DOCUMENTAȚIE

TEMA 2

Nume Student: IRINI KARINA

Grupa: 30229

CUPRINS:

1. Obiectivul temei

Obiectivul principal al temei este acela de proiectare și implementare al unei aplicații de gestionare a cozilor care atribuie clienților cozi astfel încât timpul de așteptare să fie minimizat. Aplicația trebuie sa fie însoțită de o interfață grafică dedicată prin care utilizatorul poate introduce datele de intrare și vizualiza rezultatul simulării.

Cozile sunt folosite în mod obișnuit pentru a modela domenii din lumea reală. Obiectivul principal al unei cozi este de a oferi un loc pentru un "client" pentru a aștepta înainte de a primi un "serviciu". Managementul sistemelor bazate pe cozi este interest să minimizeze timpul în care "clienții" lor așteaptă la cozi inainte de a fi serviți. O modalitate de a minimiza timpul de așteptare este de a adăuga mai mulți servitori, adică mai multe cozi în system (fiecare coadă este considerate ca având un processor asociat), dar această abordare crește costurile furnizorului de servicii.

Aplicația de gestionare a cozilor trebuie să simuleze (prin definirea unui timp de simulare $t_{simulation}$) o serie de N clienți care vin pentru serviciu, intră în Q cozi, așteaptă, sunt serviți și, în cele din urmă, părăsesc cozile. Toți clienții sun generați când simularea este pornită. Aplicația urmărește timpul total petrecut de fiecare client în cozi și calculează timpul mediu de așteptare. Fiecare client este adăugat la coada cu timpul minim de așteptare când timpul său de sosire este mai mic sau egal cu timpul de simulare.

Pentru a îndeplini obiectivul principal al temei, trebuie să ne axăm și pe obiectivele secundare. Este important să începem cu analiza problemei și identificarea cerințelor. Această etapă ne ajută să ne concentrăm asupra funcționalităților de bază ale aplicației de gestionare a cozilor, aspect detaliat în capitolul următor. Proiectarea aplicației, descrisă în cel de-al treilea capitol, este esențială pentru a începe implementarea programului. Aceasta constă în realizarea generării clienților, asignarea unei cozi, servirea și părăsirea în cele din urmă a acestora, precum și a unei interfețe grafice. Este necesară realizarea acesteia într-un mod intuitiv, pentru a asigura o utilizare cât mai plăcută a software- ului. Ultimul pas, dezvoltat în capitolul cinci, asigură buna funcționare a aplicației, fiind reprezentat de testarea codului creat.

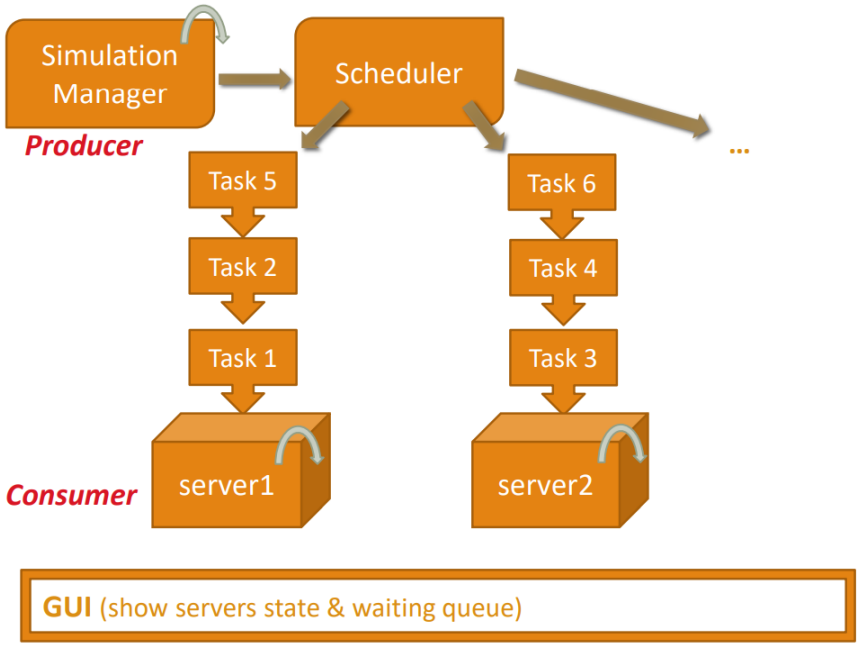
2. Analiza problemei, modelare, scenarii, cazuri de utilizare

