DOCUMENTAȚIE

TEMA 2

NUME STUDENT: Vicsi Márk-Ottó

GRUPA:30221

# CUPRINS

[1. Obiectivul temei 3](file:///C:\Users\bastard\AppData\Local\Temp\PT2021-2022_Documentation_Template.doc#_Toc95297885)

[2. Analiza problemei, modelare, scenarii, cazuri de utilizare 3](file:///C:\Users\bastard\AppData\Local\Temp\PT2021-2022_Documentation_Template.doc#_Toc95297886)

[3. Proiectare 4](file:///C:\Users\bastard\AppData\Local\Temp\PT2021-2022_Documentation_Template.doc#_Toc95297887)

[4. Implementare 3](file:///C:\Users\bastard\AppData\Local\Temp\PT2021-2022_Documentation_Template.doc#_Toc95297888)

[5. Rezultate 3](file:///C:\Users\bastard\AppData\Local\Temp\PT2021-2022_Documentation_Template.doc#_Toc95297889)

[6. Concluzii 3](file:///C:\Users\bastard\AppData\Local\Temp\PT2021-2022_Documentation_Template.doc#_Toc95297890)

[7. Bibliografie 3](file:///C:\Users\bastard\AppData\Local\Temp\PT2021-2022_Documentation_Template.doc#_Toc95297891)

# Obiectivul temei

Proiectarea și implementarea unei aplicații de gestionare a cozilor de așteptare care repartizează clienții pe cozi astfel încât timpul de așteptare să fie minimizat.

Cozile sunt utilizate în mod obișnuit pentru a modela domenii din lumea reală. Principalul obiectiv al unei cozi de așteptare este de a oferi un loc unde un "client" să aștepte înainte de a primi un "serviciu". Conducerea sistemelor bazate pe cozile de așteptare este interesată de minimizarea timpului în care "clienții" așteaptă în cozi înainte de a fi serviți. O modalitate de a minimiza timpul de așteptare este adăugarea mai multor servere, adică mai multe cozi de așteptare în sistem (se consideră că fiecare coadă are un procesor asociat), dar această abordare crește costurile furnizorului de servicii.

Obiective secundare:

* Capitolul 2: Analiza problemei și identificarea cerințelor (use case diagram, cerințe funcționale)
* Capitolul 3: Proiectarea calculatorului polinomial (alegerea structurilor de date, împărțirea pe clase)
* Capitolul 4: Implementarea aplicației de gestionare a cozilor de așteptare
* Capitolul 5: Rezultatele după testarea aplicației de gestionare a cozilor de așteptare

# Analiza problemei, modelare, scenarii, cazuri de utilizare

Cerințele funcționale sunt:

* Aplicația de simulare ar trebui să permită utilizatorilor să configureze simularea.
* Aplicația de simulare ar trebui să permită utilizatorilor să pornească simularea.
* Aplicația de simulare ar trebui să afișeze evoluția cozilor de așteptare în timp real.
* Aplicația de simulare ar trebui să afișeze rezultatele (timpul mediu de așteptare, timpul mediu de servire, ora de vârf) după simulare.

Diagrama de cazuri de utilizare (use case diagrams) este o sinteză vizuală a interacțiunilor și a relațiilor din cadrul unui sistem. Aceste diagrame arată o viziune foarte largă a unui sistem. O diagramă de caz de utilizare include câteva componente cheie, cum ar fi: actorii, care sunt persoane care au o relație în cadrul sistemului. Aceștia interacționează cu fiecare etapă a procesului. Cazurile de utilizare, adesea reprezentate printr-un oval sau un cerc care cuprind un text, descriu o funcție a sistemului pe care un actor sau o persoană o poate începe sau utiliza. Actorii se pot conecta la cazurile de utilizare și la alte persoane printr-o varietate de legături de comunicare care reprezintă diferite relații, cum ar fi: aserțiuni, generalizări, extinderi, incluziuni. (Team, 2021)

Diagram

Description automatically generatedExemplu de caz de utilizare: configurarea simulării

Actor primar: utilizatorul

Scenariu de succes:

1. Utilizatorul introduce datele necesare pentru rularea simulării: numărul clienților, numărul cozilor, timp de simulare, timp minim și maxim de sosire, durata minimă și maximă a serviciului.
2. Utilizatorul apasă butonul “Start simulation”.
3. Aplicația validează datele și afișează într-o nouă fereastră simularea.

Scenariu alternativ: argumente incorecte pentru configurarea simulării

* Utilizatorul introduce argumente incorecte pentru configurarea simulării ( numere nenaturale negative, timpii minimi sunt mai mari ca timpii maximi).
* Aplicația afișează un mesaj de eroare și solicită utilizatorului să introducă valori valide.
* Scenariul revine la subpunctul 1 din scenariul de succes precizat mai sus.

