Universitatea Tehnică Cluj-Napoca

Facultatea de Automatică și Calculatoare

Departamentul Calculatoare



**Proiect**

la disciplina

Tehnici de Programare

**Simulator de cozi de așteptare**

**Pînzariu Răzvan**

**30222**

An academic: 2020 - 2021

Cuprins

[Obiectivul temei 3](#_Toc68698266)

[Analiza problemei, modelare, scenariu, cazuri de utilizare 3](#_Toc68698267)

[Proiectare 5](#_Toc68698268)

[Implementare 7](#_Toc68698269)

[Rezultate 10](#_Toc68698270)

[Concluzii 10](#_Toc68698271)

[Bibliografie 10](#_Toc68698272)

# Obiectivul temei

Obiectivul temei curente este de a implementa în limbajul Java un simulator de cozi de așteptare (en. Queue Simulator) cu o interfață grafică interactivă ce permite utilizatorului să introducă datele de pornire și să inițializeze simualrea. Evoluția programului va fi prezentată într-un fișier text “log.txt” pentru fiecare secundă, iar la final se vor specifica date generale despre simulare (timpi de așteptare medii, ore de vârf etc.)

Pe plan secundar se vor găsi solțuii pentru:

* Analiza problemei și identificarea cazurilor de utilizare pe baza cerințelor necesare
* Proiectarea simulatorului în clase și componente specifice tehnicii de programare alese
* Implementarea simulatorului folosind clase cu funcționalități precise
* Testarea și vizualizarea evoluției programului

# Analiza problemei, modelare, scenariu, cazuri de utilizare

Problema timpului de așteptare la coadă a devenit tot mai important in ultimele decenii, cum numaărul populației a crescut, aglomerațiile au dat naștere unor cozi interminabile și nedorite de nimeni. Cum soluția brută(deschiderea mai multor cozi pentru a reduce timpul) cere numeroase resurse și sacrificii și e adesea limitată de diverși factori, s-au căutat în ultimii ani metode eficiente de gestionare a acestor cozi, fară a folosi soluții colosale.

Pentru a rezolva o astfel de problemă trebuie mai întâi să poată fi vizualizată concret. Cu ajutorul mediului digital putem simula cozi de așteptare reale și diversele perioade de timp prin care trebuie să treacă fiecare client pană va fi servit. Dintr-un astfel de program putem extrage datele necesare unei analize riguroase, urmând să decidem ce se poate face pentru a optimiza și reduce timpul de stat la coadă.

Simulatorul de cozi ia în calcul cateva date introduse de utilizator printr-o interfață interactivă și intuitivă :

1. Câți clienți sunt prezenți
2. Câte cozi sunt disponibile
3. Durata simularii
4. Timpul minim de ajungere la coadă(când clientul termină ce mai are de facut și se pune la capătul unei cozi)
5. Timpul maxim de ajungere la coadă
6. Timpul minim de servire (odată ce clientul este primul din coadă)
7. Timpul maxim de servire

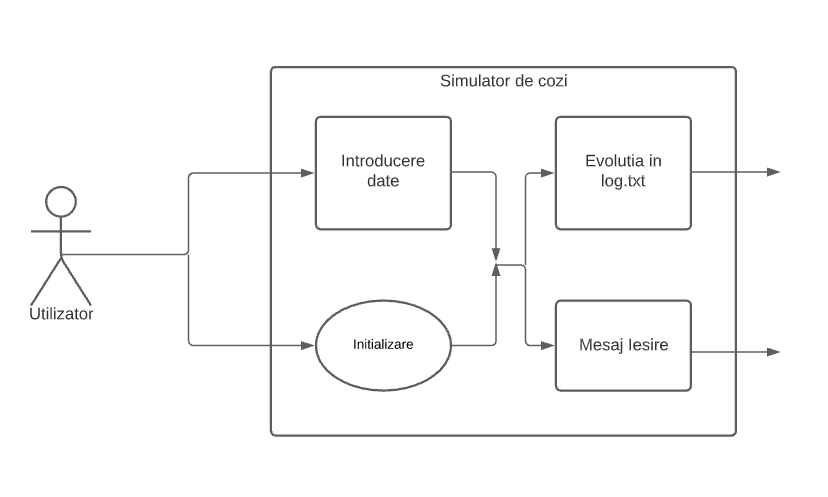
Datele acceptate sunt doar valorile numerice naturale mai mari strict ca zero.

