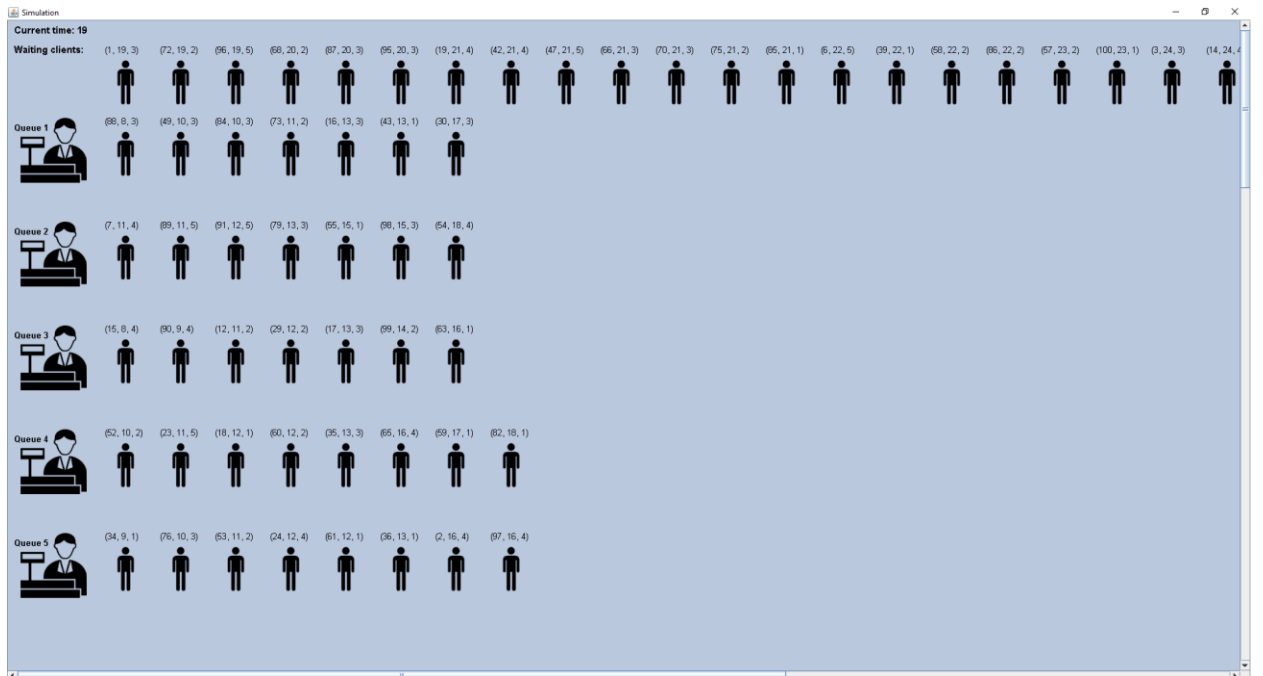
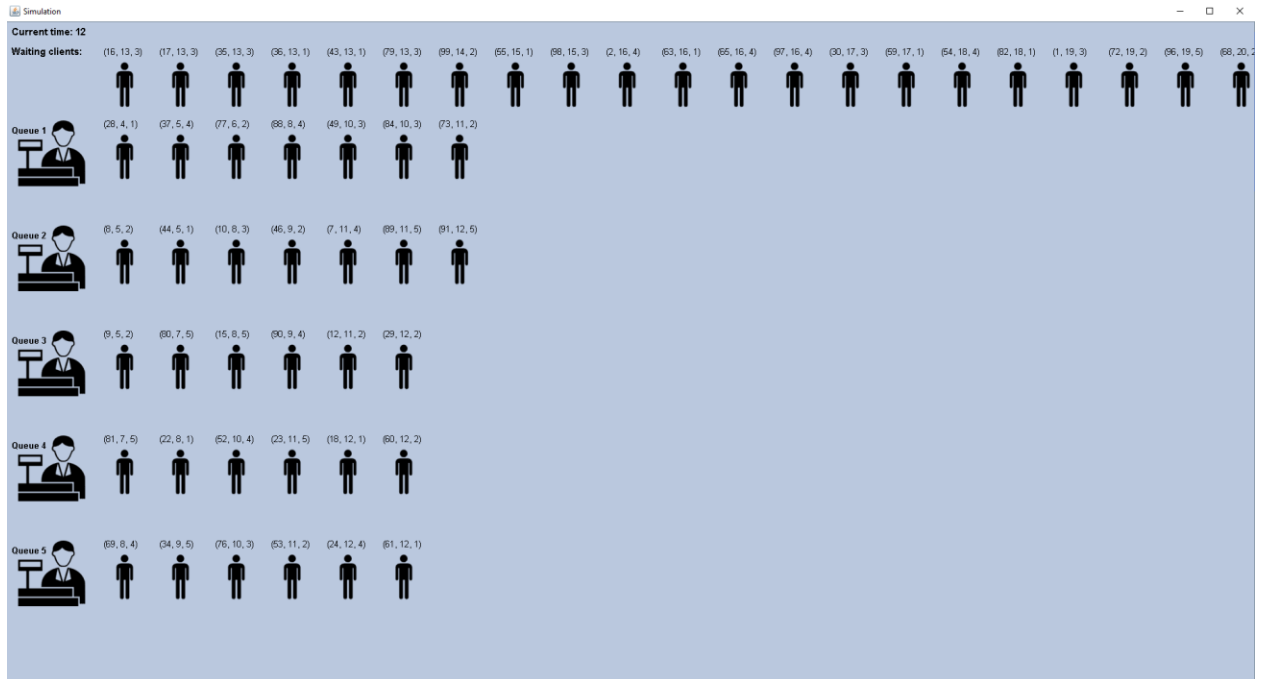
Clasă ce extinde *JPanel* și are ca variabile instanță imaginile folosite în cadrul simulării în timp real, lista de clienți în așteptare și lista cozilor cu clienții aferenți, precum și momentul curent de timp.