# Proiectare

**Modelul de proiectare MVC** (Model-View-Controller) atribuie obiectelor dintr-o aplicație unul dintre cele trei roluri: model, vizualizare (view) sau controler. Modelul definește nu numai rolurile pe care le joacă obiectele în aplicație, ci și modul în care obiectele comunică între ele. Fiecare dintre cele trei tipuri de obiecte este separat de celelalte prin limitări abstracte și comunică cu obiectele din celelalte tipuri prin aceste limitări. Colecția de obiecte de un anumit tip MVC dintr-o aplicație este uneori denumită strat / nivel - de exemplu, strat /nivel de model.

Avantajele adoptării acestui model sunt numeroase. Multe obiecte din aceste aplicații tind să fie mai ușor de reutilizat, iar interfețele lor tind să fie mai bine definite. De asemenea, aplicațiile care au un design MVC sunt mai ușor de dezvoltat decât alte aplicații.

Obiectele Model încapsulează datele specifice unei aplicații și definesc logica și calculul care manipulează și procesează aceste date. În acest caz, clasele care implementează datele aplicației sunt Task și Server. Mai multe despre acestea sunt discutate în capitolul următor. Aplicația este concepută astfel încât obiectul model să nu aibă o legătură explicită cu obiectele de vizualizare care prezintă datele sale și permit utilizatorilor să editeze aceste date - nu este preocupat de problemele legate de interfața cu utilizatorul și de prezentare.

Comunicarea: Acțiunile utilizatorului în stratul de vizualizare care creează sau modifică datele sunt comunicate prin intermediul unui obiect controler și au ca rezultat crearea sau actualizarea unui obiect model. Atunci când un obiect model se modifică (de exemplu, serverul a servit un client, apoi a fost șters din coadă) acesta notifică un obiect controler, care actualizează obiectele de vizualizare corespunzătoare.

Un obiect de vizualizare este un obiect dintr-o aplicație pe care utilizatorii îl pot vedea. Un obiect de vizualizare știe cum să se deseneze și poate răspunde la acțiunile utilizatorului. Unul dintre scopurile principale ale obiectelor de vizualizare este de a afișa datele din obiectele model ale aplicației și de a permite editarea acestor date. În ciuda acestui fapt, obiectele de vizualizare sunt de obicei decuplate de obiectele model într-o aplicație MVC. Aplicația este realizată cu Java Swing GUI widget toolkit. Un JFrame este folosit pentru introducerea datelor respectiv pentru afișarea simulării și a rezultatelor acesteia.

Comunicare: Obiectele de vizualizare află despre modificările aduse datelor modelului prin intermediul obiectelor controlerului aplicației și comunică modificările inițiate de utilizator - de exemplu, butonul ”Start simulation” a fost apăsat - prin intermediul obiectelor controlerului se transmite informația către obiectele modelului.

Un obiect controler acționează ca un intermediar între unul sau mai multe obiecte de vizualizare ale unei aplicații și unul sau mai multe obiecte model ale acesteia. Astfel, obiectele controler sunt un canal prin care obiectele de vizualizare află despre modificările din obiectele model și viceversa. De asemenea, obiectele controler pot îndeplini sarcini de configurare și coordonare pentru o aplicație și pot gestiona ciclurile de viață ale altor obiecte.

Comunicarea: Un obiect controler interpretează acțiunile utilizatorului efectuate în obiectele de vizualizare și comunică datele noi sau modificate la nivelul modelului. Atunci când obiectele modelului se modifică, un obiect controler comunică noile date ale modelului către obiectele de vizualizare, astfel încât acestea să le poată afișa (Apple Documentation Archive, 2018)

În programarea calculatoarelor**, modelul de strategie** (cunoscut și sub numele **policy pattern**) este un model de proiectare software comportamental care permite selectarea unui algoritm în timpul execuției. În loc să implementeze direct un singur algoritm, codul primește instrucțiuni în timpul execuției cu privire la care dintr-o familie de algoritmi să fie utilizat. Modelul de strategie este unul dintre tiparele incluse în cartea influentă Design Patterns de Erich Gamma, care a popularizat conceptul de utilizare a tiparelor de proiectare pentru a descrie modul de proiectare a software-ului flexibil și reutilizabil orientat pe obiecte. Amânarea deciziei cu privire la algoritmul care urmează să fie utilizat până la momentul execuției permite codului de apelare să fie mai flexibil și reutilizabil. Algoritmii de validare (strategiile), încapsulați separat de obiectul de validare, pot fi utilizați de alte obiecte de validare în diferite zone ale sistemului (sau chiar în sisteme diferite) fără duplicarea codului. În mod obișnuit, modelul de strategie stochează o referință la un anumit cod într-o structură de date și îl recuperează. Acest lucru poate fi realizat prin mecanisme precum pointerul de funcție nativ, funcția de primă clasă, clasele sau instanțele de clasă în limbajele de programare orientate pe obiecte sau accesarea stocării interne a codului de către implementarea limbajului prin reflecție. (Wikipedia, n.d.)

