MINISTERUL EDUCAȚIEI ȘI CERCETĂRII
UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

TEHNICI DE PROGRAMRE FUNDAMENTALE

TEMA 2

Simulator de cozi

Dragoș-Bogdan Lazea, grupa 30223

An universitar: 2020 – 2021

1. Obiectivul temei

Obiectivul principal al acestei teme îl reprezintă proiectarea și implementarea unei aplicații care să simuleze funcționarea sistemelor bazate pe politici de funcționare de tip coadă, minimizând timpul de așteptare al clienților și dispunând de o interfață grafică intuitivă pentru a facilita interacțiunea cu orice tip de utilizator.

Obiectivele secundare ale aceste teme sunt prezentate în tabelul de mai jos:

Obiectiv secundar	Descriere	Secțiune
1. Analiza problemei	- presupune analiza și delimitarea tuturor cerințelor pe care aplicația trebuie să le îndeplinească;	2. Analiza problemei, modelare, scenarii, cazuri de utilizare
2. Identificarea scenariilor	- presupune identificarea și analiza tuturor scenariilor în care s-ar putea regăsi aplicația în funcție de valorile de intrare introduse de către utilizator (cazuri limită, excepții, funcționare corespunzătoare);	2. Analiza problemei, modelare, scenarii, cazuri de utilizare
3. Identificarea cazurilor de utilizare	- presupune identificarea situațiilor în care aplicația își justifică funcționalitatea;	2. Analiza problemei, modelare, scenarii, cazuri de utilizare
4. Proiectarea propriu-zisă	- presupune parcurgerea etapelor de sintetizare și proiectare a aplicației conform paradigmei programării orientat pe obiecte, prin realizarea de diagrame UML, proiectarea claselor, imaginarea interfeței grafice cu utilizatorul etc.	3. Proiectare
5. Implementarea aplicației	- presupune dezvoltarea propriu zisă a claselor și metodelor aferente acestora în cod scris în limbajul de programare Java;	4. Implementare
6. Testarea aplicației implementate	- presupune testarea funcționalității aplicației pe cele trei seturi de intrări furnizate în specificație problemei și verificarea rezultatelor obținute;	5. Rezultate

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

a. Analiza problemei

Problema de soluționat este reprezentată de faptul că în lumea reală, sistemele ce utilizează politici de funcționare de tip coadă, precum magazinele, dovedesc adesea ineficiență în ceea ce privește timpul de așteptare al clienților ce urmează a fi serviți.

Prin urmare, a fost identificat, în urma analize problemei menționate, următorul cadru de cerințe funcționale:

- Aplicația trebuie să poată permite introducerea numărului de clienți și de servere disponibile, a intervalului de simulare, a limitelor de timp în ceea ce privește sosirea clienților și procesarea cerințelor acestora;
- Aplicația trebuie să informeze utilizatorul dacă inputul introdus este invalid, i. e. datele introduse nu reprezintă numere întregi pozitive sau limitele inferioare ale intervalelor de timp sunt mai mari decât limitele superioare;
- Aplicația trebuie să poată permite selectarea tipului de politică conform căreia se realizează distribuirea clienților în cozi;
- Aplicația trebuie să permită generarea aleatoare a unui număr de clienți egal cu numărul introdus de utilizator în interfața grafică asociată;
- Aplicația trebuie să permită distribuirea optimă a clienților generați spre cozile disponibile în funcție de tipul de strategie adoptat: coada cea mai scurtă sau timpul de așteptare minim.
- Aplicația trebuie să poată asigura funcționarea concurentă și independentă a celor Q cozi, astfel încât aceasta să fie o copie fidelă a sistemelor reale.

- Aplicația trebuie furnizeze în timp real fiecare pas al simulării cât și rezultatele finale în ceea ce privește timpul mediu de așteptare, timpul mediu de procesare și ora de vârf determinate în urma simulării curente;
- Aplicația trebuie să permită ștergerea inputurilor introduse în vederea pregătirii pentru efectuarea unei noi simulări;

Cadrul cerințelor nefuncționale conturate în jurul problemei de rezolvat este:

- Aplicația trebuie să dispună de o interfață grafică intuitivă.
- Aplicația trebuie să fie ușor de utilizat pentru orice tip de utilizator, indiferent de domeniul de activitate al acestuia.

