QUEUE

DOCUMENTATIE - TEMA 2 - TP

PĂCURAR IRINA
UTCN, CTI, SERIA A, GRUPA 30223
ÎNDRUMĂTORI: PROF. DR. ING. TUDOR CIOARĂ, OVIDIU MAJA
- martie 2024 -

Cuprins

1. Obiectivul temei

Tema propusă este de a proiecta și implementa o aplicație de gestionare a cozilor, care să aloce clienții în așa fel încât timpul de așteptare să fie minimizat. Aplicația trebuie să simuleze un număr de clienți *N* care sosesc la serviciu, intră într-un număr de cozi *Q*, așteaptă, sunt serviți și, în final, părăsesc cozile. Fiecare client este generat la începutul simulării și este caracterizat prin trei parametri: ID, timpul de sosire în coadă *tarrival* și timpul de servire *tservice*. Aplicația urmărește timpul total petrecut de fiecare client în cozi și calculează timpul mediu de așteptare. Fiecare client este adăugat în coada cu timpul minim de așteptare când timpul său de sosire este mai mare sau egal cu timpul de simulare curent.

Datele de intrare pentru aplicație, care trebuie introduse de utilizator prin interfața aplicației, includ numărul de clienți *N*, numărul de cozi *Q*, intervalul de simulare maxim *tsimulation_MAX*, timpul minim și maxim de sosire $tarrival\_MIN \leq tarrival \leq tarrival\_MAX$ și timpul minim și maxim de servire $tservice\_MIN \leq tservice \leq tservice\_MAX$.

Utilizarea firelor de execuție este esențială pentru procesarea în paralel a clienților în cozi, iar structurile de date sincronizate asigură siguranța firelor de execuție în gestionarea accesului la resurse comune. Aplicația va include un jurnal de evenimente care afișează starea clienților care așteaptă și a cozilor pe măsură ce timpul de simulare avansează.

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

Aplicația de gestionare a cozilor trebuie să optimizeze timpul de așteptare pentru clienți într-un sistem de cozi. Scopul este de a asigura că fiecare client este servit în cel mai scurt timp posibil, folosind un număr limitat de cozi. Principalele provocări includ echilibrarea încărcării între cozi, gestionarea concurenței și sincronizarea firelor de execuție pentru a evita condițiile de întrecere și datele corupte.

2.1.1. Modelare

- **Entități Principale:**
  - **Client:** Reprezentat de un ID unic, timp de sosire în coadă și timp de serviciu necesar.
  - **Coada:** Gestionată de un thread separat, cu propriul său set de clienți în așteptare și logică pentru procesarea clienților.
  - **Simulator:** Controlează timpul de simulare și distribuie clienții către coada cu cel mai mic timp de așteptare.
- **Date de Intrare:**
  - Numărul de clienți N
  - Numărul de cozi Q
  - Intervalul de timp pentru simulare
  - Timpul minim și maxim de sosire
  - Timpul minim și maxim de serviciu
- **Proces:**
  - La începutul simulării, clienții sunt generați cu atribute aleatoare conform specificațiilor.
  - Un thread central controlează timpul de simulare și alocă fiecare client corespunzător.
  - Fiecare coadă procesează clienții în funcție de sosirea și timpul de serviciu al acestora.

2.1.2. Scenarii

1. **Sosirea Clientului:** Un client nou sosesc la timpul $t_{arrival}$ și este plasat în coada cu timpul de așteptare minim.
2. **Procesarea Clientului:** Clientul la vârful cozii este procesat până când timpul său de serviciu expiră.
3. **Finalizarea Servirii:** Când un client este complet servit, acesta este eliminat din coadă.
4. **Calculul Timpului Mediu de Așteptare:** La finalul simulării, se calculează timpul mediu de așteptare pentru toți clienții.

2.1.3. Cazuri de Utilizare

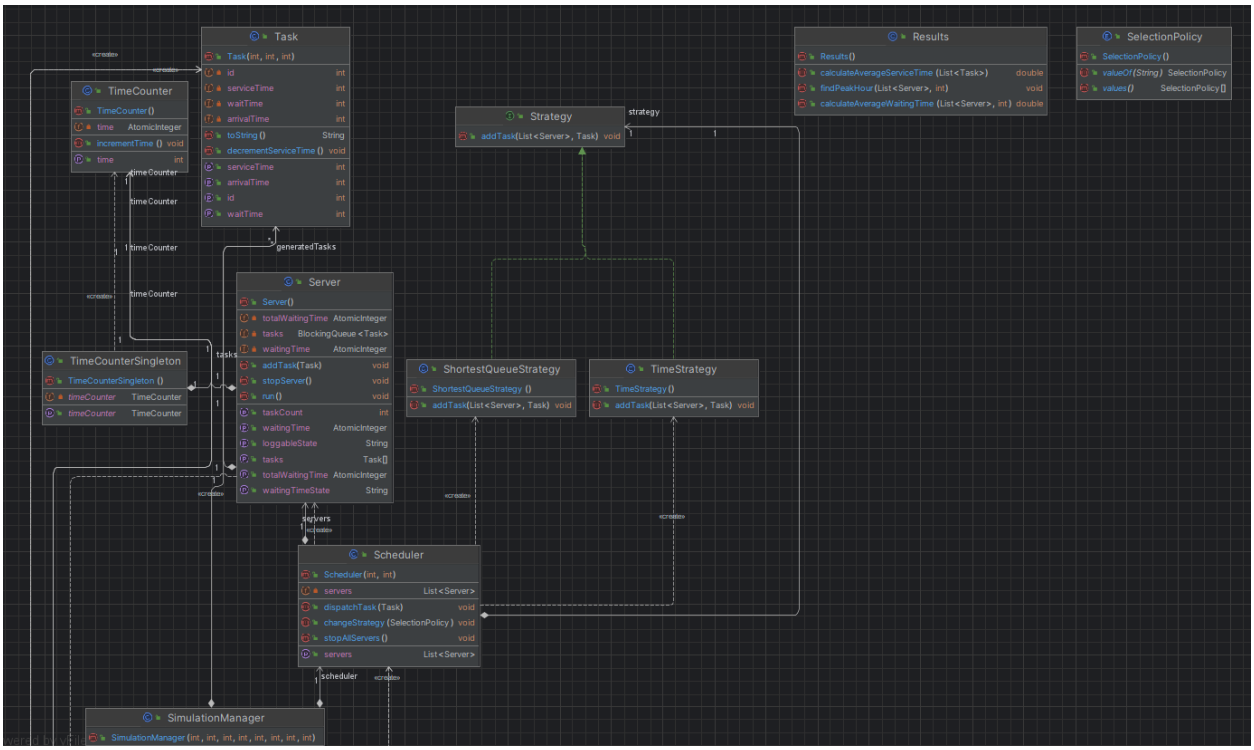
1. **Inițierea Simulării:** Utilizatorul introduce parametrii necesari și începe simularea.
2. **Vizualizarea Progresului:** Utilizatorul urmărește evoluția cozilor și a clienților în timp real prin interfața grafică.
3. **Ajustarea Parametrilor:** Utilizatorul poate modifica numărul de cozi sau intervalul de timp pentru a observa cum aceste schimbări afectează performanța sistemului.
4. **Oprirea Simulării:** Utilizatorul poate opri simularea la orice moment pentru a analiza datele colectate.

