DOCUMENTAȚIE

TEMA 2 – *Managementul cozilor*

Ciochină Cătălina- Andreea

Facultatea de Calculatoare și Tehnologia Informației

Grupa 30228

Prof. coordonator,

Chifu Viorica Rozina

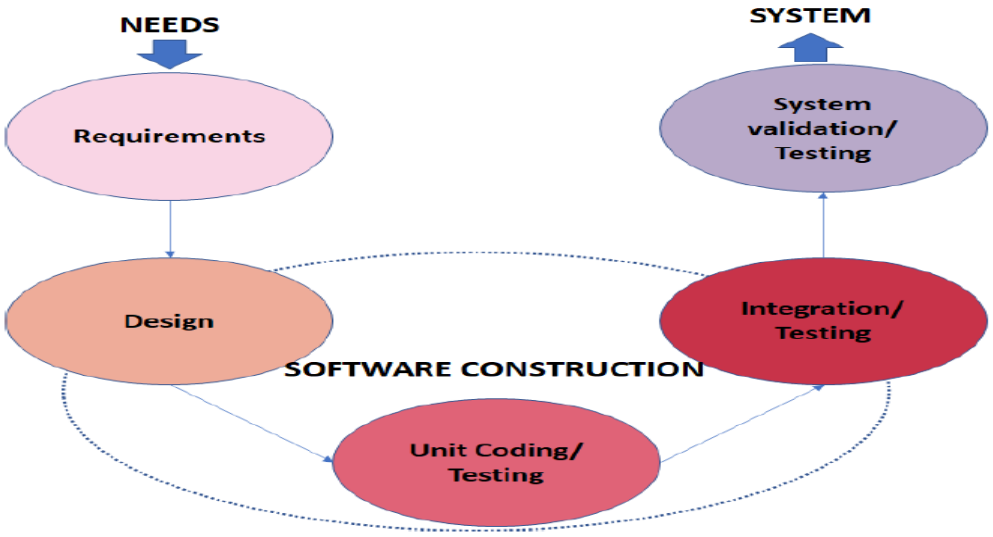
NUME STUDENT: Ciochină Cătălina-Andreea
GRUPA: 30228

CUPRINS

1. Obiectivul temei

Obiectivul principal al unei aplicații de management a cozilor este de a minimiza timpii de așteptare pentru clienți și de a îmbunătăți utilizarea eficientă a resurselor prin implementarea unor mecanisme eficiente de alocare a cozilor. Acest lucru va conduce la o experiență mai bună a clienților și la o eficiență crescută a serviciilor.

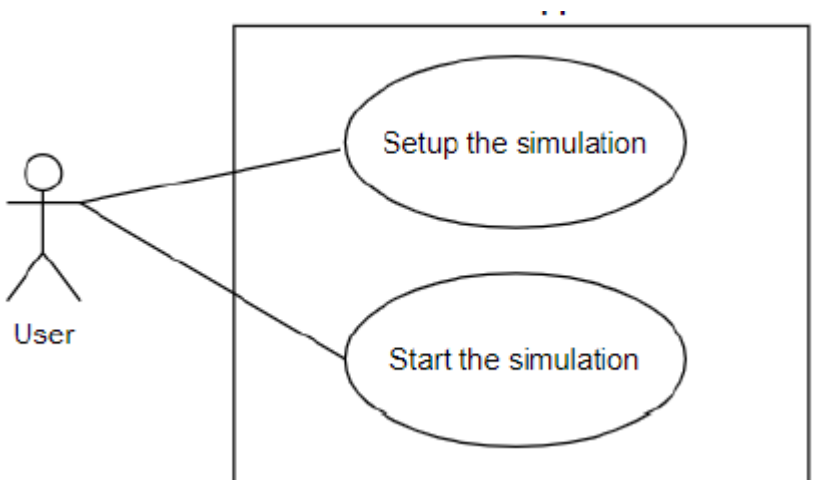
Obiective secundare	***Detalii***	***Secțiunea***
Colectarea și analiza cerințelor problemei	***Colectarea*** și ***analiza*** cerințelor problemei reprezintă procesul de înțelegere a nevoilor și cerințelor utilizatorului și transformarea acestora în specificații clare pentru dezvoltarea software-ului.	2
Proiectarea soluției	După ce cerințele au fost analizate, trebuie să se ***proiecteze*** o soluție software care îndeplinește cerințele utilizatorului.	3
Implementarea soluției	***Scriere de cod*** pentru dezvoltarea unui sistem software care să corespundă proiectării și analizei realizate anterior.	4
Testarea	Se face cu scopul de a întregii produsul software și de a ne asigura că implementarea este corectă pentru ***toate cazurile posibile***, pe care user-ul le-ar putea introduce/genera.	5



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

2.1. Cerințe functionale

- Aplicația de simulare ar trebui să permită utilizatorilor să definească parametrii de simulare, cum ar fi numărul de clienți, numărul de cozi, intervalele de sosire a clienților, timpul de servire a clienților, timpul de simulare și strategia aleasă, ordonare la cozi în funcție de cozile cu cel mai puțin clienți sau cel mai scurt timp de procesare.
- Aplicația de simulare ar trebui să permită începerea simulării, după validarea datelor introduse.
- Aplicația trebuie să ofere o vedere în timp real a tuturor cozilor simulate și să actualizeze aceste informații pe măsură ce clienții sunt adăugați în cozi sau serviciul este finalizat(la fiecare timp petrecut de către client în coadă).
- Aplicația de simulare ar trebui să permită utilizatorilor să vizualizeze datele statistice generate de simulare, cum ar fi timpul mediu de așteptare, timpul mediu de servire și timpul de varf al simulării.



2.2. Cerințe non-funcționale

1. *Performanța*: aplicația trebuie să furnizeze rezultate într-un timp rezonabil.
2. *Ușurința în utilizare*: interfața grafică trebuie să fie ușor de folosit pentru utilizatorii finali.
3. *Extensibilitate*: aplicația de simulare trebuie să permită adăugarea de noi funcționalități și opțiuni pentru a satisface nevoile viitoare ale utilizatorilor.
4. *Reutilizabilitate*: codul trebuie să fie modular și ușor de reutilizat pentru a putea fi folosit în alte proiecte viitoare.
5. *Fiabilitate*: aplicația de simulare trebuie să funcționeze fără erori și să ofere rezultate precise.
6. *Portabilitate*: sistemul trebuie să poată fi instalat și utilizat pe mai multe platforme hardware și software.

