**QUEUES**

**SIMULATOR**

**IRIMIA LAURA-MARIA**

**GRUPA 302210**

**PROFESOR LABORATOR: MITREA DAN**

Cuprins

[Obiectivul temei 3](#_Toc68593871)

[Obiectivul principal 3](#_Toc68593872)

[Obiective secundare 3](#_Toc68593873)

[Analiza problemei, modelare, scenarii, cazuri de utilizare 3](#_Toc68593874)

[Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfața utilizator) 4](#_Toc68593875)

[Diagrama UML 5](#_Toc68593876)

[Interfața 6](#_Toc68593877)

[Implementare 7](#_Toc68593878)

[Clasa „Client” 7](#_Toc68593879)

[Clasa „Queue” 7](#_Toc68593880)

[Clasa „Strategy” 9](#_Toc68593881)

[Clasa „Scheduler” 9](#_Toc68593882)

[Clasa „SimulationManager” 10](#_Toc68593883)

[Clasa „View” 13](#_Toc68593884)

[Clasa „Controller” 14](#_Toc68593885)

[Clasa „MVC” 14](#_Toc68593886)

[Rezultate (testare) 15](#_Toc68593887)

[Testul pentru 4 clienți 15](#_Toc68593888)

[Testul pentru 50 de clienți 16](#_Toc68593889)

[Testul pentru 1000 de clienți 17](#_Toc68593890)

[Concluzii 18](#_Toc68593891)

[Bibliografie 18](#_Toc68593892)

# Obiectivul temei

## Obiectivul principal

Obiectivul principal al acestei teme este proiectarea și implementarea unei aplicații de simulare cu scopul de a analiza sistemele care se bazează pe cozi pentru a determina și minimiza timpul de așteptare al clienților.

## Obiective secundare

|  |  |
| --- | --- |
| **Obiectiv secundar** | **Descriere** |
| Dezvoltarea de use case-uri și scenarii | Căutarea de scenarii în care aplicația dezvoltată o să fie folosită și de către cine va fi folosită. |
| Alegerea structurilor de date | Proiectarea unor structuri eficiente și potrivite cerințelor date. |
| Împărțirea pe clase | Alegerea unor clase care vor ușura proiectarea problemei. |
| Dezvoltarea algoritmilor | Implementarea algoritmilor necesari rezolvării problemei și rezolvării cazurilor speciale. |
| Implementarea soluției | Combinarea tuturor algoritmilor implementați in vederea rezolvării problemei. |
| Testare | Găsirea cazurilor care pot crea probleme aplicației dezvoltate și testarea acestora. |

# Analiza problemei, modelare, scenarii, cazuri de utilizare

Cozile sunt foarte des întâlnite în viața de zi cu zi. Principalul obiectiv al unei cozi este de a furniza un loc de așteptare pentru în client cu scopul obținerii unui serviciu. Administrarea sistemelor bazate pe cozi este focusată pe minimizarea timpului pe care clienții lor îl petrec la coadă până să fie serviți. Un mod de a minimiza timpul de așteptare este de a adăuga mai mulți operatori care să servească clienții, respectiv mai multe cozi, însă această abordare crește costul serviciilor oferite.

Aplicația pe care o avem de implementat trebuie să simuleze, prin definirea unui timp maxim de simulare, o serie de N clienți care solicită servicii, pentru care sunt disponibile Q cozi, care așteaptă, sunt serviți și apoi părăsesc cozile. Fiecare client este generat la începutul simulării și este caracterizat de trei parametri: un ID (un număr între 1 și N), timpul de sosire (timpul de simulare când clienții sunt gata să se așeze la cozi, adică momentul când clientul a terminat de pus în coș tot ce avea de cumpărat și se duce la o coadă de la una din casele de marcat) și timpul de servire (intervalul de timp care este necesar servirii clientului, adică timpul necesar scanării și plătirii tuturor articolelor cumpărate de client). Aplicația gestionează timpul petrecut de fiecare client la coadă și generează un timp mediu de așteptare. Fiecare client este adăugat la coadă cu cel mai mic timp de așteptare atunci când timpul de simulare este egal cu timpul lui de sosire.

Ca și date de intrare care trebuie inserate de către utilizator în interfața grafică avem următoarele:

* Numărul de clienți (N);
* Numărul de cozi (Q);
* Intervalul de simulare;
* Timpul de sosire minim și maxim;
* Timpul de servire minim și maxim.

Folosindu-ne de datele de intrare, vom genera aleator N clienți, cu ID-ul începând de la 1 până la N inclusiv, cu timpul de sosire un număr aleator între timpul minim de sosire și timpul maxim de sosire și cu timpul de servire un număr aleator între timpul minim de servire și timpul maxim de servire.

Vom porni un număr de Q (numărul de cozi) thread-uri în paralel care vor procesa clienții. Un alt thread va fi pornit pentru a susține pașii de simulare și pentru a distribui fiecare client la coada cu cel mai mic timp de așteptare când timpul de simulare este egal cu timpul de sosire al clientului respectiv.

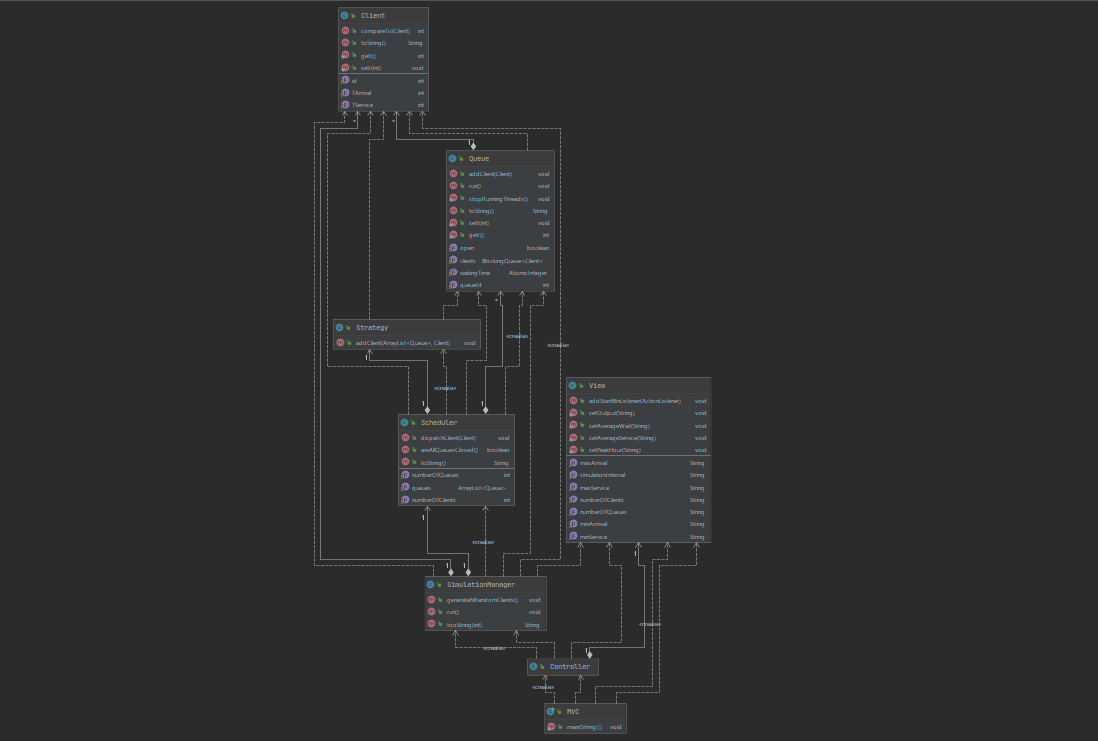
# Proiectare (decizii de proiectare, diagrame UML, structuri de date, proiectare clase, interfețe, relații, packages, algoritmi, interfața utilizator)

|  |  |
| --- | --- |
| **Package** | **Class** |
| main -> **java** | MVC |
| main -> java -> **model** | Client  Queue  Strategy  Scheduler  SimulationManager |
| main -> java -> **controller** | Controller |
| main -> java -> **view** | View |