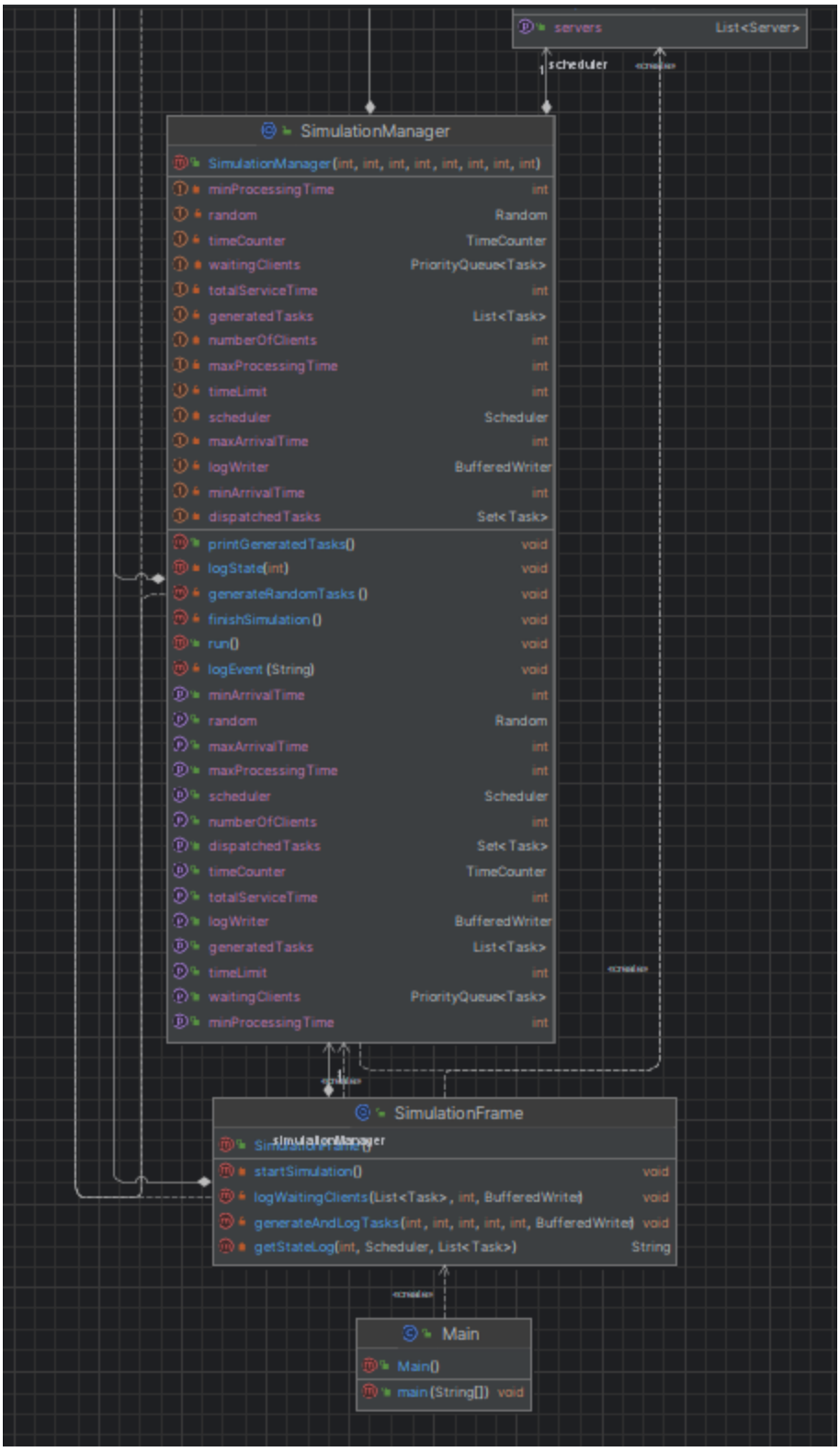
Implementarea acestei aplicații va folosi principii de programare orientată pe obiecte și tehnici de programare concurentă, asigurându-se că toate firele de execuție sunt gestionate în mod sigur și eficient.

