DOCUMENTATIE

TEMA 2

NUME STUDENT: Codorean Luca-Andrei

GRUPA: 30223

**CUPRINS**

[ Obiectivul temei 3](#_Toc163119805)

[ Analiza problemei, modelare, scenarii, cazuri de utilizare 3](#_Toc163119806)

[ Proiectare 6](#_Toc163119807)

[ Implementare 11](#_Toc163119808)

[ Rezultate 14](#_Toc163119809)

[ Concluzii 15](#_Toc163119810)

[ Bibliografie 15](#_Toc163119811)

# Obiectivul temei

Prezenta aplicație are ca scop principal simularea unui sistem de gestiune a unor cozi astfel încât timpul de așteptare corespunzător unui client este redus la minim. Aplicația generează clienții pe baza unor date de intrare transmise prin intermediul unei interfețe grafice.

Din punct de vedere tehnic, pentru implementarea cerinței se dorește folosirea unui limbaj de programare care se încadrează în paradigma programării orientate pe obiect, precum Java.

# Analiza problemei, modelare, scenarii, cazuri de utilizare

Întrucât algoritmul de gestiune trebuie să fie capabil să realizeze o distribuție cât mai corectă, este necesară definirea unor strategii eficiente de distribuție. Astfel, sunt definite două strategii:

1. Timp minim; respectiv
2. Lungime minimă.

Scenarii de distribuire a clienților:

1. Algoritmul de gestiune are ca strategie fundamentală distribuirea pe baza timpului minim. Această strategie presupune plasarea clientului curent în coada care are timpul de așteptare minim.
2. În cazul în care algoritmul nu reușește să găsească o coadă care să respecte strategia fundamentală, strategia de distribuire a clientului curent se schimbă în lungime minimă. Noua strategie presupune găsirea cozii care are lungime minimă.
3. În cazul în care nu se găsește o coadă cu lungime minimă în urma scenariului II, clientul va fi distribuit în prima coadă determinată în scenariul I.

Pentru o mai bună înțelegere a problemei, sunt necesare stabilirea unor cerințe funcționale, respectiv non-funcționale.

