DOCUMENTATIE

TEMA *2*

NUME STUDENT: Ureche Simona Elena
GRUPA: 30224

CUPRINS

1. Obiectivul temei

(i) Obiectivul principal

(ii) Dezvoltarea unei aplicații cu interfață grafică dedicată manipulării unui sistem eficient de management al cozilor și de minimizare a timpului de asteptare a clienților. Principalul obiectiv al unei cozi este să ofere un loc în care un client să aștepte înainte de a primi un serviciu.

Obiectiv secundar	*Descriere*	*Capitol*
Cerințe funcționale/nonfuncționale	Definesc comportamentul și capacitățile sistemului. Acestea pot include operațiuni specifice, funcționalități, servicii sau caracteristici pe care sistemul trebuie să le ofere pentru a satisface nevoile utilizatorilor și a atinge obiectivele stabilite. Scenariile oferă o perspectivă detaliată asupra modului în care utilizatorii și sistemul colaborează pentru a realiza sarcini.	2
Proiectarea orientată pe obiecte(POO)	Prezentarea arhitecturii generale a aplicației, evidențiind modul în care conceptele OOP au fost aplicate pentru a organiza și structura codul.	*3*
Împarțirea în pachete și clase	Modul în care clasele și alte resurse sunt grupate în pachete logice pentru a organiza și gestiona proiectul.	*3*
Concepte utilizate	Utilizate în cadrul aplicației pentru a rezolva diferite probleme sau sarcini.	*3*
Interfete grafice definite	Folosirea tehnicii Model Viev Controller pentru realizarea unui Graphical User Interface.	*3*
Descrierea claselor și metodelor	Semnăturile și descrierea claselor și metodelor relevante din cadrul aplicației de management al cozilor.	*4*
Implementarea interfeței utilizator	Modul în care interfața utilizator a fost proiectată și implementată pentru a facilita interacțiunea utilizatorului cu manipularea serverelor si a cozilor.	*4*

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

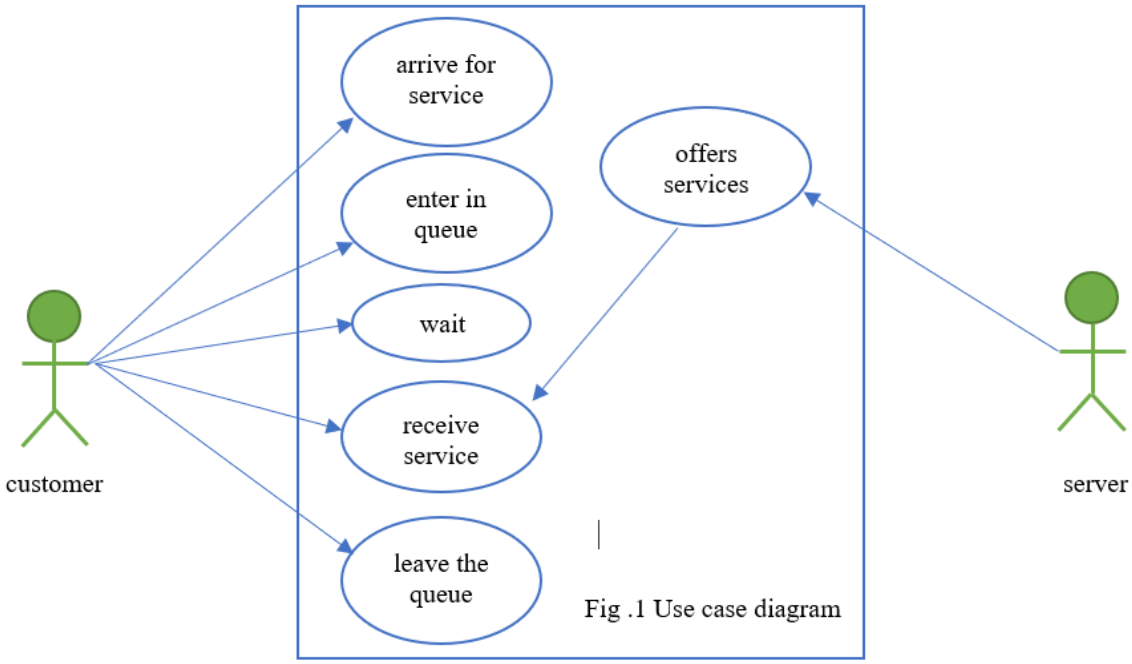


Fig .1 Use case diagram

Scenariu de Utilizare: Adăugare Polinoame

Actor Principal: Utilizatorul

Cerințe Funcționale:

- Aplicația de simulare ar trebui să permită utilizatorilor să configureze simularea –
Aplicația de simulare ar trebui să permită utilizatorilor să înceapă simularea
- Aplicația de simulare ar trebui să afișeze cozile în timp real evoluție.

