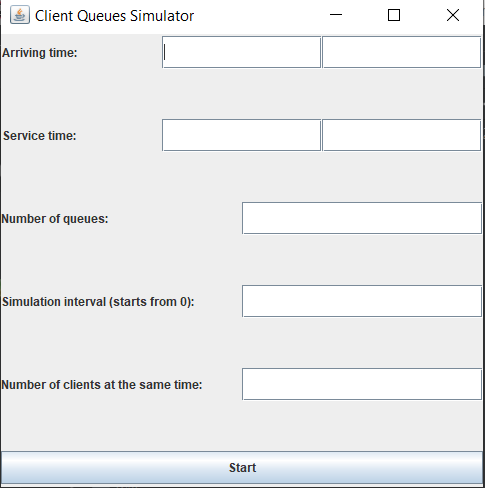
**Obiectiv principal**

Realizarea unei aplicatii capabila sa simuleze o activitate de tip client-server. Pe baza unor date initiale introduse de utilizator aplicatia simuleaza cum diversi client sosesc la momente diferite de timp, sunt asezati in cozile simbolice ale serverului urmand ca mai apoi sa fie serviti. La final aplicatia genereaza si cateva statistici legate de ce s-a intamplat pe parcursul simularii.

La deschidere, aplicatia interactioneaza cu utilizatorul prin intermediul unei interfete grafice simpliste prezentata in figura urmatoare:



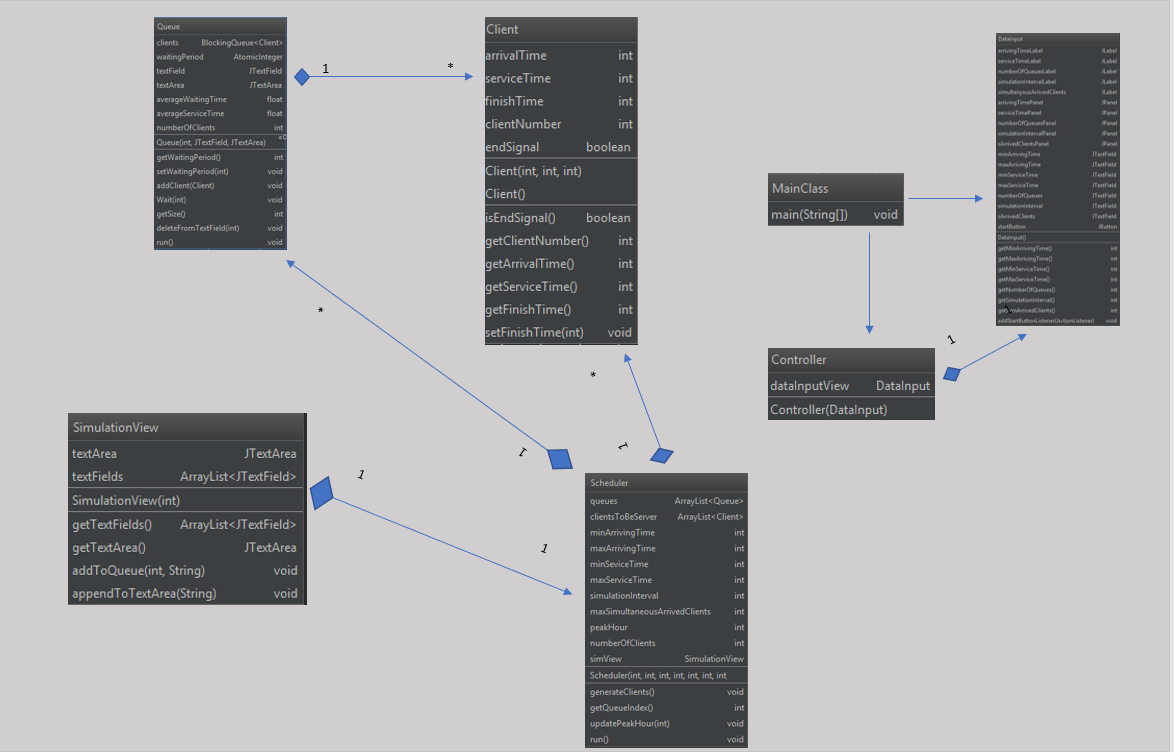
Interfata grafica constituita dintr-o fereastra ce contine:

* 2 campuri pentru introducerea timpului minim respectiv maxim de sosire intre client;
* 2 campuri pentru introducerea timpului minim respectiv maxim de servire a unui client;
* un camp destinat introducerii numarului de “cozi” pe care se va desfasura simularea;
* un camp pentru introducerea periodei de timp in care clientii pot sosi;
* un camp pentru introducerea numarului maxim de client care pot sosi in acelasi moment de timp;
* un buton de start a carui apasarea face ca aplicatia sa preia toate datele introduse de utilizator si pe baza lor sa inceapa simularea.

