DOCUMENTATIE

TEMA 2

NUME STUDENT: MATIȘ OANA-ANTONIA
GRUPA:30228

CUPRINS

1. Obiectivul temei....3
2. Analiza problemei, modelare, scenarii, cazuri de utilizare 4
3. Proiectare 6
4. Implementare 10
5. Rezultate 14
6. Concluzii....16
7. Bibliografie 16

1. Obiectivul temei

Tema 2 are ca obiective principale conceperea, proiectarea și implementarea unui program funcțional și simplu de utilizat pentru a simula un sistem de procesare a unor cozi ce primesc anumiti clienti la anumite momente de timp, obiectivul principal fiind de a determina timpul de asteptare, care se doreste a fi cat mai mic cu putinta, astfel incat aplicatia de simulare si procesare sa poata sa fie considerata eficienta.

Conceptul de thread (fir de execuție) definește cea mai mică unitate de procesare ce poate fi programată spre execuție de către sistemul de operare. Este folosit în programare pentru a eficientiza execuția programelor, executând porțiuni distincte de cod în paralel în interiorul aceluiași proces. Câteodata însă, aceste portiuni de cod care constituie corpul threadurilor, nu sunt complet independente și în anumite momente ale execuției, se poate întampla ca un thread să trebuiască să aștepte execuția unor instructiuni din alt thread, pentru a putea continua execuția propriilor instrucțiuni. Această tehnică, prin care un thread asteaptă execuția altor threaduri înainte de a continua propria execuție, se numește sincronizarea threadurilor.

Iată câteva dintre obiectivele principale ale proiectului:

- Realizarea unei interfețe grafice - aceasta are rolul de a permite interacțiunea dintre utilizator și program, prin prelucrarea datelor introduse de utilizator, dar și prin prezentarea evoluției cozilor în fiecare moment de timp din intervalul ales.
- Calcularea și afișarea unor rezultate în urma simularii: timpul mediu de așteptare și timpul mediu de procesare a comenzilor.
- Afișarea într-un fișier a rezultatelor legate de starea cozilor în fiecare moment: clienții care au fost adăugați într-o coadă, cei care au părăsit coada, timpul și datele despre clienți.
- Generarea unui număr ales de clienți, fiecare având un ID unic din intervalul [0, numărul de clienți], atribuit în ordine crescătoare în funcție de momentul generării lor. Timpul de procesare și timpul de sosire sunt două numere alese în mod aleator din intervalele stabilite de către utilizator înainte de simulare.
- Distribuirea clienților în cozi, utilizând una dintre criteriile de selectare a cozii: TimeStrategy sau ShortestQueueStrategy.
- Calcularea timpului de plecare corespunzător fiecărui client, în funcție de momentul în care a sosit, cel în care a fost adăugat în coadă și de timpul de procesare al comenzii sale.
- Ștergerea clienților din coadă atunci când timpul lor de plecare este egal cu timpul curent și actualizarea timpului de așteptare.
- Printre obiectivele secundare se pot include: reîmprospătarea cunoștințelor anterioare despre cozi, consolidarea noilor cunoștințe despre firele de execuție și interfața

Runnable, dobândirea experienței de proiectare cu modelul Model-View-Controller și dezvoltarea unei gândiri orientate pe obiect, precum și înțelegerea mai profundă a tehnicilor de programare orientată pe obiect și implementarea unor interfețe grafice de comunicare cu utilizatorul cât mai ușor de utilizat.

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

Cerințele funcționale ale proiectului sunt:

- Aplicația de simulare trebuie să permită utilizatorului să configureze simularea
- Aplicația de simulare trebuie să permită utilizatorului să pornească simularea
- Aplicația de simulare trebuie să afișeze evoluția cozilor în timp real

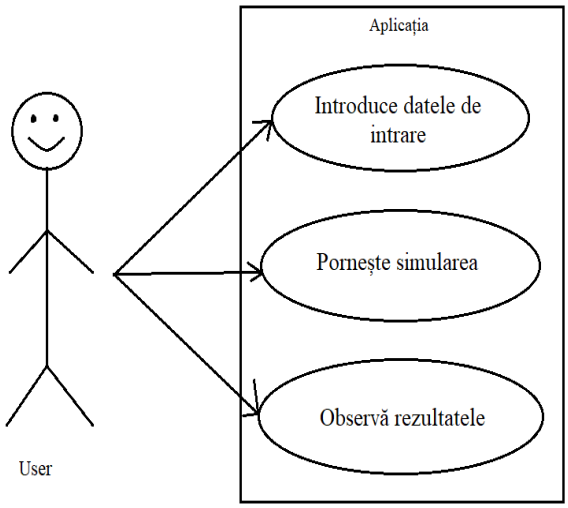
Cerințele non-funcționale ale proiectului sunt:

- Aplicația de simulare trebuie să fie intuitivă și ușoară de folosit de către utilizator

Use-caseurile sunt niște acțiuni sau evenimente care definesc interacțiunile dintre un rol și un sistem pentru a îndeplini o cerință. Aceasta diagramă use-case reprezintă use-caseurile și actorii proiectului (în cazul nostru un singur actor există, și anume userul aplicației).

Descrierea use-caseurilor:

1. Utilizatorul introduce toate datele necesare pentru simulare în câmpurile: time limit, number of clients, number of queues, min și max arrival time, min și max service time.
2. Utilizatorul apasă butonul de start
3. Aplicația afișează evoluția cozilor în timp real



Odată ce s-a terminat pasul 3, utilizatorul are două posibilități: ori apasă butonul de reset, astfel revenind la pasul 1, ori apasă butonul start din nou, pornind o nouă simulare cu aceleași date de intrare. Deși datele de intrare sunt aceleași, rezultatul probabil va fi diferit, deoarece clienții se generază aleator.