3. Proiectare

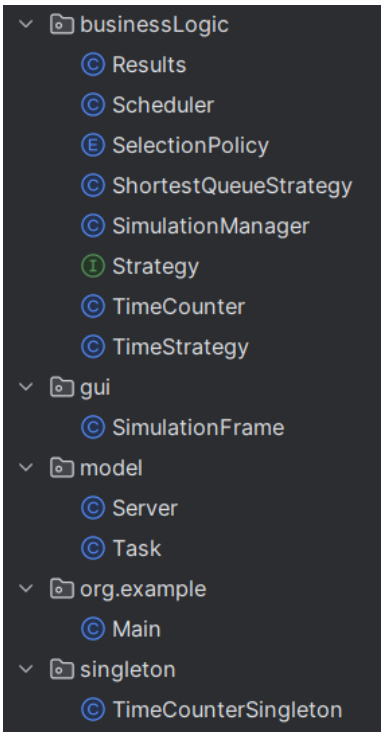
Aplicația include 11 clase principale: *MainClass*, aflată în pachetul *org.example*, responsabilă de inițializarea interfeței grafice; clasa *SimulationFrame*, din cadrul pachetului *GUI*, care conține toate elementele grafice-vizuale și facilitează interacțiunea utilizatorului cu aplicația; *Server*, *Task* din pachetul *model*, *TimeCounterSingleton* din pachetul singleton, iar în pachetul businessLogic se regăsește enumul *SelectionPolicy*, precum și clasele: *Results*, *Scheduler*, *SimulationManager*, *TimeCounter*, *ShortestQueueStrategy* și *TimeStrategy*, ultimele 2 implementând interfața *Strategy*.

Clasele




servers List<Server>
scheduler
SimulationManager
SimulationManager(int, int, int, int, int, int, int, int)
minProcessingTime int
random Random
timeCounter TimeCounter
waitingClients PriorityQueue<Task>
totalServiceTime int
generatedTasks List<Task>
numberOfClients int
maxProcessingTime int
timeLimit int
scheduler Scheduler
maxArrivalTime int
logWriter BufferedWriter
minArrivalTime int
dispatchedTasks Set<Task>
printGeneratedTasks() void
logState(int) void
generateRandomTasks() void
finishSimulation() void
run() void
logEvent(String) void
minArrivalTime int
random Random
maxArrivalTime int
maxProcessingTime int
scheduler Scheduler
numberOfClients int
dispatchedTasks Set<Task>
timeCounter TimeCounter
totalServiceTime int
logWriter BufferedWriter
generatedTasks List<Task>
timeLimit int
waitingClients PriorityQueue<Task>
minProcessingTime int
SimulationFrame
startSimulation() void
logWaitingClients(List<Task>, int, BufferedWriter) void
generateAndLogTasks(int, int, int, int, int, BufferedWriter) void
getStateLog(int, Scheduler, List<Task>) String
Main
Main()
main(String[]) void

Pachetele


businessLogic
Results
Scheduler
SelectionPolicy
ShortestQueueStrategy
SimulationManager
Strategy
TimeCounter
TimeStrategy
gui
SimulationFrame
model
Server
Task
org.example
Main
singleton
TimeCounterSingleton

4. *Implementare*

4.1. Clasa Main

Clasa **Main** din pachetul **org.example** servește drept punct de intrare în aplicația de gestionare a cozilor. Aceasta conține metoda **main**, care este metoda standard de lansare pentru orice aplicație Java.

```java
package org.example;

import businessLogic.SimulationManager;
import gui.SimulationFrame;

public class Main {
    public static void main(String[] args) {
//...
        new SimulationFrame();
    }
}
```

4.2. SimulationFrame

Clasa SimulationFrame este o componentă GUI (interfață grafică pentru utilizator) pentru aplicația de gestionare a cozilor, care este implementată ca un JFrame în Java Swing. Aceasta gestionează toate interacțiunile utilizatorului cu aplicația, permițând configurarea și monitorizarea unei simulări a managementului cozilor.

4.2.1. Componente

- *Câmpuri de Text*: SimulationFrame include mai multe câmpuri de text pentru introducerea datelor necesare simulării: numărul de clienți, numărul de cozi, intervalul de simulare, și intervalele pentru timpul de sosire și de servire (minim și maxim).
- *Butonul de Start*: Există un buton startButton care inițiază simularea când este apăsat.
- *Aria de Text pentru Log-uri*: O zonă de text logArea, unde sunt afișate logurile simulării, este inclusă pentru a vizualiza progresul și rezultatele simulării.