2.3. Scenarii de utilizare/ Use cases

Use case: Configurare simulare
Primary Actor: User-ul

Main Success Scenario:

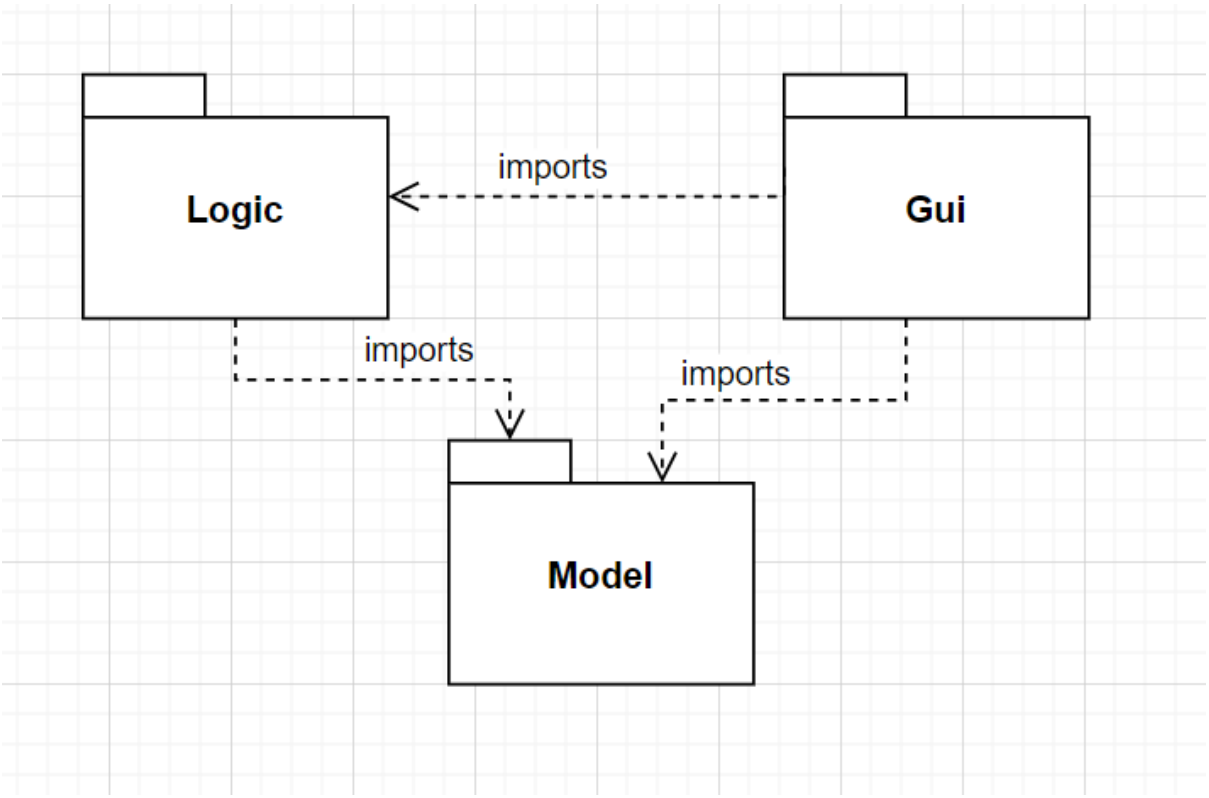
1. Utilizatorul introduce valorile pentru: numărul de clienți, numărul de cozi, intervalul de simulare, timpul minim și maxim de sosire și timpul minim și maxim de servire.
2. Utilizatorul alege strategia de aranjare la cozi, în funcție de timpul cel mai scurt sau în funcție de coada cea mai scurtă.
3. Utilizatorul apasă butonul de validare a datelor de intrare, care pornește și simularea.
4. După ce simularea afișeaza rezultatele(se încheie) se revine la pasul 1, utilizatorul putând introduce din nou alte date și să înceapă altă simulare exact ca în pașii anteriori.

Alternative Sequence:

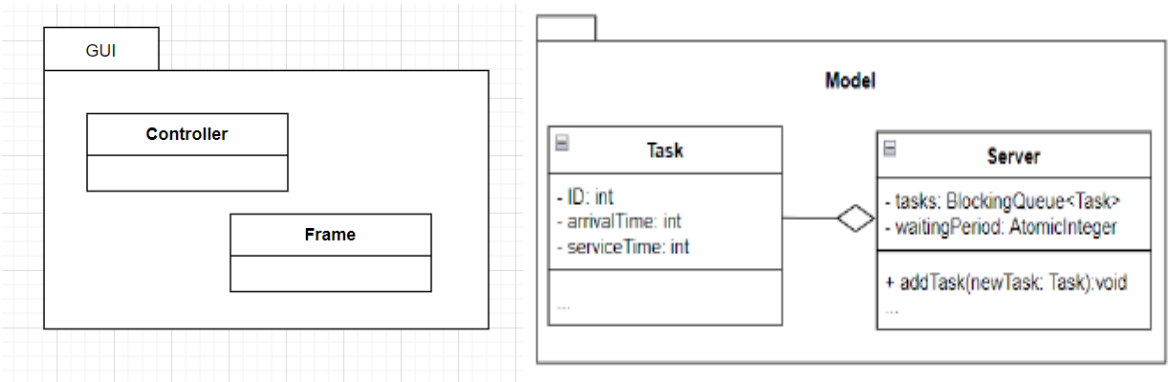
1. Utilizatorul introduce valori invalide sau uită să introducă/selecteze parametrii de configurare ai aplicației.
2. Aplicația afișează un mesaj de eroare și solicită utilizatorului să introducă valori valide.
3. Scenariul revine la pasul 1.