Cerințe Non-Funcționale:

- Aplicația de simulare trebuie să fie intuitiva și ușor de utilizat de către utilizator.
- Aplicația de simulare trebuie să poată gestiona eficient un număr mare de clienți și cozi.
- Aplicația de simulare trebuie să fie robustă și să nu fie predispusă la căderi sau comportamente neașteptate.

Scenariul Principal de Succes:

- Utilizatorul introduce valorile pentru urmatorii parametrii:
- Numarul de clienti, numarul de cozi, intervalul de simulare, timpul minim si maxim de sosire si timpul minim si cel maxim de servire.
- Utilizatorul apasa butonul de "Valideaza Datele"
- Daca datele sunt valide, aplicatia afiseaza un mesaj, informand ca incepe simularea.

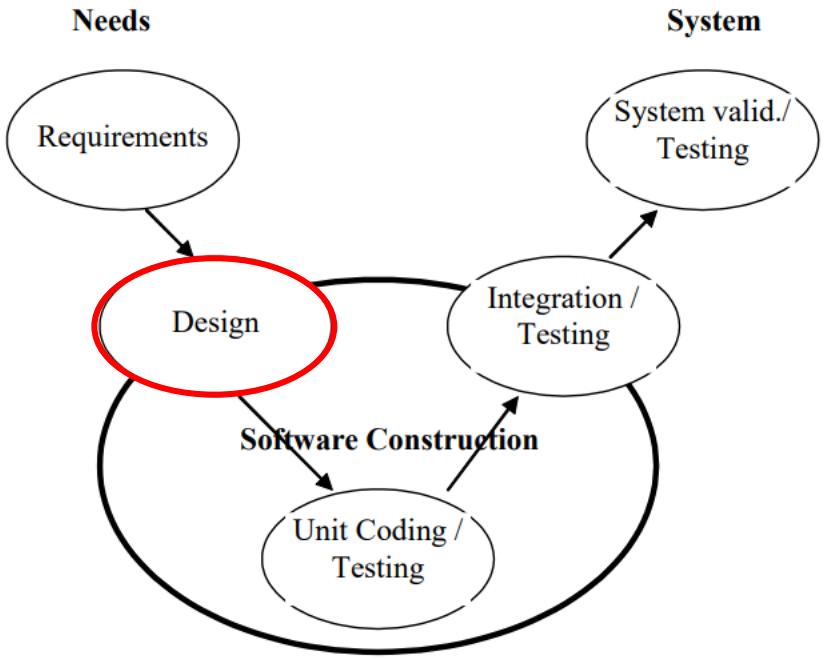
Secvență Alternativă: Date Invalide

- Utilizatorul introduce valori invalide pentru parametrii de configurare ai aplicatiei.
- Aplicatia afisează mesaj de eroare și îi permite utilizatorului să introducă alte date.
- Scenariul revine la pasul 1.