Cerințele funcționale sunt cele care descriu funcționalitățile pe care programul trebuie să le îndeplinească, iar cerințele non funcționale sunt cele care descriu caracteristicile programului.

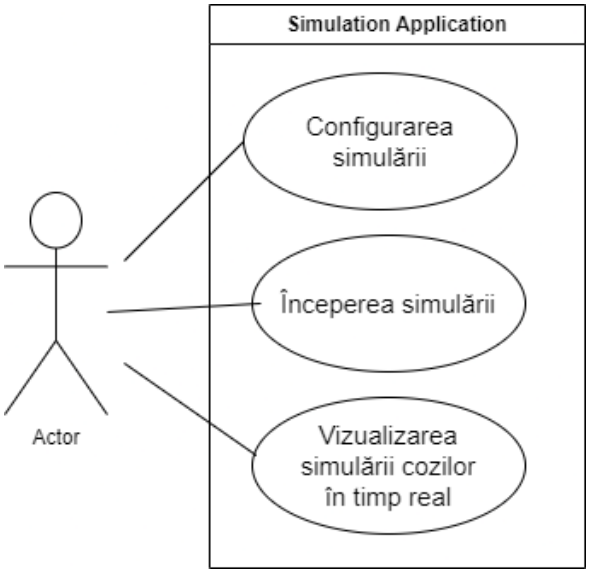
Cerințele problemei subliniază dezvoltarea unei interfețe grafice prin care utilizatorul să aibă posibilitatea de a introduce datele de intrare, de a selecta butonul de începere a simulării și de a vizualiza evoluția cozilor în timp real. Aplicația trebuie să folosească algoritmi potriviți, astfel încât toate aceste operații să fie realizate cu succes, iar utilizatorul să poată observa rezultatul generat sub o formă cât mai clară.

Una dintre cerințele non funcționale esențiale constă în utilizarea cât mai ușoară și intuitivă a programului. Astfel, utilizatorul nu ar trebui să reușească să introducă date de intrare invalide. În cazul acestor scenarii, el va fi avertizat de greșeala făcută, prin mesaje sugestive. O altă cerință non funcțională este reprezentată de fiabilitatea programului, siguranța acestuia față de posibilele erori în timpul rulării.

Pentru a înțelege mai bine modelarea calculatorului de polinoame, am folosit următoarea schemă:



Pentru a descrie funcționalitatea unei aplicații, este importantă evidențierea diferitelor cazuri de utilizare ale acesteia. Pentru aplicația de simulare, funcționalitatea se bazează pe operațiile implementate, care trebuie să ofere utilizatorului un rezultat final satisfăcător.



Scenariu de utilizare: Configurarea simulării

Actorul principal: Utilizatorul

Scenariul principal de succes:

1. Utilizatorul introduce valorile pentru: numărul de clienți, numărul de cozi, intervalul de simulare, timpul minim și maxim de sosire și timpul minim și maxim de servire și selectează politica de simulare.

2. Utilizatorul apasă butonul de validare a datelor de intrare.
3. Aplicația validează datele și începe simularea.

Scenariu alternativ: Valori invalide pentru parametrii de configurare.

- Utilizatorul introduce valori invalide pentru parametrii de configurare a aplicației.
- Aplicația afișează un mesaj de eroare și solicită utilizatorului să introducă valori valide.
- Scenariul se întoarce la pasul 1.