Aplicația implementează două strategii pentru gestionarea cozilor: ConcreteStrategyQueue și ConcreteStrategyTime. Cele două algoritmi returnează din lista de servere pe cel care are cel mai scurt timp de așteptare / coadă de așteptare. **Algoritmii** sunt implementați cu ajutorul Stream API-ului.

ConcreteStrategyQueue

Optional<Server> serverOptional = servers.stream().min(Comparator.comparingInt(Server::getCurrentTasksNumber));

serverOptional.ifPresent(server -> server.addTask(task));

ConcreteStrategyTime

Optional<Server> serverOptional = servers.stream().min(Comparator.comparingInt(Server::getWaitingPeriod));

serverOptional.ifPresent(server -> server.addTask(task));

Fiecare metodă creează un stream de obiecte din argumentul “servers” și încearcă să returneze elementul minim din lista respectivă prin compararea acestora cu proprietatea specificată a serverului cu ajutorul clasei Comparator. Apoi, algoritmii verifică dacă există un server corespunzător și adaugă task-ul la server.

Ca **structură de dată** principală aplicația folosește interfața BlockingQueue și implementarea ei ArrayBlockingQueue. Este un Queue în Java care suportă operații, care așteaptă ca coada să nu fie goală atunci când extrage și elimină un element și așteaptă ca spațiul să devină disponibil în coadă atunci când adaugă un element. Implementările Java BlockingQueue sunt thread-safe. Toate metodele de coadă sunt de natură atomică și utilizează “lock”-uri interne sau alte forme de control ale concurenței. Fiecare server are la dispoziție o structură de dată de acest tip.

**Diagrama de clase**

A screenshot of a video game

Description automatically generated

# Implementare

**Clasa DataInputWindow**

Este o subclasă de tip JFrame, conține 7 câmpuri de intrare pentru cele 7 date necesare pentru configurarea simulării (numărul clienților, numărul cozilor, timp de simulare, timp minim și maxim de sosire, durata minimă și maximă a serviciului). Are un buton cu textul “Start simulation”. Butonul este dotat de un ActionListener: când clientul îl apasă, metodele *getInputs()* și *checkInput()* sunt apelate. Dacă utilizatorul a introdus date de configurare greșite ( de exemplu un număr zecimal sau negativ, timpul minim de sosire este mai mare decât timpul maximal de sosire), aplicația afișează un mesaj de eroare prin care avertizează utilizatorul. Clasa are un atribut de tip boolean “*simulationNotStarted*”, care are valoarea adevărată / true în mod implicit și este setat ca fals / false când se apasă butonul de pornire. Această proprietate a obiectului este verificată în fiecare 100 de milisecunde de către aplicație și nu pornește simulația până când aceasta nu are valoare falsă.

**Clasa SimulationFrame**

Este o subclasă de tip JFrame, conține o zonă de text care poate fi derulată vertical. Aici, prin metoda *updateTextArea()* obiectul care apelează poate seta zona de text a cadrului. Tot aici sunt afișate și rezultatele după simulare.

**Clasa Server**

Serverul conține un BlockingQueueQueue în care sunt salvate clienții care aștept să fie serviți. Serverul verifică în fiecare secundă dacă în coadă se află sau nu un client. În caz afirmativ, decrementează timpul de servire a clientului aflat la capătul cozii. Atributul waitingPeriod arată cât timp îi ia serverului să servească toți clienții, adică de cât timp este nevoie pentru ca coada de așteptare să devină goală. În cazul în care clientul care urmează să fie servit are un timp de serviciu de 1 secundă, acesta este scos automat din coada de așteptare de către server. Fiecare server la adăugarea unui nou client salvează datele necesare pentru a calcula ulterior rezultatele simulării. În lista *waitingTimes* sunt salvate timpii pe care clienții sunt nevoiți să îi așteapte până să părăsească coada ( adică timpul de așteptare la coada respectivă + timpul de serviciu a clientului). În lista *serviceTimes* sunt salvate timpii necesari de servire a fiecărui client. Variabila *peakHour* salvează timpul în care serverul a fost cel mai ocupat, adică atunci când coada de așteptare era cea mai lungă. Aceste date sunt la sfârșitul simulării adunate de Scheduler, care cu ajutorul acestora calculează timpul de așteptare mediu, timpul de servire mediu și ora de vârf. Fiecare server are o referință la variabila *currentTime* a unui obiect Scheduler. Această variabilă este împărtășită pentru toate serverele și servește ca timp global.