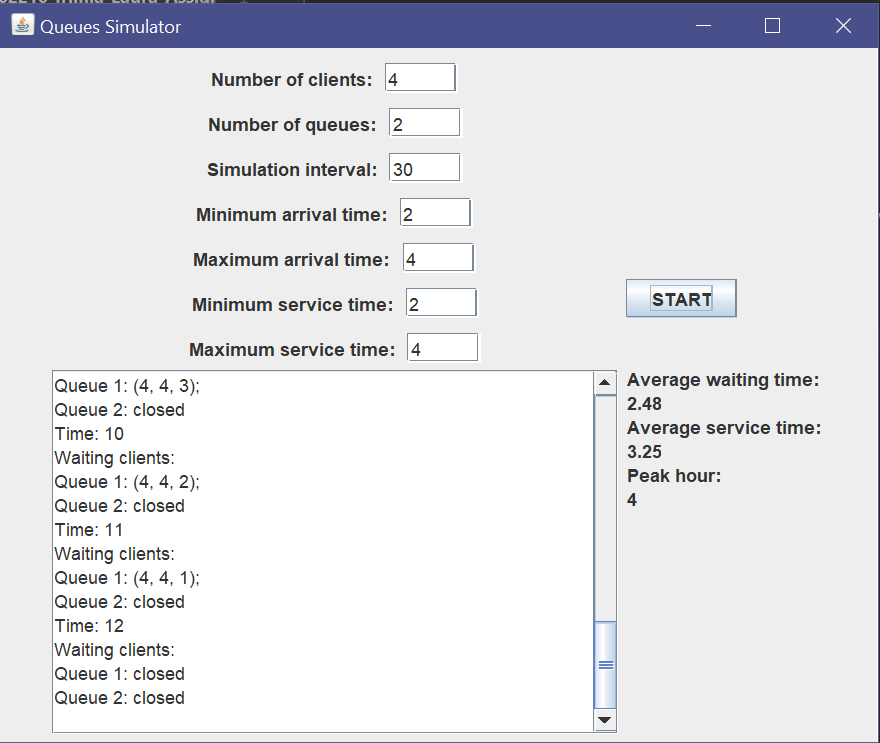
|  |  |
| --- | --- |
| **Class** | **Descriere** |
| Client | Clasa are trei atribute principale, respectiv ID-ul, timpul de sosire și timpul de servire, un constructor și metodele corespunzătoare de set și get. Pe lângă metodele de bază am mai adăugat suprascrierea metodei compareTo din interfața Comparable și suprascrierea metodei toString. |
| Queue | Clasa are patru atribute principale, respectiv lista de clienți, timpul de așteptare, ID-ul corespunzător și o variabilă care să ne arate dacă este deschisă coada sau nu, un constructor și metodele standard de set și get. Pe lângă metodele de bază am adăugat o metodă pentru a adăuga clienții la coadă, o metodă care suprascrie metoda de run din interfața Comparable, o metodă care mă ajută să opresc thread-urile și metoda suprascrisă toString. |
| Strategy | Clasa nu are niciun atribut, dar are o singură metodă foarte importantă, metoda cu ajutorul căreia adaug clienții la coada cu cel mai mic timp de așteptare. |
| Scheduler | Clasa are patru atribute principale, numărul de clienți, numărul de cozi, lista de cozi aferentă și un obiect de tipul clasei Strategy. Pe lângă metodele de bază de set și get și constructorul, am adăugat o metodă care apelează metoda de adăugare din clasa Strategy, o metodă care verifică dacă toate cozile sunt goale și metoda suprascrisă toString. |
| SimulationManager | Clasa are ca atribute toate datele de intrare din cerința problemei plus câteva atribute care să mă ajute la implementare: timpul mediu de așteptare, timpul mediu de servire, ora de vârf, un obiect de tipul Scheduler și lista de clienți generată aleator. Pe lângă metodele de bază și constructor, am adăugat metoda care generează clienții aleator, metoda run suprascrisă din interfața Comparable și metoda suprascrisă toString. |
| View | Clasa are ca atribute toate componentele de la nivel de interfață care necesită manipularea ulterioară. Conține constructorul interfeței, metode care returnează textul din TextField-uri, metode care setează rezultatul în eticheta corespunzătoare și în TextArea și metoda care adaugă ascultător pentru butonul de start. |
| Controller | Clasa care leagă clasele din pachetul model cu clasa View. Are un singur atribut, și anume un obiect de tip View. În cadrul constructorului inițializez atributul de tip View și asociez metoda ascultătoare pentru butonul de start cu o clasă anonimă. |
| MVC | Clasa care conține metoda main. Aici creez un obiect de tip View pe care îl pasez constructorului Controller-ului. |

Proiectarea claselor în acest mod m-a ajutat să înțeleg mai bine legăturile dintre clase și să dezvolt algoritmii necesari în clase corespunzătoare. În pachetul „model” am pus clasele „Client”, „Queue”, „Scheduler”, „SimulationManager” și „Strategy” pentru că aceste clase se ocupă de „calculele” aplicației dezvoltate. În pachetul „view” se află clasa „View” cu implementarea interfeței și metodele necesare, iar în pachetul „controller” am doar clasa „Controller” care leagă „calculele” aplicației cu interfața.