3. Proiectare

(i) Decizii de proiectare

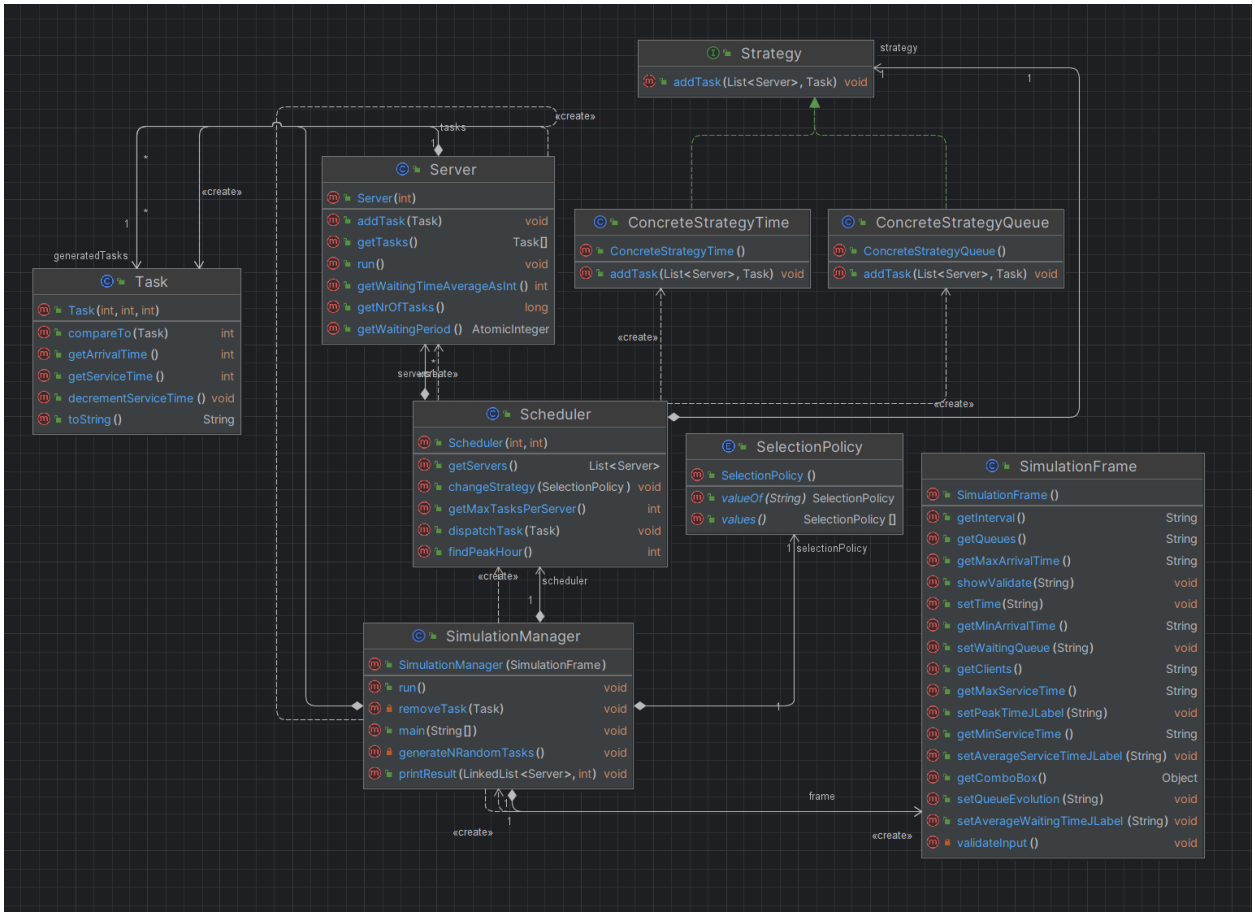
Aplicația dedicată manipulării unui sistem eficient de management al cozilor și de minimizare a timpului de asteptare a clienților este structurată folosind conceptele de programare orientată pe obiecte (OOP) pentru a organiza și structura codul într-un mod coerent și eficient. Vom separa atât componentele logice cât si modelul de date de interfață de control, ceea ce face mai ușoară gestionarea și întreținerea codului.



(ii) Împărțirea în pachete

- Pachetul GUI: Aici se va afla clasa “GUI”, care gestionează interfața grafică a aplicației.
- Pachetul Logic: Acest pachet este alcătuit din clasa “Simulation Manager”, care conține bucla de simulare dar și generatorul random de task-uri, clasa “Scheduler”, care trimite task-uri la server si stabilește strategia si clasele ConcreteStrategyQueue si ConcreteStrategyTime care implementeaza interfata Strategy, personalizând metoda în funcție de caz.

- Pachetul Model: Aici vor fi definite clasele “Task”, care reprezinta clientii și “Server`”, care modelează task-urile, reprezentand cozile.



(iii) Împărțirea în clase

- Clasa GUI: Această clasă gestionează aspectele legate de interfața grafică a aplicației. Ea definește elementele vizuale și interacțiunile utilizatorului cu acestea, precum și logica necesară pentru a interpreta și a răspunde la acțiunile utilizatorului.

- Clasa SimulationManager: Gestionează simularea întregului sistem, inclusiv inițializarea parametrilor, generarea de sarcini și gestionarea interacțiunii cu interfața utilizatorului și cu scheduler-ul.

- Clasa Scheduler: Se ocupă de distribuirea sarcinilor către servere folosind o anumită strategie, precum și de monitorizarea serverelor și a sarcinilor. Calculează, de asemenea, timpul mediu de așteptare pentru toate sarcinile.