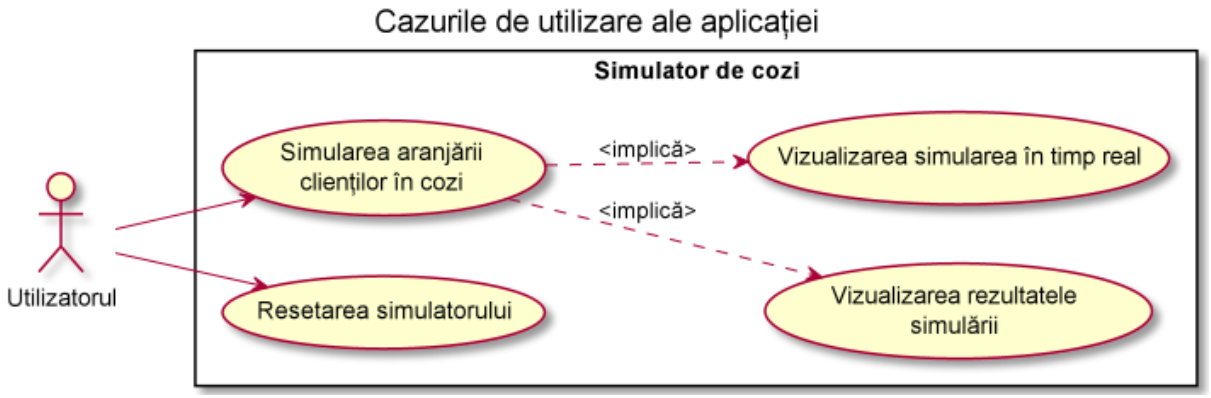
b. Modelare

Aplicația a fost proiectată pe modelul unui sistem cu nouă intrări și trei ieșiri, intrările fiind reprezentate numărul de clienți considerați pentru simulare (***N***), numărul de cozi (***Q***), timpul asociat simulării ($t_{simulation}$), limitele intervalului pentru timpul de sosire a clienților ($t_{arrivalMin}$, $t_{arrivalMax}$), cele pentru timpul de procesare ($t_{serviceMin}$, $t_{serviceMax}$) și o intrare care să permită resetarea în vederea efectuării unui nou pas de simulare, în timp ce ieșirile sunt reprezentate de timpul mediu de așteptare ($t_{avgWaiting}$), timpul mediu de procesare ($t_{avgService}$) și ora de vârf (***peakHour***).

Modelul problemei este astfel translatat într-un model orientat pe obiecte, ce are la bază clasele Client, Server și Scheduler prin intermediul cărora se vor defini și transpune în modelul obiectual intrările și ieșirile aplicației, definite pe baza modelului din lumea reală.

c. Cazuri de utilizare și scenarii posibile

Diagrama cazurilor de utilizare este prezentată mai jos:



În cele ce urmează vor fi descrise cazurile de utilizare alte aplicației dezvoltate:

Use Case: Simularea aranjării clienților în cozi
Actorul principal: Utilizatorul
Scenariul de succes:

1. Utilizatorul introduce corect informațiile necesare simulării în fereastra de introducere a datelor din interfața grafică.
2. Utilizatorul pornește simularea prin apăsarea butonului „Start Simulation”.
3. Aplicația permite vizualizarea în timp real a simulării în fereastra dedicată, etapele și rezultatele acesteia fiind scrie în fișierul „Log-of-events.txt”, acest caz de utilizare fiind astfel constituit din două subcazuri elementare: vizualizarea simulării în timp real în interfața grafică și vizualizarea rezultatelor obținute la finalul simulării.

Scenarii alternative (posibile erori): Cel puțin una dintre valorile introduse de către utilizator în interfața grafică nu este un număr întreg sau cel puțin una dintre limitele inferioare ale celor două intervale de timp este mai mare decât limita superioară corespunzătoare.

1. Utilizatorul este notificat prin apariția unei ferestre cu un mesaj de eroare corespunzător excepției generate.

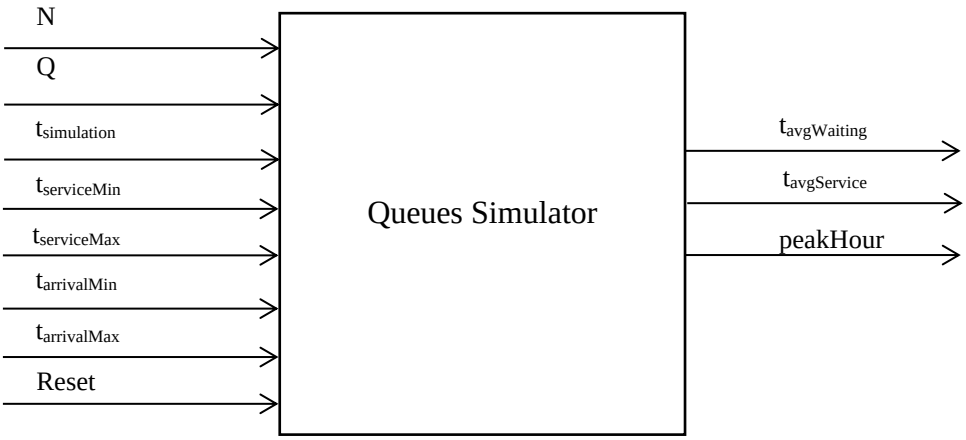
2. Aplicația este închisă automat, scenariul întorcându-se mai apoi, după redeschiderea acesteia, la pasul de introducere a informațiilor necesare simulării de către utilizator.