3. Proiectare

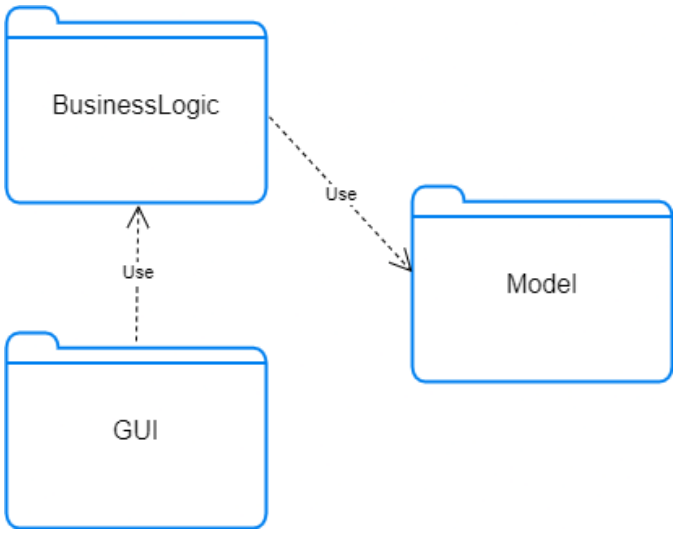
Programarea orientată pe obiecte (OOP) este o paradigmă de programare ce se bazează pe conceptul de obiecte, componente software care incorporează atât atributele cât și operațiile care se pot efectua asupra câmpurilor. Unul dintre principiile fundamentale ale OOP este încapsularea datelor, ascunderea detaliilor unui obiect și, astfel, protejarea informațiilor. Prin utilizarea principiilor OOP se beneficiază de o dezvoltare mai rapidă și mai eficientă a software-ului dorit, cu o structură clară și organizare bună a datelor. Acest aspect implică identificarea obiectelor și a interacțiunilor dintre acestea, definirea claselor și a relațiilor dintre ele, implementarea metodelor și a datelor pentru fiecare clasă, precum și testarea codului.

Astfel, pentru această tema am ales să utilizez două clase principale pentru modelarea obiectelor:

1. Clasa Task: Această clasă este corespondenta unui client. Astfel acesta este caracterizat de un ID unic, timpul de sosire și timpul necesar servirii acestuia. Primele două sunt reprezentate de numere întregi, fiind definite prin tipul primitiv int. Al treilea câmp este definit prin AtomicInteger pentru a oferi siguranță thread-urilor, fără a utiliza sincronizare.

2. Clasa Server: Această clasă este corespondenta unei cozi de magazin. Astfel aceasta este caracterizată de o coadă de clienți, task-uri definită prin structura de date BlockingQueue, care suportă modelele de proiectare producător-consumator și de o perioadă de așteptare de tipul AtomicInteger. Structurile de date utilizare facilitează siguranța firelor de execuție, fiind mai performate decât colecțiile sincronizate. Clasa implementează interfața Runnable, fapt care necesită suprascrierea metodei run, metodă care face posibilă procesarea clienților.