**Implementare**

Aplicatia este implementata cu ajutorul limbajului de programare Java, drept urmare aceasta respecta paradigmele programarii orientate pe obiect.

In continuare avem diagrame de clase UML corespunzatoare:



Identificam cele 7 clase principale ale aplicatiei care pot fi grupate in urmatoarele categorii:

* simularea propriu-zisa la nivel de program: Client, Queue, Scheduler;
* interfata grafica: Controller, DataInput, SimulationView;
* clasa principala: MainClass.

Identificam relatii de asociere si relatii de compozitie. Relatiile de compozitie marcheaza faptul ca o anumita clasa nu poate existe fara alta.

**Descrierea claselor**

* **Client**

Clasa care reprezinta la nivel simbolic un client din interiorul simularii.

Identificam 5 atribute la nivel de clasa(arrivalTime, serviceTime, finishTime, clientNumber, endSignal) care au urmatoarele semnificatii:

-arrivalTime: momentul de timp in care clientul respective a sosit;

-serviceTime: timpul necesar clientului sa fie servit odata ce ajunge la inceputul cozii;

-finishTime: momentul de timp la care clientul a parasite simularea;

-clientNumber: identificator numeric unic al clientului;

-endSignal: camp auxiliar care serveste doar in interiorul programului la oprirea thread-urilor ce reprezinta cozila la care clientii asteapta

Doi constructori, unul pentru a crea un client “normal” in functie de timpul de sosire, cel de servire si de indetificatorul unic si altul pentru a crea un client menit doar sa marcheze inchiderea cozilor de clienti.

Metode getters&setters pentru a putea lucra cu variabilele din interiorul clasei.

* **Queue**

Clasa care reprezinta la nivel symbolic o coada(se pot instantia mai multe) din cadrul simularii.

Identificam urmatoarele atribute:

-clients: atribut de tipul “BlockingQueue<Client>” care reprezinta o lista cu clientii dispusi la coada repsectiva. Tipul special al acestui atribut permite ca thread-ul sa nu se incheie in caz ca coada ramane goala la un moment ci sa astepte dupa sosirea altor clienti in coada.

-waitingPeriod: atribut de tipul “AtomicInteger” care reprezinta timpul de asteptare total al cozii respective, timp in care toti clientii aflati in coada urmeaza sa fie serviti. Acest atribut este folosit pentru a determina in ce coada sa plasam un client nou venit.

-textField si textArea: atribute care fac legatura cu interfata grafica pentru a permite cozilor sa-si afiseze in timp real continutul si alte statistici.

-averageWaitingTime, averageServiceTime, numberOfClients: attribute folosite pentru generarea de statistici individuale pentru fiecare coada

Constructor pentru instantierea unei noi cozi pe baza capacitatii maxime dar si pe baza celor 2 campuri textArea si textField pentru a se face legatura dintre coada curenta si reprezentantul ei din interfata grafica.

Metoda de tipul getter&setter pentru campul waitingPeriod.

Metode addClient care permite clasei Scheduler sa adauge client intr-o anumita coada.

Metoda getSize care returneaza numarul din clienti dintr-o coada la un moment dat.

Metoda deleteFromTextField care simuleaza in interfata grafica plecarea unui client.

Metoda implicita run care permite clasei Queue sa fie executata ca un thread, astfel putem avea mai multe cozi care lucreaza in paralel.

* **Scheduler**

Clasa ce se ocupa cu generarea de clienti si simularea sosirii lor la intervale diferite de timp iar mai apoi distribuirea acestora in coada cu cel mai mic timp de asteptare.

Identificam urmatoarele atribute:

-queue: variabila de tip ArrayList<Queue> care pastreaza referinta fiecarei cozi din simularea noastra pentru a putea adauga clienti in ele.

