DOCUMENTATIE

TEMA 2



NUME STUDENT: Sîrbu Cosmin-Claudiu

GRUPA: 30221(2)

Cuprins

[DOCUMENTATIE 1](#_Toc133375299)

[TEMA 2 1](#_Toc133375300)

[1. Obiectivul temei 3](#_Toc133375301)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 5](#_Toc133375302)

[3. Proiectare 7](#_Toc133375303)

[4. Implementare 9](#_Toc133375304)

[5. Rezultate 13](#_Toc133375305)

[6. Concluzii 13](#_Toc133375306)

[7. Bibliografie 13](#_Toc133375307)

# Obiectivul temei

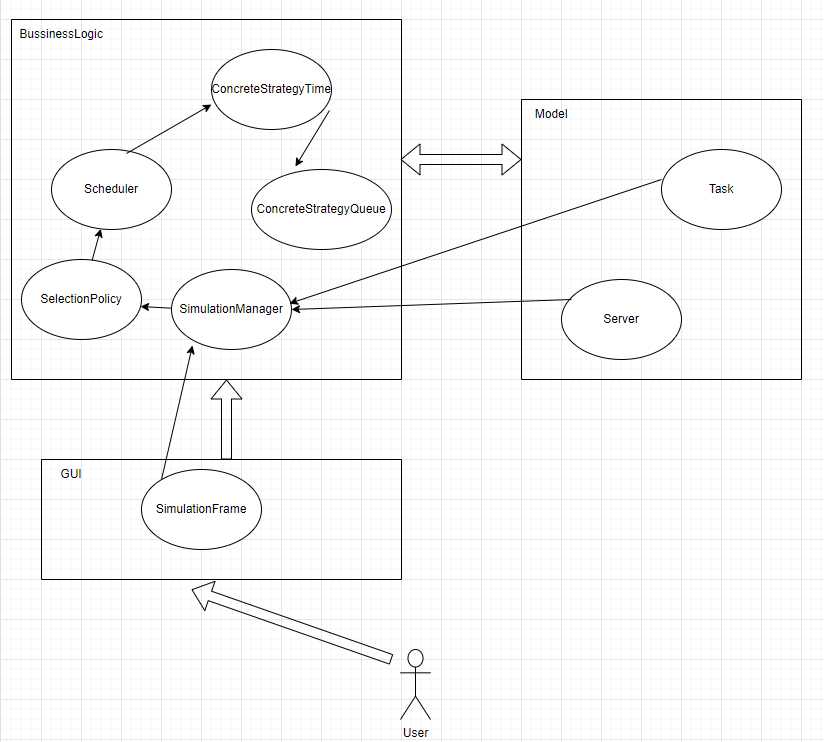
Obiectivul principal al proiectului este de a implementa un numar dat de personae, într-o

coadă (sau mai multe) într-un magazin. Pentru dezvoltarea și implementarea funcțiilor necesare realizării acestui proiect, s-a folosit limbajul de programare Java, folosind ca IDE, aplicația IntelliJ Idea de la NetBrains. Conceptul este asemănător ca în viața reală. Persoanele care se așează la coadă, se așează în funcție de 2 posibile criterii. După achiziționarea obiectelor dorite, se așează ori la coada cea mai scurtă, ori la cea mai rapidă, în funcție de preferința clientului. Pentru a putea observa cum decurg procesele de terminare a cozilor, s-a implementat o interfață GUI, o clasă Java prin care utilizatorul interacționează în mod direct cu programul. Acestea sunt obiectivele principale ale proiectului.