- Clasa ConcreteStrategyQueue: Este o implementare a interfeței Strategy și definește o strategie de alegere a serverului în funcție de lungimea cozii de așteptare. Selectează serverul cu cea mai scurtă coadă pentru a procesa sarcina.

- Clasa ConcreteStrategyTime: Implementează, de asemenea, interfața Strategy, dar în acest caz alege serverul în funcție de timpul estimat pentru procesarea sarcinii. Selectează serverul cu cel mai scurt timp de procesare disponibil pentru sarcina dată.

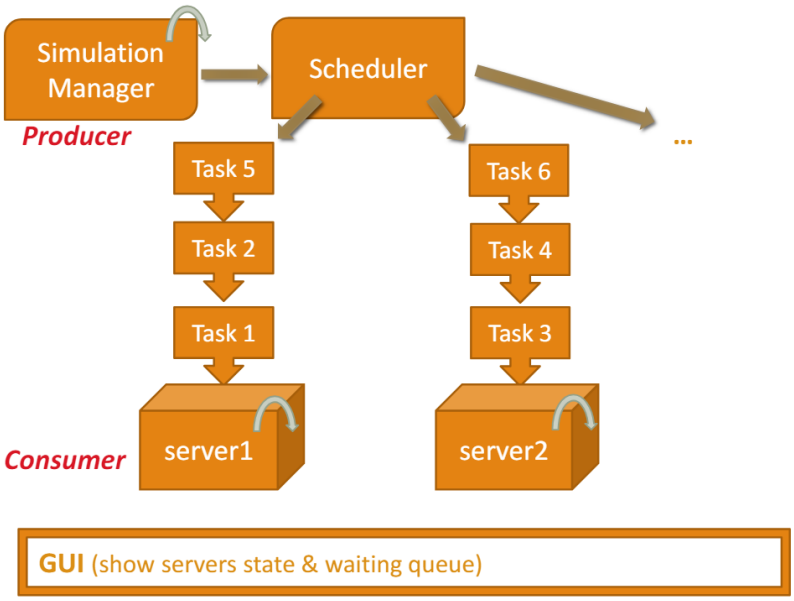
- Clasa Server: Reprezintă un server în cadrul sistemului și este responsabil pentru procesarea task-urilor. Monitorizează sarcinile primite și timpul de așteptare mediu pentru acestea.

- Clasa Task: Definește un task care trebuie procesată de un server. Include informații precum id, timpul de sosire și timpul necesar pentru a fi procesată.

(iv) Concepte utilizate

- Thread-uri: Utilizate pentru gestionarea concurenței și executarea operațiilor în paralel. Folosite în special pentru gestionarea serverelor și task-urilor asociate cu acestea în aplicație.

- BlockingQueue este o structură de date în Java utilizată pentru comunicarea și sincronizarea între thread-uri într-un mediu concurent. Aceasta oferă o colecție care se comportă ca o coadă, permițând adăugarea și eliminarea elementelor într-un mod sigur pentru thread-uri. Atunci când coada este plină, operațiile de adăugare sunt blocate până când devine disponibil spațiu, iar când coada este goală, operațiile de eliminare sunt blocate până când devin disponibile elemente. Această abordare este utilă pentru implementarea modelelor de producător-consumator și pentru gestionarea sarcinilor între thread-uri, precum în simularea serverelor. Utilizarea BlockingQueue asigură o sincronizare corectă și evită conflictele între thread-uri într-un mediu concurent.