**Clasa Task**

Este clasa care modelează clienții.

**Interfața Strategy**

Conține o singură metodă *add()* care este implementată de clasele ConcreteStrategyTime și ConcreteStrategyQueue folosindu-se de algoritmii menționați în capitolul 2, secția algoritmi.

**Clasa Scheduler**

Este clasa care este responsabilă pentru instanțierea, implementarea și gestionarea serverelor. La crearea unui nou obiect de tip Scheduler, sunt instanțiate implicit și serverele. La apelarea comenzii *startAll()* fiecare server din listă este rulată în cadrul unui nou fir de execuție (thread). La apelarea comenzii *killAll()* fiecare server din listă este întreruptă de Scheduler. Ultimele două metode menționate sunt folosite de SimulationManager pentru rularea simulării. Funcția *dispatchTask()* adaugă un client la a o coadă selectată de Scheduler după strategia predefinită în cod sursă. Variabila *currentTime* este împărtășită pentru toate serverele și servește ca timp global.

**Clasa SimulationManager**

Este clasa care se ocupă de rularea simulării. La crearea unui nou obiect de tip SimulationManager, prin intermediul datelor primite de la GUI, în constructor este creat un nou obiect de tip Scheduler, care inițiază serverele. Totodată, sunt generate și utilizatori aleatorii în funcție de parametrii introduși de utilizator. Când obiectul te tip SimulationManager este rulat ca fir de execuție în cadrul funcției main din clasa App, aceasta începe simularea. În fiecare secundă adăugă clienții în serverele potrivite cu ajutorul unui Scheduler. Apoi afișează starea curentă a simulării pe ecran. Când simularea s-a terminat, obiectul de tip Scheduler oprește serverele, adună datele salvate de acestea și calculează rezultatele finale.

# Rezultate

Aplicația a fost testată în trei medii diferite, fiecare cu parametri diferiți, și în fiecare caz a trecut testele. Rezultatele simulării sunt înregistrate într-un fișier, aici este atașat doar un exemplu simplu.

Time 0  
Waiting clients: (0, 2, 3);(1, 2, 2);(2, 2, 3);(3, 3, 2);(4, 4, 3)  
Queue 0:closed  
Queue 1:closed  
Time 1  
Waiting clients: (0, 2, 3);(1, 2, 2);(2, 2, 3);(3, 3, 2);(4, 4, 3)  
Queue 0:closed  
Queue 1:closed  
Time 2  
Waiting clients: (3, 3, 2);(4, 4, 3)  
Queue 0:( 0, 2, 3);  
Queue 1:( 1, 2, 2); ( 2, 2, 3);  
Time 3  
Waiting clients: (4, 4, 3)  
Queue 0:( 0, 2, 2); ( 3, 3, 2);  
Queue 1:( 1, 2, 1); ( 2, 2, 3);  
Time 4  
Waiting clients:  
Queue 0:( 0, 2, 1); ( 3, 3, 2); ( 4, 4, 3);  
Queue 1:( 2, 2, 3);  
Time 5  
Waiting clients:  
Queue 0:( 3, 3, 2); ( 4, 4, 3);  
Queue 1:( 2, 2, 2);  
Time 6  
Waiting clients:  
Queue 0:( 3, 3, 1); ( 4, 4, 3);  
Queue 1:( 2, 2, 1);  
Time 7  
Waiting clients:  
Queue 0:( 4, 4, 3);  
Queue 1:closed  
Time 8  
Waiting clients:  
Queue 0:( 4, 4, 2);  
Queue 1:closed  
Time 9  
Waiting clients:  
Queue 0:( 4, 4, 1);  
Queue 1:closed  
Time 10  
Waiting clients:  
Queue 0:closed  
Queue 1:closed  
Avg waiting time: 4.2  
Avg service time: 2.6  
Peak hour: 4

1. Concluzii

Aplicația de gestionare a cozilor de așteptare a trecut faza de testare, gestionând chiar și cazurile cu mai multe cozi și mulți clienți. Funcționalități care ar putea fi incluse în proiect:

* O interfață mai intuitivă (aranjarea etichetelor de text și a câmpurilor de intrare, aranjarea textului pe ecran în timpul simulării pentru cozi de așteptare lungi).
* Mutarea fișierelor de logare într-un folder separat.

# Bibliography

*Apple Documentation Archive*. ( 2018, 04 06). Retrieved from Cocoa Core Competencies: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

Team, I. E. (2021, April 22). *Indeed*. Retrieved from https://www.indeed.com/career-advice/career-development/use-case-diagram

*Wikipedia*. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Strategy\_pattern