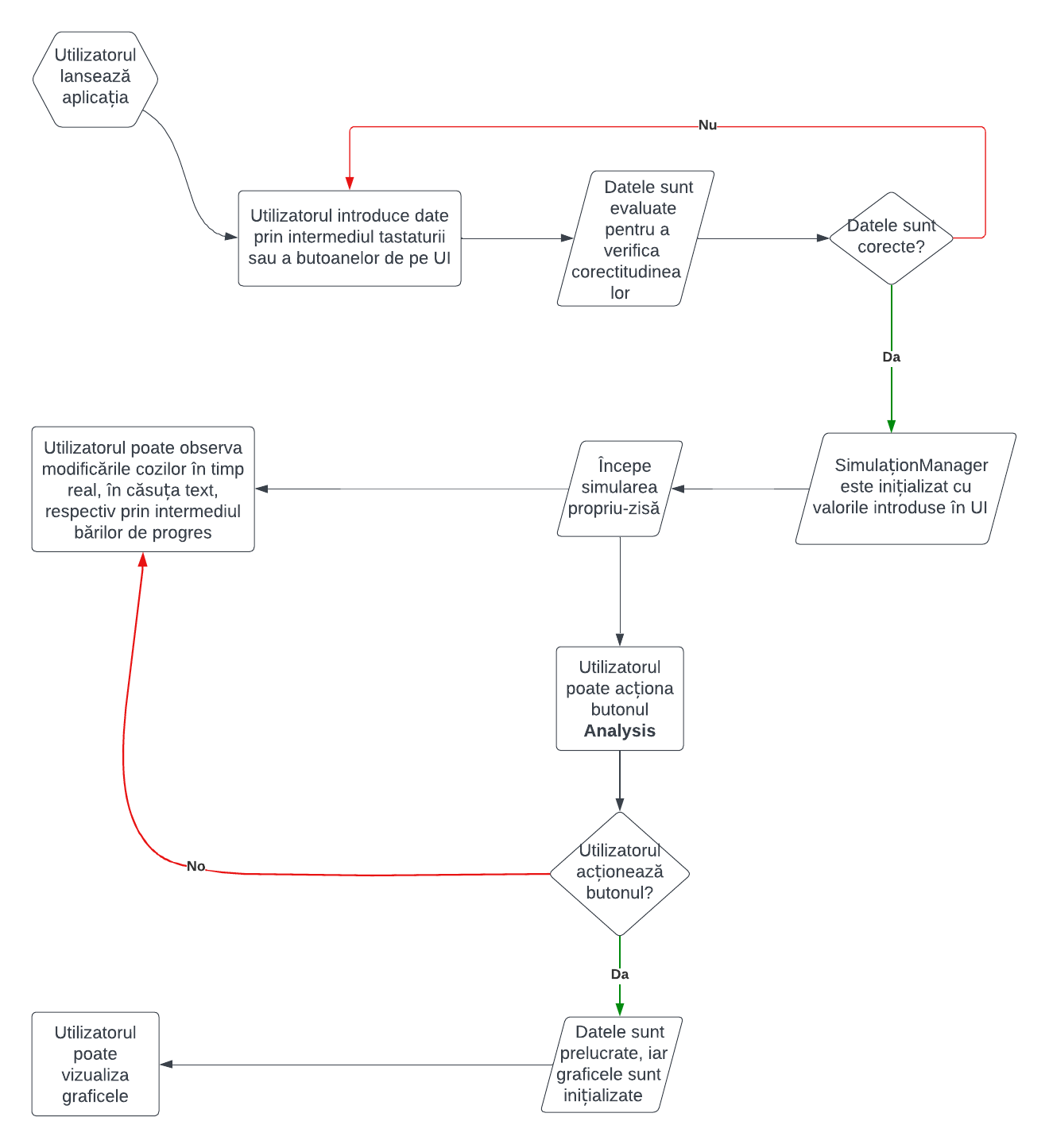
1. Cerințe funcționale

|  |  |
| --- | --- |
| **Nr crt.** | **Descrierea cerinței** |
| 1 | Utilizatorul ar trebui să aibă spații care să îi permită introducerea datelor de intrare. |
| 2 | Utilizatorul ar trebui să aibă o modalitate prin intermediul căreia să începe simularea. |
| 3 | Utilizatorul ar trebui să aibă o modalitate prin intermediul căreia să începe vizualizeze datele generate și evoluția lor grafic |
| 4 | Utilizatorul ar trebui să aibă o modalitate prin intermediul căreia să începe vizualizeze evoluția datelor în timp real. |

1. Cerințe non-funcționale

|  |  |
| --- | --- |
| **Nr crt.** | **Descrierea cerinței** |
| 1 | Aplicația ar trebui să aibă un timp de răspuns relativ mic. |
| 2 | Aplicația ar trebui să fie capabilă să gestioneze cazurile în care datele de intrare primite de la utilizator sunt incorecte, și în acest caz să informeze utilizatorul. |
| 3 | Aplicația trebuie să fie sigură – prin intermediul inputului, utilizatorul nu ar trebui să poată modifica codul sursă |
| 4 | Aplicația ar trebui să aibă niște cerințe de sistem minime, astfel încât să poată să fie rulată pe orice dispozitiv pe care este accesată. Timpul său de reacție nu ar trebui să fie foarte influențat de componentele hardware ale dispozitivului pe care este rulată. |

Procesul prin care datele de intrare sunt prelucrate este descris de următoarea diagramă. Această diagramă prezintă întreg fluxul de date. De asemenea, este descrisă inclusiv modalitatea de reacția a aplicației în cazul în care datele introduse sunt incorecte din punct de vedere sintactic.



Figură 1. Diagrama fluxului datelor.

*Cazuri de utilizare*

Caz de utilizare: inițializare simulare

Actor primar: utilizatorul

Pași către succes:

1. Utilizatorul introduce date în căsuțele de text de pe UI.
2. Utilizatorul apasă pe butonul „Run Simulation”
3. Datele introduse de către client sunt verificate și validate de către controller.