E pus la dispoziție un buton de “Initialize Simulation” (Inițializează Simularea) și un camp de text needitabil etichetat “Output Message” (Mesaj de Ieșire) care afișează raspunsul programului la datele introduse.

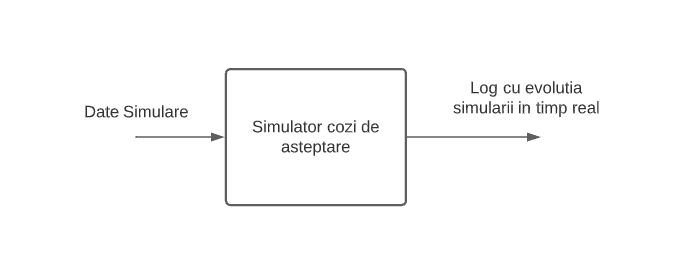
Interacțiunea cu simulatorul se rezumă la următorii pași:

1. Utilizatorul introduce de la tastatură datele necesare în campurile de text editabile (etichetate corespunzător)
2. Utilizatorul apasă butonul de “Initialize Simulation” și se afisează în campul de text “Output Message” un mesaj corezpunzător din care rezultă urmatoarele scenarii:
   1. Datele introduse nu sunt corecte în întregime, se afișeaza un mesaj ce menționează greșelile de introducere, simularea nu incepe
   2. Datele introduse sunt corecte în întregime, se afișează un mesaj de succes și incepe simularea.

Odată ce incepe simularea, eticheta “NOT RUNNING” (nu ruleaza) se va schimba în “RUNNING”(ruleaza) pentru a arăta că simularea e in curs de desfășurare.

Diagrama de mai jos descrie succint cazurile de utilizare:

# Proiectare

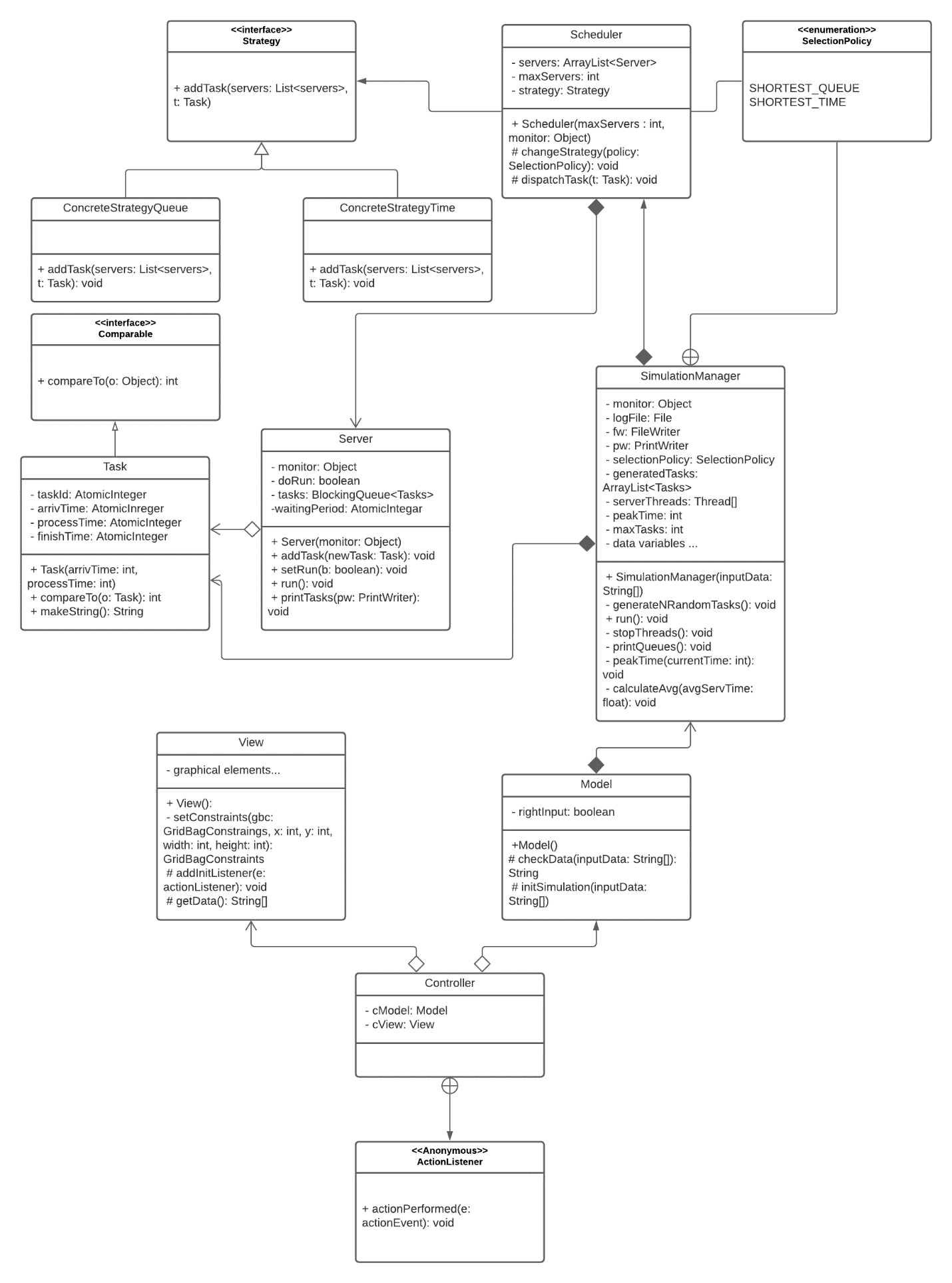
La cel mai înalt nivel de abstractizare avem un simulator ce primeste câteva date de intrare, afișează un răspuns pe baza corectitudinii lor și noteaza intr-un fisier “log.txt” evoluția în timp real a simularii, precum sugerează și diagrama atașată:

Coborând un nivel mai jos, organizarea internă se rezumă la interacțiuniile dintre patru pachete: gui, dataTypes, control și strategies.

* **gui –** conține clasele ce se ocupă de interacțiunea cu interfața grafică, structurat dupa un model arhitectural MVC (Model-View-Controller)
* **dataTypes** – cuprinde cozile(servers) și clienții (tasks) și datele corespunzătoare, celelalte pachete operează cu aceste date
* **control –** cuprinde clasa principală de simulare și organizatorul (en. “scheduler”) care determină întreg comportamentul simulării
* **strategies –** conține strategiile de distribuire în cozi

În cadrul pachetului gui avem o altă subdiviziune datorată arhitecturii MVC. Sumar explicat, Controller-ul gestionează datele primite în cadrul interfeței(clasa View) și le trimite mai departe catre Model pentru procesare. Același Controller primește rezultatele returnate de Model si actualizează corespunzător interfața (View).

In cadrul organizatorului (“Scheduler”) s-a folosit o arhitectură de tip “Strategy Pattern” ce perminte selectarea unui algoritm dintr-o listă dată la începutul execuției. In loc să folosim un singur algoritm direct, sunt oferite instrucțiuni cu privire la care dintre cele disponibile trebuie utilziat, putând fi schimbat oricand chiar și in mijlocul execuției(dacă programatorul asigură o bună desfașurare în cadrul unei astfel de schimbări). În cadrul proiectului s-au folosit două strategii, selectate de “Scheduler” : distribuirea după cel mai scurt timp și distribuirea după cea mai scurtă coadă. Implicit selectată este distribuirea dupa timp, aceasta putand fi schimbată din cod. Orice strategie implementează interfata “Strategy” ( cu o singura metodă “addTask”), iar lista strategiilor implementate se gasește în enumerarea “SelectionPolicy”.