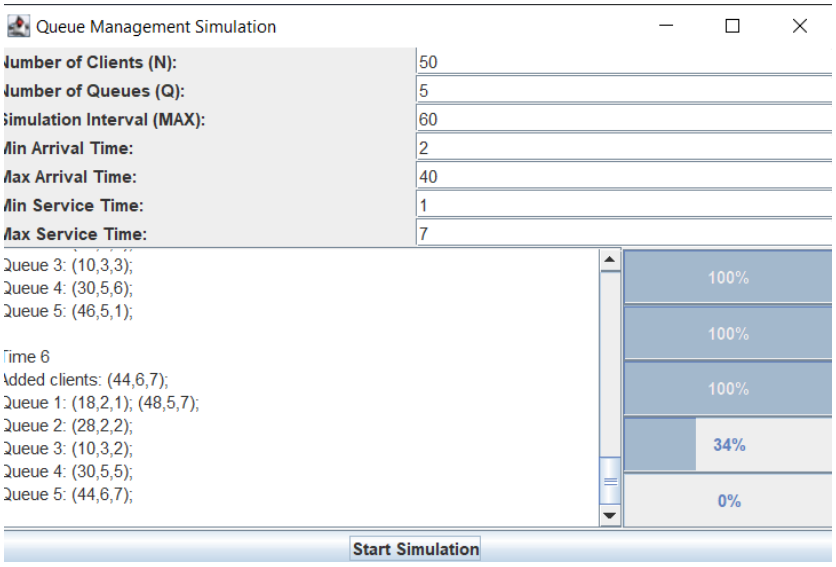
4.2.2. Funcționalități

- *Constructor*: Stabilește configurarea inițială a ferestrei, inclusiv dimensiunea, titlul, operația de închidere, layout-ul și adăugarea componentelor.
- *Interfața pentru Introducerea Datelor*: Panele și câmpurile pentru introducerea datelor permit utilizatorului să specifice parametrii necesari pentru simulare.

- *Acțiunea Butonului de Start*: Definește comportamentul aplicației la apăsarea butonului de start, declanșând metoda startSimulation().
- *Logica de Simulare*: Metoda startSimulation() inițializează și execută simularea într-un thread separat pentru a menține interfața grafică responsivă. Simularea include:
    - Crearea și scrierea în fișiere log pentru sarcini și starea simulării.
    - Distribuția sarcinilor (Task) în cozi în funcție de timpul lor de sosire, folosind un Scheduler.
    - Actualizarea continuă a interfeței grafice și a logurilor pe baza stării actuale a simulării.
- *Tratarea Excepțiilor și Întreruperilor*: Gestionează erorile ce pot apărea în timpul rulării simulării și se asigură că resursele sunt eliberate corespunzător la finalul simulării.

4.2.3. Utilizare

Această clasă este punctul central de interacțiune pentru utilizatorul final, oferindu-i posibilitatea de a configura parametrii simulării, de a iniția procesul și de a urmări evoluția acestuia în timp real printr-o interfață grafică intuitivă. Prin integrarea elementelor de control și vizualizare, SimulationFrame facilitează utilizarea aplicației într-un mod eficient și accesibil.



4.3. Server

Clasa Server din pachetul model este concepută pentru a gestiona sarcinile într-o coadă în cadrul unei simulări de gestionare a cozilor. Această clasă implementează interfața Runnable, ceea ce îi permite să fie executată ca un thread separat în cadrul aplicației. Scopul principal al clasei este de a procesa Task-urile adăugate la coada sa, gestionând timpul de așteptare și de serviciu al fiecărei sarcini.

4.3.1. Caracteristici

- *Coada de Sarcini*: Folosește un BlockingQueue<Task> pentru a stoca sarcinile care trebuie procesate. Acest tip de coadă este ales pentru capacitatea sa de a gestiona accesul concurent, fiind adecvat pentru operațiuni în medii multi-thread.
- *Timp de Așteptare*: Utilizează un AtomicInteger pentru waitingTime, care reprezintă timpul total de așteptare pentru sarcinile din coadă, permițând modificări atomice și sigure din punct de vedere al concurenței.
- *Timp Total de Așteptare:* Similar, totalWaitingTime este un AtomicInteger care acumulează timpul total de așteptare pentru toate sarcinile procesate de server.
- *Contorizare Timp*: Utilizează o instanță de TimeCounter printr-un singleton pentru a obține timpul curent în sistem, ceea ce este esențial pentru calculul timpilor de așteptare.

4.3.2. Funcționalități

- *Adăugarea de Sarcini*: Metoda addTask adaugă o sarcină la coadă și actualizează timpul de așteptare corespunzător duratei de serviciu a sarcinii.
- *Procesarea Sarcinilor*: În metoda run, serverul extrage și procesează sarcinile din coadă. Pentru fiecare sarcină, calculează timpul de așteptare, actualizează timpul total de așteptare și reduce timpul de serviciu al sarcinii curente la fiecare secundă până când acesta ajunge la zero.
- *Gestionarea Întreruperii*: Serverul verifică în mod constant dacă thread-ul a fost întrerupt, caz în care iese din bucla de procesare pentru a permite oprirea corectă a serverului.
- *Logarea Stării*: Metoda getLoggableState generează o reprezentare textuală a stării actuale a cozii, listând toate sarcinile curente într-un format ușor de citit.

```java
package model;
import businessLogic.TimeCounter;
import singleton.TimeCounterSingleton;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.atomic.AtomicInteger;

27 usages
public class Server implements Runnable {
    8 usages
    private BlockingQueue<Task> tasks;
    4 usages
    private AtomicInteger totalWaitingTime;
    5 usages
    private AtomicInteger waitingTime;
    1 usage
    private TimeCounter timeCounter = TimeCounterSingleton.getTimeCounter();
    1 usage
    public AtomicInteger getTotalWaitingTime() { return totalWaitingTime; }
    1 usage
    public Server() {...}
    2 usages
    public int getTaskCount() { return tasks.size(); }
    2 usages
    public void addTask(Task task) {...}
    6 usages
    public AtomicInteger getWaitingTime() { return waitingTime; }
    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            try {
                Task currentTask = tasks.peek();
                if (currentTask != null) {
                    currentTask.setWaitTime(timeCounter.getTime() - currentTask.getArrivalTime());
                    System.out.println(currentTask.getWaitTime() + " task:" + currentTask.getId());
                    totalWaitingTime.set(totalWaitingTime.get() + currentTask.getWaitTime());
                    while (currentTask.getServiceTime() > 0) {
                        waitingTime.decrementAndGet();
                        Thread.sleep( millis: 1000);
                        currentTask.decrementServiceTime();
                    }
                    tasks.poll();
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            }
        }
    }
    1 usage
    public Task[] getTasks() { return tasks.toArray(new Task[0]); }
    3 usages
    public String getLoggableState() {...}
    1 usage
    public void stopServer() { Thread.currentThread().interrupt(); }
}
```

4.4. Task

Clasa Task din pachetul model este o structură de date folosită pentru a reprezenta o sarcină sau un client în contextul unei simulări de gestionare a cozilor. Aceasta stochează informații esențiale despre fiecare sarcină, inclusiv timpul de sosire, timpul necesar pentru serviciu și timpul de așteptare.