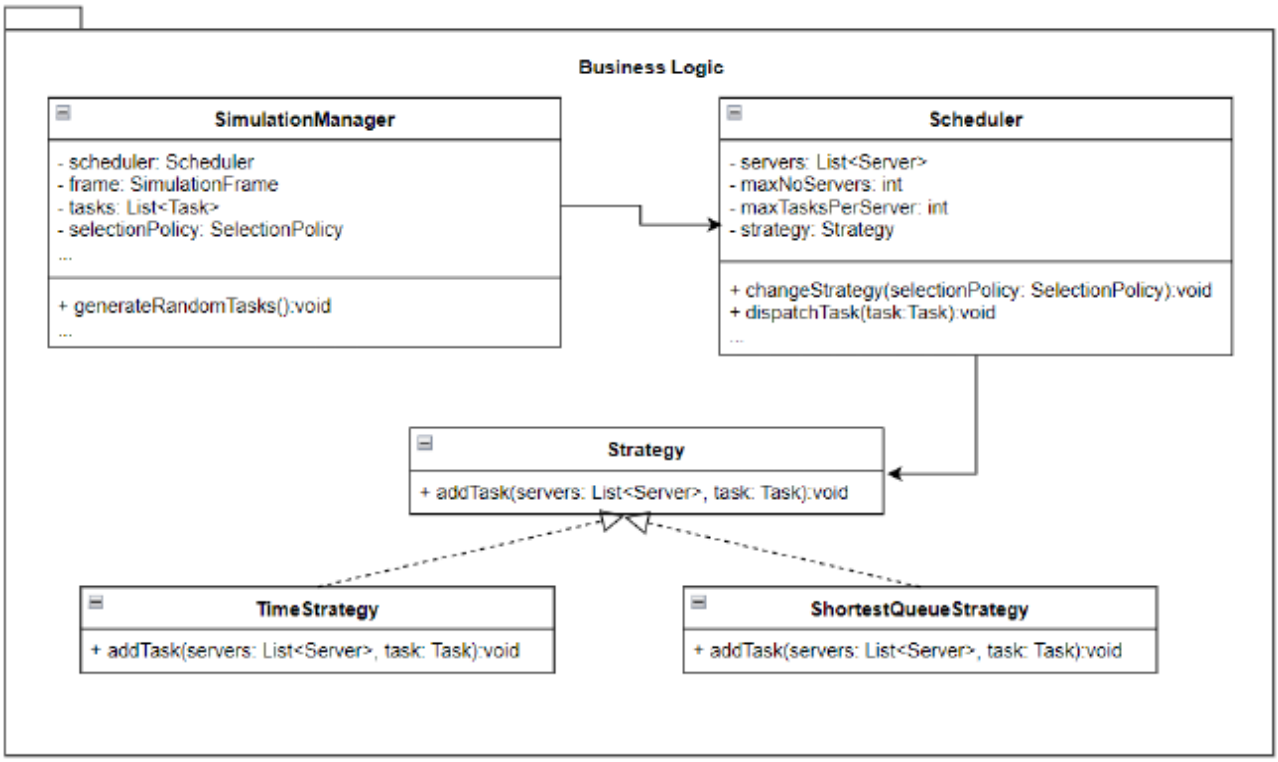
3. Proiectare

Am organizat codul în pachete asemenea organizării Model-View-Controller, denumirea în cadrul proiectului meu este Model(cu clasa Server și Task)-BussinessLogic(cu clasa ConcreteStrategyQueue, ConcreteStrategyTime, Scheduler, SimulationManager) si GUI(cu clasa Frame și Controller). Diagrama de pachete



Împărțirea în pachete:



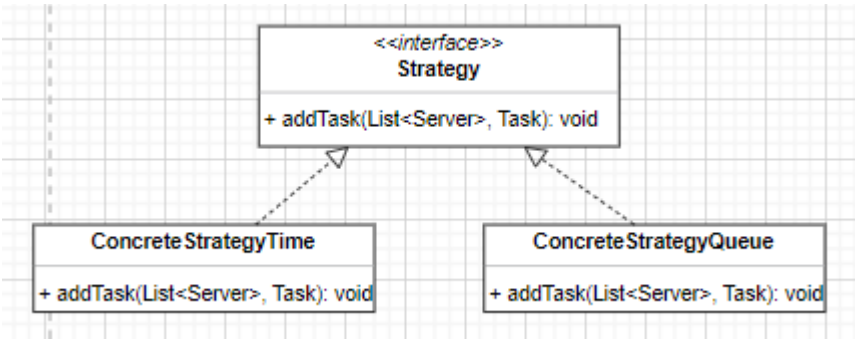


Clasele pe care le-am creat/folosit sunt descrise în următoarea secțiune(4. Implementare)

1. Clasa ConcreteStrategyQueue - bussinessLogic
2. Clasa ConcreteStrategyTime – bussinessLogic
3. Clasa Scheduler – bussinessLogic
4. Clasa SimulationManager – bussinessLogic
5. Clasa Controller - gui
6. Clasa Frame - gui
7. Clasa Server - model
8. Clasa Task - model
9. Enum SelectionPolicy – bussinessLogic
10. Interfața Strategy– bussinessLogic
11. Clasa Thread
12. Interfața Runnable

Diagrama UML de clase corespunzătoare implementării pe care am făcut-o este următoarea:

Clasele ConcreteStrategyTime si ConcreteStrategyQueue implementează interfața Strategy.