-clientsToBeServed: variabila de tip ArrayList<Client> care o lista cu clientii ce trebuie serviti in timpul simularii, list ace este parcursa la fiecare moment de timp pentru a vedea ce clienti sosesc in respectivul moment.

-minArrivingTime, maxArrivingTime, minServiceTime, maxServiceTime, simulationInterval, maxSimultaneousArrivedClients: variabilele in care se retin datele introduse de utilizator. Acestea reprezinta paramaetrii in care se desfasoara simularea.

-peakHour, numberOfClients: variabila in care se retin date pentru statistici(ora de varf respective numarul total de clienti serviti).

-simView: variabila de tip SimulationView care leaga clasa Scheduler de interfata grafica unde se desfasoara simularea.

Constructor pentru instantierea clasei pe baza tuturor datelor introduse de utilizator dar si pe baza referentei la variabila instanta a clase SimulationView.

Metoda generateClients care genereaza un numar random de clienti in urmatorul mod: pentru fiecare moment de timp de la 0 la valoare retinuta in variabila simulationInterval se genereaza un numar de clienti care sosesc in acel moment(numar cuprins intre 0 si maxSimultaneousArrivedClients) apoi fiecare client primeste un serviceTime generat random si ID-ul sau. Momentele de timp la care pot aparea clienti nu sunt neaparat consecutive, acestea sunt ales pe baza unui numar random generat in intervalaul [minArrivingTime; maxArrivingTime]

Metoda getQueueIndex care ne returneaza indexul acelei cozi care are timpul de asteptare cel mai mic la momentul cand este apelata metoda pentru a sti unde vrem sa adaugam urmatorul client.

Metoda updatePeakHour care in funtie de numarul de clienti din fiecare coada stabileste care a fost “ora de varf” a simualrii.

Metoda run care executa urmatoarele operatii: apeleaza metoda generateClients pentru generarea de clienti. Pentru fiecare moment de timp parcurge lista de clienti generate ca sa verifice ce clienti au sosit in acel moment. Pentru fiecare client sosit se cauta coada in care trebuie adaugat si se adauga. In acelasi timp se face si un update in interfata grafica unde sunt trecute date despre clientul nou venit. Se asteapta o secunda apoi se scade din fiecare coada o secunda din timpul total de asteptare pentru a simula trecerea unui moment de timp. La finalul timpului de simulare metoda run trimite o instanta speciala a clasei client in cozi care nu este contorizata la nivel de statistici dar care are rolul de a marca sfarsitul cozii respective,.

* **DataInput**

Clasa care serveste ca interfata grafica initiala unde utilizatorul is va introduce datele de simulare.

Toate atributele din clasa reprezinta elementele constitutive ale interfetei grafice respectiv 5 label-uri de text si 7 textField-uri pentru introducerea datelor plus panel-uri auxiliare care servesc la ordonarea corecta a interfetei si in final butonul de start al simularii.

Constructorul care instantiaza clasa respective pentru ca utilizatorul sa poata vedea zona de introducere a datelor si care totodata creaza toata legaturile specifice intre atributele clasei si elementele grafice.

Metode de tip getter care de aceasta data preiau input-ul utilizatorului din fiecare textField unde acesta a introdus un alt parametru al simularii.

Metoda addStartButtonListener care adauga functionalitatea descrisa de noi butonului de start.

* **SimulationView**

Clasa care serveste ca interfata grafica principala pentru simularea noastra si care apare doar dupa introducerea informatilor despre simulare si apasarea butonului de start.

Atributul textArea de tip JTextArea in care este trecut tot log-ul simularii si care este transmis mai departe atat clasei Schduler cat si fiecarei instante a clasei Queue.

Lista de atrbitue de tip JTextField, textFields care retine corespundentul din interfata grafica al fiecarei cozi si reprezinta locul unde aceasta isi afiseaza continutul.

Constructorul care asigura initializarea clasei si genereaza un numar de textField-uri egal cu numarul de cozi pentru simulare respectiva, numar introdus de catre utilizator.

Metoda getTextArea care ofera si celorlalte clase posibilitatea sa scrie in log-ul principal al simularii atat informatii despre clienti cat si statisticile de la final de simulare.

Metoda addToQueue care simuleaza operatia de adaugare a unui client intr-o coada pentru ca aceasta sa fie vizibila utilizatorului in interfata grafica.

Metoda appendToTextArea care printeaza in log-ul principal al interfetei noastre grafice.

