**DOCUMENTAȚIE TEMA 4**

**FOOD DELIVERY MANAGEMENT SYSTEM**

**Nume prenume: Copoț Raluca**

**Grupa: 30227**

**Profesor laborator: Mitrea Dan**

Cuprins

1. **Obiectivul temei3**
   1. Obiectivul principal3
   2. Obiective secundare3
2. **Analiza problemei3**

2.1 Analiza principală 3

2.2 Cerințe funcționale 4

2.3 Cerințe non-funcționale 5

1. **Proiectare5**

3.1 Determinarea structurilor de date5

3.2 Împărțirea în clase și pachete5

3.3 Algoritmi7

1. **Implementare8**

4.1 Clase și metode8

4.2 GUI10

1. **Rezultate12**

5.1 Funcționare12

5.2 Testare12

1. **Concluzii12**
2. **Bibliografie13**
3. **Obiectivul temei**
   1. **Obiectivul principal**

Proiectarea și implementare unui sistem de management al livrării alimentelor pentru o companie de catering.

* 1. **Obiective secundare**

|  |  |  |
| --- | --- | --- |
| **Obiectiv secundar** | **Descriere** | **Capitol** |
| Analizarea problemei și identificarea cerințelor | Realizarea diagramei use-case și determinarea scenariilor posibile de funcționare; | 2 |
| Determinarea structurilor de date | Pentru a îndeplini cerințele impuse trebuie determinate tipurile de date pentru a respecta Composite Design Pattern și structurile pentru a stoca informațiile legate de comenzi în DeliveryService; | 3 |
| Împărțirea în clase si pachete | Determinarea unui scop bine definit pentru fiecare clasă și distribuirea acestora în diferite pachete, în funcție de rol pentru a urma un anumit design pattern; | 3 |
| Determinarea și dezvoltarea algoritmilor | Având trei tipuri de utilizatori cu diferite permisiuni trebuie implementați algoritmi pentru operațiile pe care le poate efectua fiecare dintre aceștia; | 3 |
| Implementarea aplicației | Scrierea propriu-zisă a codului, implementarea efectivă a algoritmilor determinați anterior; | 4 |
| Testarea | Verificarea corectitudinii funcționării programului; | 5 |