Diagrama UML**:**



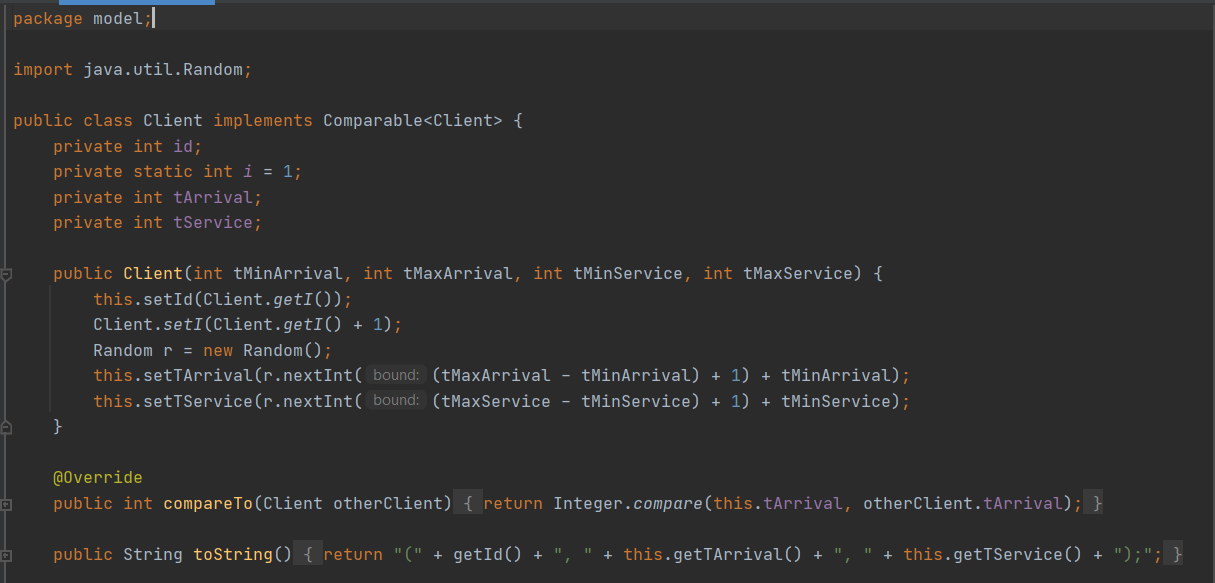
Interfața**:**



# Implementare

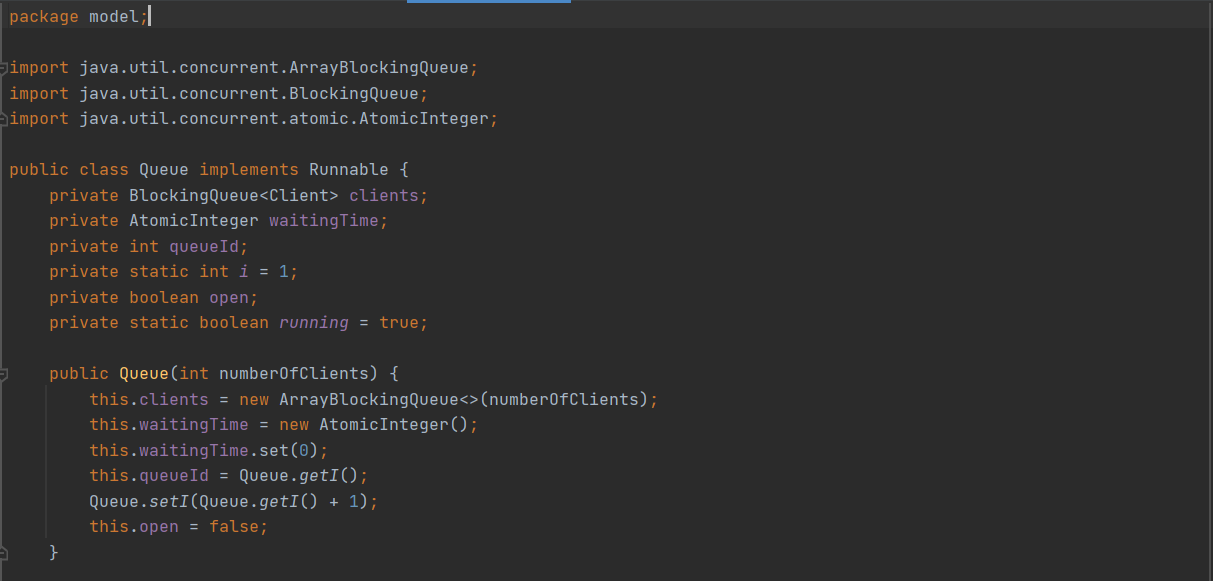
Clasa „Client”**:**

Clasa „Client” are cele trei atribute principale care descriu un client și încă un atribut static care mă ajută să setez ID-ul corespunzător pentru fiecare client creat. În constructor îmi generez un timp de sosire aleator între timpul minim de sosire și timpul maxim de sosire și un timp de servire aleator între timpul minim de servire și timpul maxim de servire cu ajutorul metodei „Random”. Metoda „compareTo” am suprascris-o pentru a-mi ordona clienții crescător după timpul de sosire, iar metoda „toString” am suprascris-o pentru o afișare mai frumoasă și mai ușor de înțeles a rezultatelor generate.

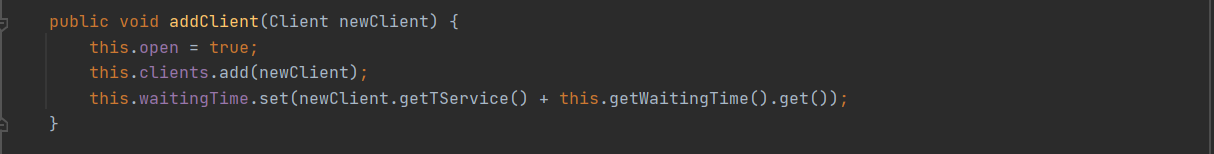


Clasa „Queue”**:**

Clasa „Queue” reprezintă practic thread-ul corespunzător unei liste de clienți și are patru atribute principale, lista de clienți, timpul de așteptare pentru fiecare coadă, ID-ul cozii (de la 1 la numărul de cozi, Q) și o variabilă de tip boolean care îmi spune dacă respectiva coadă este deschisă sau nu. Restul atributelor le folosesc pentru a seta corect ID-ul și pentru a opri thread-urile pornite din „SimulationManager”. În constructorul clasei asociez fiecare atribut principal cu valoarea necesară.



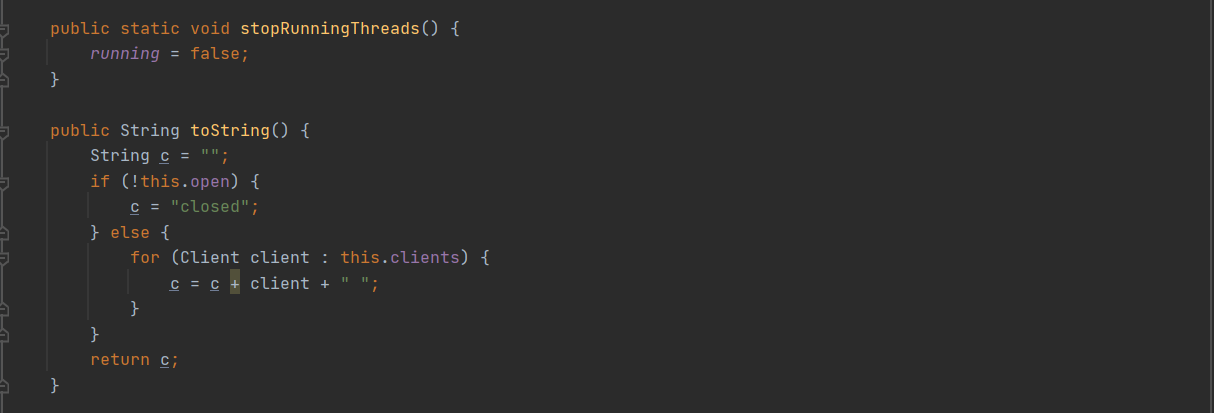
Metoda „addClient” adaugă un client în lista de clienți a unei cozi prin setarea cozii ca fiind deschisă, prin adăugarea lui în listă și prin incrementarea timpului de așteptarea a cozii cu timpul de servire a clientului adăugat.