Relațiile dintre clase sunt reprezentate de diagrama UML:

Un “SimulationManager” e compus dintr-un “Scheduler” și o lista de “Task”-uri ce sunt redirecționate către “Servere” (un server are mai multe “Task”-uri). Un “Scheduler” este compus din mai multe servere și contine un obiect ce implementează interfața “Strategy”. “SimulationManager” este la randul său parte componentă a “Model”-ului (acolo e initializat și rulat), iar modelul este deținut de un “Controller” care mai deține și un “View” și implementeazaă o clasă anonimă “ActionListener”.

Pentru a asigura o bună sincronizare a firelor de lucru, s-au folosit unde a fost nevoie variabile și structuri “thread-safe” precum “AtomicBoolean”, “AtomicInteger”, “BlockedList” etc. Toate cozile de așteptare sunt controlate de același cronometru al managerului, folosind funcția “notifyAll” pe monitorul “Monitor” (obiect declarat doar pentru acest scop) al cărui control îl preia după ce toate cozile au terminat operațiile de decrementare și adăugare. După “notifyAll” serverele(cozile) continuă operațiile și paseasă controlul monitorului până ajunge din nou la manager, care așteaptă o secundă până a iși continua execuția(și execuția tuturor celorlalte cozi de așteptare). S-au folosit blocurile sincronizate doar acolo unde a fost strict nevoie de ele(în rest ne-am bazat pe obiecte “Thread-Safe” și alte mecanisme de gestionare a firelor de lucru).

# Implementare

Pentru implementare s-au folosit urmatoarele clase:

1. Task (pachet – dataTypes)

Un task reprezintă un client, s-a păstrat denumirea de “task” pentru a sugera cadrul general al problemei. Sunt păstrate date despre ID, timp de procesare, de sosire și de terminare(când clientul părasește sistemul). Este implementată interfața “Comparable” cu metoda de sortare specifică, pe lângă alte metode auxiliare (“makeString”, “getters”, etc.).

1. Server (pachet – dataTypes)

Un server reprezintă o coadă, păstrându-se denumirea generală pentru același scop ca și la “task”. Se pastrează o referință la obiectul “monitor”(cu scopul sincronizarii), un obiect de tip “PrintWriter” pentru scrierea în fișierul log, o variabilă de control “doRun”, lista de “task”-uri corespunzătoare și timpul de așteptare curent.

Metodele principale sunt cele de adaugare a unui nou task și metoda “run”, preluată din interfața “Runnable” implementată. În “run” se ia pe rând câte un “task” din lista și se decrementează timpul de așteptare la fiecare secundă(timp determinat de clasa SimulationManager). De asemenea se printează în fișier.

3. Controller (pachet - gui)

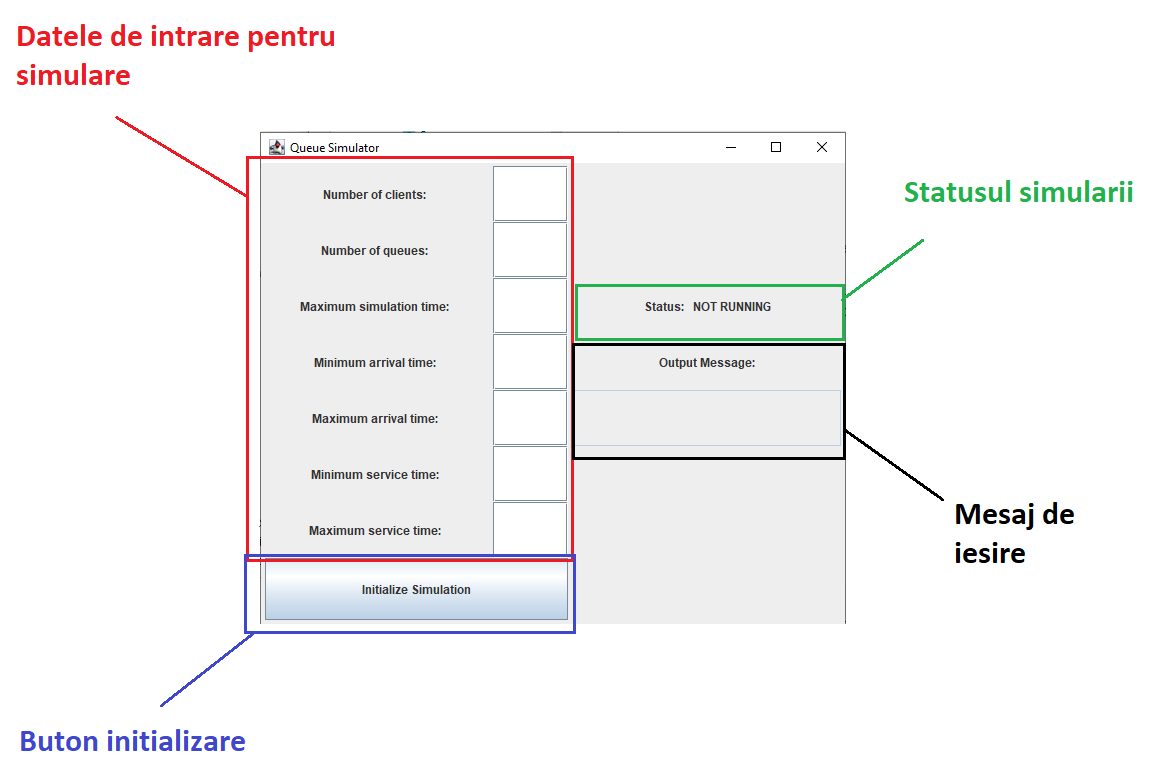
Scopul “Controller”-ului a fost deja specificat în cadrul proiectarii. Singura metodă prezentă e cea gestionează răspunsul apăsarii pe butonul de “Initialize Simulation”. Odata apăsat, se apelează metoda “initSimulation” din “Model” ce pornșeste simularea propriu-zisă. Variabilele prezente sunt auxiliare.