Use Case: Resetarea
Actorul principal: Utilizatorul
Scenariul de succes:

1. Utilizatorul efectuează click pe butonul „Reset Simulator”.
2. Aplicația va reveni în starea inițială prin golirea câmpurilor de text din fereastra dedicată introducerii informațiilor în vederea pregătirii pentru o eventuală simulare viitoare.

3. Proiectare

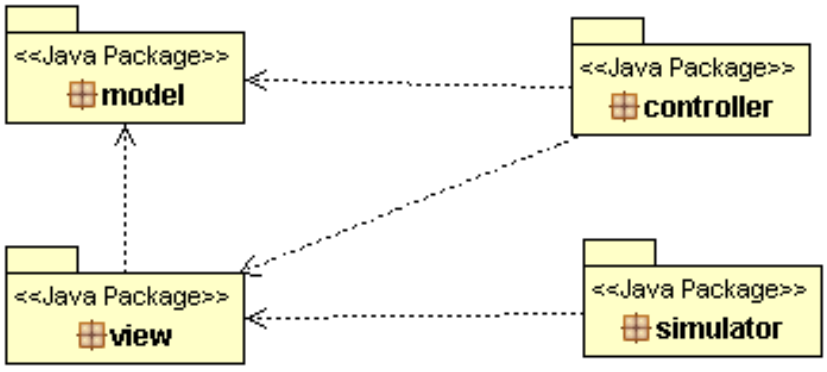
Primul nivel de proiectare îl reprezintă imaginea de ansamblu asupra aplicației prin realizarea schemei bloc a aplicației (cutia neagră a sistemului), care poate fi vizualizată mai jos.



Următoarea etapă a proiectării este reprezentată de divizarea aplicației în subaplicații cu funcționalități specializate. Subsistemele componente ale aplicației fiind organizate conform unui pattern arhitectural de tip MVC (**Model-View-Controller**), următorul nivel de divizare al aplicației putând fi identificat în diagrama de pachete prezentată mai jos.

Modelul încapsulează datele specifice aplicației și definește obiectele, logica și computațiile care stochează, manipulează și procesează datele furnizate de utilizator, în timp ce vederea redă conținutul modelului, fiind responsabilă de realizarea interfeței grafice cu utilizatorul, iar controllerul este responsabil pentru legarea celor două subsisteme anterior menționate, având drept scop simularea propriu-zisă și furnizarea rezultatelor cerute.

Astfel, simulatorul de cozi va fi compus din cele trei componente menționate mai sus, interconectate astfel încât să formeze un ansamblu unitar, fapt ce poate fi observat în următoarea diagramă de pachete.