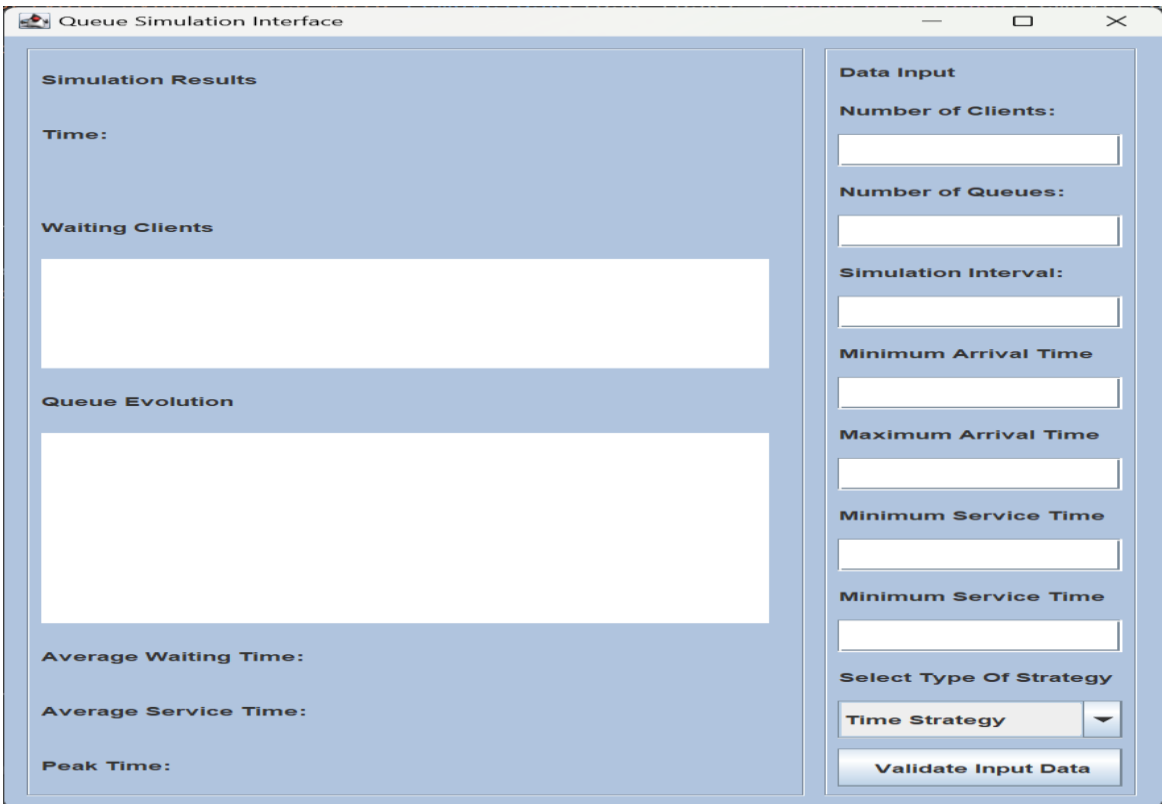


- Variabile atomice: Utilizate pentru a asigura operațiuni atomice pe date partajate între thread-uri. Evită condițiile de cursă și asigură consistența datelor în medii concurente.

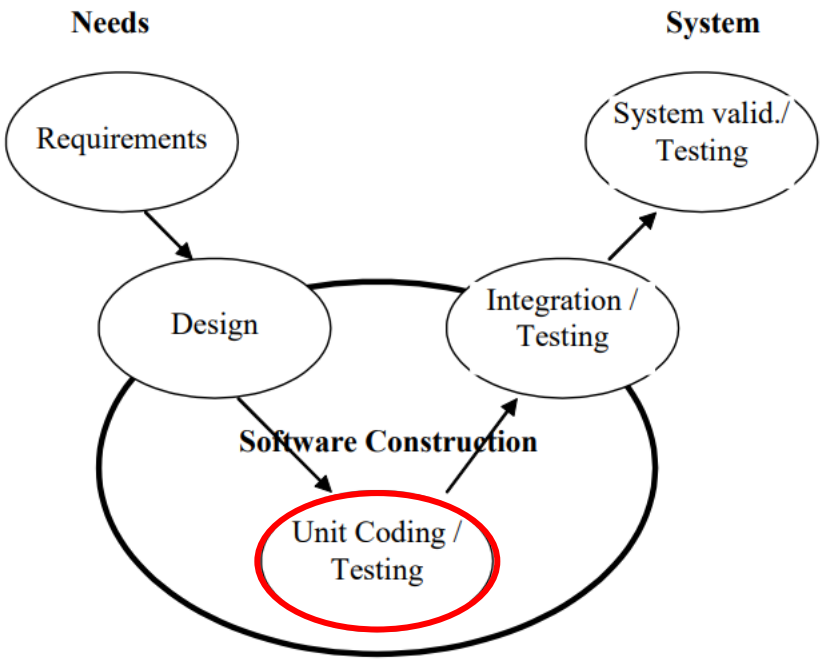
- Instrucțiunea synchronized: Folosită pentru sincronizarea accesului la resurse partajate între thread-uri. Previene accesul simultan din mai multe thread-uri la resurse comune, reducând riscul de conflicte și condiții de cursă.