Simularea modului de desfasurare a evenimentelor:

Actor principal: utilizatorul

Scenariul principal de success:

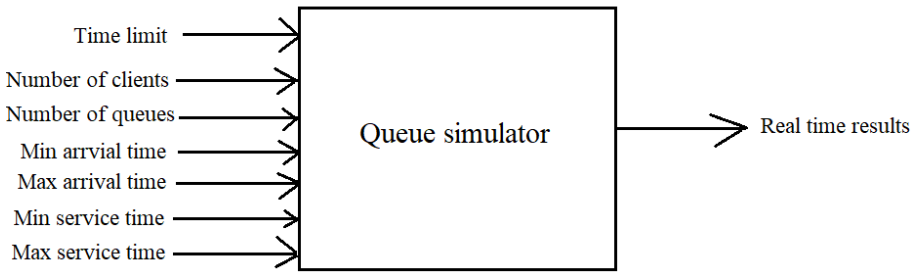
1. Se alege de catre utilizator numarul de clienti care trebuie diStribuit in cozile de asteptare.
2. Se alege de catre utilizator care este numarul de cozi in care sa fie distribuiti clientii.
3. Se introduc de catre utilizator valorile pentru timpul minim de sosire a clientilor si timpul maxim pentru sosirea lor in text field-urile potrivite.
4. Se introduce de catre utilizator in interfata grafica o valoare pentru timpul minim de procesare a unei comenzi.
5. Se alege de catre utilizator care este valoarea timpului maxim de procesarea a comenzilor si il introduce in locul potrivit.
6. Se introduce de catre utilizator valoarea momentului de timp pana la care are loc simularea in interfata grafica.
7. Se alege de catre utilizator modalitatea de distribuire a clientilor in cozi, prin alegerea a uneia din cele doua optiuni: criteriul prin care este aleasa coada in care o sa fie adaugat noul client(timpul de asteptare minim sau coada in care se afla cel mai mic numar de persoane).
8. Se apasa de catre utilizaor butonul cu textul SUBMIT/START.
9. Simulatorul afiseaza in interfata reprezentarea cozilor de asteptare care se actualizeaza in fiecare moment.
10. Simulatorul scrie in fisierul .txt o descriere a evenimentelor pentru fiecare coada in parte.

Secvență alternativă: date de intrare introduse gresit

1. Se introduc texte care nu pot fi convertite din string in integer
   -Sirurile de caractere contin caractere care nu sunt cifre
   -Datele de intrare lipsesc
   -Numerele introduse nu sunt in concordanta: timpul de simulare este prea mic, sau timpul maxim de sosire este mai mic decat cel minim.
2. Avem exceptie de tipul IOException si simularea e intrerupta si in interfata nu se afiseaza nimic.

3. Proiectare

1) Proiectarea OOP



Pentru a realiza funcționalitățile simulatorului de cozi, avem nevoie de 7 intrări: **Time Limit**, care e durata simulării, **Number of Clients** (numărul de clienți), **Number of Queues** (numărul cozilor), **Min arrival time** (timpul minim de sosire a clienților), **Max arrival time** (timpul maxim de sosire a clienților), **Min service time** (timpul minim de servire a unui client), **Max service time** (timpul maxim de servire a unui client).

Modelele arhitecturale definesc structuri pentru sisteme software în termeni de subsisteme predefinite și responsabilităților acestora.

Pentru proiectarea simulatorului de cozi am ales modelul arhitectural de implementare **MVC (Model View Controller)**, care împarte aplicația noastră în trei părți:

- **Componentele Model:** încapsulează datele de bază și funcționalitățile
- **Componentele View:** afișează informații utilizatorului, obține datele pe care le afișează de la Model
- **Controller:** Fiecărui View îi este asociată o componentă Controller. Controllerele primesc intrarea de obicei sub forma unor evenimente care denotă mișcarea mouse-ului, activarea butoanelor mouse-ului sau intrarea de la tastatură. Evenimentele sunt traduse în cereri de service, care sunt trimise fie la Model sau la View

Clasele pe care le-am utilizat pentru realizarea acestei structuri sunt următoarele:

- **Clasa Task:** Clienții reprezintă obiectul principal care participă la simulare. Pe ei se vor efectua acțiuni: se vor adăuga și se vor șterge din cozi. Clienții aplicației se generează aleator. Această clasă conține atributele și metodele caracteristice unui client.
- **Clasa Server:** Clasa conține atributele și metodele caracteristice unei cozi. O coadă poate conține mai mulți clienți.

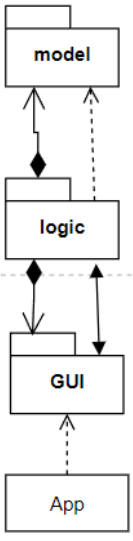
- **Clasa Scheduler:** În această clasă se vor crea atâtea cozi câte se precizează de către utilizator. Tot aici, pentru fiecare coadă, se pornește câte un thread, iar când se termină simularea, se vor opri aceste threaduri.
- **Clasa Strategy:** Această clasă va adăuga clienții în coada potrivită, adică în coada unde timpul de așteptare este minim în momentul respectiv (adică unde trebuie să aștepte cel mai puțin).
- **Clasa SimulationManager:** Aceasta este clasa principală care realizează funcționalitatea aplicației de simulare. Aici se va crea threadul principal, se vor genera clienți aleatorii, respectiv e clasa Main a proiectului.