Obiectivele secundare vor fi prezentate în tabelul de mai jos:

|  |  |  |
| --- | --- | --- |
| ***Obiectiv secundar*** | ***Descriere*** | ***Capitol*** |
| Implementarea clasei SimulationManager | În această clasă s-a apelat interfața prin care s-a putut determina timpul minim si maxim pentru care un client poate sta la coadă, numarul maxim de clienți, case de marcat, timpul maxim pentru care se pot efectua încasări. | 3, 4 |
| Implementarea clasei enum SelectionPolicy | Prin această clasă se enumeră strategiile care pot fi folosite de clienți ca să se așeze la casele de marcat. | 4 |
| Implementarea Clasei Scheduler | Sunt implementate strategiile (se face alegerea). | 4 |
| Clasa ConcreteStrategyTime  ***Obiectiv secundar*** | Prin această clasa este descrisă metoda de adaugare de client la casă pe baza strategiei de cel mai scurt timp de așteptare la casă.  ***Descriere*** | 4  ***Capitol*** |
| Clasa ConcreteStrategyQueue | Prin această clasa este descrisă metoda de adaugare de client la casă pe baza strategiei de cea mai scurtă coadă de așteptare la casă. | 4 |
| Interfață Strategy | Implementează metoda de adaugare client. | 4 |
| Clasa Task | Descrierea obiectului Task, adică acțiunile clientului. | 4 |
| Clasa Server | Descrierea obiectului Server, sau a casei de marcat. | 4 |
| Clasa SimulationFrame | Conține GUI-ul și elementele introduse pentru realizarea interfeței. | 4 |
| Implementarea Main-ului | Rularea obiectivului principal | 1 |

# Analiza problemei, modelare, scenarii, cazuri de utilizare



**Cerințe funcționale:**

1. Interfața implementată care face legătura dintre utilizator și program permite accesul introducerii de la tastatură a timpului minim și maxim de așteptare, timpul minim și maxim de sosire la casă, numărul de case disponibile, numărul total de clienți care se află în magazin și timpul total cât se lucrează la case.
2. Prin butonul introdus în interfață, se dă start-ul la cozi.
3. Clasele implementate nu depășesc 300 de linii de cod.
4. Metodele nu depășesc 30 de linii.
5. Interfața grafică s-a realizat prin Java Swing.
6. S-au folosit thread-uri în realizarea programului. Sincronizări. Generare aleatoare pentru intervalele de timp.

**Liste pași:**

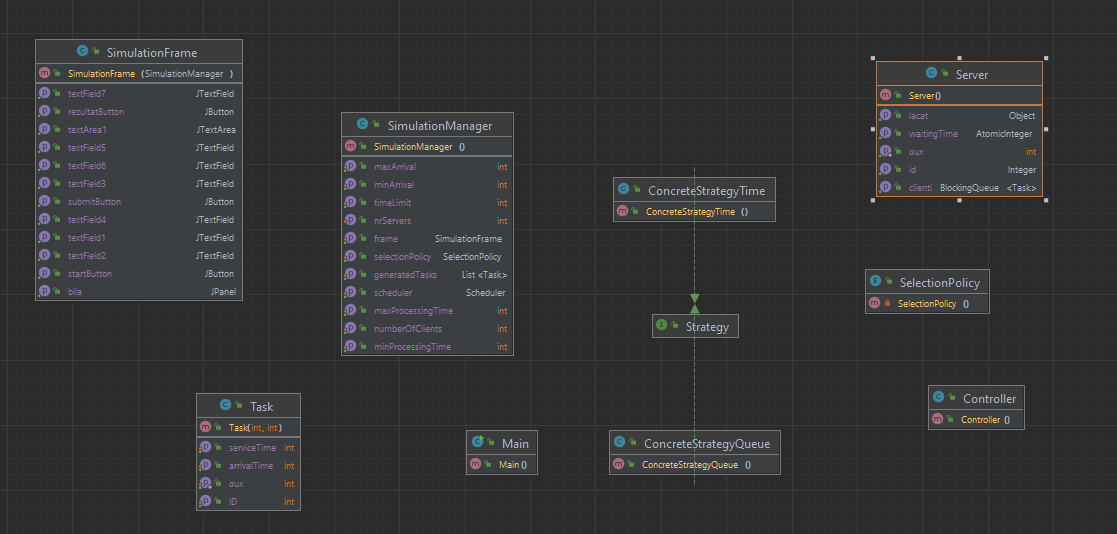
1. Prin intermediul interfeței GUI utilizatorul aare posibilitatea de a introduce de la tastatură datele necesare care sunt cerute in Label-uri.
2. Interfața este apelată în clasa SimulationManager, iar prin crearea unei simulări noi în Main, odată cu rularea programului, aceasta este lansată. În urma introducerii datelor de la tastatură și apăsarea butonului de “Start”, se activează funcția afișare() din clasa SimulationManager. În afișare se printează felul cum interacționează thread-urile. Se va afișa situația în magazin în fiecare moment. Rularea se va opri doar când toți clienții și-au finalizat cumpărăturile.
3. Clasele implementate nu depășesc niciuna numărul maxim admis de 300 de linii de cod. Clasa cu cel mai mare număr de linii de cod este clasa “SimulationManager” care conține 212 linii.
4. Metodele au fost implementate citeț și ușor de înțeles, niciuna din ele nedepășind 30 de linii.
5. S-a folosit Java Swing pentru crearea interfeței grafice. Această interfață folosește 7 Text-Field-uri, 1 buton, 1 JTextArea, 1 JScrollPane. Java Swing-ul permite crearea de aplicații pentru Desktop, prin care utilizatorul interacționează cu mediul de lucru. Interfața arată în felul următor:

O imagine care conține text

Descriere generată automat

# Proiectare

Diagrama UML a proiectului:



**Structuri de date:**

În codul prezentat, se folosesc următoarele structuri de date:

1. ArrayList<Task>: pentru a stoca toate task-urile generate în timpul simulării.
2. Scheduler: pentru a gestiona procesul de alocare a task-urilor la servere.
3. List<Task>: pentru a stoca temporar task-urile generate într-un anumit moment de timp, înainte de a fi trimise către Scheduler pentru a fi alocate unui server.
4. ArrayList<Server>: pentru a stoca toate serverele disponibile.
5. Synchronized blocuri și obiecte de tipul Lacăt (clasa java.util.concurrent.locks.ReentrantLock): pentru a asigura accesul exclusiv la anumite resurse (cum ar fi lista de clienți ai unui server) în timpul simulării cu mai multe fire de execuție.

**Interfețe:**

Această interfață reprezintă o fereastră de simulare pentru un sistem. Ea conține un total de opt câmpuri de text (JTextField), unul dintre ele fiind un câmp de text mai mare (JTextArea) pentru afișarea rezultatelor. De asemenea, are un buton (JButton) - pentru a obține rezultate (rezultatButton). Toate elementele interfeței sunt afișate pe un panou (JPanel) numit Auchan, care este atașat la fereastra JFrame. În plus, interfața este asociată cu un obiect de tip SimulationManager (m), care este pasat prin intermediul constructorului, și cu un obiect de tip Controller (c) utilizat pentru gestionarea evenimentelor din interfață.

**Algoritmii folosiți:**

Clasa SimulationManager implementează interfața Runnable și are rolul de a gestiona simularea de cozi folosind un obiect de tip Scheduler și o interfață grafică (SimulationFrame). Mai jos sunt descriși algoritmii principali din această clasă:

generateNRandomTasks() - metoda generează un număr specific de obiecte Task cu timp de sosire și de procesări aleatorii și le adaugă la lista de obiecte generatedTasks.

run() - metoda conține o buclă while care rulează simularea până când timpul simulării (timeLimit) este atins. În fiecare iterație, metoda verifică dacă există vreun client în magazin sau dacă vreo coadă este goală. În caz afirmativ, se așteaptă 1 secundă și se notifică toate coziile folosind metoda notifyAll(). Apoi, metoda își actualizează interfața grafică cu ajutorul metodei afisare() și incrementează timpul curent. Dacă niciun client nu mai există în magazin și toate cozile sunt goale, simularea se oprește.

afisare(int timp) - metoda generează un șir de caractere care reprezintă starea actuală a magazinului și a cozilor într-un anumit moment de timp. Aceasta afișează numărul și timpul de sosire și procesare a fiecărui client și numărul coziilor, precum și clienții care așteaptă la fiecare coadă.

Setările variabilelor de instanță ale clasei (maxProcessingTime, minProcessingTime, numberOfClients, etc.) sunt gestionate prin metodele de getter și setter corespunzătoare.