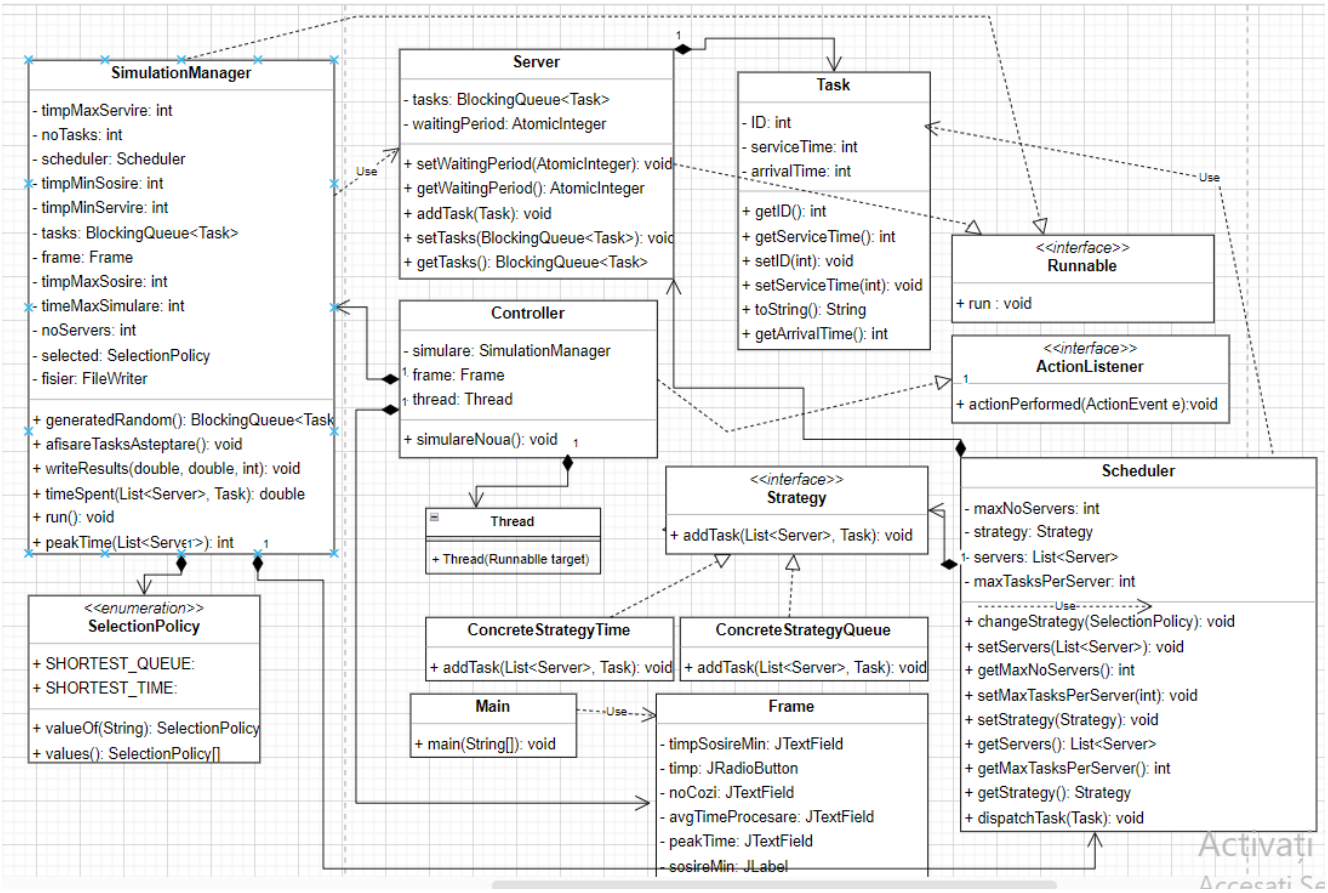


Un Controller are un SimulationManager, un Frame, un Thread și implementează interfața ActionListener.

Un Server are unul sau mai multe Task-uri și implementează interfața ActionListener.

Un Scheduler are o strategie și unul sau mai multe Servere, folosește și clasa Task.

Clasa SimulationManager are un SelectionPolicy, un Scheduler, una sau mai multe Task-uri.



4. Implementare

❖ *Clasa TASK*

Este un dintre clasele fără de care nu ar putea fi realizată aplicația, pentru că reprezintă tocmai actantul simulării, clientul. Are trei variabile instanță, un ID, prin care este identificat obiectul, un arrivalTime, care este de fapt timpul la care clientul trebuie să fie pus în coadă și un serviceTime, care reprezintă timpul, pe care trebuie să îl stea la coadă clientul. Mai sunt settere, gettere și metoda suprascrisă toString.

❖ *Clasa Server*

Este o altă clasă importantă a aplicației, deoarece reprezintă un alt actant al simulării și anume coada. Fiecare obiect creat de tipul acestei clase are o coadă de task-uri, am folosit BlockingQueue, care este o colecție concurentă care poate fi utilizată într-un mediu cu mai multe thread-uri, fără a afecta integritatea datelor sau a genera erori, și un timp de așteptare pe coadă. Această clasă implementează interfața Runnable. Metaoda addTask adaugă la coada de Task-uri, task-ul trimis ca parametru și incrementează timpul de așteptare al cozii. Clasa este obligată să implementeze metoda run(), care rulează în buclă infinită și verifică dacă sunt sarcini în așteptare, în cazul in care sunt, se scoate primul task din coadă și se procesează, cât timp timpul de procesare este mai mare decât zero, la fiecare iterație se pune Thread-ul la somn timp de 1 secundă și este decrementat timpul de procesare al task-ului. După ce timpul de service este epuizat, task-ul este eliminat din coadă și are loc trecerea la următorul task din coadă. Clasa mai are gettere și settere.

```
@Override
public void run() {
    while (true) {
        try {
            if (tasks.size() > 0) {
                Task nextTask = tasks.peek();
                int serviceTime = nextTask.getServiceTime();
                while (serviceTime > 0) {
                    Thread.sleep( millis: 1000);
                    waitingPeriod.getAndDecrement();
                    nextTask.setServiceTime(serviceTime - 1);
                    serviceTime = nextTask.getServiceTime();
                }
                tasks.remove(nextTask);
            }

        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
```

❖ *Clasa Scheduler*

Are o listă de cozi/servere, un număr maxim de servere și un număr maxim de task-uri pe server și strategia după care se adaugă în cozi. În constructor creez cele N cozi și pentru fiecare coadă câte un obiect thread trecând obiectul server ca parametru, și se pornește firul de execuție apelând metoda start(). La final, obiectul server este adăugat în lista de servere.

```
for (int i = 0; i < maxNoServers; i++) {
    Server server = new Server(maxTasksPerServer);
    Thread thread = new Thread(server);
    thread.start();
    servers.add(server);
}
```