Clasele prezentate mai sus realizează funcționalitățile aplicației, astfel se încadrează la componenta de **Model** a structurii MVC. Pe lângă clasele Model, avem următoarele clase:

- **Clasa SimulationFrame:** în această clasă este realizată interfața utilizatorului (GUI). În modelul MVC, aceasta reprezintă componenta **View**.
- **Clasa SimulationController:** această clasă "controlează" acțiunile aplicației, adică în cazul apăsării unui buton se ocupă de evenimentele, acțiunile care trebuie să urmeze pentru ca aplicația noastră să îndeplinească cerințele și să funcționeze corect. În arhitectura MVC, această clasă reprezintă **Controller**-ul.
- **Clasa FilesUtil:** această clasă nu are un rol esențial în funcționalitatea proiectului, are doar rolul de a scrie în fișiere rezultatele simulărilor.

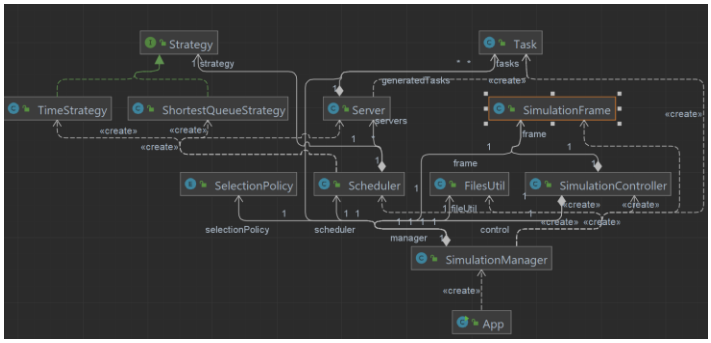
2) Diagrama UML

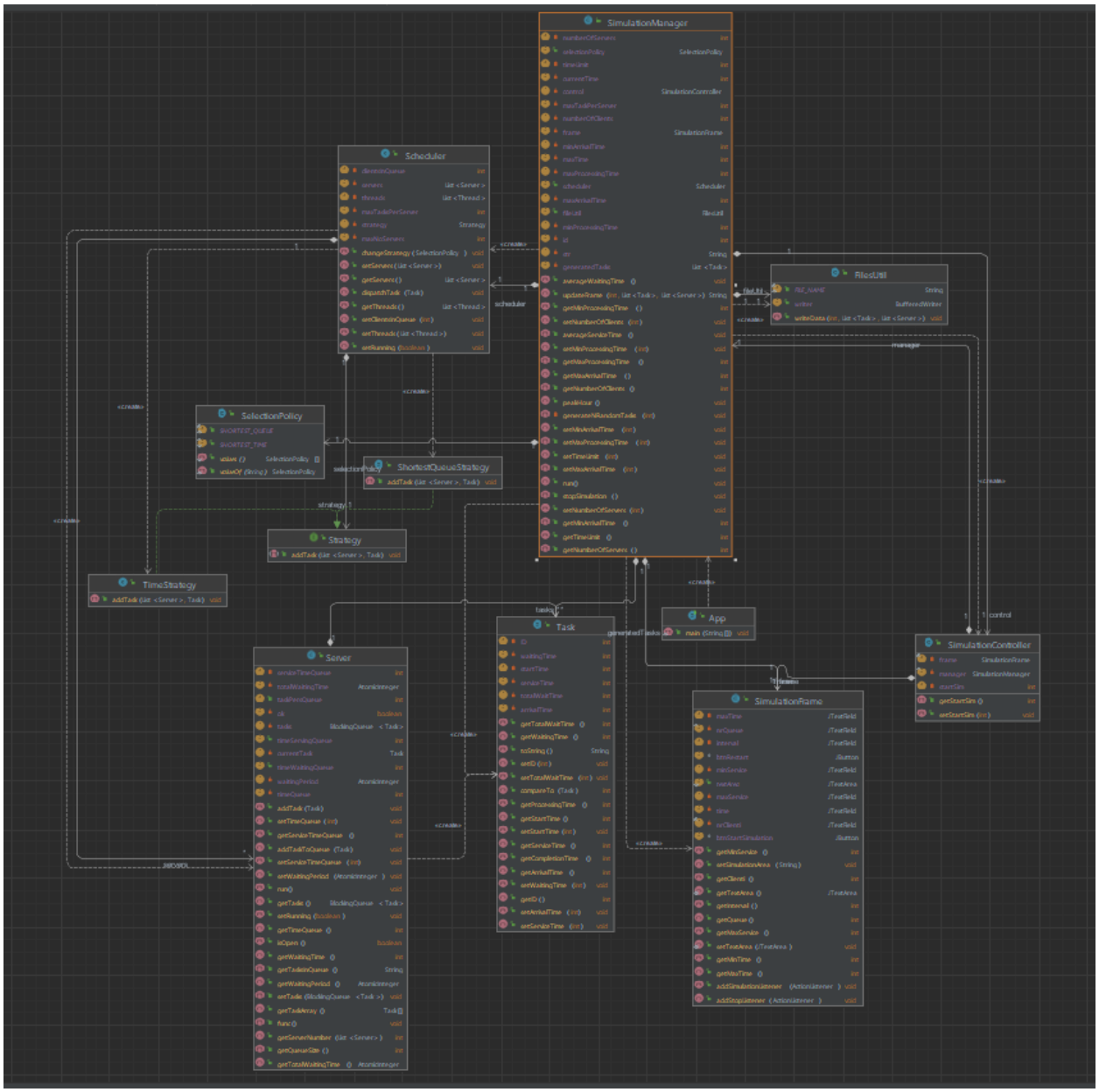
Diagrama de pachete



Aplicația este organizată în trei pachete: Model, GUI și Logic, care grupează împreună clase și interfețe legate între ele. Pachetul model conține clasele "Server" (care reprezintă cozile) și "Task" (care reprezintă clienții). Pachetul logic conține clasele care sunt importante pentru simulare, cum ar fi "SimulationManager" sau "Scheduler". Pachetul GUI conține interfețele grafice de utilizator și controlorul asociat.