Metodele *paintComponent* (*Graphics g*) și *repaintComponent* (*CopyOnWriteArrayList <Client> waitingClients*, *CopyOnWriteArrayList <ClientsQueue> queues*, *AtomicInteger currentTime*) sunt folosite pentru a asigura aspectul de animație al simulării în timp real, adică pentru a actualiza simularea la fiecare secundă.

#### 4.6.6. SimulationFrame

Clasa ce extinde *JFrame* și reprezintă fereastra de simulare în timp real. Are ca variabile instanță un *JPanel* principal, un *JScrollPane* pentru a permite derularea sus-jos și stânga-dreapta în fereastră, și un *SimulationPanel* care să redea simularea efectivă.





#### 4.8. MainClass

Clasa principală este singura care are implementată metoda *public static void main (String[] args)*, deci aceasta este clasa care se va rula pentru pornirea programului.

În metoda *main* a acestei clase se declară și se instanțiază un obiect de tip *InputData* și unul de tip *Controller*, care are obiectul *InputData* ca variabile instanță. În această metodă principală este făcută vizibilă fereastra datelor de intrare.

### 5. Rezultate

În urma definitivării proiectului, pe lângă multitudinea de teste executate doar prin rularea de exemple aleatorii și compararea, la fiecare secundă, a rezultatului afișat în fereastra de simulare, cu rezultatul scris în fișierul de ieșire, cu rezultatul așteptat pe baza datelor, s-au efectuat și cele 3 teste predefinite în specificația proiectului, teste pentru care datele de intrare introduse și fișierele text sunt anexate la finalul lucrării de față, în Anexa 1, Anexa 2 și Anexa 3.

### 6. Concluzii

Tema de față a prezentat elemente de noutate în primul rând prin thread-urilor și a tipurilor de date thread-safe.

Elementele de grafică au fost, de asemenea, o provocare în realizarea proiectului.

Alte direcții de dezvoltare ulterioară ar putea include:

- îmbunătățirea modului de vizualizare a simulării în timp real (rezolvarea decalajului dintre o ștergere și o redesenare succesivă a JPanel)
- adăugarea unor efecte 3D butoanelor ca upgrade al interfeței cu utilizatorul
- adăugarea unui buton de reset care să pornească o nouă simulare pe baza datelor introduse de utilizator
- inserarea în interfața de simulare a unui buton de stop, care să oprească rularea simulării
- includerea posibilității de a pune pauză simulării un timp nedefinit, și de a o relua la un moment de timp ulterior

### 7. Bibliografie

- TP2020-2021\_Descriere\_Laborator.pdf
- ASSIGNMENT\_2\_SUPPORT\_PRESENTATION.pdf
- Tema2.pdf
- <https://docs.oracle.com/javase/tutorial/2d/images/drawimage.html>
- [https://www3.ntu.edu.sg/home/ehchua/programming/java/J4b\\_CustomGraphics.html](https://www3.ntu.edu.sg/home/ehchua/programming/java/J4b_CustomGraphics.html)
- <https://www.codejava.net/java-se/file-io/how-to-read-and-write-text-file-in-java>
- <https://www.baeldung.com/java-atomic-variables>
- <https://www.baeldung.com/java-thread-stop>
- <https://www.educative.io/edpresso/how-to-generate-random-numbers-in-java>
- <https://www.geeksforgeeks.org/enum-in-java/>