Metoda changeStrategy este folosită în constructor, este setter-ul strategiei, pe baza unui obiect primit ca parametru de tipul SelectionPolicy.

```
public void changeStrategy(SelectionPolicy sel) {
    switch (sel) {
        case SHORTEST_QUEUE:
            this.strategy = new ConcreteStrategyQueue();
            break;
        case SHORTEST_TIME:
            this.strategy = new ConcreteStrategyTime();
            break;
        default:
            throw new IllegalArgumentException("Nu exista alta strategie");
    }
}
```

Metoda dispatchStrategy cheamă metoda de adăugare a unui task la un server din clasele care implementează interfața Strategy.

```
public void dispatchTask(Task task) { this.strategy.addTask(servers, task); }
```

❖ *Clasa SimulationManager*

Este cea care se ocupă de simulare în sine, adunând majoritatea claselor implementate, are un Scheduler, o lista(BlockingQueue) de task -uri si Frame-ul, pentru afișarea în timp real a datelor. Clienții se generează random pe baza datelor primite de la utilizator, timp minim și maxim de sosire, timp minim și maxim de procesare.

```
public BlockingQueue<Task> generatedRandom() {
    BlockingQueue<Task> tasks = new ArrayBlockingQueue<>(noTasks);
    for (int i = 0; i < noTasks; i++) {
        Random rand = new Random();
        int sosire = rand.nextInt( bound: timpMaxSosire - timpMinSosire + 1) + timpMinSosire;
        int servire = rand.nextInt( bound: timpMaxServire - timpMinServire + 1) + timpMinServire;
        Task task = new Task( ID: i + 1, sosire, servire);
        tasks.add(task);
    }
    List<Task> taskList = new ArrayList<>(tasks);
    Collections.sort(taskList, new Comparator<Task>() {
        @Override
        public int compare(Task task1, Task task2) {
            return Integer.compare(task1.getArrivalTime(), task2.getArrivalTime());
        }
    });
    tasks = new ArrayBlockingQueue<>(noTasks,  fair: true, taskList);
    return tasks;
}
```

Metoda de run iterează prin lista de task-uri generate random cât timpul nu este mai mare decât timpul de simulare. La fiecare pas se ia un task și se trimite la scheduler și apoi se șterge.

```
public void run() {
    AtomicInteger timp = new AtomicInteger( initialValue: 0);
    int peakTime = 0, maxTasks = 0;
    double averageTimeSpent = 0.0, averageTimeService = 0.0;
    int auxTime = Integer.parseInt(frame.getTimpSimulare().getText());
    Iterator<Task> iter;
    frame.getSTART().setEnabled(false);
    while (timp.get() <= timeMaxSimulare) {
        iter = tasks.iterator();
        while (iter.hasNext()) {
            Task nextTask = iter.next();
            if (nextTask.getArrivalTime() == timp.get()) {
                scheduler.dispatchTask(nextTask);
                averageTimeSpent += timeSpent(scheduler.getServers(), nextTask);
                averageTimeService += (timeSpent(scheduler.getServers(), nextTask) + nextTask.getServiceTime());
                this.tasks.remove(nextTask);
            }
        }
```

Am făcut timpul de așteptare mediu ca o medie între toți timpii de aștepare pe fiecare task. Un task așteaptă să intre în coadă suma timpilor de procesare ai task-urilor anterioare și la timpul de procesare mediu am luat suma timpilor de procesare a tuturor timpilor plus a task-ului current.

```java
public double timeSpent(List<Server> servers, Task taskSearched) {
    double finalAverageTime = 0.0;
    for (Server server : servers) {
        for (Task task : server.getTasks()) {
            if (task.getServiceTime() > 0 && taskSearched.equals(task)) {
                BlockingQueue<Task> queue = server.getTasks();
                List<Task> subList = new ArrayList<>(queue);

                int index = subList.indexOf(task);
                if (index >= 1) {
                    List<Task> previousTasks = subList.subList(0, index);

                    AtomicInteger totalTimeSpent = new AtomicInteger( initialValue: 0);
                    for (Task prevTask : previousTasks) {
                        totalTimeSpent.addAndGet(prevTask.getServiceTime());
                    }
                    finalAverageTime += totalTimeSpent.get();
                }
            }
        }
    }
    return finalAverageTime;
}
```

Iar timpul de varf l-am stabilit ca momentul când sunt cele mai multe task-uri pe cozi, în total.

```java
public int peakTime(List<Server> servers) {
    int maxxNrPerTime = servers.get(0).getTasks().size();
    int auxNrTasks=0;
    for (Server server : servers) {
        auxNrTasks += server.getTasks().size();
        if (auxNrTasks > maxxNrPerTime) {
            maxxNrPerTime = auxNrTasks;
        }
    }
    return maxxNrPerTime;
}
```

❖ *Clasa ConcreteStrategyQueue*

Implementează interfața Strategy și implicit metoda addTask, care primește o listă de severe/cozi și un Task. Strategia se bazează pe adăugarea task-ului la coada cu cele mai puține task-uri. Am ales să fac minimul, iterez printer cozi si la fiecare iau nr de task-uri, iar dacă este mai mic decât minimul atunci restabilesc minimul și noul server/coadă minima. La sfârșit după ce parcurg toate cozile, adaug task-ul la coada cu număr minim de task-uri găsită, apelând metoda addTask din Server.

```
@Override
public void addTask(List<Server> servers, Task newTask) {
    Server server = servers.get(0);
    int minQueue = servers.get(0).getTasks().size();
    for (Server it : servers) {
        int size = it.getTasks().size();
        if (minQueue > size) {
            minQueue = size;
            server = it;
        }
    }
    server.addTask(newTask);
}
```