(v) Interfața grafică

Interfața grafică a aplicației este împărțită în două coloane. În prima coloană, utilizatorul poate vizualiza rezultatele simulării, cum ar fi timpul, clienții așteptând și evoluția cozilor. În plus, sunt afișate și informații precum timpul mediu de așteptare și timpul mediu de servire. În cea de-a doua coloană, utilizatorul poate introduce datele necesare pentru simulare, cum ar fi numărul de clienți, numărul de cozi, intervalul de simulare și timpii minim și maxim de sosire și de servire. Utilizatorul poate valida datele introduse și porni simularea apăsând butonul corespunzător.



4. Implementare



(i) Descrierea claselor și metodelor

(i.1) *Clasa GUI*

Clasa "GUI" este responsabilă pentru gestionarea interfeței grafice a aplicației de simulare a cozilor. Aceasta conține următoarele metode:

-. getClients(), getQueues(), getInterval(), getMinServiceTime(), getMaxServiceTime(), getMinArrivalTime(), getMaxArrivalTime(), getComboBox(): Aceste metode returnează diferite informații introduse de utilizator în interfața grafică pentru simulare. Ele sunt utilizate pentru a obține datele necesare pentru inițierea simulării și selectarea strategiei.

-. showValidate(String errMessage): Această metodă afișează un mesaj de validare în cazul în care datele introduse de utilizator nu sunt valide.
setTime(String time), setWaitingQueue(String queue), setQueueEvolution(String queue), setAverageWaitingTimeJLabel(String average),

setAverageServiceTimeJLabel(String service), setPeakTimeJLabel(String peak): Aceste metode setează diferite informații și rezultate ale simulării în interfața grafică. De exemplu, timpul, listele de așteptare ale clienților, evoluția cozilor și timpul maxim de vârf sunt afișate în interfață pentru a oferi utilizatorului informații despre starea simulării.

(i.2) *Clasa Task*
Clasa Task este responsabilă pentru reprezentarea și gestionarea sarcinilor în cadrul simulării cozilor. Aceasta conține următoarele metode:

- Metoda getArrivalTime(): Returnează timpul de sosire al sarcinii.

- Metoda getServiceTime(): Returnează timpul de servire al sarcinii.

- Metoda decrementServiceTime(): Scade timpul de servire al sarcinii cu o unitate în fiecare apel, simulând progresul în servirea sarcinii..

- Metoda compareTo(Task otherTask): Compară două sarcini pe baza timpului lor de sosire. Metoda este utilizată pentru sortarea sarcinilor în ordinea sosirii lor în sistem..

- Metoda toString(): Returnează o reprezentare sub formă de șir de caractere a sarcinii, conținând ID-ul, timpul de sosire și timpul de servire. Această reprezentare este utilă pentru afișarea detaliilor sarcinilor în diverse contexte, cum ar fi afișarea listei de așteptare sau a evoluției cozilor în interfața grafică.

(i.3) *Clasa Server*
Clasa Server este responsabilă pentru gestionarea sarcinilor și a timpului de așteptare asociat în cadrul unui server din sistemul de simulare a cozilor. Aceasta conține următoarele metode:

- Constructorul Server(int capacity): Inițializează un nou server cu o capacitate dată pentru coada de sarcini și inițializează timpul de așteptare și timpul de așteptare mediu..

- Metoda addTask(Task newTask): Adaugă o nouă sarcină în coada de sarcini a serverului și actualizează timpul de așteptare și timpul de așteptare mediu..

- Metoda run(): Implementează logica de procesare a sarcinilor într-un fir de execuție separat. Serverul rulează în mod continuu, procesând sarcinile din coadă, actualizând timpul de așteptare și timpul de așteptare mediu în consecință.

- Metoda getTasks(): Returnează un tablou de sarcini care se află în prezent în coada de sarcini a serverului.
- Metoda getNrOfTasks(): Returnează numărul de sarcini din coada de sarcini a serverului..
- Metoda getWaitingTimeAverageAsInt(): Returnează timpul mediu de așteptare al sarcinilor sub formă de întreg. Dacă nu există timp de așteptare mediu disponibil, se returnează valoarea 0

(i.4.1) *Clasa ConcreteStrategyQueue:*

Clasa ConcreteStrategyQueue implementează interfața Strategy și definește o strategie specifică pentru adăugarea sarcinilor la servere, bazată pe numărul minim de sarcini din cozi. Aceasta conține următoarele metode:

- Metoda addTask(List<Server> servers, Task t): Metodă sincronizată care adaugă o sarcină la serverul cu cel mai mic număr de sarcini în coadă din lista dată de servere. Parcurge lista de servere și identifică serverul cu cel mai mic număr de sarcini, adăugând apoi sarcina la acesta.