# Implementare

Aceasta clasă reprezintă un tip de enumerare (enum) în limbajul de programare Java, denumit "SelectionPolicy". Enum-urile permit programatorilor să definească un set de constante cu nume, care pot fi utilizate în cod în loc de valori literale. În acest caz, există două constante definite: "SHORTEST\_QUEUE" și "SHORTEST\_TIME". Acestea reprezintă două politici de selecție utilizate în cadrul unei simulări a unui sistem de cozi cu mai multe servere. Politica "SHORTEST\_QUEUE" selectează serverul cu cel mai mic număr de clienți în coadă, iar politica "SHORTEST\_TIME" selectează serverul cu cel mai scurt timp de așteptare pentru a procesa un client.

Clasa "Scheduler" reprezinta un planificator de sarcini care are ca scop gestionarea resurselor unui numar maxim de servere si distribuirea sarcinilor primite catre acestea, in functie de o anumita strategie de selectie.

Aceasta clasa contine urmatoarele atribute:

- "servers" reprezinta o lista de servere;

- "maxNoServers" reprezinta numarul maxim de servere;

- "strategy" reprezinta o instanta a unei clase de strategie de selectie a serverelor (initializata cu "ConcreteStrategyTime").

Constructorul clasei primeste ca parametru "maxNoServers" si initializeaza lista de servere cu numarul maxim de servere alocat.

Metoda "changeStrategy" primeste ca parametru un tip de politica de selectie ("SelectionPolicy"), iar in functie de aceasta, instanta de strategie este inlocuita cu o noua instanta corespunzatoare acestei politici (instantiata cu "ConcreteStrategyQueue" sau "ConcreteStrategyTime").

Metoda "dispatchTask" primeste o sarcina ("Task") si o adauga la coada unui server, folosind strategia de selectie a serverului corespunzatoare. Daca strategia este de tipul "SHORTEST\_TIME", se creaza o noua instanta a clasei "ConcreteStrategyTime", iar daca strategia este de tipul "SHORTEST\_QUEUE", se creaza o noua instanta a clasei "ConcreteStrategyQueue".

Metoda "getServers" returneaza lista de servere.

Aceasta clasă implementează interfața `Strategy` și reprezintă o strategie concretă de alegere a serverului pentru a fi adăugat în coada de așteptare a unui server, bazată pe timpul minim de așteptare.

Metoda `addTask` primește o listă de servere și o sarcină (task) și determină serverul cu cel mai scurt timp de așteptare pentru a adăuga sarcina. În primul rând, se inițializează variabila `minim` cu primul server din listă, apoi se compară timpul de așteptare al fiecărui server cu timpul de așteptare al serverului minim. Dacă timpul de așteptare al serverului curent este mai mic decât timpul de așteptare al serverului minim, atunci serverul minim devine serverul curent. La final, sarcina este adăugată în coada de așteptare a serverului minim.

Clasa "ConcreteStrategyQueue" implementează interfața "Strategy" și conține o metodă numită "addTask". Această clasă implementează strategia de adăugare a unei sarcini (Task) în coada de așteptare a unui server.

Metoda "addTask" primește ca parametri o listă de obiecte Server și o sarcină (Task). Această metodă alege serverul cu cel mai mic număr de clienți din lista de servere și adaugă sarcina în coada sa de așteptare. În cazul în care mai multe servere au același număr de clienți, este ales primul server din listă.

Această clasă este parte a unui design pattern numit "Strategy Pattern", care permite modificarea comportamentului unui obiect prin selectarea uneia dintre mai multe strategii posibile la runtime. În acest caz, "ConcreteStrategyQueue" reprezintă o implementare a unei strategii de adăugare a sarcinilor în coada de așteptare a serverelor, dar există și alte implementări posibile ale acestei strategii sau alte strategii pentru gestionarea sarcinilor.

Clasa "Server" reprezintă un server într-un sistem de procesare a sarcinilor (Task). Această clasă implementează interfața "Runnable", ceea ce înseamnă că poate fi rulată într-un fir de execuție separat.