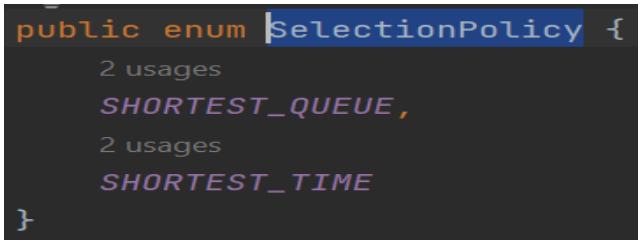
❖ *Clasa ConcreteStrategyTime*

Implementează interfața Strategy și implicit metoda addTask, care primește o listă de severe/cozi și un Task. Strategia se bazează pe adăugarea task-ului la coada cu cel mai scurt timp de așteptare. Aici am folosit waitingPeriod-ul instanțiat în Server. La fel ca la cealaltă strategie am făcut timpul minim iterând prin toate serverele și salvând atunci când găsesc alt minim și serverul current. La final, se adaugă task-ul la server-ul cu timp minim de așteptare, apelând metoda addTask din Server.

```
public void addTask(List<Server> servers, Task newTask) {
    Server server = servers.get(0);
    AtomicInteger minWaitTime = servers.get(0).getWaitingPeriod();
    for (Server it : servers) {
        AtomicInteger time = it.getWaitingPeriod();
        if (minWaitTime.get() > time.get()) {
            minWaitTime = time;
            server = it;
        }
    }
    server.addTask(newTask);
}
```

❖ *Enum SelectionPolicy*

Este un enum cu 2 valori, care stabilesc modul de management al cozilor:



❖ *Clasa Controller*

Implementează interfața de ActionListener. Butonul de START din Frame are adaugat drept ActionListener un obiect de tipul clasei Controller, care implementează interfața ActionListener. În cadrul Frame, la butonul START am adăugat și comanda "Start", ca să identificăm în Controller pentru care acțiune trebuie să reacționeze. Atunci când butonul este apăsat se încearcă citirea tuturor datelor introduse, în cazul în care nu sunt corecte/introduse apar pop-uri de mesaje pentru utilizator.

```
try {
    aux = Integer.parseInt(frame.getNoCozi().getText());
    noServers.set(aux);
} catch (NumberFormatException ex) {
    JOptionPane.showMessageDialog(this.frame, message: "Numarul de cozi nu este corect!", title: "Notification", JOpti
}
```

După ce citirea tuturor datelor este făcută, se lansează o simulare cu datele citite si se pornește un thread pentru simularea începută.

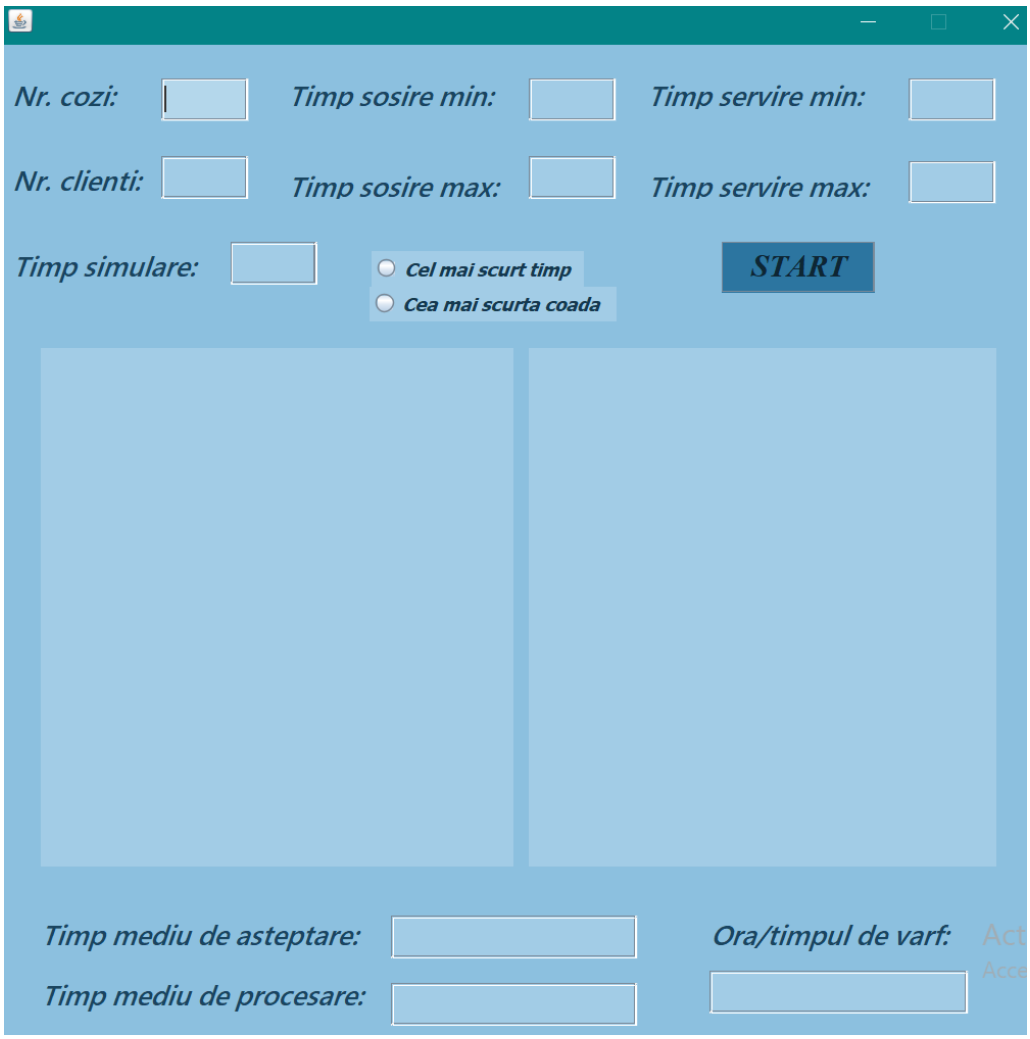
```
///pornirea simularii
try {
    simulare = new SimulationManager(this.frame, timpSimulare.get(), timpServireMin.get(), timpServireMax.g
} catch (IOException ex) {
    throw new RuntimeException(ex);
}

thread = new Thread(simulare);
thread.start();
```