Diagrama de clase





3) Structuri de date folosite

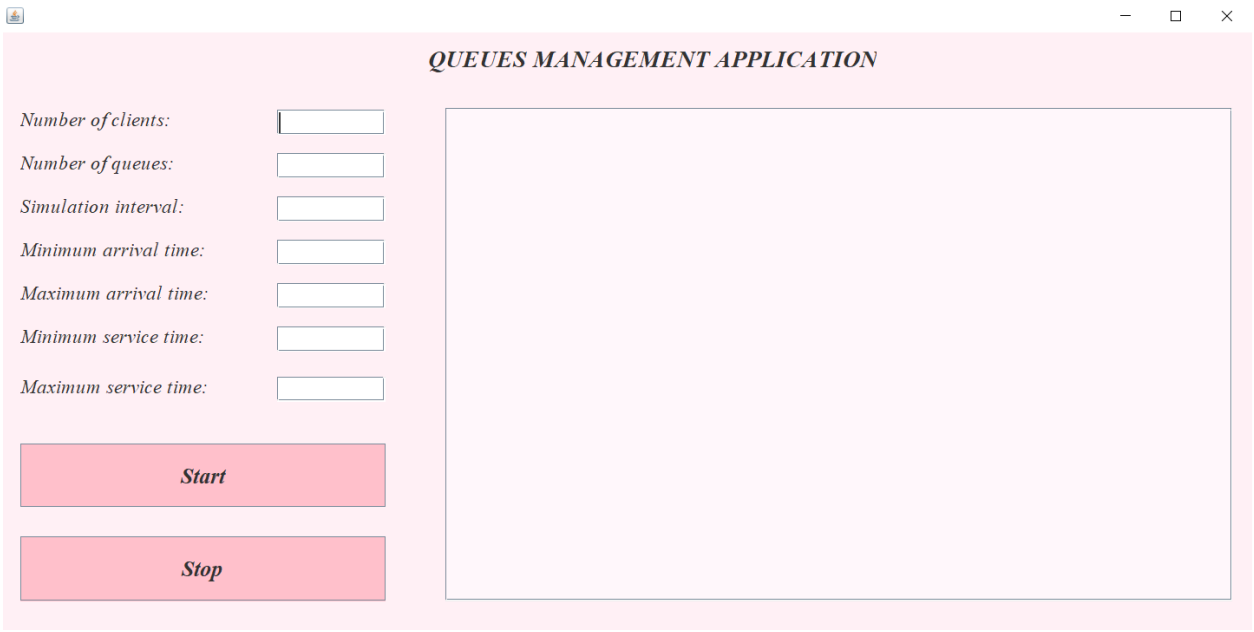
In acest proiect am folosit structurile: BlockingQueue si AtomicInteger.

Metodele BlockingQueue vin în patru forme, cu moduri diferite de a gestiona operațiuni care nu pot fi satisfăcute imediat, dar pot fi satisfăcute la un moment dat în viitor: unul aruncă o excepție, al doilea returnează o valoare specială (fie nulă, fie falsă, în funcție de operațiunea), al treilea blochează firul curent pe termen nelimitat până când operațiunea poate reuși, iar al patrulea blochează doar pentru o limită de timp maximă dată înainte de a renunța. Aceste metode sunt rezumate în următorul table.

BlockingQueue ete folosit pentru a adauga si a sterge concurrent clienti , acesta fiind capabil sa blocheze firele de lucru care incearca sa insereze sau sa ia elemente din coada.Spre exemplu daca thread-ul vrea sa ia un element insa acesta nu exista va astepta pana la aparitia unuia.

Clasa AtomicInteger oferă o variabilă int care poate fi citită și scrisă atomic și care conține, de asemenea, operații atomice avansate precum compareAndSet(). Clasa AtomicInteger se află în pachetul java.util.concurrent.atomic, deci numele complet al clasei este java.util.concurrent.atomic.AtomicInteger . Acest text descrie versiunea AtomicInteger găsită în Java 8, dar prima versiune a fost adăugată în Java 5.

Reprezentarea interfeței grafice



4. Implementare

Clasa Task este definită în pachetul "model" și implementează interfața "Comparable". Aceasta are cinci variabile de instanță: "ID", "arrivalTime", "serviceTime", "waitingTime" și "totalWaitTime", toate acestea fiind private.

Constructorul clasei "Task" primește trei parametri și inițializează valorile ID-ului, timpului de sosire și timpului de serviciu.

- Metodele "getWaitingTime" și "setWaitingTime" returnează și setează timpul de așteptare al obiectului "Task".
- Metoda "getTotalWaitTime" calculează și returnează timpul total de așteptare al obiectului "Task".
- Metoda "getID" returnează ID-ul obiectului "Task", iar metoda "setID" setează ID-ul obiectului "Task".
- Metoda "getArrivalTime" returnează timpul de sosire al obiectului "Task", iar metoda "setArrivalTime" setează timpul de sosire al obiectului "Task".
- Metoda "getServiceTime" returnează timpul de serviciu al obiectului "Task", iar metoda "setServiceTime" setează timpul de serviciu al obiectului "Task".
- Metoda "toString" returnează o reprezentare sub formă de șir de caractere a obiectului "Task".
- Metoda "compareTo" compară două obiecte "Task" pe baza timpului de sosire.
- Metoda "getProcessingTime" calculează și returnează timpul de procesare al obiectului "Task".
- Metoda "getCompletionTime" calculează și returnează timpul de finalizare al obiectului "Task".
- Metoda "getStartTime" returnează timpul de începere al obiectului "Task", iar metoda "setStartTime" setează timpul de începere al obiectului "Task".