* Minimum Processing Time < Maximum Processing Time;
* Minimum Arrival Time < Maximum Arrival Time
* Toate căsuțele text sunt completate.

Pași către eroare:

1. Utilizatorul apasă pe butonul „Run simulation” prea devreme.
2. Utilizatorul completează căsuțele de text parțial.
3. Utilizatorul completează în totalitate căsuțele text, însă invalid.

Caz de utilizare: vizualizare simulare

Actor primar: utilizatorul

Pași către succes:

1. Utilizatorul urmărește actualizarea în timp real a căsuței text și a progressbar-urilor.
2. Utilizatorul apasă pe butonul „Analysis”.
3. Utilizatorul vizualizează graficele care descriu datele generate de simularea curentă.
4. Utilizatorul deschide fișierul text în care se află date legate de simularea curentă.

Pași către eroare:

1. Utilizatorul apasă pe butonul „Analysis” înainte de inițializarea simulării.
2. Utilizatorul nu așteaptă terminarea simulării pentru deschiderea fișierului corespondent simulării curente.
3. Utilizatorul deschide fișierul greșit și vizualizează date legate de o simulare anterioară.

Dependența celor două cazuri de utilizare este descrisă de diagrama:

O imagine care conține text, diagramă, captură de ecran, cerc

Descriere generată automat

Figură 2. Diagrama de usecase

# Proiectare

Pentru ca principiile programării orientate pe obiect să fie respectate în totalitate. În consecință, un o coadă va fi descrisă sub forma unui obiect de tip Server, iar clienții vor fi instanțe a unei clase numite Task. Astfel, între clasa Server și clasa Task se stabilește o relație de dependență.

Pentru managementul dependențelor proiectului, se folosește Maven. Maven reprezintă un instrument de project management, capabil să gestioneze inclusiv dependențele proiectului. În cazul de față, dependența principală este cea folosită pentru conceperea interfeței grafice, anume JavaFX.

În ceea ce privește structura aplicației, este implementată și respectată arhitectura MVC. Eficiența acestui tip de arhitectură este dată de oportunitatea de a refolosi codul scris, pe care o oferă dezvoltatorului de software. Acest pattern este descris de imaginea atașată mai jos:



Figură 3. Descrierea arhitecturii Model View Controller

Pentru a susține modelul MVC și principiile SOLID, funcționalitatea a fost împărțită în pachete și subpachete. Prin urmare, logica este distribuită astfel:

backend

controllers

models

services

repositories

policies

singletons

resources

Structurarea claselor de implementare a logicii de calcul în repositories și services reprezintă un mecanism suplimentar de abstractizare a datelor. De asemenea, oferă dezvoltatorului oportunitatea de a-și organiza produsul astfel încât principiile SOLID să fie respectate.

Pachetul backend reprezintă spațiul dedicat implementării logicii din spatele interfeței grafice. Aici se regăsesc toate clasele care îndeplinesc funcții de manipulare a datelor introduse de către utilizator.

O imagine care conține text, diagramă, captură de ecran, linie

Descriere generată automat

Figură 4. Implementarea pachetului controllers

* Diagrama UML de mai sus prezintă implementarea pachetului controllers, care realizează legătura funcțională dintre logica de business și interfața grafică. Datele sunt aduse în interfață prin intermediul clasei statice StateManager, care are rol de distribuire a datelor prin intreaga logică dezvoltată.
* Pachetul models găzduiește definirea obiectelor (conceptului) de Server și Task;

O imagine care conține text, captură de ecran, diagramă, Font

Descriere generată automat

Figură 5. Implementarea pachetului models.

* repositories este pachetul în care este implementată logica de distribuire, și algoritmul propriu-zis de manipulare a datelor, respectiv funcționalitatea claselor model. Aici există un *contract* pe care fiecare repository al fiecărei clase model trebuie să-l respecte. Implementarea propriu-zisă a acestora se realizează în sub-pachetul repositories.implementation.