4.4.1. Atribute

- *id*: Un identificator unic pentru sarcină, utilizat pentru a urmări și distinge sarcinile individuale.
- *arrivalTime*: Timpul la care sarcina sosește în sistem sau este pregătită pentru procesare.
- *serviceTime*: Durata necesară pentru completarea sarcinii, reflectând cât timp aceasta trebuie să fie procesată înainte de a fi considerată completă.
- *waitTime*: Timpul pe care sarcina îl petrece așteptând în coadă înainte de a fi procesată.

4.4.2. Constructor

- Constructorul clasei Task primește trei parametri (id, arrivalTime, serviceTime) și inițializează atributele corespunzătoare ale sarcinii.

4.4.3. Metode

- *Gettere și Settere*: Clasa oferă metode standard de tip getter și setter pentru toate atributele sale, permițând manipularea și accesarea valorilor acestora:
  - *getWaitTime(), setWaitTime(int waitTime)*
  - *getId(), setId(int id)*
  - *getArrivalTime(), setArrivalTime(int arrivalTime)*
  - *getServiceTime(), setServiceTime(int serviceTime)*
- *decrementServiceTime():* Această metodă este folosită pentru a decrementa timpul de serviciu cu o unitate. Este utilă în simularea procesului de serviciu, unde la fiecare unitate de timp (de exemplu, secundă), timpul de serviciu rămas necesar pentru completarea sarcinii este redus până când acesta ajunge la zero.
- *toString()*: Suprascrie metoda toString() pentru a oferi o reprezentare textuală a sarcinii, utilă pentru logare și afișarea stării. Formatează informațiile sarcinii într-un șir de caractere sub forma (id, arrivalTime, serviceTime).

```java
package model;

42 usages
public class Task {
    3 usages
    private int id;
    3 usages
    private int arrivalTime;
    5 usages
    private int serviceTime;
    2 usages
    private int waitTime;
    2 usages
    public Task(int id, int arrivalTime, int serviceTime) {...}
    2 usages
    public int getWaitTime() { return waitTime; }
    1 usage
    public void setWaitTime(int waitTime) { this.waitTime = waitTime; }
    2 usages
    public int getId() { return id; }
    8 usages
    public int getArrivalTime() { return arrivalTime; }
    4 usages
    public int getServiceTime() { return serviceTime; }

    @Override
    public String toString() { return "(" + id + "," + arrivalTime + "," + serviceTime + ")"; }
    1 usage
    public void decrementServiceTime() {...}
}
```

4.5. SimulationManager

Clasa SimulationManager din pachetul businessLogic joacă un rol central în gestionarea simulării unui sistem de cozi. Aceasta coordonează crearea și procesarea sarcinilor (tasks), gestionarea timpului și logarea stării sistemului. Clasa implementează interfața Runnable, permițându-i să fie executată într-un thread separat.

4.5.1. Constructor și Inițializare

Constructorul clasei SimulationManager primește mai mulți parametri esențiali pentru configurarea și inițializarea simulării:

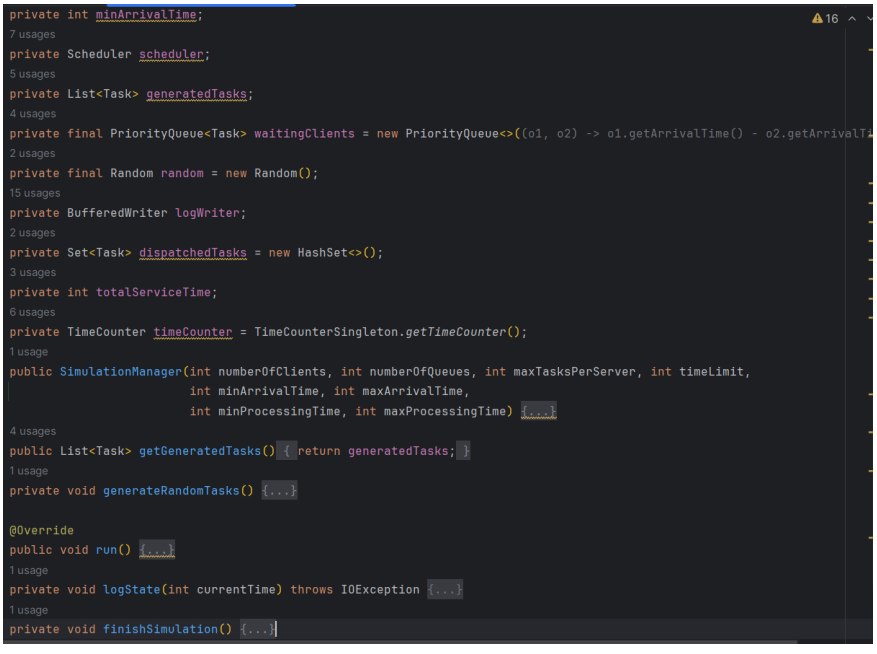
- *numberOfClients*: Numărul total de clienți (sarcini) care vor fi procesați.
- *numberOfQueues*: Numărul de cozi disponibile pentru procesarea sarcinilor.