Clasa "Server" implementează interfața **Runnable**, ceea ce înseamnă că poate fi rulată într-un fir de execuție separat.

Clasa **Server** conține câteva variabile de instanță, cum ar fi:

- **tasks** - o coadă de obiecte **Task** (unde **Task** este o altă clasă din programul dat);
- **waitingPeriod** - o valoare atomică întreagă care reprezintă perioada de așteptare totală a tuturor sarcinilor din coadă (în secunde);
- **timeQueue**, **serviceTimeQueue**, **timeWaitingQueue**, **timeServingQueue**, **taskPersQueue** - alte variabile întregi utilizate pentru a calcula diferite statistici.

Clasa **Server** conține, de asemenea, mai multe metode, printre care:

- **getTimeQueue()**, **getTasks()**, **setTimeQueue(int)**, **getServiceTimeQueue()**, **setServiceTimeQueue(int)**, **getTotalWaitingTime()**, **getWaitingPeriod()**, **getQueueSize()**, **getTasksInQueue()**, **getWaitingTime()** - aceste metode sunt folosite pentru a accesa și a gestiona variabilele de instanță menționate anterior;
- **Server()**, **Server(BlockingQueue<Task>, AtomicInteger)**, **Server(int)** - acestea sunt constructorii clasei;

- **addTask(Task)**, **addTaskToQueue(Task)** - aceste metode sunt utilizate pentru a adăuga sarcini la coadă;
- **getTaskArray()** - această metodă convertește coada de sarcini într-un tablou de obiecte **Task**;
- **run()** - această metodă este apelată atunci când firul de execuție asociat instanței curente este pornit și începe să proceseze sarcinile din coadă;
- **getServerNumber(List<Server>)** - această metodă este utilizată pentru a obține numărul unui server dintr-o listă dată de servere;
- **isOpen()** - această metodă verifică dacă coada de sarcini a serverului este goală sau nu;
- **func()** - această metodă este utilizată în scopuri de testare și setează variabila **ok** la **false**;
- **setRunning(boolean)** - această metodă nu face nimic și pare să fie nefolosită în program.

Clasa "TimeStrategy" implementează interfața "Strategy" și definește strategia de alocare a task-urilor la servere. Această strategie se bazează pe timpul de așteptare în coadă al fiecărui server, alegându-se serverul cu cel mai mic timp de așteptare.

Metoda "addTask" primește o listă de servere și un obiect de tip "Task", apoi parcurge fiecare server din listă și găsește serverul cu cel mai mic timp de așteptare în coadă. Apoi adaugă timpul de servire al noului task la timpul de așteptare din coadă și setează noul timp de așteptare pentru serverul respectiv.

Această strategie garantează o alocare echilibrată a task-urilor, evitând supraaglomerarea unor servere și sub-utilizarea altora.

Clasa *"ConcreteStrategyQueue"* reprezinta stategia alegerii cozii in functie de lungimea cozii, adica in aceasta clasa se verifica cati client se afla intr o coada. Aceasta clasa implementeaza interfata Strategy. Astfel aceasta implementeaza codul pentru adaugarea unui client "*adaugaClient"*.

Clasa"FilesUtil" reprezinta clasa pentru a scrie intr-un fisier denumit "*simulationOutput.txt"* date simularii aplicatiei in timp real. Pentru a putea implementa aceasta metoda am folosit metode si obiecte de tipul BufferWriter pe care le-am folosit incluzand bilbiotecile :

```
import java.io.BufferedWriter;
import java.io.FileWriter;
```

Pentru a scrie in fisier am creat un obiect de tip "*BufferWriter"* , am creat un string care sa fie de forma celuia pe care dorim sa il afisam. Adica acesta contine timpul simularii, clinetii care asteapta sa fie introdusi intr o coada, adica care au *arrivalTime<timpCurent* si mai apoi pe randuri noi fiecare coada cu clientii din acesta la timpul curent al simalarii. Acestui string i-am facut append in fisier folosind apelul motodei implementate in java, care a fost totodata sugerat

de intellij. Astfel in fisier nu se vor suprascrie datele de mai sus ci se vor adauga la finalul fisierului, aceasta avand toate datele simularii.

Clasa *"Scheduler"* este clasa unde am creat metodele unde se apeleaza alegerea pentu fiecare client a strategiei si a cozii in care urmeaza sa fie introdus. Aceasta clasa are ca si instante: o lista de servere(queues), numarul maxim de servere, numarul maxim de clienti dintr-un server si strategia aleasa in functie de care se face alegerea cozii in car va fi pus urmatorul client, adica timpul lui de sosire sa fie egal cu timpul de simulare curent.

Clasa sau altfel spui enumeratia ***"SelectionPolicy"*** contine strategiile implementate mai sus, si anume:

```
public enum SelectionPolicy {
   SHORTEST_QUEUE, SHORTEST_TIME
}
```

Clasa *"Server"* implementeaza interfata runnable intrucat pentru fiecare server am creat un thread. Adica pentru fiecare coada. Aceasta clasa are ca si instante: o coada de client, o perioada de asteptare, timpul petrecut in coada si timpul de servire. In aceasta clasa am implementat metoda *"adaugaClient"* prin care adaugam clintii in coada si calculam timpul de asteptare pentru coada respective.