Metoda suprascrisă de „run” reprezintă funcționalitatea unui thread reprezentativ unei cozi la care clienții vor să intre și să aștepte. Variabila „running” o setez pe „false” în metoda de „run” din clasa „SimulationManager”. Atâta timp cât mai avem clienți în lista de clienți, luăm timpul de servire al primului client și punem thread-ul să doarmă 1 secundă, fapt care ne ajută la sincronizarea thread-urilor fără a folosi numele „synchronized”. După ce thread-ul se trezește, decrementăm timpul de așteptare al cozii, decrementăm și timpul de servire al clientului la care am ajuns și îi și setăm timpul de servire cu cel decrementat. Dacă timpul de servire al unui client este zero îl scoatem din lista de clienți pentru că nu are rost să-l mai afișăm. Dacă lista de clienți este goală, setăm timpul de așteptare corespunzător la 0 și spunem că lista este închisă.

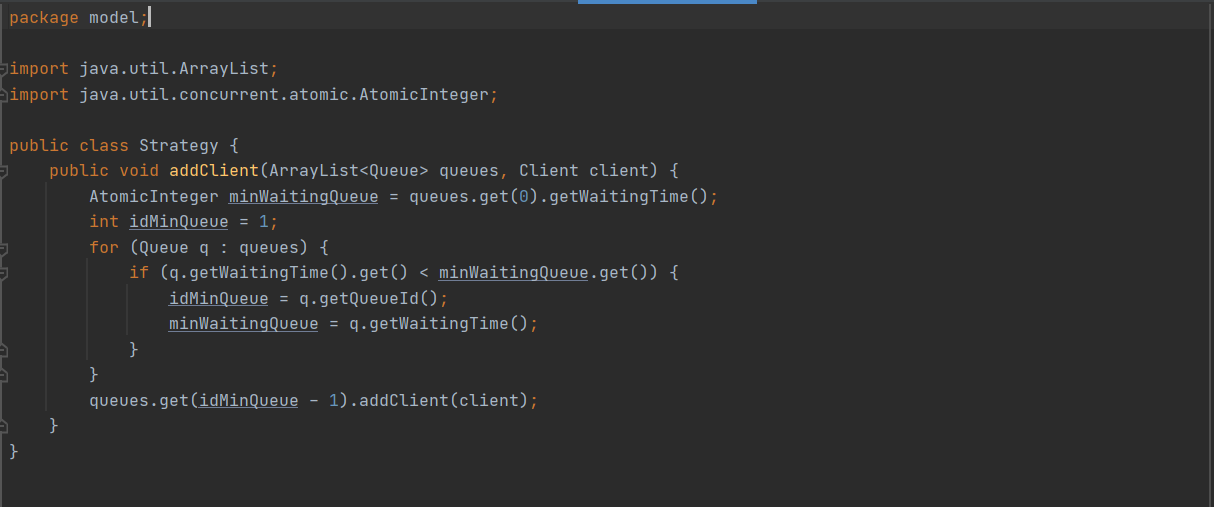


Metoda „stopRunningThreads” este apelată în metoda „run” din clasa de simulare și mă ajută să opresc thread-urile tuturor cozilor pornite atunci când se termină simularea.

Metoda suprascrisă „toString” mă ajută să afișez corespunzător starea unei cozi, respectiv dacă este închisă sau dacă are clienți la coadă pe care îi și afișez.

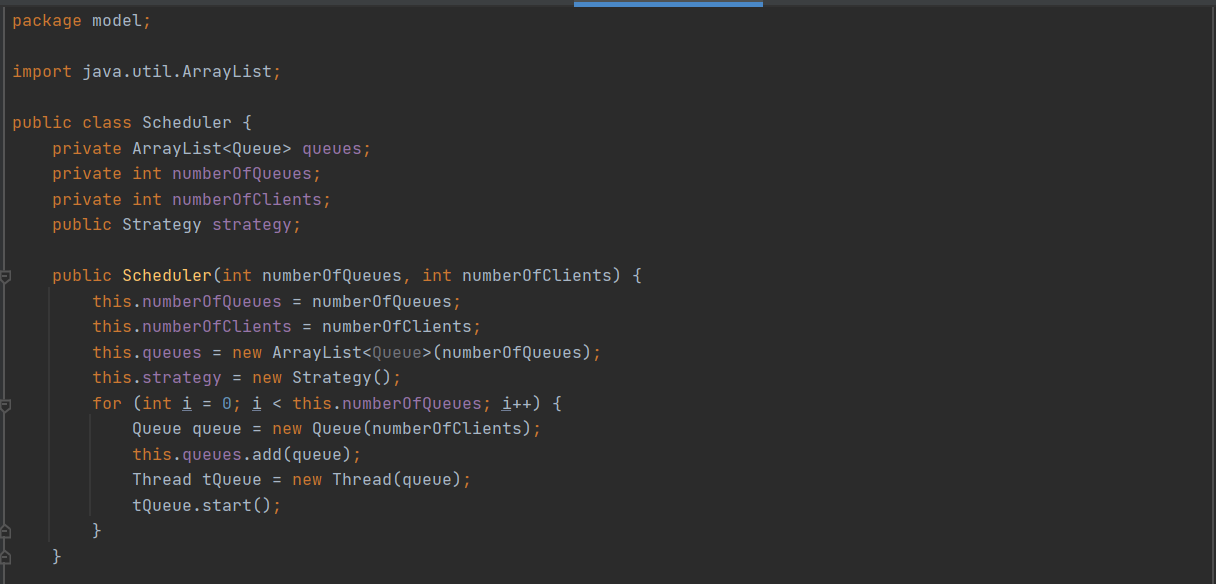
Clasa „Strategy”**:**

Clasa „Strategy” are o singură metodă care reprezintă metoda cu care adaug un client la o listă de cozi. Îmi iau o variabilă care reprezintă timpul de așteptare și ID-ul pentru prima coadă, parcurg restul cozilor și dacă găsesc una cu un timp de așteptare mai mic, salvez ID-ul ei și timpul de așteptare în variabilele de mai sus. Apoi, după parcurgerea tuturor cozilor, adaug clientul în coada găsită cu cel mai mic timp de așteptare.

****

Clasa „Scheduler”**:**

Clasa „Scheduler” are patru atribute principale, lista de cozi, numărul de clienți ca dată de intrare, numărul de cozi ca dată de intrare și o legătură către clasa „Strategy”. În constructorul acestei clase fac legătura atributelor cu valorile necesare și apoi pentru numărul de cozi dat de utilizator, creez coada, o adaug la lista de cozi de mai sus și pornesc și thread-ul corespunzător acelei cozi.