Clasa "Server" conține următoarele atribute:

- "clienti": o coadă de sarcini (Task) care trebuie procesate de server

- "waitingTime": un obiect AtomicInteger care reprezintă timpul total de așteptare a clienților pentru a fi serviți de către server

- "lacat": un obiect folosit pentru sincronizare

- "id": un identificator unic al serverului

- "aux": un contor static folosit pentru a asigna id-uri unice fiecărui server creat

Constructorul clasei "Server" inițializează atributul "clienti" cu o coadă de sarcini de dimensiune maximă 200, "waitingTime" cu valoarea 0, "lacat" cu un obiect nou și atributul "id" cu o valoare unică asignată de contorul static "aux".

Clasa "Server" conține și următoarele metode:

- "adaugareClient": adaugă o sarcină (Task) în coada de așteptare a serverului și actualizează timpul total de așteptare al clienților

- "run": reprezintă logica de procesare a sarcinilor de către server. Metoda rulează într-un ciclu infinit și preia prima sarcină din coada de așteptare a serverului. Dacă timpul de servire al sarcinii este 1, aceasta este eliminată din coada de așteptare, altfel timpul de servire al sarcinii este decrementat cu 1. Metoda actualizează timpul total de așteptare al clienților și așteaptă până când un alt fir de execuție îi notifică că a fost adăugată o nouă sarcină în coada de așteptare.

- "getAux": o metodă statică care returnează valoarea contorului "aux"

- "getWaitingTime": o metodă care returnează timpul total de așteptare a clienților

- "getClienti": o metodă care returnează coada de așteptare a serverului

- "getId": o metodă care returnează identificatorul unic al serverului

- "getLacat": o metodă care returnează obiectul folosit pentru sincronizare

- "eGoala": o metodă care returnează true dacă coada de așteptare a serverului este goală sau false altfel.

Clasa `Task` este o clasă simplă care modelează un task ce trebuie să fie procesat într-un sistem de coadă.

Are câteva variabile de instanță private:

- `ID` - identificatorul unic al task-ului

- `aux` - un contor static care reține numărul total de obiecte create din această clasă. Aceasta este utilizată pentru a asigna o valoare unică câmpului `ID` al fiecărui nou obiect.

- `arrivalTime` - timpul la care task-ul a fost adăugat în sistem

- `serviceTime` - timpul necesar pentru a procesa task-ul

Clasa are un constructor public care primește `arrivalTime` și `serviceTime`, setându-le pe amândouă și setând `ID`-ul obiectului curent cu valoarea din `aux`, apoi incrementând `aux`.

Mai are și getteri și setteri pentru variabilele private, care sunt utilizate pentru a accesa și actualiza valorile acestora din alte clase.

Celelalte clase: Main, SimulationManager, SimulationFrame, au fost prezentate în capitolele anterioare.

# Rezultate

Rezultatele au fost încărcate pe GitHub.

# Concluzii

Am creat o aplicatie de gestiune a cozilor care atribuie clientilor cozile astfel incat timpul de asteptare sa fie minimizat. Am implementat un simulare a sosirii clientilor, introducerii lor in cozi, asteptarii, servirii si parasirii cozilor, tinand cont de trei parametrii specifici fiecarui client: ID-ul, timpul de sosire si intervalul de timp necesar pentru servire. Am urmarit si am calculat timpul total petrecut de fiecare client in cozi si am calculat timpul mediu de asteptare. Am adaugat fiecare client in coada cu cel mai mic timp de asteptare atunci cand timpul de sosire al clientului a fost mai mare sau egal cu timpul de simulare. Aplicatia mea de gestiune a cozilor este capabila sa primeasca de la utilizator datele de intrare necesare, cum ar fi numarul de clienti, numarul de cozi, intervalul de simulare, timpul minim si maxim de sosire si timpul minim si maxim de servire. Toate cerintele au fost atinse cu succes.

# Bibliografie

http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html http://www.tutorialspoint.com/java/util/timer\_schedule\_period.htm http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-andthreadpoolexecutor.html