4. Model (pachet - gui)

In “Model” se verifica dacă datele de intrare sunt corecte și se returnează un răspuns corespunzător ce va fi ulterior afișat în gui. Dacă datele sunt corecte, e apelată metoda “initSimulation” ce se folosește de clasa SimulationManager pentru a porni simularea(pornește un nou thread). Variabilele prezente sunt auxiliare.

5. View (pachet – gui)

In “View” se pastrează elementele ce țin de interfața grafică și metodele ce preiau datele introduse în elementele respective, funcții ce vor fi apelate de “Controller”. În constructor sunt stabilite detalii privind aceste elemente(aliniere, format text, dimensiune etc.) precum și organizarea lor în panouri.



Sunt prezente șapte câmpuri de text editabile etichetate pentru datele de intrare ale simulării(chenarul rosu), un singur buton de inițialziare(albastru), o etichetă de status(verde) care se schimba în “RUNNING” când simularea rulează și un câmp de text needitabil(negru) etichetat “Output message” pentru mesajele de iesire ale sistemului (erori la introducere, date corecte etc.).

6. ConcreteStrategyTime (pachet - strategies)

Clasa reprezintă strategia principală a sistemului, componentă esențiala a “strategy pattern”-ului folosit pentru a distribui clienții în cozi. Singura metodă prezentă, “addTask” primește lista de cozi deschise și un singur client, determină care dintre cozi are cel mai mic timp de așteptare și adaugă clientul în lista acelei cozi. Astfel de fiecare dată cand un nou client trebuie distribuit, se va alatura cozii cu timpul de așteptare minim.

7. ConcreteStrategyQueue (pachet -strategies)

Clasa reprezintă o alta strategie a sistemului, componentă a “strategy pattern”-ului folosit pentru a distribui clienții in cozi. Aici, “addTask” primește lista de cozi deschise și un singur client, determină care dintre cozi are cei mai puțini clienti și adaugă clientul în lista acelei cozi. Astfel de fiecare dată cand un nou client trebuie distribuit, se va alătura cozii celei mai scurte.

8. SelectionPolicy (pachet -strategies)

Enumerare a celor doua tipuri de strategii posibile : “SHORTEST\_QUEUE” și “SHORTEST\_TIME”.

9. Strategy (pachet - strategies)

Interfață ce stă la baza oricarei strategii prezente sau ulterior implementate. Impune folosirea metodei “addTask”.