****

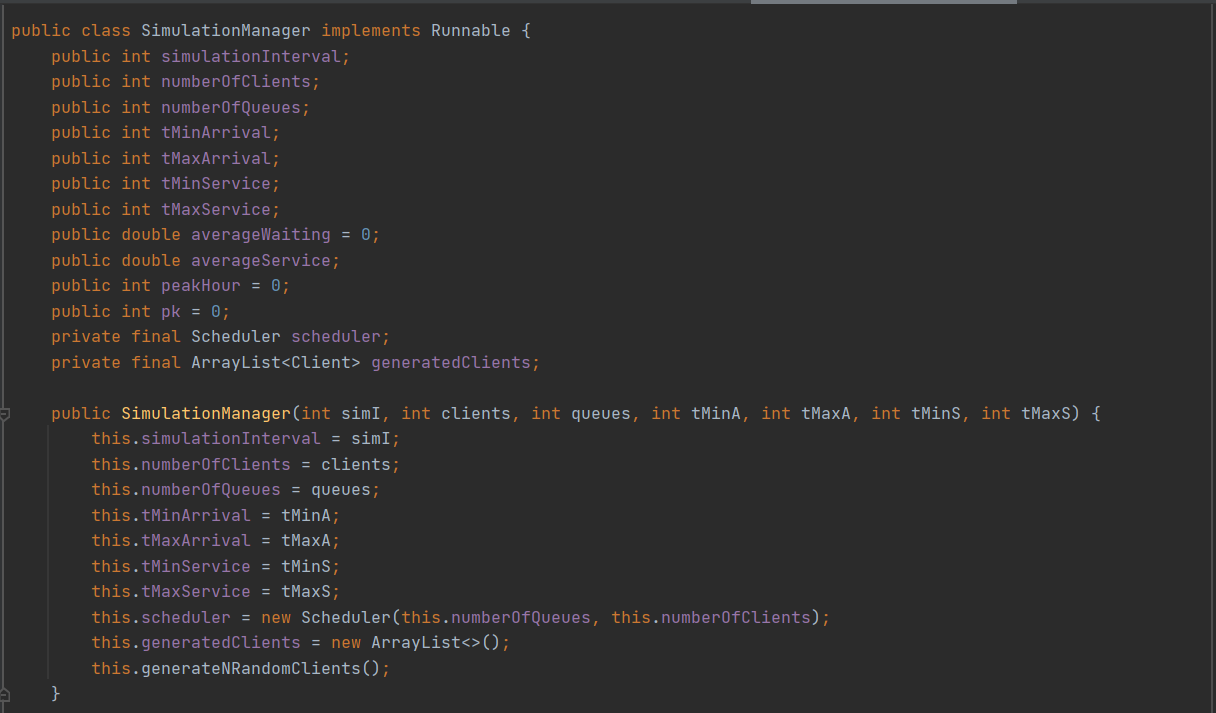
Metoda „dispatchClient” este apelată în momentul simulării. Această metodă apelează metoda de adăugare client din clasa „Strategy” pentru un anumit client și pentru lista de cozi din clasa în care suntem acum, respectiv „Scheduler”.

Metoda „areAllQueuesClosed” verifică dacă toate cozile din listă sunt închise și returnează un boolean „true” dacă este așa. Și această metodă mă ajută la simulare pentru a opri simularea dacă toate cozile sunt închise și nu mai sunt nici clienți care trebuie să ajungă la cozi.

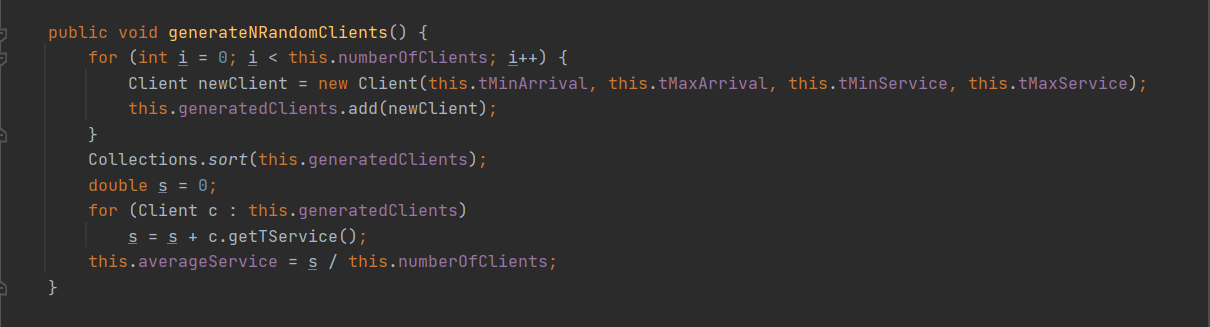
Metoda „toString” este suprascrisă pentru afișarea corespunzătoare a tuturor cozilor pornite de simulare.

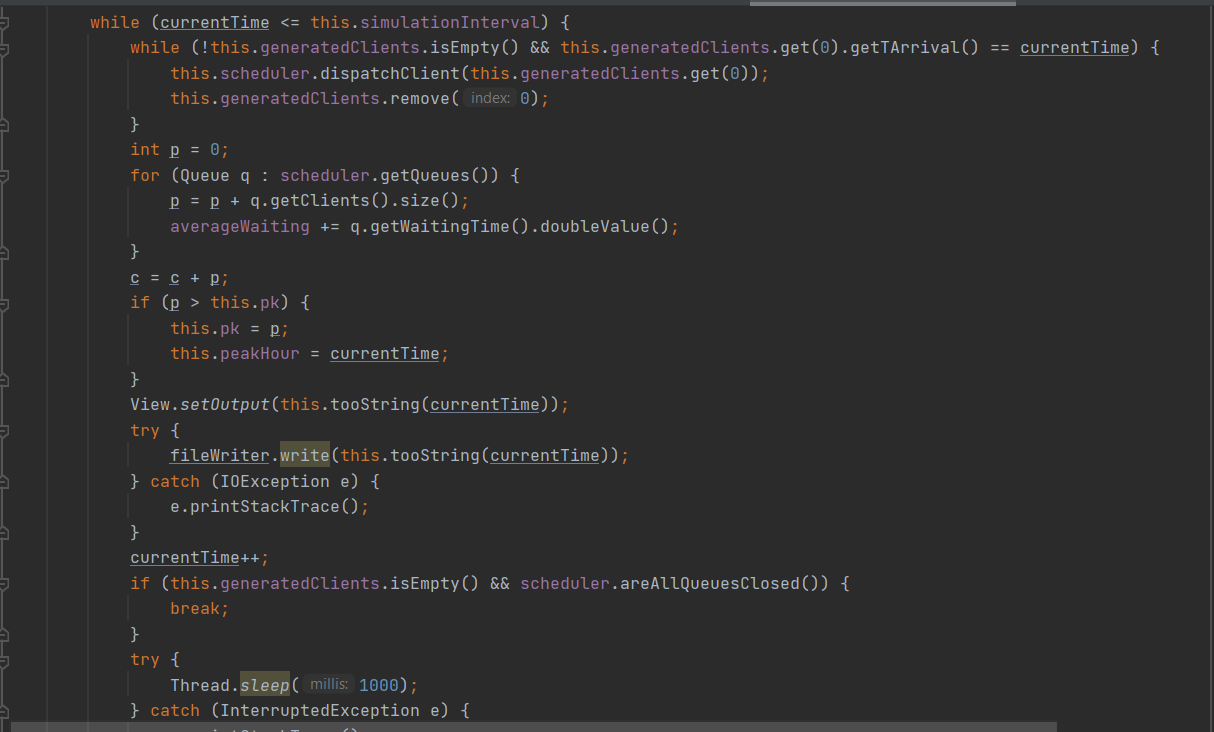
****

Clasa „SimulationManager”**:**

Clasa „SimulationManager” are ca atribute toate datele de intrare ale aplicație și încă câteva atribute importante pentru implementarea mea: timpul mediu de așteptare, timpul mediu de servire, ora de vârf, o instanță a clasei „Scheduler” și o listă de clienți care vor să intre la cozi generată aleator. În constructor setez fiecare atribut cu valorile corespunzătoare și apelez metoda care generează și setează lista de clienți.****

Metoda „generateNRandomClients” este metoda care generează o listă de N clienți. Pentru numărul de clienți N, îmi creez un nou client care este generat aleator cu ajutorul constructorului din clasa „Client” pe care îl adaug în lista de clienți generați. După ce mi-am generat toți clienții, îi sortez cu ajutorul metodei „sort” din clasa „Collections” care se folosește de metoda „compareTo” care este suprascrisă în clasa „Client”. Tot în această metodă calculez timpul mediu de servire ca fiind suma dintre timpii de servire a tuturor clienților împărțită la numărul de clienți.