❖ *Clasa Frame*

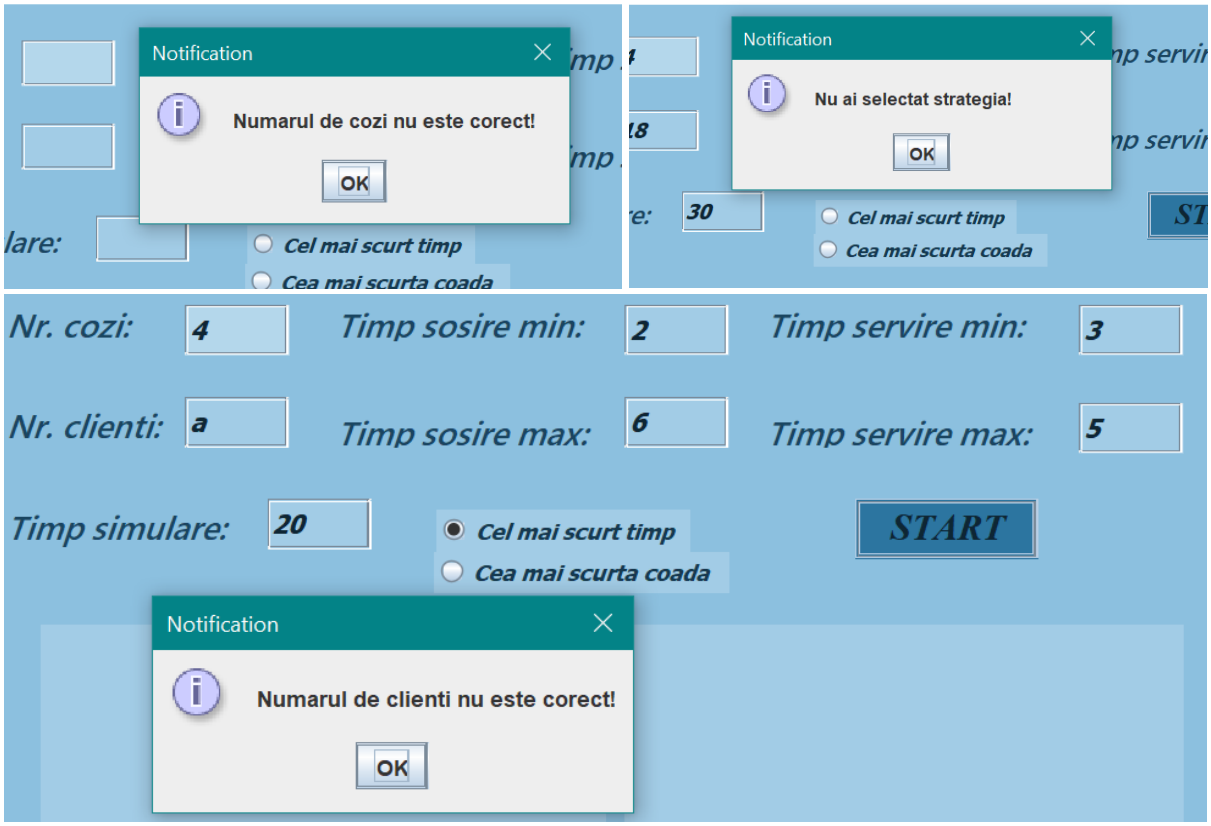
Extinde clasa JFrame și împreună cu clasa Controller au drept scop implementarea unei interfețe user-friendly de management a M clienți generați random la N cozi, într-un timp maxim de simulare, pe baza unei anumite strategii. După introducerea datelor, mai exact numărul de cozi, numărul de clienți, timpul minim de sosire și timpul maxim de sosire, timpul minim de servire și timpul maxim de servire, se introduce și timpul de

simulare și se selectează strategia de aranjare a clinților la cozi. Simularea începe atunci când este apăsat butonul de START, care are rol și de validare a datelor introduse de user. Un obiect de tipul Controller este creat și îi este trimis ca parametru fereastra curentă.

Aplicația se poate închide prin apăsarea butonului de exit/ închidere a programului, adică se închide fereastra de simulare. Atunci când simularea începe butonul de START este blocat până când counter-ul atinge timpul maxim de simulare, moment în care butonul de START devine din nou active și simularea se poate repeata(se pot schimba valorile/strategia, însă se schimbă și clienții care sunt generați random).



În cazul în care datele introduse sunt incorecte/ neintroduse sau dacă strategia nu a fost selectată vor apărea căsuțe pop-up prin care îi este comunicată user-ului problema și îi este cerut să introducă datele corect.



❖ *Clasa Main*

Creează un obiect de tipul Frame, care lansează fereastra de interfață a aplicației.

5. Rezultate

Se observă cum rezultatele sunt influențate de generarea random. De asemenea sunt direct proporționale, cu cât crește numărul de task-uri cu atât mai mare este timpul mediu de așteptare și timpul mediu de procesare. Rezultatele simulării curente sunt afișate în simulation.txt și în interfață

Primul test presupune un număr de 2 cozi, la care sunt repartizați 4 clienți, cu timpul de sosire între 2 și 30 și timpul de procesare între 2 și 4, iar timpul de simulare este de 60 de secunde.

```
Timpul mediu de asteptare este 0.0
Timpul mediu de procesare este 3.5
Timpul de varf este 30
```

Al doilea test presupune un număr de 5 cozi, la care sunt repartizați 50 clienți, cu timpul de sosire între 2 și 40 și timpul de procesare între 1 și 7 , iar timpul de simulare este de 200 de secunde.

```
Timpul mediu de asteptare este 3.0
Timpul mediu de procesare este 7.34
Timpul de varf este 35
```