10. Scheduler (pachet – control)

Organizatorul(en “Scheduler”) se ocupă de gestiunea propriu-zisa a clientilor, apelând clasele din pachetul “strategies” în mod corespunzător. Dispune de lista de cozi a simularii, de numărul maxim de cozi și de strategia aleasă. Odata inițializat, organizatorul creează atâtea cozi câte indică numarul maxim și le adaugă listei, trimițând mai departe referințe la monitorul principal și la printer. Prima metodă importantă (“changeStrategy”) determină strategia folosită, urmând ca metoda “dispatchTask” (distribuie client) să apeleze funcția de adăugare a acelei strategii.

11. SimulationManager (pachet – control)

Principala clasă a programului, managerul de simulare pastrează toate datele introduse de utilizator cu privire la simulare, precum și un monitor principal, referință la fișierul în care se vor scrie rezultatele, printer-ul asociat, metoda de distribuire, organizatorul, lista de clienți și lista de “Thread”-uri a simularii.

Odata inițializat, managerul deschide fișierul de scriere și seteaza datele de intrare primite, urmând să genereze aleator clienți cu date in limitele specificațiilor date de utilizator. Este apoi inițializat organizatorul (“scheduler”) și se setează implicit strategia “SHORTEST\_TIME”. Folosind lista de cozi (“servers”) creată de organizator, se creează o listă de “Threaduri” pentru fiecare coadă în parte și se pornește fiecare “thread”.

Metoda “generateNRandomTasks” folosește datele primite pentru a genera clienți (“tasks”) și a ii sorta intr-o listă. Se setează astfel și “ID”-ul fiecărui client în lista sortată pentru o mai ușoară identificare.

Metoda “stopThreads” oprește toate firele de execuție asociatate(cozi) care altfel ar rula la infinit datorită modului în care au fost implementate. Cum managerul se oprește din executie odata ce se depașeste timpul maxim de simulare, e firesc ca și cozile(“Servers”) să se oprească din execuție odata cu el.

Metodele “peakTime” și “calculateAvg” au scopul de a extrage informații generale pe baza simulării: ora de vârf(când sunt cei mai mulți clienti la cozi), timpul mediu de procesare(servire) si timpul mediu de așteptare pentru fiecare client.

Metoda “printQueues” trece pe la fiecare coadă și apeleaza funcția “printTasks” pentru o afișare corectă și ordonată în “log.txt”.

# Rezultate

Testarea s-a realizat pe mai multe variabile, cu numarare și verificare manuală a datelor din “log” la nevoie, fiind folosite atât valori mici cât și altele foarte mari.

S-au rulat datele model din cerința proiectului. Conținuturile fișierelor “log” sunt prezente în același director în care se află și documentația.

# Concluzii

Putem folosi mediul digital pentru a simula scenarii din lumea reală și pentru a cauta moduri în care să le optimizăm. Reducem astfel analiza obositoare și manuală a evoluției unor cozi reale la câteva apăsări de butoane cu rezultate imediate pe baza datelor introduse. Se pot extrage astfel foarte ușor diverse date ce sunt mai apoi folosite de specialiști pentru a găsi soluții.

Din tema curentă am învățat importanța folosirii mai multor fire de lucru, precum și metode de gestionare și sincronizare a acestora. Am înțeles importanța executarii în paralel a programelor și modul în care se optimizează timpul de lucru. Am observat incă o data importanța folosirii tehnicilor de programare și rezultatul optimizat al acestora spre deosebire de cel al unei implementări naive.

Programul se poate extinde pentru a calcula și alte date: cum influențeaza numărul de clienți/ numărul de cozi timpul de așteptare, cât de bine aleg oamenii coada cu cel mai scurt timp de așteptare(de obicei o iau pe cea mai scurtă), care e timpul minim realist la care se poate ajunge etc. Cu mai multe variabile si funcționalități, putem ajunge foarte aproape de desfășurarea reală a situației unor cozi de așteptare.

# Bibliografie

1. <https://docs.oracle.com/en/java/> - documentație Java

2. <https://www.informit.com/articles/article.aspx?p=336264&seqNum=3> – pentru UML

3. Prezentarea suport și Cursurile de Tehnici de Programare

4. http://tutorials.jenkov.com - informații pentru lucrul cu fire de executie