(i.4.2) *Clasa ConcreteStrategyTime:*

Clasa ConcreteStrategyTime implementează interfața Strategy și definește o strategie specifică pentru adăugarea sarcinilor la servere, bazată pe timpul minim de așteptare. Aceasta conține următoarele metode:

- Metoda addTask(List<Server> servers, Task t): Metodă sincronizată care adaugă o sarcină la serverul cu cel mai mic timp de așteptare din lista dată de servere. Parcurge lista de servere și identifică serverul cu cel mai mic timp de așteptare, verificând apoi dacă acesta are loc disponibil pentru o nouă sarcină. În caz afirmativ, adaugă sarcina la acesta. În caz contrar, căută un alt server disponibil cu timp de așteptare mai mare decât minimul identificat și adaugă sarcina la acesta, dacă este găsit.

(i.5) *Clasa Sheduler*

Clasa Scheduler gestionează serverele și strategiile de distribuire a sarcinilor. Aceasta conține următoarele metode:

- Constructorul Scheduler(int maxNoServers, int maxTasksPerServer): Inițializează Scheduler cu numărul maxim de servere și numărul maxim de sarcini per server. Creează și pornește thread-urile pentru fiecare server din lista de servere.

- Metoda changeStrategy(SelectionPolicy policy): Schimbă strategia de distribuire a sarcinilor în funcție de politica specificată. Dacă politica este SHORTEST_TIME, se folosește ConcreteStrategyTime, iar dacă politica este SHORTEST_QUEUE, se folosește ConcreteStrategyQueue.

- Metoda dispatchTask(Task task): Distribuie o sarcină către serverele din lista de servere folosind strategia curentă.

- Metoda getServers(): Returnează lista de servere.

- Metoda getMaxTasksPerServer(): Returnează numărul maxim de sarcini per server.

- Metoda findPeakHour(): Calculează și returnează numărul total de clienți (sarcini) din toate serverele, indicând astfel ora de vârf a simulării.

(i.6) *Clasa SimulationManager*

Clasa SimulationManager coordonează întreaga simulare a sistemului de cozi și servere. Aceasta conține următoarele metode:

- Constructorul SimulationManager(SimulationFrame frame): Inițializează simularea cu valorile primite de la interfața grafică. Creează obiectul Scheduler cu numărul de servere și strategia specificate. Generează sarcini aleatorii și inițializează fișierul de scriere.
- Metoda generateNRandomTasks(): Generează un număr specificat de sarcini cu timpuri de sosire și de procesare aleatorii și le adaugă în lista de sarcini generate.

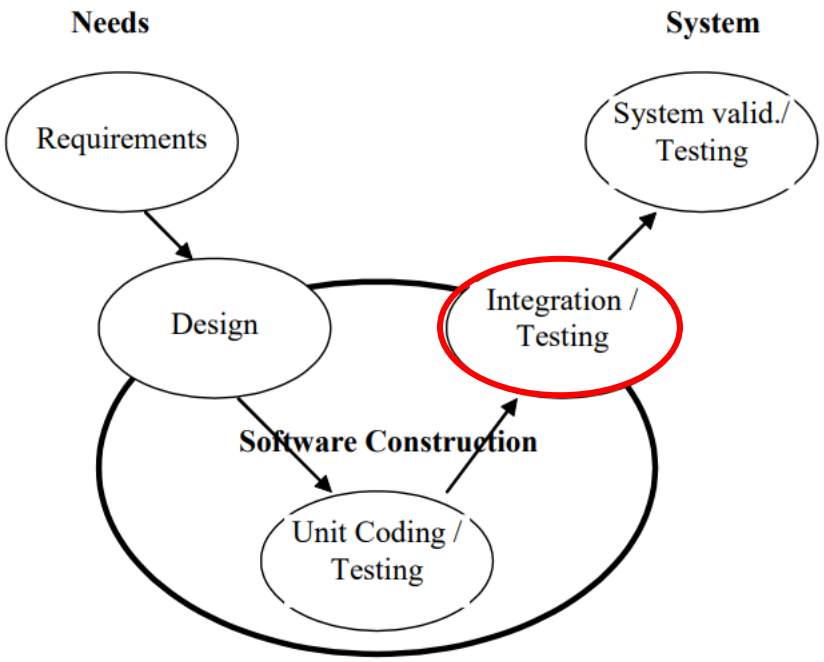
- Metoda run(): Rulează simularea, gestionând distribuirea sarcinilor către servere, actualizând interfața grafică și scriind rezultatele în fișier.