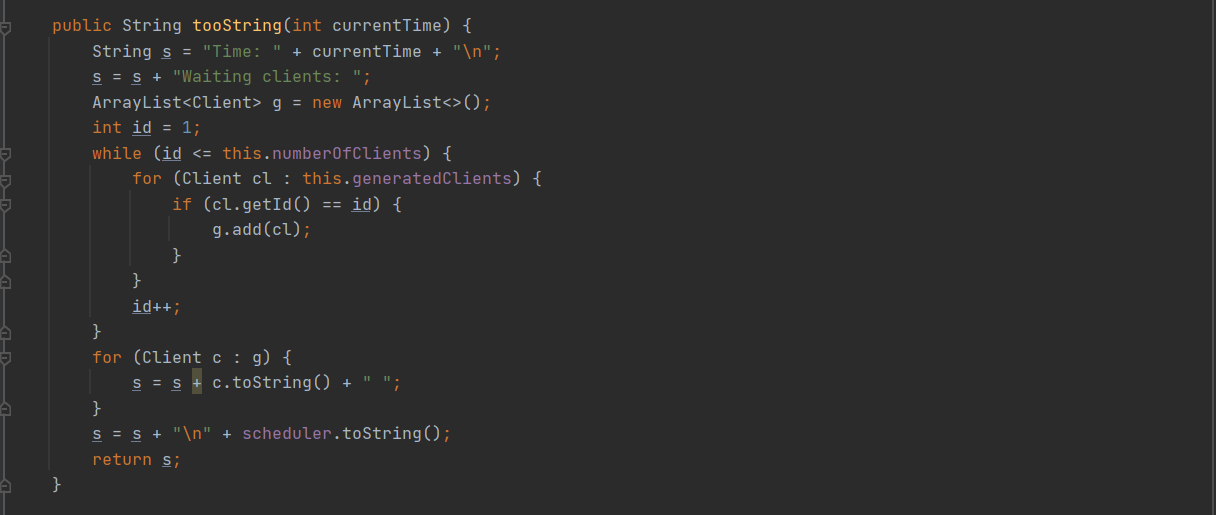
****

Metoda „run” este cuprinsă mai jos. Nu am cuprins-o pe toată pentru că mai sunt blocurile de „try-catch” și liniile de afișare a rezultatelor. Simularea trebuie să ruleze atâta timp cât timpul curent de simulare este mai mic sau egal decât timpul maxim de simulare dat de către utilizator. Apoi, pentru fiecare client din lista de clienți generată mai sus care are timpul de sosire egal cu timpul de simulare curent, apelez metoda „dispatchClient” care îl adaugă la coada cu cel mai mic timp de așteptare și apoi îl șterg din lista de clienți. Mă folosesc mereu de clientul de la indexul 0 pentru ca la acea poziție se află clientul cu timpul cel mai mic de sosire deoarece lista este sortată. După calculez ora de vârf ca fiind suma tuturor clienților la momentul respectiv de timp din toate cozile. Dacă suma calculată este mai mare decât ce am calculat la un moment anterior de timp, setez variabila care mă ajută în acest sens, respectiv „pk”, cu valoarea sumei calculate și setez și ora de vârf cu valoarea timpului curent. În „for”-ul în care calculez câți clienți sunt în toate cozile, adun la valoarea „averageWaiting” timpul de așteptare pentru fiecare coadă. Timpul de așteptare mediu va fi suma tuturor timpul de așteptare a tuturor cozilor la fiecare moment de timp de simulare împărțită la câți clienți au fost în total la toate cozile în timpul simulării. Apoi afișez în interfață și în fișierul de output corespunzător și cresc timpul de simulare. Apoi verific dacă ar fi cazul ca simularea să se oprească dacă lista de clienți generați este goală și dacă sunt toate cozile închise. Apoi thread-ul doarme o secundă pentru sincronizarea aplicației.****

Mai jos este continuarea metodei de „run” descrisă mai sus, unde se vede că după ce se termină timpul de simulare, opresc toate thread-urile cozilor pornite și afișez rezultatele calculate în interfață și în fișierul de output.

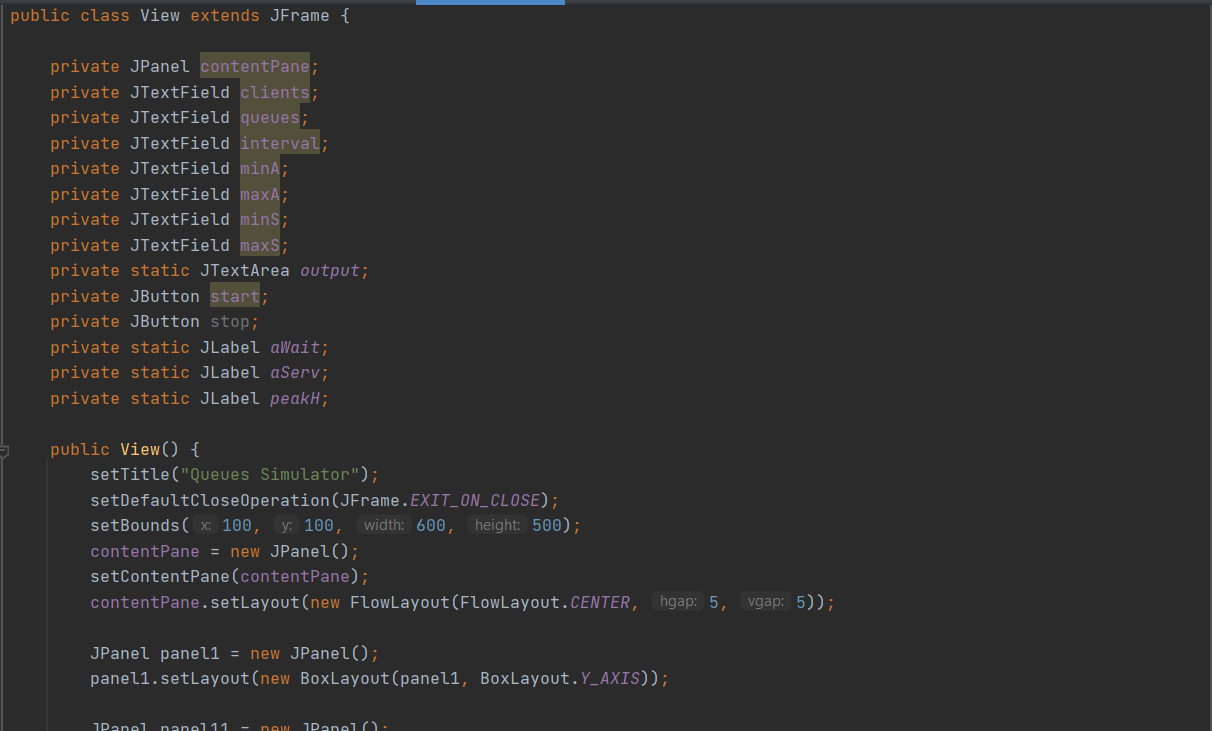
****

Metoda suprascrisă „toString” mă ajută să afișez lista de clienți care așteaptă în ordinea ID-ului, nu în ordinea timpului de sosire.