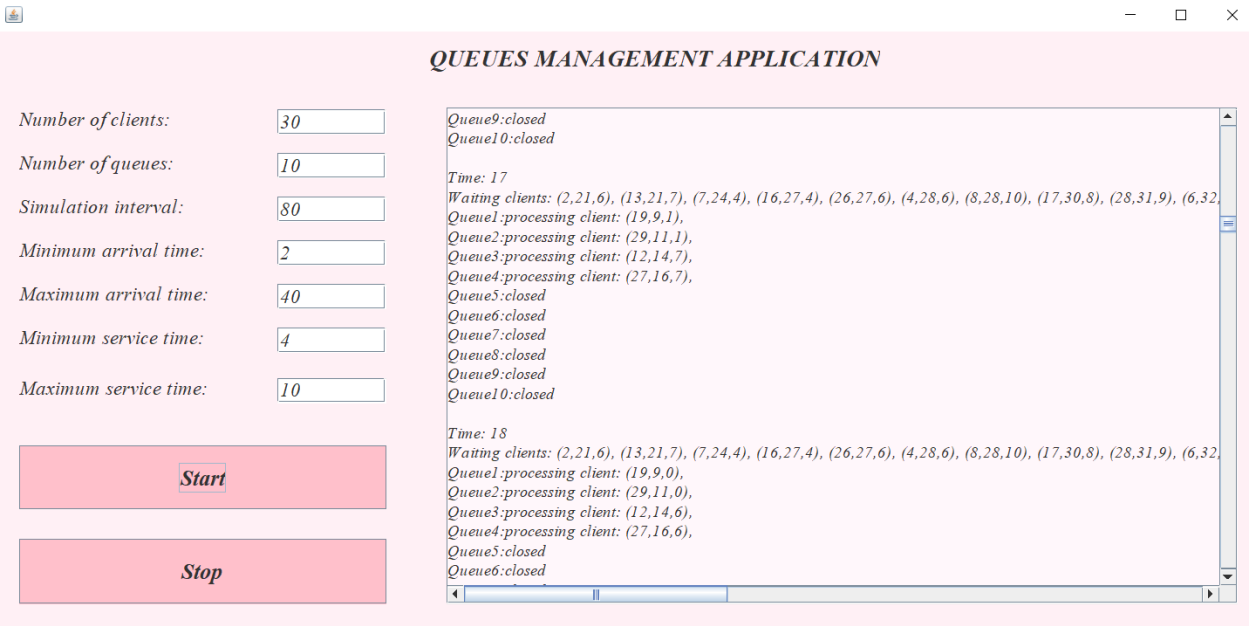
Clasa *"SimulationManager"* este clasa in care am implementat metoda pentru a genera random clientii si pentru a ii sorta pe acestea in fucntie de timpul lor de sosire in coada. Totodata in aceasta clasa avem un constructor *SimulatonManager*() care ia din interfata creata datele necesare simularii si le proceseaza astfel incat sa creeze o lista de clienti apeland metoda *generateRandomClients*, si cozile apelanda constructorul Scheduler:

Si mai apoi alege strategia care va fi folosita pentru adaugarea in cozi a clientilor. Tot in clasa SimulationManager contine si urmatoarele metode: "*calcTimpMediuAsteptare"*, *"calcTimpMediuServire"(*care calculeaza timpul de asteptare, respective timpul de servire dupa cum le spune numele*), "oraVarf"(*calculeaza momentul cand a fost cea mai mare aglomeratie in cozi), si metoda *"updateFrame*"(updateaza permanent interfata grafica si afiseaza modificarile care se face in cozi la fiecare moment de timp al simularii). Aceasta clasa implementeaza interfata *Runnable*, astfel ca in interiorul ei am creat si metoda run(). In aceasta am implementat codul pentru simularea cozilor in sine, adica pentru a introduce clientii in cozi atunci cand timpul de sosire este egal cu timpul curent de simulare si de scoatere din coada atunci cand a trecut timpul de procesare al clinetului din capul cozii. Tot in acealsa clasa am scris codul pentru actualizarea listei de clienti care asteapta sa fie introdusi in cozi, astfel daca un client a intrat intr-o coada va fi scos din lista de asteptare.

Clasa *"SimulationFrame"* este clasa unde am creat design-ul interfetei, la aceasta am adaugat butonul de simulare care va incepe simularea in momentul apasarii. Tot aici avem panoul in care se vor afisa rezultatele simularii, si 7 casute de text in care introduce datele: numarul de clineti(N), numarul de cozi pe care le avem la dispozitie(Q), timpul limita, adica timpul maxim de simulare, timpul minim de sosire al clientilor care urmeaza sa fie generate, timpul minim de servire sau de procesare al clientilor care urmeaza sa fie generate random. Timpul maxim de servire sau de procesare al clientilor care urmeaza sa fie generate.

Clasa "*SimulationController*" este clasa in care am implementat o subclasa SimuateListener care implementeaza interfata ActionListener si care creaza ActionListener pentru butonul de simulare din interfata grafica. Aici am creat o variabila *startSimulation* care va fi egala cu '0' daca butonul de simulare din interfata nu a fost apasat sau altfel poate lua valoarea '1' daca apasam butonul de Start simulation de la mouse. Acest buton are rorlul de a verifica daca a inceput sau nu simularea ca sa stim cand putem lua din castuele de test din interfata datele pe care le-am introdus de la tastatura. Daca nu am folosi acest buton ar aparea niste erori pe parcursul rularii programului sau anumite exceptii datorita faptului ca nu avem de unde extrage datele pentru a crea cozile si pentru a genera clientii si deci acestea din urma nu se vor crea si vor ramane nule sau neinitializate putand fi posibile tosui afisari incomplete, fara sens, sau care sa aiba anumite lipsuri. Asadar este foarte important sa avem in vedere daca simularea este sau nu pornita, pentru a putea extrage corect datele din casutele completate de noi din interfata grafica.