Al treilea test presupune un număr de 20 cozi, la care sunt repartizați 1000 clienți, cu timpul de sosire între 10 și 100 și timpul de procesare între 3 și 9, iar timpul de simulare este de 200 de secunde.

```
Timpul mediu de asteptare este 21.834
Timpul mediu de procesare este 27.872
Timpul de varf este 100
```

```
Clienti care trebuie pusi la coada:(48,100,3)(119,100,5)(173,100,6)(237,100,3)(316,100,4)(339,100,8)(429,100,5)(712,100,5)(768,100,5)(798,100,7)(800,100,3)(901,100,5)
Timp 100
Coada 1 -> (938,77,4) (142,79,7) (520,79,6) (49,83,4) (981,83,7) (233,85,3) (116,88,6) (126,89,5) (274,90,3) (522,91,4) (912,92,8) (833,95,9) (584,97,4) (93,99,3) (316,100,4)
Coada 2 -> (5,73,1) (850,73,6) (949,77,5) (16,78,3) (737,79,6) (281,81,5) (477,84,3) (276,85,4) (282,89,3) (311,90,4) (530,91,7) (728,94,5) (854,95,7) (643,97,5) (339,100,8)
Coada 3 -> (363,74,3) (969,77,5) (153,78,9) (803,79,3) (508,82,9) (621,84,6) (729,86,9) (299,89,4) (63,91,8) (555,91,4) (544,93,4) (856,95,8) (667,97,4) (378,98,5) (429,100,5)
Coada 4 -> (997,73,4) (470,75,4) (166,78,3) (859,79,5) (801,80,9) (60,83,9) (847,84,5) (94,87,5) (313,89,9) (575,91,4) (725,93,9) (889,95,5) (669,97,5) (180,99,4) (712,100,5)
Coada 5 -> (415,77,7) (186,78,5) (868,79,7) (374,81,8) (874,84,9) (851,86,9) (437,89,3) (578,91,8) (825,94,5) (196,95,5) (919,95,9) (199,97,5) (673,97,9) (48,100,3) (768,100,5)
Coada 6 -> (32,76,4) (232,78,9) (69,80,8) (959,80,8) (262,83,4) (905,84,9) (224,88,3) (607,89,4) (144,91,7) (802,91,3) (879,94,8) (976,95,6) (705,97,6) (491,98,5) (798,100,7)
Coada 7 -> (527,75,4) (279,78,9) (255,80,3) (377,81,5) (532,82,3) (983,84,5) (188,87,5) (647,89,8) (438,90,4) (269,94,3) (312,95,9) (2,96,4) (512,96,8) (751,97,4) (800,100,3)
Coada 8 -> (106,76,2) (370,78,4) (52,81,4) (603,81,8) (26,85,3) (556,85,6) (280,87,5) (657,89,7) (159,91,8) (360,95,6) (23,96,7) (626,96,8) (941,97,6) (200,99,5) (901,100,5)
Coada 9 -> (431,77,3) (408,78,9) (303,79,9) (67,81,7) (617,81,3) (178,84,9) (569,85,3) (247,88,3) (816,89,7) (158,93,7) (381,95,6) (95,96,7) (697,96,7) (982,97,5)
Coada 10 -> (445,77,1) (1,78,5) (483,78,3) (272,80,7) (691,81,3) (351,83,8) (636,85,8) (857,86,9) (924,89,9) (439,90,4) (389,95,4) (118,96,4) (709,96,8) (693,98,6)
Coada 11 -> (506,78,4) (306,79,8) (811,81,3) (763,82,6) (203,84,7) (726,85,9) (472,87,7) (594,90,3) (174,91,9) (192,93,4) (394,95,7) (713,96,4) (930,98,8) (427,99,6)
Coada 12 -> (355,80,8) (831,81,6) (293,84,6) (933,85,7) (764,87,9) (419,88,4) (28,90,4) (696,90,3) (76,92,7) (414,95,3) (221,96,9) (772,96,3) (36,99,5) (562,99,6)
Coada 13 -> (706,78,3) (369,79,6) (547,80,6) (849,81,6) (794,82,3) (963,85,3) (932,87,6) (81,90,4) (788,90,5) (601,94,8) (504,95,4) (773,96,8) (46,98,8) (612,99,8)
Coada 14 -> (809,78,5) (590,80,9) (900,81,5) (476,83,4) (461,84,5) (973,85,8) (660,88,4) (42,91,6) (205,91,8) (277,93,6) (545,95,4) (776,96,5) (185,98,6) (613,99,6)
Coada 15 -> (921,75,2) (467,76,8) (832,78,3) (599,80,3) (962,81,9) (757,83,9) (37,87,9) (966,87,5) (206,91,5) (376,92,7) (565,95,4) (873,96,5) (304,97,6) (677,99,9)
Coada 16 -> (922,75,4) (692,77,3) (916,78,4) (103,81,8) (967,81,5) (83,86,9) (990,87,7) (840,88,8) (225,91,4) (649,94,7) (579,95,7) (877,96,3) (357,97,8) (678,99,6)
Coada 17 -> (718,77,5) (991,78,9) (172,81,8) (971,81,6) (84,86,3) (898,88,6) (59,91,5) (229,91,6) (478,93,9) (627,95,3) (951,96,6) (375,98,3) (739,99,3) (119,100,5)
Coada 18 -> (820,77,1) (436,79,8) (778,80,9) (464,82,9) (797,82,5) (365,86,6) (948,88,4) (263,90,7) (292,91,3) (494,92,7) (698,95,7) (30,97,5) (362,97,7) (846,99,5)
Coada 19 -> (24,77,4) (836,77,6) (497,79,9) (219,81,6) (827,83,6) (55,85,8) (56,87,7) (79,89,9) (342,91,6) (492,93,3) (783,95,6) (302,96,8) (383,97,4) (173,100,6)
Coada 20 -> (152,75,1) (77,77,4) (878,77,6) (509,79,5) (241,81,4) (954,83,9) (622,86,3) (68,88,5) (122,89,6) (386,91,5) (699,94,8) (804,95,5) (397,97,4) (237,100,3)
```

6. Concluzii

Consider că acest proiect m-a ajutat să îmi aprofundez cunoștințele legate de programarea cu fire de execuție. A fost un proiect complex din care am învățat foarte multe lucruri noi.

7. Bibliografie

https://dsrl.eu/courses/pt/
https://users.utcluj.ro/~igiosan/teaching_poo.html
https://www.draw.io/
https://www.tutorialspoint.com/java/java_multithreading.htm
https://www.geeksforgeeks.org/java-threads/