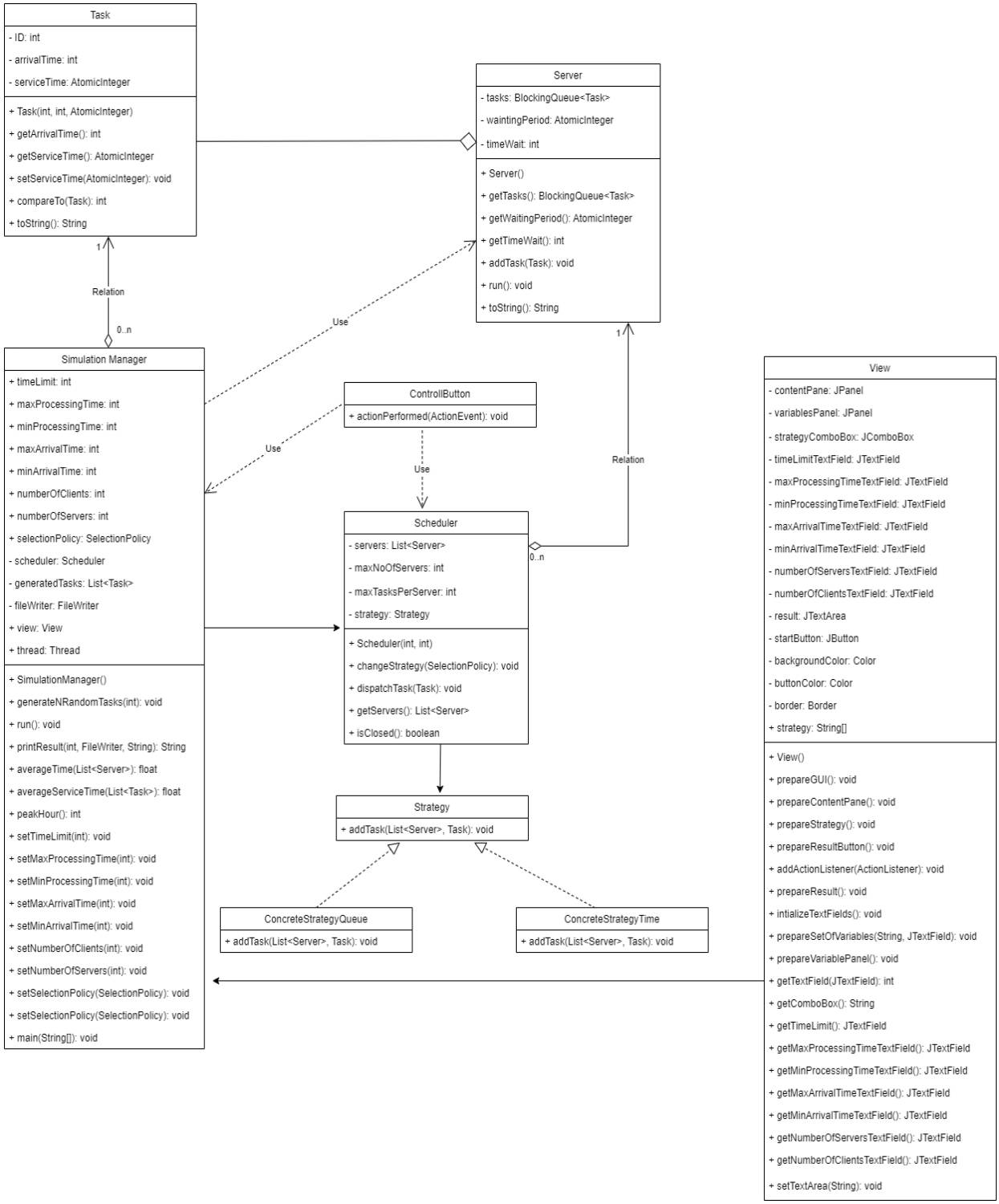
Proiectul este de asemenea împărțit în mai multe pachete:



Din diagrama UML de pachete se poate observa că între pachetele Graphical User Interface, respectiv pachetul Business Logic există o relație de dependență , analog între pachetul Business Logic și pachetul Data Models.

1. Pachetul Business Logic: Acest pachet conține două clase corespunzătoare strategiilor de așezare a clienților la cozi: ConcreteStrategyTime, respectiv ConcreteStrategyQueue, o clasă Scheduler și o clasă SimulationManager. De asemenea, în cadrul acestui pachet există și o interfață Strategy, clasele ConcreteStrategyQueue și ConcreteStrategyTime implementâd-o. Acest pachet definește logica aplicației, conținând toate metodele ce stau la baza funcționării proiectului. Clasa SimulationManager este de asemenea și clasa de bază, în care este pornit firul de execuție principal, fapt care determină execuția aplicației de simulare a cozilor.
2. Pachetul Model: Acest pachet conține două clase: Task și Server, corespunzătoare gândirii programării orientate pe obiecte. Clasa Task implementează interfața Comparable, aspect esențial în ordonarea clienților în funcție de timpul de sosire.
3. Pachetul GUI: Acest pachet conține clasa care implementează interfața grafică – View. Aceasta extinde clasa JFrame din biblioteca standard Swing, fiind responsabilă pentru toate interferențele vizuale, pe când gestionarea interacțiunilor utilizatorului și a datelor de intrare și ieșire în cadrul aplicației este realizată de clasa ControlButton din clasa SimulationManager din pachetul BusinessLogic care implementează interfața ActionListener.