- Metoda graphicalInterfaceConnection (List<Server> serverList, int time): Actualizează starea interfeței grafice cu informații despre sarcinile în așteptare și starea fiecărui server.

- Metoda removeTask(Task task): Șterge o sarcină din lista de sarcini generate.

- Metoda main(String[] args): Punctul de intrare în program, inițializează interfața grafică și începe simularea.

5. .Rezultate



Test 1	Test 2	Test 3
N = 4 Q = 2 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [2, 30] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [2, 4]	N = 50 Q = 5 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [2, 40] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [1, 7]	N = 1000 Q = 20 $t_{simulation}^{MAX}$ = 200 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [10, 100] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [3, 9]

Test1:

```
Time: 11
Waiting clients: (1, 12, 4); (2, 12, 3); (4, 16, 3); (3, 17, 3);
Queue 1: closed
Queue 2: closed

Time: 12
Waiting clients: (4, 16, 3); (3, 17, 3);
Queue 1: (1, 12, 4);
Queue 2: (2, 12, 3);

Time: 13
Waiting clients: (4, 16, 3); (3, 17, 3);
Queue 1: (1, 12, 3);
Queue 2: (2, 12, 2);

Time: 14
Waiting clients: (4, 16, 3); (3, 17, 3);
Queue 1: (1, 12, 2);
Queue 2: (2, 12, 1);

Time: 15
Waiting clients: (4, 16, 3); (3, 17, 3);
Queue 1: (1, 12, 1);
Queue 2: closed

Time: 16
Waiting clients: (3, 17, 3);
Queue 1: (4, 16, 3);
Queue 2: closed

Time: 17
Waiting clients:
Queue 1: (4, 16, 2);
Queue 2: (3, 17, 3);

Time: 18
Waiting clients:
Queue 1: (4, 16, 1);
Queue 2: (3, 17, 2);
```

```
Time: 15
Waiting clients: (4, 16, 3); (3, 17, 3);
Queue 1: (1, 12, 1);
Queue 2: closed

Time: 16
Waiting clients: (3, 17, 3);
Queue 1: (4, 16, 3);
Queue 2: closed

Time: 17
Waiting clients:
Queue 1: (4, 16, 2);
Queue 2: (3, 17, 3);

Time: 18
Waiting clients:
Queue 1: (4, 16, 1);
Queue 2: (3, 17, 2);

Time: 19
Waiting clients:
Queue 1: closed
Queue 2: (3, 17, 1);

Time: 20
Waiting clients:
Queue 1: closed
Queue 2: closed


Average Waiting Time: 3.75
Average Service Time: 3.25
Peak Time: 2
```

6. Concluzii

În concluzie, acest proiect a reprezentat o oportunitate de a învăța și de a experimenta noi concepte și tehnici în programarea concurentă în Java. Utilizarea clasei ArrayBlockingQueue pentru gestionarea cozilor a fost esențială pentru implementarea eficientă a simulării serverelor și a distribuirii sarcinilor între acestea. De asemenea, lucrul cu thread-uri a fost un aspect central al proiectului, permițând gestionarea concurenței și execuția operațiilor în paralel. În cadrul acestui proiect, am observat importanța utilizării variabilelor atomice și a instrucțiunii synchronized pentru a asigura consistența și sincronizarea corectă a datelor în medii concurente. Cu toate acestea, în timpul implementării, s-au întâmpinat unele provocări legate de gestionarea concurenței și de sincronizarea corectă a operațiilor între thread-uri.

Pentru a îmbunătăți proiectul, putem lua în considerare aspect precum optimizarea performanței, gestionarea mai eficienta a excepțiilor sau îmbunătățirea interfeței utilizator.

7. Bibliografie

- Name-Conventional: https://google.github.io/styleguide/javaguide.html
- UML: https://www.jetbrains.com/help/idea/class-diagram.html
- Thread Pool: https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html
    - https://www.youtube.com/watch?v=ZcKt5FYd3bU&t=490s
- Maven: https://www.jetbrains.com/help/idea/maven-support.html
-