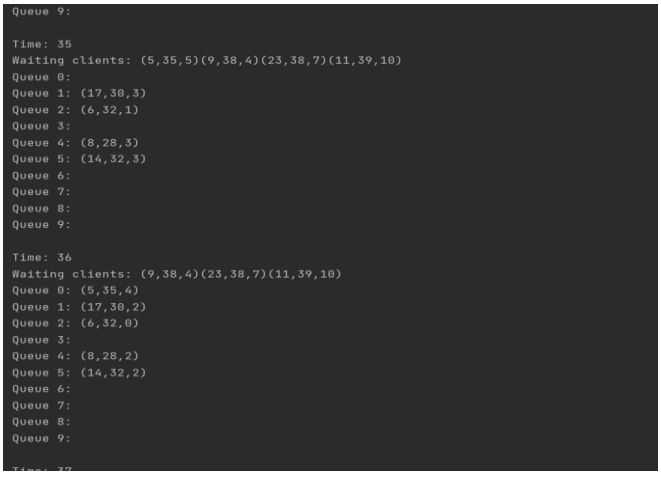
5. Rezultate

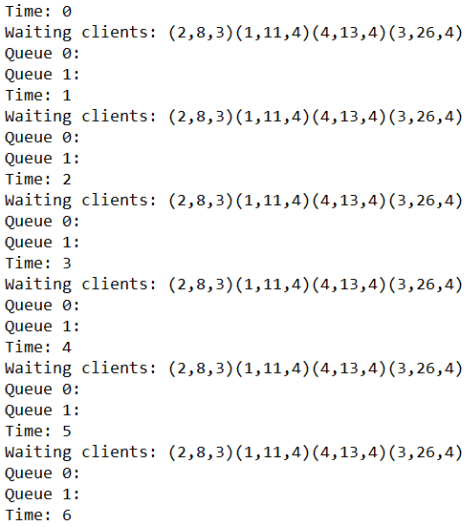


Se va genera timp aleator de sosire si cand se va fi egal cu timpul curent se va adauga in coada, apoi se va decrementa waitingProcess, iar cand va ajunge la 0 se va scoate din coada. Totodata, se va calcula si timpul mediu de asteptare.

Programul ilustreaza utilizarea cozilor pentru reducerea timpul de asteptare si de servire.

Utilizatorul poate folosi programul foarte usor cu ajutorul interfetei grafice, in partea stanga introducand datele, iar in partea dreapta se vora fisa rezultatele intr-un text area.





Testele pe care le-am incercat sunt cele primate la laborator unde sunt definite si cerintele necesare si minime cat si baremul de notare si cateva mici indicatii legat de proiectarea aplicatiei. Astfel cele 3 teste pe care le am incercat sunt:

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

```
Time: 13
Waiting clients: (4,13,4)(3,26,4)
Queue 0:
Queue 1:
Time: 14
Waiting clients: (3,26,4)
Queue 0: (4,13,3)
Queue 1:
Time: 15
Waiting clients: (3,26,4)
Queue 0: (4,13,2)
Queue 1:
Time: 16
Waiting clients: (3,26,4)
Queue 0: (4,13,1)
Queue 1:
Time: 17
Waiting clients: (3,26,4)
Queue 0: (4,13,0)
Queue 1:
Time: 18
Waiting clients: (3,26,4)
Queue 0:
Queue 1:
Time: 19
Waiting clients: (3,26,4)
```

```
Time: 44
Waiting clients:
Queue 0: (38,38,3)(13,40,3)
Queue 1: (39,38,2)(25,40,3)
Queue 2: (12,38,5)(28,39,7)
Queue 3: (27,36,0)(33,38,1)
Queue 4: (44,37,0)(35,38,1)
Time: 45
Waiting clients:
Queue 0: (38,38,2)(13,40,3)
Queue 1: (39,38,1)(25,40,3)
Queue 2: (12,38,4)(28,39,7)
Queue 3: (33,38,1)
Queue 4: (35,38,1)
Time: 46
Waiting clients:
Queue 0: (38,38,1)(13,40,3)
Queue 1: (39,38,0)(25,40,3)
Queue 2: (12,38,3)(28,39,7)
Queue 3: (33,38,0)
Queue 4: (35,38,0)
Time: 47
Waiting clients:
Queue 0: (38,38,0)(13,40,3)
Queue 1: (25,40,3)
Queue 2: (12,38,2)(28,39,7)
Queue 3:
Queue 4:
Time: 48
```

```
Time: 28
Waiting clients: (20,28,9)(92,28,5)(113,28,9)(116,28,4)(133,28,6)(187,28,5)(474,28,4)(482,28,6)(584,28,4)(627,28,6)(4,29,8)(114,29,6)(124,29,8)(293,29,8)(425,29,8)(586,29,9)(650,29,4)(710,2
5)(469,37,8)(514,37,7)(550,37,7)(640,37,4)(700,37,4)(851,37,4)(861,37,3)(927,37,4)(951,37,8)(968,37,9)(9,38,5)(59,38,7)(354,38,5)(400,38,3)(609,38,8)(770,38,3)(812,38,3)(920,38,6)(46,39,8)(
34,45,4)(451,45,6)(504,45,9)(642,45,8)(674,45,9)(696,45,5)(786,45,9)(802,45,7)(808,45,6)(917,45,4)(959,45,9)(7,46,4)(125,46,5)(161,46,4)(176,46,3)(352,46,8)(403,46,5)(415,46,9)(439,46,7)(46
(681,53,9)(722,53,7)(753,53,4)(922,53,5)(978,53,6)(248,54,4)(266,54,7)(271,54,7)(417,54,6)(442,54,5)(493,54,6)(727,54,6)(986,54,6)(989,54,8)(105,55,4)(136,55,8)(220,55,8)(453,55,5)(471,55,3
52,63,6)(273,63,8)(480,63,7)(539,63,3)(554,63,9)(677,63,6)(684,63,7)(701,63,5)(749,63,4)(960,63,5)(239,64,4)(313,64,8)(337,64,4)(376,64,4)(490,64,8)(850,64,4)(971,64,8)(8,65,4)(32,65,8)(362
3)(173,73,4)(254,73,3)(363,73,6)(421,73,7)(593,73,5)(807,73,8)(832,73,6)(866,73,3)(146,74,5)(194,74,6)(267,74,6)(339,74,8)(601,74,7)(669,74,7)(705,74,3)(712,74,7)(717,74,3)(720,74,4)(925,74
86,82,6)(981,82,8)(66,83,4)(75,83,6)(95,83,8)(205,83,6)(298,83,3)(343,83,9)(473,83,3)(477,83,3)(552,83,7)(576,83,8)(873,83,4)(883,83,9)(887,83,5)(952,83,9)(975,83,3)(247,84,7)(295,84,6)(360
92,8)(55,93,4)(126,93,8)(229,93,5)(274,93,7)(347,93,6)(388,93,3)(789,93,4)(898,93,8)(80,94,7)(170,94,3)(240,94,5)(277,94,4)(317,94,5)(517,94,8)(572,94,3)(621,94,6)(649,94,4)(811,94,5)(246,9
Queue 0: (692,17,4)(662,19,7)(36,22,9)(447,22,8)(738,23,4)(974,26,5)
Queue 1: (773,17,3)(12,19,9)(804,19,3)(675,22,7)(814,23,9)(365,24,3)(279,27,8)
Queue 2: (912,17,0)(68,20,5)(135,21,6)(734,22,7)(522,24,4)(110,25,3)(318,27,3)
Queue 3: (943,17,2)(307,18,3)(73,20,4)(737,22,8)(744,24,5)(99,27,3)(342,27,6)
Queue 4: (35,18,1)(349,18,4)(74,20,8)(758,22,4)(792,24,5)(121,25,7)(361,27,5)
Queue 5: (774,15,1)(357,18,3)(90,20,6)(50,22,3)(829,22,9)(290,25,4)(395,27,9)
Queue 6: (819,15,0)(397,18,9)(258,20,3)(169,22,6)(881,22,4)(449,25,3)(543,27,4)
Queue 7: (422,18,5)(166,19,7)(268,20,4)(896,22,6)(479,25,7)(598,27,8)
Queue 8: (871,15,1)(459,18,7)(334,20,8)(253,22,5)(963,22,8)(825,25,9)(725,27,4)
Queue 9: (24,17,1)(476,18,6)(401,20,3)(61,23,4)(198,24,6)(878,25,8)(756,27,4)
Queue 10: (370,16,0)(523,18,5)(409,20,9)(302,22,5)(299,23,3)(48,26,8)(854,27,6)
Queue 11: (668,18,8)(430,20,7)(380,22,3)(333,23,9)(217,26,4)(965,27,7)
Queue 12: (972,18,5)(214,19,4)(721,20,6)(424,23,7)(309,26,7)(127,27,9)
Queue 13: (803,16,0)(1000,18,4)(241,19,9)(888,20,7)(455,23,4)(369,26,5)
Queue 14: (889,16,0)(300,19,4)(942,20,9)(427,22,4)(486,23,6)(396,26,6)
Queue 15: (108,17,0)(315,19,8)(150,21,4)(6,23,6)(499,23,8)(585,26,4)
Queue 16: (259,17,0)(332,19,3)(77,21,6)(177,21,5)(654,23,8)(606,26,4)
Queue 17: (261,17,1)(575,19,3)(107,21,8)(505,21,7)(686,23,7)(666,26,6)
Queue 18: (465,17,4)(644,19,5)(655,21,9)(691,23,5)(218,24,9)(945,26,8)
Queue 19: (636,17,0)(656,19,3)(867,21,5)(441,22,6)(732,23,3)(967,26,8)
Time: 29
```

6. Concluzii

În acest proiect am dobândit cunoștințe despre lucrul cu cozi și thread-uri, ceea ce consider că este extrem de util, deoarece acestea sunt folosite în multe domenii și scopuri în viața reală. Programul ar putea fi îmbunătățit prin implementarea unor situații neprevăzute care ar putea duce la creșterea timpului de așteptare sau de procesare.

În ceea ce privește această aplicație, am întâmpinat anumite dificultăți, însă din acestea am reușit să extrag concluzii și idei valoroase care îmi pot fi de ajutor în implementarea altor proiecte în viitor. Un aspect crucial constă în identificarea soluțiilor pentru sub-obiectivele necesare atingerii obiectivului principal, deoarece acesta este alcătuit, în general, din mai multe astfel de sub-obiective. De asemenea, am învățat că, pentru a rezolva o problemă, este necesar să alegem structurile de date optime în funcție de specificul cazului. În contextul nostru, am descoperit că utilizarea BlockingQueue s-a dovedit a fi mult mai eficientă decât utilizarea unui vector.

7. Bibliografie

https://stackoverflow.com/
https://www.w3schools.com/java/java_threads.asp
http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html
https://www.javatpoint.com/how-to-create-a-file-in-java
https://www.w3schools.com/java/java_files.asp