Diagrama UML de clase, din care se poate observa relațiile dintre acestea, este următoarea:



4. Implementare

Clasa Task: Clasa are drept câmpuri ID, un număr întreg, unic calculat la crearea unei noi instanțe de clasă, timpul de sosire (arrivalTime) și timpul de procesare (serviceTime), tot numere întregi. Timpul de sosire reprezintă momentul în care clientul va fi pus la coadă, în timp ce al doilea reprezintă cât va sta la coadă. Folosind încapsularea datelor, a fost necesară implementarea setterelor și getterelor, împreună cu un constructor corespunzător.

Clasa Server: Clasa are drept câmpuri o coadă de task-uri , de clienți, respectiv suma timpilor de așteptare a clienților distribuiți la coada respectivă, de tipul AtomicInteger. Această clasă implementează interfața Runnable, astfel am suprascris metoda run în care preluăm task-ul următor, îl procesăm decrementând serviceTime-ul și oprim thread-ul pentru o perioadă egală cu waitingPeriod-ul, apoi decrementăm și acest atribut. În cazul în care serviceTime-ul unui task ajunge la zero, clientul se consideră servit și îl scoatem din coadă. Pentru a putea calcula timpul mediu petrecut la coadă de un client am creat metoda getSize care returnează numărul de clienți care așteaptă să fie procesat. Average Waiting Time se estimează ca fiind suma rezultatului metodei anterioare pentru toate cozile existente împărțită la numărul clienților care au fost distribuiți. Pentru a putea calcula timpul mediu de procesare a unui client, am declarat atributele time și nbOfClientsProcessed, numere întregi. Average Service Time se estimează ca fiind suma rezultatelor împărțirii timpului în care au fost serviți clienții de la coadă și numărul celor care au ajuns să fie procesați, mai exact, primi în coada respectivă, și numărul cozilor disponibile.

Clasa Scheduler: Clasa are drept câmpuri o listă de tipul Servers, numărul maxim de cozi, maxNoOfServers, numărul maxim de clienți pentru o coadă, maxTasksPerServer, ambele de tipul primitiv int, precum și strategia de așezare a clienților la. Această clasă trimite clienții la coadă în funcție de metoda aleasă din interfața grafică. În constructorul acestei clase am creat maxNoOfServers instanțe de clasa Server și pentru fiecare am creat și am pornit câte un fir de execuție. Metoda changeStrategy setează strategia corespunzătoare politicii transmise ca argument. Politica este reprezentată în aplicație de două constante SHORTEST_TIME și SHORTEST_QUEUE declarate în SelectionPolicy de tipul enum. În funcție de politica aleasă metoda dispatchTask apelează metoda addTask din clasa corespunzătoare. Metoda isClosed ne transmite starea cozilor, fapt ce ne ajută să oprim simularea în situația în care nu mai sunt clienți care așteaptă să fie distribuiți la o coadă și toate cozile sunt închise.

Interfața Strategy conține metoda addTask. Aceasta este implementată în clasele ConcreteStrategyTime și ConcreteStrategyQueue, fiecare reprezentând un mod de așezare a clienților la cozi. O strategie se bazează pe timpul de așteptare la coadă, iar cealaltă se bazează pe numărul clienților așezați la fiecare coadă în parte, alegând în fiecare caz minimul.

Clasa View se ocupă de interfața grafică și de toate interacțiunile vizuale. Pentru o organizare bună a codului, au fost create mai multe metode. Fiecare dintre acestea ajută la pregătirea componentelor. Interfața grafică este alcătuită din 8 etichete, fiecare cu un nume sugestiv pentru datele de intrare care trebuie introduse de către utilizator în casetele text aferente. Strategia dorită se poate alege prin intermediul unui comboBox. Odată ce datele sunt validate, butonul START începe simularea, care poate fi vizualizată în timp real în caseta text din fereastră.