O imagine care conține text, captură de ecran, Font, Paralel

Descriere generată automat

Figură 6. Clasă implementată în pachetul repositories

* services reprezintă pachetul prin care se realizează efectiv logica de business. Aici, fiecare clasă model are un service, prin intermediul căruia este manipulat repository-ul corespondent acesteia.

O imagine care conține text, captură de ecran, diagramă, linie

Descriere generată automat

Figură 7. Implementarea pachetului services

* singletons este pachetul care găzduiește implementarea claselor responsabile cu inițializarea, respectiv manipularea claselor din pachetele de repositories, respectiv services.

O imagine care conține text, captură de ecran, Font, număr

Descriere generată automat

Figură 8. Implementarea pachetului singletons

* policies reprezintă pachetul care găzduiește implementarea policy-urilor. În logica dezoltată, un policy reprezintă un set de restricții în baza căruia aplicația adoptă un anume comportament. De pildă, aici se găsesc dezvoltările celor două strategii de distribuire a obiectelor de tip Task, în obiectele de tip Server.

O imagine care conține text, captură de ecran, linie, Font

Descriere generată automat

Figură 9. Implementarea pachetului policies

# Implementare

Structurarea în pachete precum repositories și services oferă un nivel avansat de abstractizare a datelor. De asemenea, respectarea principiilor SOLID presupune separarea funcționalității astfel încât fiecare clasă, respectiv metodă să îndeplinească doar o singură funcționalitate, sau, dacă este necesar, o gamă de funcționalități din aceeași arie.

Unul dintre principalele principii care întărește acest tip de organizare este *Open-Closed Principle[[1]](#footnote-1).*

Tabelul următor surprinde cele mai importante clase, respectiv rolul lor în logica aplicației.

|  |  |  |
| --- | --- | --- |
| **Nr. crt** | **Numele clasei / interfeței** | **Descrierea metodelor importante** |
| 1 | SimulationManagerRepository (i) | Are rol de a inițializa simularea și a genera datele necesare simulării. Procesul de simulare propriu-zis este descris în metoda run().  De asemenea, are rol de a inițializa și fișierul în care se salvează date asociate cu simularea. Tot el este însărcinat cu scrierea în fișier, prin intermediul metodei writeAnalysis() |
| 2 | SchedulerRepository (i) | Preia task-urile generate de simulator și ulterior le distribuie, prin intermediul metodei dispatchTask() către cel mai convenabil server. |
| 3 | StateManagerSingleton | Este o colecție de gettere și settere care au rolul de a stabili o cale de transmitere a datelor între interfața grafică și pachetele de backend. |
| 4 | TimeStrategyPolicy (i) | Implementează mecanismul în baza căruia se realizează distribuirea unui task pentru a obține timpul de așteptare minim.  *(în mod analog funcționează ShortestQueueStrategyPolicy)* |
| 5 | SimulationManagerService (i) | Descrie o serie de metode prin intermediul cărora sunt accesate metode ce aparțin SimulationManagerRepository.  *(în mod analog funcționează SchedulerService)* |
| 6 | ServiceSingleton | Prezintă modalitatea de a apela service-urile implementate în aplicație, într-o manieră statică, cu scopul de a evita inițializarea mai multor obiecte de acest tip, la fiecare apel.  Prin această metodă, întreaga aplicație funcționează pe baza unui singur obiect din fiecare obiect tip de service.  *(în mod analog funcționează RepositorySingleton, respectiv PolicySingleton)* |
| 7 | HelloController | Are rolul de a prelua datele de la interfața grafică, respectiv redirecționarea lor către StateManager. De asemenea, acesta are rol de a genera anumite obiecte de pe interfața grafică, în manieră dinamică.  În metodele simulate(), respectiv openAnalysis() se găsesc conexiuni către pachetele de backend responsabile cu începerea simulării, respectiv deschiderea unei ferestre ghidate de ChartViewController. |
| 8 | ChartViewController | Are rolkul de a dispune, sub formă grafică. prin intermediul metodei populateChart(), datele colectate de StateManager.  Adăugarea unor noi grafice este ușoară, întrucât este necesară doar modificarea acestui controller. |

Interfața grafică a fost realizată cu ajutorul Java Scene Builder, un software dedicat realizării interfețelor grafice pentru aplicații Java. În sine, interfața este un fișier de tip XML, în care sunt regăsite detalii despre totalitatea obiectelor ce constituie UI.