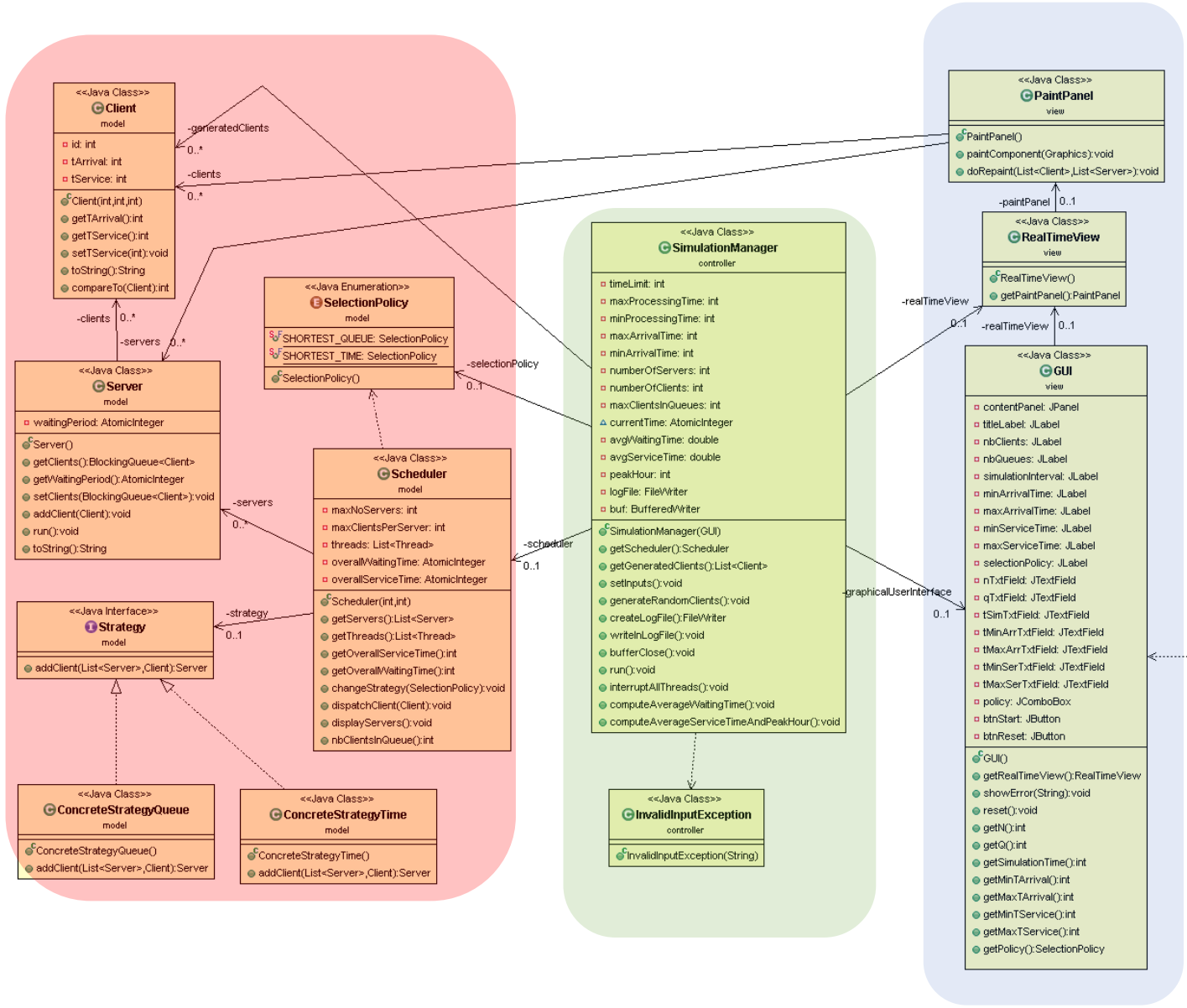


Trecând la cea de-a treia etapă de proiectare, se continuă descompunerea problemei și astfel se definesc cele mai de jos unități din descompunerea aplicației conform paradigmei programării orientate pe obiecte (POO), și anume clasele definite de dezvoltator.

Astfel, în cadrul modelului, responsabil de efectuarea computațiilor necesare îndeplinirii cerințelor funcționale, se dovedesc a fi necesare clase pentru definirea unor obiecte precum Client, Server, reprezentat printr-o lisă de clienți, reprezentând coada propriu zisă, clasa Scheduler realizează distribuirea clienților la servere conform politicii de selectare alese, în timp ce clasa SimulationManager execută simularea propriu-zis, iar clasele GUI și RealTimeView sunt responsabile de realizarea interfeței grafice. De asemenea, singura interfață definită de către dezvoltator este interfața Strategy, implementată de clasele ConcreteStrategyTime și ConcreteStrategyQueue, necesare diferențierii în funcție de politica se selectare a clienților adoptată.

Descrierea tuturor claselor va fi detaliată în secțiunea dedicată implementării aplicației, împreună cu metodele importante și definitorii fiecărei clase din componența proiectului.

În cele ce urmează va fi expusă diagrama UML de clase ale aplicației, denumirile acestora fiind sugestive și inspirate din domeniul problemei. Colorarea diferită a grupurilor de clase din diagrama de mai jos indică apartenența acestora la pachetele expuse în diagrama de pachete, prezentată anterior.



Dintre algoritmii utilizați la proiectarea soluției, relevanți pentru analiză în contextul dezvoltării aplicației se dovedesc algoritmii pentru calculul timpilor medii de așteptare și procesare. Aceștia sunt reprezentați calculul unei medii aritmetice, fiind însă necesară diferențierea tipurilor de clienți luați în calcul în funcție de faptul dacă aceștia au fost sau nu procesați, respectiv a intervalelor de timp asociate. Astfel, numărul de clienți considerați pentru calculul timpului mediu de procesare este numărul clienților care, până la momentul finalizării simulării, au fost deja procesați, în timp ce pentru timpul mediu de așteptare este luată în considerare întreaga perioadă de timp în care clienții așteaptă la rând (sunt atribuiți unui server de către planificator).