Clasa SimulationManager: Clasa are drept câmpuri cele șapte date de intrare, un fileWriter care facilitează scrierea într-un fișier a datele de ieșire, o instanță de clasa Scheduler, thread-ul principal al aplicației, precum și o listă de clienți, care este generată aleator în metoda generateNRandomTasks. Această clasă are ca și metodă, metoda main în care creăm o instanță de clasa SimulationManger, prin care reușim să creăm firul de execuție. Ea implementează interfața Runnable, astfel am suprascris metoda run în care alegem clienții cu timpul de sosire egal cu timpul curent al simulării, iar cu ajutor metodei dispatchTask trimitem clientul la coada

potrivită în funcție de politica de gestionare aleasă, după care ștergem clienții aleși din listă. În această metodă calculăm și timpul mediu de servire care reprezintă rezultatul împărțirii sumei timpilor de procesare a clienților la numărul acestora. Pentru a determina ora de vârf am verificat la fiecare moment de timp numărul clienților aflați la cozi, iar pentru a calcula timpul mediu de așteptare la coadă am împărțit suma timpilor de așteptare la coadă realizată în clasa Server la numărul clienților.

Clasa ControlButton implementează interfața ActionListener și metoda actionPerformed care în cazul apăsării butonului START din interfața grafică, preia datele de intrare și pornește firul principal de execuție.

5. Rezultate

Pentru verificarea corectitudinii codului am folosit cele trei exemple de date de intrare date din prezentarea temei. În primul test sunt furnizați patru clienți, două cozi, o limită de timp pentru simularea aplicației de 60 de secunde, intervalul timpului de sosire [2, 30] și intervalul timpului de servire [2, 4] a clienților. După rularea programului cu aceste date, se poate observa că aplicația funcționează corect, conform așteptărilor. De asemenea, am verificat funcționalitatea folosind și alte exemple, pentru a asigura corectitudinea ambelor strategii de așezare a clienților la coadă.

Pentru a reduce riscul de erori și de comportament imprevizibil al programului, am încercat să mă gândesc la toate cazurile posibile. De exemplu, în cazul în care utilizatorul introduce date de intrare invalide, rezultatul este automat eronat. Astfel, dacă acesta nu furnizează numere întregi în casetele text, se va vizualiza un mesaj de eroare în locul simulării cozilor în timp real.

6. Concluzii

Această tema m-a ajutat să percep importanța etapei de implementare a unui proiect. Înainte de a începe partea de proiectare și scriere de cod pentru o aplicație, este esențial să ne gândim la ce dorim să obținem și, mai ales, în ce fel vrem să lucrăm, ce structuri de date ne sunt de folos, ce scenarii am putea să gestionăm, precum și alte detalii importante care trebuie puse la punct. Timpul meu a fost, astfel, mult mai bine organizat, știind anterior cum este nevoie să acționez pentru un rezultat cât mai satisfăcător.

Posibilitatea de a lucra cu fire de execuție mi-a întărit cunoștințele legate de acestea și mi le-a aprofundat prin folosire unor noi structuri de date precum AtomicInteger și BlockingQueue.

O posibilă dezvoltare ulterioară ar fi capabilitatea aplicației de gestionare a cozilor ar consta în vizualizarea grafică a rezultatului. Acest aspect ar mări spectrul de utilizare al aplicației, chiar dacă necesită o cultură mult mai profundă a implementării interfeței grafice

7. Bibliografie

1. https://dsrl.eu/courses/pt/
2. https://app.diagrams.net/
3. https://www.geeksforgeeks.org/java-program-to-write-into-a-file/
4. https://docs.oracle.com/javase/tutorial/uiswing/components/combobox.html
5. https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
6. https://www.tutorialspoint.com/java/util/timer_schedule_period.htm
7. https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html