- *maxTasksPerServer*: Capacitatea maximă de sarcini pe care fiecare server (coadă) o poate gestiona.
- *timeLimit*: Limita de timp pentru durata simulării.
- *minArrivalTime și maxArrivalTime*: Intervalele pentru timpul de sosire al sarcinilor.
- *minProcessingTime și maxProcessingTime*: Intervalele pentru timpul de procesare al sarcinilor.

Inițializează de asemenea un Scheduler pentru alocarea sarcinilor la cozi, generează sarcinile aleatoriu și pregătește un BufferedWriter pentru logarea evenimentelor.

4.5.2. Funcționalități

- *Generarea Sarcinilor Aleatorii*: Metoda generateRandomTasks creează un set de sarcini bazate pe parametrii specificați, fiecare având un timp de sosire și un timp de serviciu generat aleatoriu. Aceste sarcini sunt stocate într-o listă și într-o coadă de priorități (PriorityQueue), sortată după timpul de sosire.
- *Rularea Simulării*: Metoda run controlează fluxul principal al simulării. La fiecare increment de timp, verifică dacă există sarcini gata de a fi procesate, le scoate din coada de așteptare și le trimite la Scheduler pentru alocare la cozi. De asemenea, actualizează starea curentă în log.
- *Logarea Stării*: Logarea periodică include timpul de simulare, starea cozilor, sarcinile așteptate și timpul total de așteptare.
- *Calculul Statisticilor*: La finalul simulării, calculează timpul mediu de așteptare și timpul mediu de serviciu, scriind aceste informații în log.
- *finishSimulation*: Asigură închiderea corectă a BufferedWriter, gestionând eventualele excepții.



4.6. Scheduler

Clasa Scheduler din pachetul businessLogic este proiectată pentru a coordona distribuția sarcinilor (Task) către diferite servere (Server) în cadrul unei simulări de management al cozilor. Aceasta implementează logicile de alocare a sarcinilor folosind strategii diferite, în funcție de politica de selecție specificată.

4.6.1. Atribute

- *servers*: O listă de servere care procesează sarcinile.
- *maxNoServers*: Numărul maxim de servere care pot fi create.
- *maxTasksPerServers*: Capacitatea maximă de sarcini care pot fi alocate fiecărui server.
- *strategy*: Strategia actuală de alocare a sarcinilor la servere.

4.6.2. Constructor și inițializare

- Constructorul primește maxNoServers și maxTasksPerServers ca parametri și inițializează lista de servere. Pentru fiecare server, se pornește un nou thread, permițând serverelor să proceseze sarcinile în paralel.
- Strategia implicită de alocare a sarcinilor este setată la ShortestQueueStrategy, care alege serverul cu cea mai scurtă coadă pentru alocarea noilor sarcini.

4.6.3. Metode

- *getServers():* Returnează lista de servere.
- changeStrategy(SelectionPolicy policy): Schimbă strategia de alocare a sarcinilor în funcție de politica specificată (SHORTEST_QUEUE sau SHORTEST_TIME). Aceasta permite schimbarea dinamică a comportamentului de alocare în timpul rulării simulării.
- *dispatchTask(Task task)*: Distribuie o sarcină către servere bazându-se pe strategia curentă. Metoda verifică mai întâi dacă strategia a fost setată. Apoi, identifică serverele cu timpul de așteptare minim sau cu cel mai mic număr de sarcini, în funcție de numărul de servere cu aceleași valori minime ale timpului de așteptare:
    - Dacă există un singur server cu timpul de așteptare minim, se folosește strategia TimeStrategy.
    - Dacă există mai multe servere cu același timp de așteptare minim, se folosește strategia ShortestQueueStrategy.

```java
package businessLogic;

import model.Server;
import model.Task;

import java.util.*;

6 usages
public class Scheduler {
    7 usages
    private List<Server> servers;
    1 usage
    private int maxNoServers;
    1 usage
    private int maxTasksPerServers;
    6 usages
    private Strategy strategy;
    2 usages
    public Scheduler(int maxNoServers, int maxTasksPerServers) {...}
    8 usages
    public List<Server> getServers() { return servers; }
    4 usages
    public void changeStrategy(SelectionPolicy policy) {...}
    2 usages
    public void dispatchTask(Task task) {...}
}
```

4.7. Results

Clasa Results din pachetul businessLogic este o clasă utilitară destinată calculului și monitorizării diferitelor metrici de performanță în cadrul unei simulări de gestionare a cozilor. Această clasă conține metode statice care permit evaluarea eficienței sistemului de cozi, calculând timpul mediu de așteptare, timpul mediu de servire și identificând ora de vârf a sistemului.

4.7.1. Atribute statice

- *peakHour*: Memorează ora la care s-a înregistrat numărul maxim de clienți procesați.
- *maxClients*: Numărul maxim de clienți observați într-o oră specifică, folosit pentru a determina peakHour.