Algoritmul pentru determinarea orei de vârf îl reprezintă cel de determinare a maximului dintr-o mulțime de numere, ora de vârf fiind momentul de timp în care în cozile asociate planificatorului se găsește un număr maxim de clienți.

4. Implementare

4.1. Clase, interfețe și metode definite

Denumirile claselor, ale variabilelor instanță și ale metodelor ce urmează a fi sintetizate și prezentate în cele ce urmează sunt sugestive, nefiind necesară explicitarea semnificației acestora.

a. Pachetul model

- **Clasa Client** modelează entitatea client, reprezentând-o prin trei variabile instanță: identificator unic (ID), timpul de sosire și timpul necesar procesării cerințelor sale. Metodele cele mai reprezentative ale clasei fiind:

Antetul metodei	Utilizarea
`String toString()`	- afișarea rezultatelor în interfața grafică sub formă de string-uri și scrierea acestora în fișierul „Log-of-events.txt”
`int compareTo(Client o)`	- compararea a doi clienți, necesară sortării în funcție de timpul de sosire

- **Clasa Server** modelează conceptul de coadă, având drept atribute o listă de clienți, declarată de tip BlockingQueue, și o perioadă de așteptare asociată, de tip AtomicInteger. Tipurile celor două variabile instanță ale clasei sunt utilizate cu scopul menținerii siguranței și a corectitudinii execuției concurente, distribuite pe mai multe fire de lucru: BlockingQueue suportă o operație de așteptare înainte de a extrage un element din listă astfel încât să asigure extragerea unui element nenul, iar AtomicInteger permite realizarea sincronizării în ceea ce privește returnarea și actualizarea automată a valorii variabilei, prin metode de tipul getAndIncrement() sau getAndDecrement(). Această clasă implementează interfața Runnable, permițând astfel procesarea în paralel a fiecărei cozi asociate planificatorului. Metodele semnificative ale clasei Server sunt:

Antetul metodei	Utilizarea
`void addClient(Client c)`	- adăugarea unui client în coadă și actualizarea perioadei de așteptare asociate cozii
`public void run()`	- eliminarea primului client din coadă când timpul său de procesare devine 0

- **Enumerarea SelectionPolicy** conține cele două politici posibile de selectare a cozii la care se va adăuga clientul: ***SHORTEST_QUEUE*** (în coada cea mai scurtă), ***SHORTEST_TIME*** (în coada a cărei perioadă de așteptare este minimă).

- **Interfața Strategy** conține antetul metodei de adăugare a unui client la unul dintre serverele asociate planificatorului, `Server addClient(List<Server> serversList, Client c)`,

care va fi implementată, în funcție de strategia abordată, în clasele **ConcreteStrategyTime** și **ConcreteStrategyQueue**.

- **Clasa Scheduler** este reprezentată printr-o listă de servere și o listă thread-uri, fiecare thread fiind asociat unui server. Practic, în această clasă se realizează adăugarea clienților la servere, în funcție de câmpul strategy. Pentru variabilele de tip listă s-a folosit tipul CopyOnWriteArrayList pentru a garanta siguranță în gestiunea execuției concurente. Metodele cele mai reprezentative ale aceste clase sunt:

Antetul metodei	Utilizarea
`void changeStrategy(SelectionPolicy policy)`	- selectarea strategiei de adăugare a clienților în cozi în funcție de politica de selecție transmisă ca parametru
`void dispatchClient(Client c)`	- adăugarea efectivă a clientului conform cu strategia adoptată
`int nbClientsInQueue()`	- determină numărul total de clienți din care se află într-una din cozile asociate planificatorului

b. Pachetul view

- **Clasa PaintPanel** este definește panoul în care se vor reprezenta grafic serverele și clienții folosind metodele `void paintComponent(Graphicds g)` și `void doRepaint(List<Client> cl, List<Server> sr)`, permițând vizualizarea simulării în timp real în interfața grafică a aplicației, reprezentând panoul de conținut pentru fereastra dedicată simulării, definită de clasa **RealTimeView** din același pachet.
- **Clasa GUI** este responsabilă de fereastra în care utilizatorul introduce informațiile necesare simulării, definite anterior ca intrări ale sistemului, având drept atribut și un manager de simulare care suportă execuția propriu-zisă a simulării. Metoda `void reset()` aduce interfața grafică în starea inițială, caracterizată prin faptul că sunt necompletate câmpurile de text, iar metoda `void showError(String errMessage)` permite apariția unei ferestre ce afișează un mesaj de eroare în cazul apariției unei excepții de tipul **InvalidInputException** sau a unei situații critice, când una dintre limitele inferioare ale unuia dintre intervalele de timp este mai mare decât limita superioară aferentă. De asemenea, clasa **GUI** conține numeroase metode de accesare a textului din câmpurile de text, pentru parsarea intrării sub forma acceptată de către model. Cele două butoane, Start Simulation și Reset Simulator, au definiți ascultători sub forma unor clase interne anonime: la apăsarea butonului Start Simulation, dacă nu apare nicio situație excepțională, se creează un nou obiect de tipul SimulationManager care va lansa în execuție întreaga simulare a procesului de minimizare a timpului de așteptare în coadă, iar la apăsarea butonului Reset Simulator, se va apela metoda `void reset()`.