În esență, interfața grafică este formată din două pagini:

1. Interfața principală de manipulare a datelor; respectiv
2. Interfața de vizualizare a tabelelor.

O imagine care conține text, captură de ecran, software, Software multimedia

Descriere generată automat

Figură 10. Interfața principală.

*O imagine care conține text, captură de ecran, software, proiectare

Descriere generată automat*

Figură 11. Interfața grafică la momentul 21 a simulării.

*O imagine care conține text, captură de ecran, Interval, diagramă

Descriere generată automat*

Figură 12. Vizualizarea unui grafic generat.

# Rezultate

Aplicația returnează mai multe tipuri de rezultate. Astfel, există:

1. Rezultate observabile în timp real, pe interfața grafică.
2. Rezultate post-simulare, în directorul resources/simulations; respectiv
3. Rezultate dispuse sub formă tabelară.

Salvarea fișierelor post-simulare este realizată sub forma unui document cu extensia .txt, care are un nume standard. Astfel, fiecare document din această categorie respectă structura numelui:

simulation\_report\_xxxxxxxx\_xxxx\_xxxx\_xxxx\_x.txt, unde

structura xxxxxxxx\_xxxx\_xxxx\_xxxx\_x reprezintă o cheie generată random pentru fiecare simulare rulată. Conținutul unui asemenea fișier se aseamănă cu:

O imagine care conține text, captură de ecran, Font

Descriere generată automat

Figură 13. Conținutul unui fișier de timp simulation\_report.

Pentru rezultatele descrise de punctul 1 și 2, se pot verifica figurile 11, respectiv 12.

# Concluzii

În concluzie, aplicația reușește cu succes să implementeze simularea unei distribuiri de clienți în cozi. Din punct de vedere tehnic, aplicația ar trebui să răspundă oricărei operații într-un timp relativ scurt, aproape insesizabil utilizatorului (a nu se lua în calcul timpul în care se așteaptă ca un client să elibereze coada).

Posibile dezvoltări ulterioare versiunii curente ar fi implementarea mai multor grafice, care să permită o vizualizare mai clară a datelor, eventual și în timp real. Din punct de vedere a interfeței grafice, s-ar putea lucra la o variantă mai prietenoasă cu temele sistemelor de operare curente, întrucât actuala formă a aplicației are un format trivial, totuși capabil să răspundă cerințelor clientului.

Din perspectivă personală, acest assignment reprezintă o modalitate de foarte bună de asimilare de noi cunoștințe în ceea ce privește manipularea thread-urilor. Mai mult decât atât, arhitectura MVC și structura pachetelor a fost un exercițiu bun pentru startul unor proiecte mai complexe. Consider familiarizarea cu aceste concepte extrem de utilă, atât din punct de vedere educațional, cât și din punct de vedere profesional.

# Bibliografie

1. [*https://lucid.app/*](https://lucid.app/)
2. [*https://docs.oracle.com/javafx/2/charts/css-styles.htm*](https://docs.oracle.com/javafx/2/charts/css-styles.htm)
3. [*https://docs.oracle.com/javase/8/javafx/api/javafx/scene/chart/XYChart.Data.html*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/chart/XYChart.Data.html)
4. [*https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/VBox.html#VBox--*](https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/VBox.html#VBox--)
5. [*https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/progress.htm*](https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/progress.htm)

1. Principiul Open-closed presupune că entitățile (în cazul de față clasele) trebuie să fie disponibile pentru extindere, însă în cazul modificărilor, ele trebuie făcute intern, astfel încât acestea să nu afecteze structura, respectiv logica. [↑](#footnote-ref-1)