****

Clasa „View”**:**

Clasa „View” care extinde „JFrame” are ca atribute cele mai importante componente din cadrul interfeței (unele componente au fost declarate numai în constructor). În cadrul constructorului este creată fereastra cu anumite dimensiuni și un anumit nume, apoi este creat „Panel”-ul de bază la care se adaugă treptat restul „Panel”-urilor care conțin componentele necesare.

****

Metodele de mai jos din clasa „View” sunt folosite pentru a extrage input-ul adăugat de utilizator, pentru a suprascrie etichetele cu rezultatele corespunzătoare și pentru a adăuga ascultător pentru butonul de „start”.

****

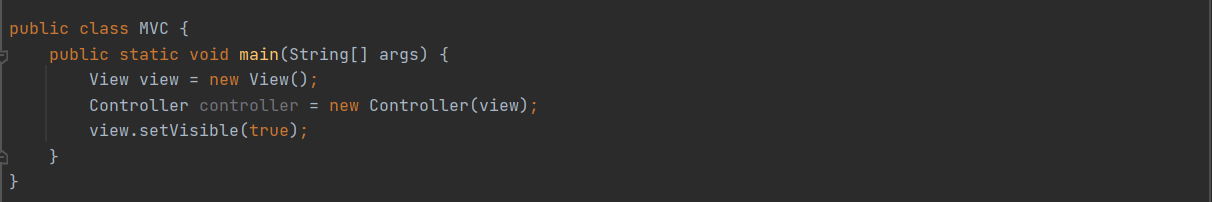
Clasa „Controller”**:**

Clasa „Controller” are un singur atribut, și anume o instanță de obiect către clasa „View”. În cadrul constructorului clasei „Controller” se inițializează singurul atribut și se implementează metoda de adăugare de ascultător pentru butonul de „start”. Atunci când se apasă butonul de „start”, se culeg datele corespunzătoare adăugate de către utilizator în interfață și se creează un obiect de simulare de tipul „SimulationManager”. Apoi este creat un thread de tipul obiectului de simulare care este și pornit. Afișarea rezultatelor la fiecare moment de timp al simulării este realizat de thread-ul de simulare.

****

Clasa „MVC”**:**

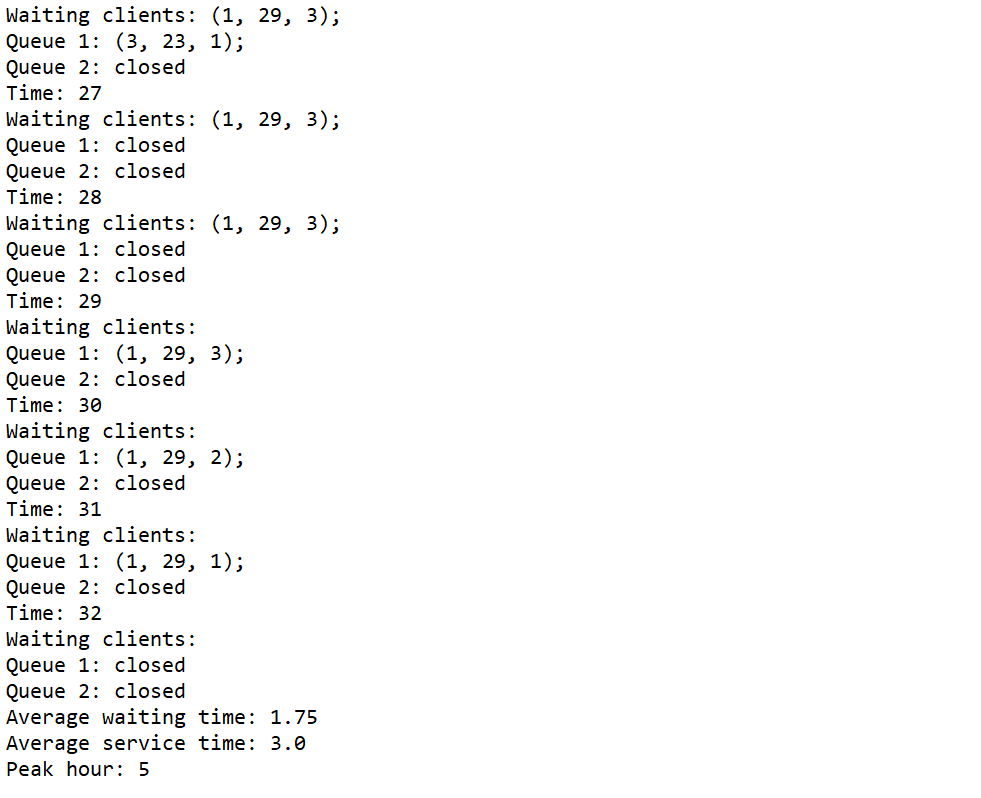
În clasa „MVC” creez o instanță a unui obiect de clasă „View” care este pasat ca argument constructorului „controller”-ului și apoi setez vizibilitatea „view”-ului.

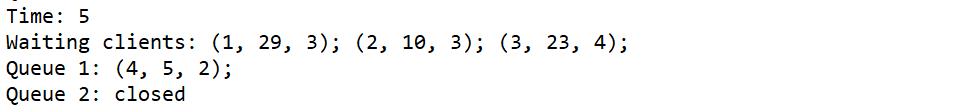
****

# Rezultate (testare)

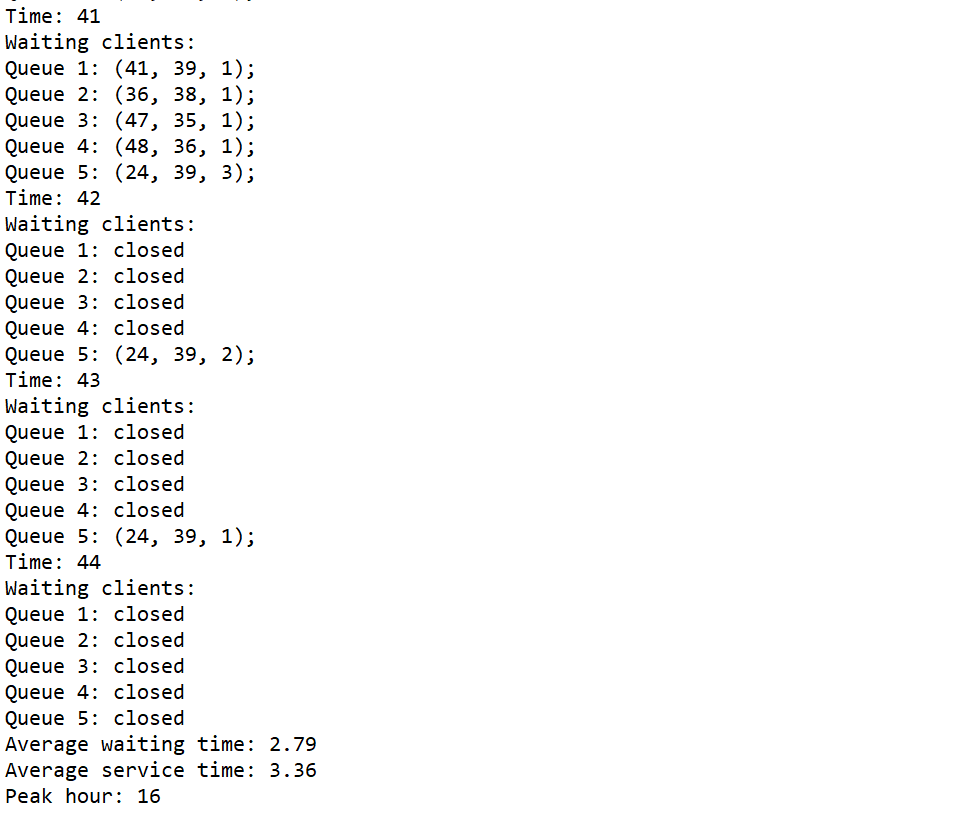
Am testat implementarea mea cu ajutorul datelor de intrare care ne-au fost spuse în cerința problemei. Aplicația mea a trecut fiecare test și a generat date de ieșire corespunzătoare pe care le adaug mai jos.