c. Pachetul controller

- **Clasa InvalidInputException** reprezintă o clasă excepție care este aruncată în momentul în care utilizatorul nu introduce date valide (numere întregi pozitive) în interfața grafică dedicată.

- **Clasa SimulationManager** realizează execuția propriu zisă a simulării și calcularea rezultatelor obținute în urma acesteia, având drept metode reprezentative:

Antetul metodei	Utilizarea
`void setInputs() throws InvalidInputException, NumberFormatException`	- setează variabilele instanță la valorile extrase din interfața grafică a aplicației
`void generateRandomClients()`	- generează o listă aleatoare de clienți, conform informațiilor specificate de utilizator în ceea ce privește

	intervalele pentru timpul de sosire și timpul de procesare
`void writeInLogFile()`	- scrie etapele și rezultatele simulării în fișierul „Log-of-events.txt”
`void run()`	- execuția propriu-zisă (se adaugă clienții la servere, se procesează, se elimină din listele aferente serverelor)
`void computeAverageWaitingTime()`	- calculează și scrie în fișier timpul mediu de așteptare
`void computeAverageServiceTimeAndPeakHour()`	- determină și scrie în fișier timpul mediu de procesare și ora de vârf

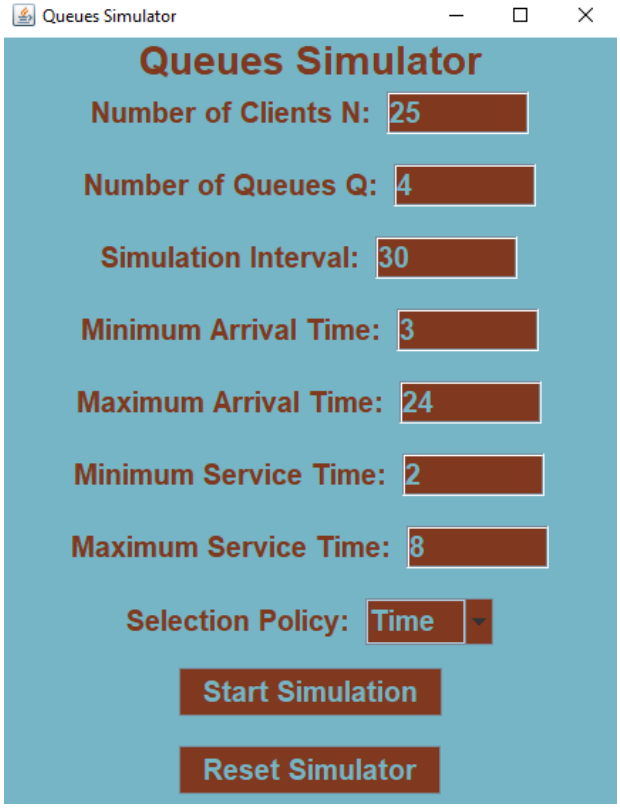
d. Pachetul simulator

- **Clasa QueuesSimulator** instanțiază într-o metodă ***`main`*** componenta GUI a unui simulator de cozi, care, prin crearea și instanțierea tuturor componentelor necesare, permite rularea aplicației dezvoltate.