* **Controller**

Clasa care face legatura intre program si interfata grafica.

Atributul dataInputView care retine referinta la interfata grafica responsabila cu simularea principala.

Constructorul care intializeaza interfata grafica si adauga butonului de start actiunea pe care noi dorim sa o execute.

Subclasa startButtonListener care citeste din interfata dataInputView datele introduse de utilizator apoi face interfata de introducere invisible initializand totodata interfata de simulare cu datele primite si proneste treadul corespunzator clasei Sceduler.

* **MainClass**

Clasa care porneste tot programul principal prin intermediul clasei Controller prin intializarea unei variabile de tipul Controller care primeste ca paramtru o noua instanta a clasei DataInput.

**Mod de functionare**

Aplicatia porneste si afiseaza interfata de introducere a datelor. Utilizatorul introduce datele minime necesare simularii si apasa apoi butonul de start. Daca datele introduse nu sunt corecte se va arunca o exceptie iar aplicatia nu isi va continua rularea.

Daca datele introduse sunt corecte acestea sunt citite prin intermediul clasei Controller apoi trimise mai departe atat la interfata grafica de simulare cat si la clasele Scheduler si Queue.

Clasa Scheduler isi incepe excutia sub forma unui nou thread care da start la randul lui unui numar de thread-uri egal cu numarul de cozi pe care dorim sa-l simulam. Apoi acesta genereaza in mod aleator o lista de clienti care sosesc la momente diferite de timp si care au timp de servire generate tot in mod aleator. La fiecare moment de timp de la momentul 0 pana la momentul “simulationInterval” se parcurge lista de clienti generats anterior pentru a vedea ce clienti au sosit la momentul curent. Se parcurge mai apoi lista de cozi si se alege indexul cozii cu perioada de asteptare cea mai mica. Plasam clientul current in coada cu indicele calculat anterior si repetam procesul pentru fiecare client sosit deoarece putem avea mai multi clienti care sosesc in acelasi moment de timp. Punem thread-ul sa astepte o secunda pentru a simula trecerea unei unitati de timp. Totodata, la fiecare pas, scadem unu din perioada de asteptare a fiecarei cozi care are aceasta perioada mai mare ca zero pentru a simula si in cozi trecerea unei perioade de timp. La finalul timpului de simulare se trimite catre fiecare coada un client cu o caracteristica speciala care marcheaza oprirea thread-ului pentru coada respective. Acest ultim client nu este luat in considerare in cadrul statisticilor generate. Mentionam ca pentru fiecare actiune descrisa mai sus clasa va afisa in log-ul din cadrul interfetei de simulare informatii utile pentru utilizator.

Datorita variabilei de tip BlockingQueue din cadrul lor instantele claselor Queue care functioneaza in acest moment ca niste thread-uri vor fi intr-o stare de asteptare pana cand primesc primul client. In continuare lista de clienti ce trebuie serviti este parcursa si pentru fiecare client in parte, pe baza timpului de asteptare total al cozii si pe baza timpului sau de servire si sosire, se calculeaza timpul in care acesta va parasi coada. Mai apoi se pune thread-ul pe sleep pentru un numar de secunde egal cu timpul de servire al clientului respective tocmai pentru a se simula servirea acestuia. Tot pentru fiecare client servit se actualizeaza statisticile ce urmeaza sa fie calculate la final. In cazul in care o coada ramane goala la un moment dat iar simularea nu s-a terminat inca, aceasta va astepta in continuare sosirea altui client sau sosirea clientului ce detine caracteristica speciala ce marcheaza finalul excutiei pentru thread-ul respectiv. La fel ca in cazul clasei Scheduler, la fiecare pas se vor afisa in interfata grafica informatii utile pentru utilizator si totodata se va actualiza la nivel vizual continutul fiecarei cozi. La final, fiecare coada va afisa statistici propri despre timpul mediu de servire si timpul mediu de asteptare din cadrul ei.

**Concluzii**

Aplicatia dezvoltata satisface functia de simulator de activitati de tip client-serer pentru un numar aleator de clienti, sositi fiecare la intervale aleatoare de timp si cu timpi de servire alesi de asemenea aleator.

Cerinta de distribuire a acestora pe mai multe cozi aduce, la nivel de programator, o mai buna intelegere a lucrului cu thread-uri si asupra metodelor de sincronizare a acestora.