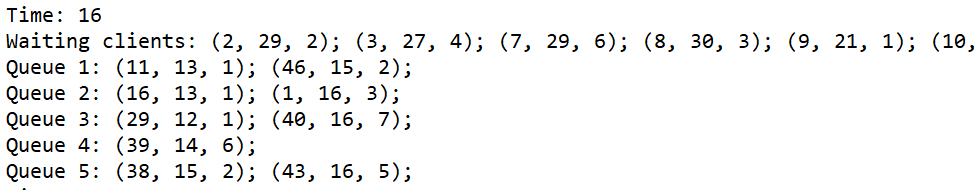
Testul pentru 4 clienți**:**

****

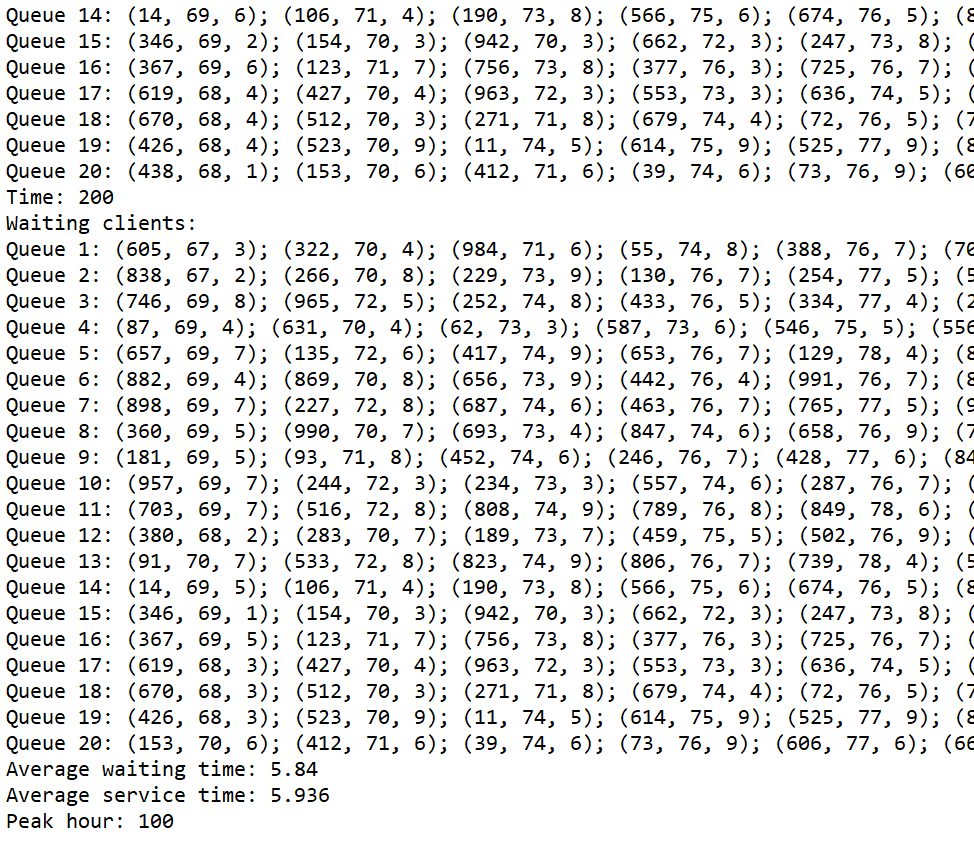
****

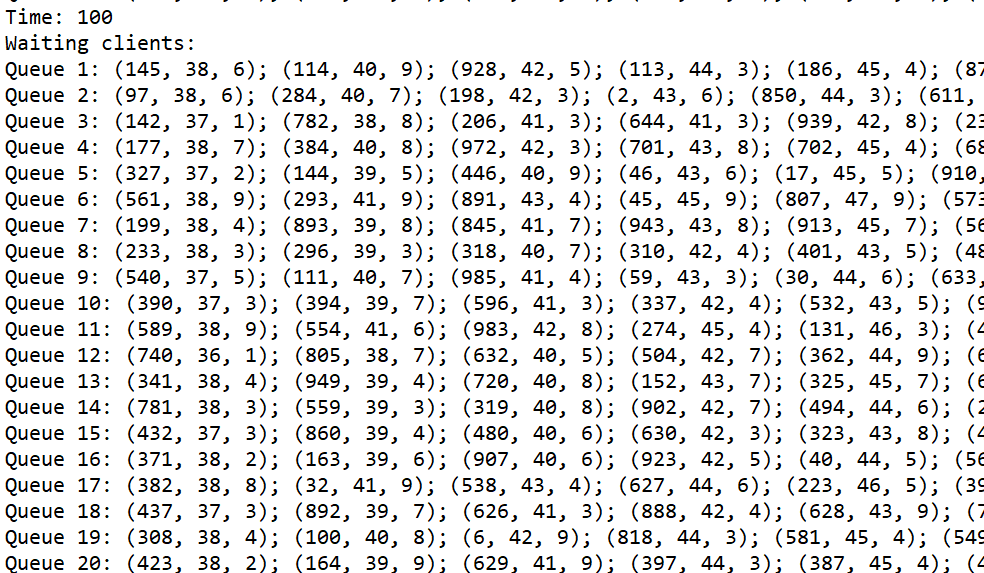
Testul pentru 50 de clienți**:**

****

****

Testul pentru 1000 de clienți**:**

****

****

# Concluzii

În urma rezolvării acestei teme, pot spune că mi-am însușit mai bine cunoștințele din semestrul întâi despre Java și am acumulat și informații noi, cum ar fi utilizarea thread-urilor. Dezvoltarea treptată a aplicației m-a ajutat să vin cu idei cât mai creative și eficiente pentru fiecare problemă care apărea.

Ca și dezvoltări ulterioare, aș mai finisa metoda de simulare și aș calcula cu o altă formulă timpul mediu de așteptare și poate aș face și o interfață mai plăcută vizual, nu doar minimalistă.

# Bibliografie

Nu m-am ajutat de nicio sursă de inspirație pentru redactarea documentație. Tot ce am scris aici sunt ideile care mi-au venit pe moment. Nu pot garanta că în aceste idei nu se găsesc rezumate a altor idei citite pe diverse site-uri de domeniu sau în diverse cărți.