4.7.2. Metode statice

- *calculateAverageWaitingTime(List<Server> servers, int totalClients)*: Calculează timpul mediu de așteptare al clienților. Aceasta sumează timpurile totale de așteptare raportate de fiecare server și le împarte la numărul total de clienți procesați pentru a obține media.
- *calculateAverageServiceTime(List<Task> tasks)*: Determină timpul mediu de servire pentru toate sarcinile completate. Suma tuturor timpilor de servire este împărțită la numărul de sarcini pentru a calcula media.
- *findPeakHour(List<Server> servers, int simulationTime)*: Identifică ora de vârf în cadrul simulării. Funcția parcurge toate serverele și numără toate sarcinile active într-un moment dat. Dacă numărul sarcinilor procesate în ora curentă depășește recordul anterior, ora curentă este marcată ca fiind cea mai aglomerată, și se actualizează peakHour.

```
package businessLogic;

import model.Server;
import model.Task;

import java.util.List;

6 usages
public class Results {
    2 usages
    public static int peakHour = 0;
    2 usages
    public static int  maxClients = 0;
    2 usages
    public static double calculateAverageWaitingTime(List<Server> servers, int totalClients) {...}
    1 usage
    public static double calculateAverageServiceTime(List<Task> tasks) {...}
    2 usages
    public static void findPeakHour(List<Server> servers, int simulationTime) {...}
}
```

4.8. SelectionPolicy

Enumul SelectionPolicy din pachetul businessLogic conține cele 2 strategii: Shortest_Queue și Shortest_Time. Acesta va fi folosit pentru selectarea strategiei curente.

```
package businessLogic;

7 usages
public enum SelectionPolicy {
    2 usages
    SHORTEST_QUEUE, SHORTEST_TIME
}
```

4.9.Strategy

Interfața Strategy din pachetul businessLogic definește un contract pentru implementarea diferitelor strategii de alocare a sarcinilor (Task) către servere (Server) în cadrul unui sistem de gestionare a cozilor. Această interfață este crucială pentru a permite flexibilitatea în modul în care sarcinile sunt distribuite între servere, facilitând implementarea diferitelor politici de scheduling fără a modifica logica centrală a simulatorului.

4.9.1. Metoda definită

- addTask(List<Server> servers, Task t): Această metodă este responsabilă pentru adăugarea unei sarcini la unul dintre serverele disponibile în listă, conform unei anumite strategii de alocare. Parametrii metodei includ:
    - servers: Lista de servere disponibile între care sarcina poate fi alocată.
    - t: Sarcina care trebuie adăugată în sistem.

4.9.2. Rolul interfetei Strategy

- *Abstracție*: Interfața Strategy abstrage detaliile de implementare ale diferitelor strategii de alocare, permițând sistemului principal să utilizeze diverse strategii fără a depinde de implementările specifice. Aceasta promovează principiul de design "open/closed", unde software-ul este deschis pentru extensie, dar închis pentru modificare.
- *Flexibilitate*: Prin folosirea acestei interfețe, sistemul poate schimba dinamic strategia de alocare a sarcinilor în funcție de condițiile de simulare sau parametri configurabili, adaptându-se la cerințe variate sau la scenarii diferite de test.

- *Usurința în testare și întreținere*: Implementările bazate pe o interfață comună sunt mai ușor de testat și de întreținut. De asemenea, permite dezvoltatorilor să adauge noi strategii fără a afecta restul codului.

4.9.3. Strategiile

Implementările interfetei Strategy sunt:

- Shortest Queue Strategy: Alege serverul cu cea mai scurtă coadă pentru a minimiza timpul de așteptare.
- Shortest Time Strategy: Alege serverul cu cel mai scurt timp estimat de așteptare, bazându-se pe timpul de serviciu al sarcinilor curente.

```
package businessLogic;

import model.Server;
import model.Task;

import java.util.List;

3 usages  2 implementations
public interface Strategy {
    2 usages  2 implementations
    public void addTask(List<Server> servers, Task t);
}
```

4.10. ShortestQueueStrategy

Clasa ShortestQueueStrategy din pachetul businessLogic implementează interfața Strategy și definește o strategie specifică pentru alocarea sarcinilor într-un sistem de cozi. Această strategie alege să aloce fiecare sarcină nouă serverului (sau cozii) care are cel mai mic număr de sarcini în așteptare la momentul respectiv. Scopul acestei abordări este de a balansa încărcătura între servere și de a minimiza timpul de așteptare pentru sarcinile noi adăugate.

Metoda *addTask* este implementarea concretă a logicii de distribuire a sarcinilor conform strategiei de coadă cea mai scurtă:

- Parametri:
  - *servers*: Lista de servere disponibile pentru alocarea sarcinilor.
  - *t*: Sarcina care trebuie adăugată.
- Proces:
  - Inițial, metoda identifică primul server din listă ca având coada cea mai scurtă.
  - Apoi, iterează prin lista de servere pentru a găsi serverul cu numărul cel mai mic de sarcini în așteptare. Acest lucru se face comparând numărul de sarcini ale fiecărui server cu numărul de sarcini ale serverului curent considerat a avea coada cea mai scurtă.

După identificarea serverului cu cea mai scurtă coadă, sarcina t este adăugată la acest server.

```java
package businessLogic;

import model.Server;
import model.Task;

import java.util.List;

2 usages
public class ShortestQueueStrategy implements Strategy {

    2 usages
    @Override
    public void addTask(List<Server> servers, Task t) {
        Server shortestQueue = servers.get(0);

        for (Server server : servers) {
            if (server.getTaskCount() < shortestQueue.getTaskCount()) {
                shortestQueue = server;
            }
        }
        shortestQueue.addTask(t);
    }
}
```

4.11. TimeStrategy

Clasa TimeStrategy din pachetul businessLogic este o implementare concretă a interfeței Strategy, specializată în alocarea sarcinilor (Task) pe baza timpului de așteptare minim. Această strategie își propune să minimizeze timpul de așteptare pentru fiecare sarcină prin alegerea serverului (Server) cu cel mai scurt timp estimat de așteptare totală în momentul alocării sarcinii.

Strategia se concentrează pe distribuirea eficientă a sarcinilor pentru a asigura o gestionare echilibrată a încărcării și a timpului de răspuns în sistem.