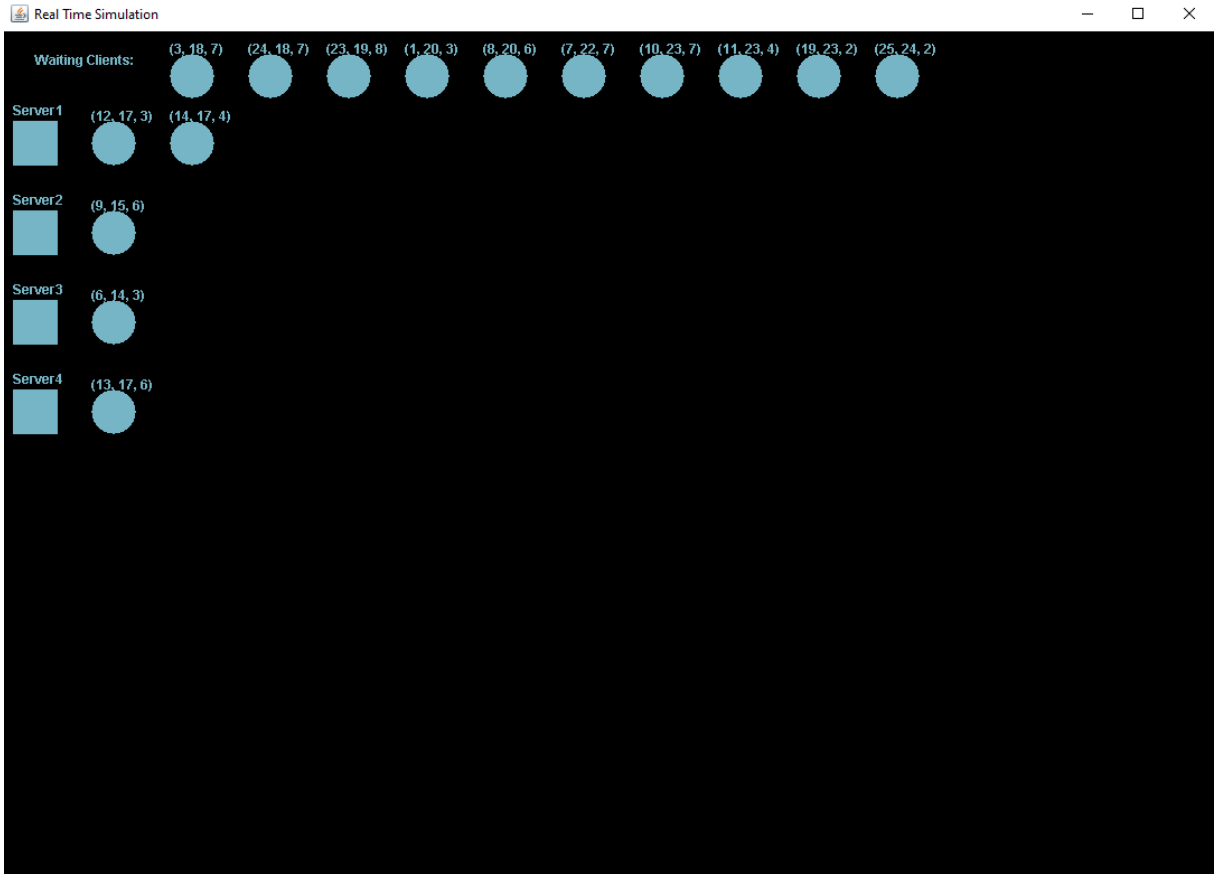
4.2. Interfața grafică cu utilizatorul (GUI)

Aplicația dispune de o interfață grafică intuitivă, dezvoltată folosind Swing și fiind construită pe baza a două JFrame: o fereastră dedicată introducerii informațiilor necesare simulării și o fereastră pentru vizualizarea în timp real a etapelor de simulare a procedeului de distribuire optimă a clienților în cozi.

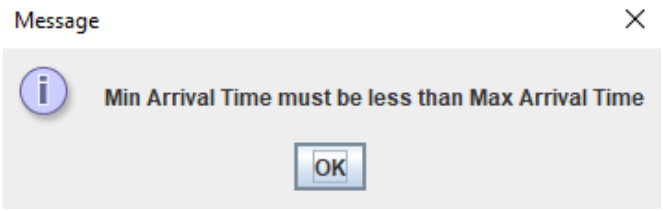
Fereastra dezvoltată cu scopul introducerii datelor de către utilizator are un panou principal în care au fost adăugate diferite subpanouri, implementate folosind JPanel. Astfel, interfața grafică are în componența sa șapte câmpuri de text editabile pentru introducerea informațiilor de simulare de către utilizator. Etichetele sunt realizate folosind JLabel, în timp ce câmpurile de text au la bază componente de tip JTextField. De asemenea, în interfața grafică a aplicației se pot regăsi două butoane, „Start Simulation” și „Reset Simulator”. De asemenea, aceasta conține un element de tip JComboBox ce permite selectarea strategiei de adăugare în coadă. Această componentă componentă a interfeței grafice poate fi vizualizată în figura de mai jos:

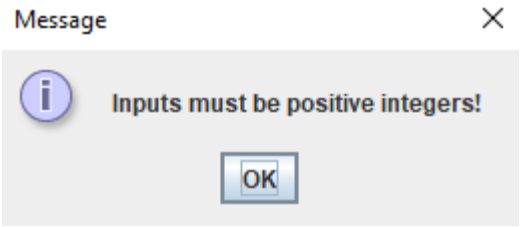


Fereastra pentru vizualizarea simulării în timp real are la bază un JPanel în care clienții sunt desenați folosind cercuri deasupra cărora sunt scrise informațiile specifice fiecăruia dintre ei, iar serverele sunt reprezentate prin pătrate etichetate cu indicele corespunzător. Clienții neasociați unei cozi se află în partea superioară a panoului, fiind identificați prin „Waiting Clients". La fiecare moment de timp informația referitoare la listele de clienți se actualizează, fapt ce poate fi observat în timp real în această fereastră a interfeței grafice:



Aplicația semnalizează o eroare prin apariția unei ferestre cu un mesaj specific în cazul în care utilizatorul introduce date invalide. Ferestrele ce afișează mesajele de eroare au următorul design:





Interfața grafică facilitează folosirea simulatorului de către orice tip de utilizator și oferă o interacțiune mai bună a utilizatorului modelul obiectual al soluției.

5. Rezultate

Testarea aplicației a fost realizată pe cele trei scenarii de test furnizate în specificația problemei, etapele simulării și rezultatele obținute în fișierul „Log-of-events.txt" pentru fiecare set de date de intrare impus urmând să fie atașat în secțiunea **ANEXĂ – Rezultatele celor trei cazuri de testare impuse**.

Aplicația a reacționat corect, având comportamentul așteptat în toate cazurile de testare furnizate, a respectat cadrul de cerințe funcționale impus prin specificația problemei, în fișierul „Log-of-events.txt" putând fi regăsite pentru fiecare pas al simulării: timpul curent, lista de clienți care nu au fost asociați unui server și lista serverelor împreună cu clienții asociați, iar la final fiind atașate rezultatele calculate reprezentând timpul mediu de așteptare, timpul mediu de procesare și ora de vârf, sub forma

```
------------------------------------------------------------------------------
Time: 22
Waiting clients: (10, 23, 7); (11, 23, 4); (19, 23, 2); (25, 24, 2);
Queue 1: (14, 17, 2); (1, 20, 3); (8, 20, 6);
Queue 2: (9, 15, 1); (24, 18, 7);
Queue 3: (3, 18, 5); (7, 22, 7);
Queue 4: (13, 17, 1); (23, 19, 8);
```

```
------------------------------------------------------------------------------
END OF SIMULATION
Average Waiting Time: 2.76
Average Service Time: 5.5
Peak Hour: 23
```

Testarea proiectului nu s-a rezumat doar la cele trei teste impuse în specificația problemei, aplicația având un răspuns pozitiv la multitudinea de cazuri de test furnizate, printre acestea numărându-se, de asemenea, situații de tip excepție care au fost interceptate și tratate corect. De asemenea, la fiecare pas de testare s-a verificat concordanța rezultatelor scrise în fișierul „Log-of-events.txt" cu cele reprezentate vizual în interfața grafică de care aplicația dispune pentru a asigura corectitudinea informațiilor vizualizate.

6. Concluzii

Concluzionând, se poate afirma că această temă vine să soluționeze problema minimizării timpului de așteptare în sistemele construite pe o politică de tip coadă, fiind implementată conform paradigmei programării orientate pe obiect și proiectându-se ca o mini-aplicație ce poate fi folosită de orice tip de utilizator, dispunând de un pattern arhitectural care permite separarea modelului obiectual, de interfața grafică cu care utilizatorul interacționează. Pentru dezvoltatorul proiectului, implementarea acestuia s-a dovedit utilă, având drept rezultat o mai bună însușire a manipulării și dezvoltării aplicațiilor Object Oriented, prin exersarea implementării practice a conceptelor specifice acestei paradigme de programare, dar și acumularea de noi cunoștințe referitoare la folosirea thread-urilor, a tipurilor de date și obiecte folosite în execuția concurentă sincronizată și a elementelor de grafică și vizualizare a rezultatelor în timp real în dezvoltarea unei aplicații susținute de o interfață grafică cu utilizatorul.

Printre posibilele dezvoltări ulterioare ale aplicației se numără afișarea rezultatelor obținute în urma simulării și în fereastra dedicată vizualizării rezultatelor din interfața grafică, dar și stabilirea și implementarea unui cod de culori pentru diferitele tipuri de clienți diferențiați din punctul de vedere al statusului: verde – în procesare, galben – în coadă, roșu – în așteptare).

7. Bibliografie

- ASSIGNMENT_2_SUPPORT_PRESENTATION.pdf (furnizat pe Microsoft Teams)
- Tema2.pdf (furnizat pe Microsoft Teams)
- Cursuri și lucrări de laborator de la disciplina Programare Orientată pe Obiecte, studiată în semestrul I al anului universitar 2020-2021
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html
- https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html
- https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html
- https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html
- https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html