Metoda *addTask* este implementarea concretă a logicii de distribuire a sarcinilor conform strategiei de coadă cu timpul de așteptare cel mai mic:

- Parametri:
  - *servers*: Lista de servere disponibile în care sarcina poate fi alocată.
  - *t*: Sarcina care trebuie adăugată în sistem.
- Procesul de Alocare:
  - Inițializare: Se stabilește inițial primul server din listă ca având timpul de așteptare minim (minTimeQueue). Valoarea timpului de așteptare al acestui server este stocată în minTime.
  - Identificarea Serverului cu Timp Minim: Metoda iterează prin lista de servere pentru a identifica serverul cu cel mai mic timp de așteptare. Compară timpul de așteptare curent (serverTime) al fiecărui server cu minTime. Dacă găsește un server cu un timp mai mic decât minTime, actualizează atât minTimeQueue cât și minTime cu noile valori ale serverului respectiv.
  - Alocarea Sarcinii: După determinarea serverului cu timpul de așteptare minim, sarcina t este adăugată la acest server.

```java
package businessLogic;

import model.Server;
import model.Task;

import java.util.List;

1 usage
public class TimeStrategy implements Strategy {

    2 usages
    @Override
    public void addTask(List<Server> servers, Task t) {
        Server minTimeQueue = servers.get(0);
        int minTime = minTimeQueue.getWaitingTime().get();

        for (Server server : servers) {
            int serverTime = server.getWaitingTime().get();
            if (serverTime < minTime) {
                minTime = serverTime;
                minTimeQueue = server;
            }
        }
        minTimeQueue.addTask(t);
    }
}
```

4.12. Time Counter

Clasa TimeCounter este proiectată pentru a ține evidența timpului în cadrul aplicației. Aceasta utilizează un AtomicInteger pentru a menține și actualiza valorile timpului într-un mod thread-safe, asigurând că actualizările sunt atomic realizate și vizibile instant tuturor thread-urilor.

Funcționalități:

- *getTime():* Returnează valoarea curentă a timpului.
- *incrementTime():* Incrementează timpul cu o unitate. Această operație este thread-safe, ceea ce este esențial într-un mediu multi-threaded, cum ar fi o simulare de cozi care operează pe mai multe servere simultan.

```
package businessLogic;

import java.util.concurrent.atomic.AtomicInteger;

8 usages
public class TimeCounter {
    3 usages
    private AtomicInteger time;

    1 usage
    public TimeCounter() { this.time = new AtomicInteger( initialValue: 0); }

    7 usages
    public int getTime() { return time.get(); }

    2 usages
    public void incrementTime() { this.time.incrementAndGet(); }
}
```

4.13. TimeCounterSingleton

TimeCounterSingleton implementează pattern-ul de design Singleton pentru a asigura că există o singură instanță a clasei TimeCounter accesibilă global în întreaga aplicație. Aceasta garantează că toate componentele aplicației referă și utilizează același contor de timp, asigurând consistența și eliminând riscurile legate de discrepanțe temporale între diferitele părți ale aplicației. Funcționalități principale:

- getTimeCounter(): Furnizează accesul global la instanța unică a TimeCounter, permițând tuturor componentelor să acceseze și să utilizeze același cronometru.

În aplicația de simulare a gestionării cozilor, gestionarea corectă a timpului este esențială. Fiecare sarcină (Task) poate avea un timp de sosire și un timp de serviciu, iar serverele (Server) trebuie să proceseze sarcinile în concordanță cu aceste timpi. TimeCounter permite aplicației să avanseze secvențial prin fiecare unitate de timp, actualizând starea fiecărui server și sarcină într-un mod sincronizat.

Prin utilizarea unui singleton pentru TimeCounter, aplicația asigură că toate deciziile legate de procesarea sarcinilor sunt bazate pe același cadru temporal, evitând astfel erorile care ar putea rezulta din utilizarea multiple instanțe de timp diferit configurate.

```
package singleton;

import businessLogic.TimeCounter;

6 usages
public class TimeCounterSingleton {
    1 usage
    private static final TimeCounter timeCounter = new TimeCounter();

    3 usages
    public static TimeCounter getTimeCounter() { return timeCounter; }
}
```

5. Rezultate

În urma simulării sistemului de gestionare a cozilor, rezultatele obținute reflectă eficiența strategiilor implementate și capacitatea sistemului de a gestiona fluxurile variabile de clienți. Timpul mediu de așteptare indică o distribuție relativ echitabilă a sarcinilor între servere. Timpul mediu de serviciu demonstrează capacitatea serverelor de a procesa sarcinile eficient. Ora de vârf a fost indică momentul în care sistemul a gestionat cel mai mare număr de clienți. Aceste măsurători oferă o bază solidă pentru evaluarea performanței sistemului și pentru identificarea zonelor care necesită îmbunătățiri.

6. *Concluzii și dezvoltări ulterioare*

Analiza rezultatelor simulării sugerează că sistemul de gestionare a cozilor funcționează eficient sub parametrii actuali, dar există oportunități de optimizare. Pentru a îmbunătăți gestionarea vârfurilor de sarcini, ar putea fi implementarea mai multor strategii, pentru a eficientiza și mai tare distribuirea în cozi. În plus, implementarea unui sistem de predare a comportamentului clienților pe baza datelor istorice ar putea permite anticiparea și gestionarea mai eficientă a fluctuațiilor de aflux. Continuarea dezvoltării și integrarea feedbackului clienților vor fi esențiale pentru ajustările strategice și pentru asigurarea unei performanțe robuste în viitor.

7. *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-and-threadpoolexecutor.html