1. **Analiza problemei**

**2.1. Analiza principală**

Exemplu de use-case pentru importarea meniului

**Use case**: importarea meniului

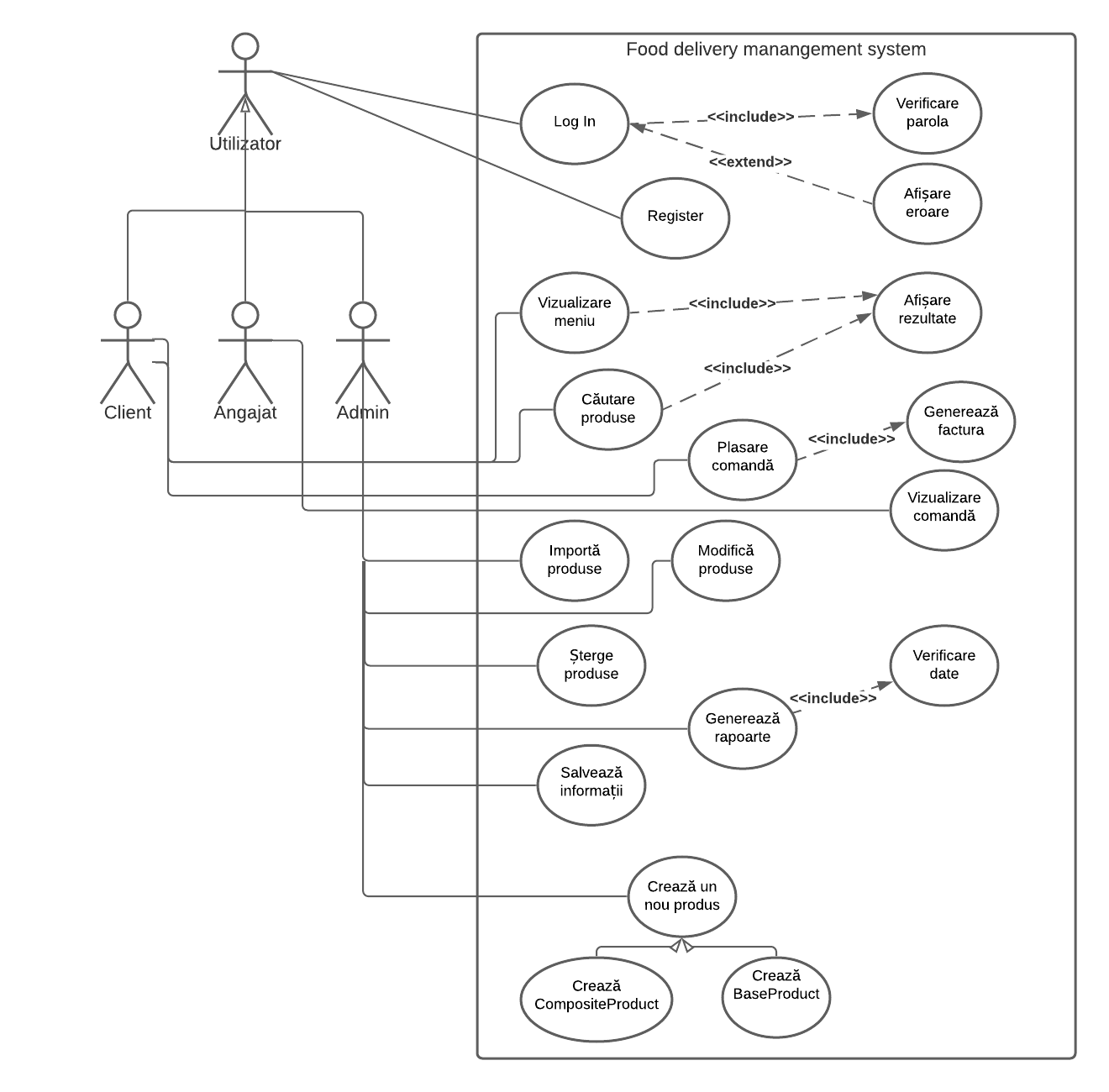
**Actorul primar**: Administratorul

**Scenariul principal în caz de succes**:

1. Utilizatorul introduce datele de intrare pentru a se loga în cont: numele de utilizator și parola
2. Utilizatorul apasă pe butonul „Submit”
3. Datele sunt corecte și este deschisă fereastra pentru operațiile admin-ului
4. Admin-ul apasă pe butonul „Import products”
5. Produsele sunt importate cu succes
6. Utilizatorul apasă pe butonul „Log Out” pentru a părăsi aplicația

**Scenariul alternativ**: Introducerea unor date de intrare invalide

* Utilizatorul introduce un nume de utilizator care nu există sau o parolă greșită
* Aplicația afișează un mesaj de eroare
* În cazul în care nu există deja un cont, acesta poate fi creat
* Scenariul se întoarce la pasul 1



**Diagrama use case:**

**2.2. Cerințe funcționale**

Proiectați și implementați o aplicație pentru simularea unui sistem de management al livrării alimentelor pentru o companie de catering. Aplicația trebuie să îndeplinească următoarele cerințe:

* Conține trei tipuri de utilizatori: client, administrator și angajat;
* Citirea datelor de la tastatură;
* Utilizarea de lambda expressions și procesarea stream-urilor;
* Utilizarea design pattern-urilor: Composite Design Pattern și Observer Design Pattern;
* Afișarea rezultatelor în interfața grafică;
* Clientul poate:
  + Crea un nou cont;
  + Vizualiza lista de produse din meniu;
  + Căuta produse în funcție de anumite criterii;
  + Plasa o comandă pentru care este generată o chitanță;
* Adminul poate:
  + Importa lista inițială de produse;
  + Efectua operații pe produse;
  + Genera patru tipuri de rapoarte;
* Angajatul este notificat de fiecare dată când este plasată o comandă;

**2.3. Cerințe non-funcționale**

* Aplicația ar trebui să fie intuitivă și ușor de folosit de către utilizator;
* Designul ar trebui să fie unul compact, dar care conțină toate elementele dorite, fără să devină prea încărcat;
* Aplicația ar trebui să atenționeze utilizatorul atunci când introduce date de intrare invalide;
* Aplicația ar trebui să permită crearea unui nou cont pentru toate tipurile de utilizatori;
* Rezultatul afișat ar trebui sa fie lizibil, ușor de înțeles și interpretat;

1. **Proiectare**
   1. **Determinarea structurilor de date**

Pentru acest proiect, una dintre primele cerințe menționate este aceea de a avea trei tipuri de utilizatori, Client, Admin și Employee, motiv pentru care am implementat aceste clase. Însă, un lucru pe care îl au în comun toți utilizatorii sunt datele de logare, motiv pentru care toate cele trei clase vor moșteni clasa abstractă User. Utilizând același principiu, am decis ca MenuItem să fie superclasă pentru BaseProduct și CompositeProduct. În plus, mai am nevoie de o clasă Order, și de cele prezentate în diagrama UML de clase din pdf-ul cu cerințele.

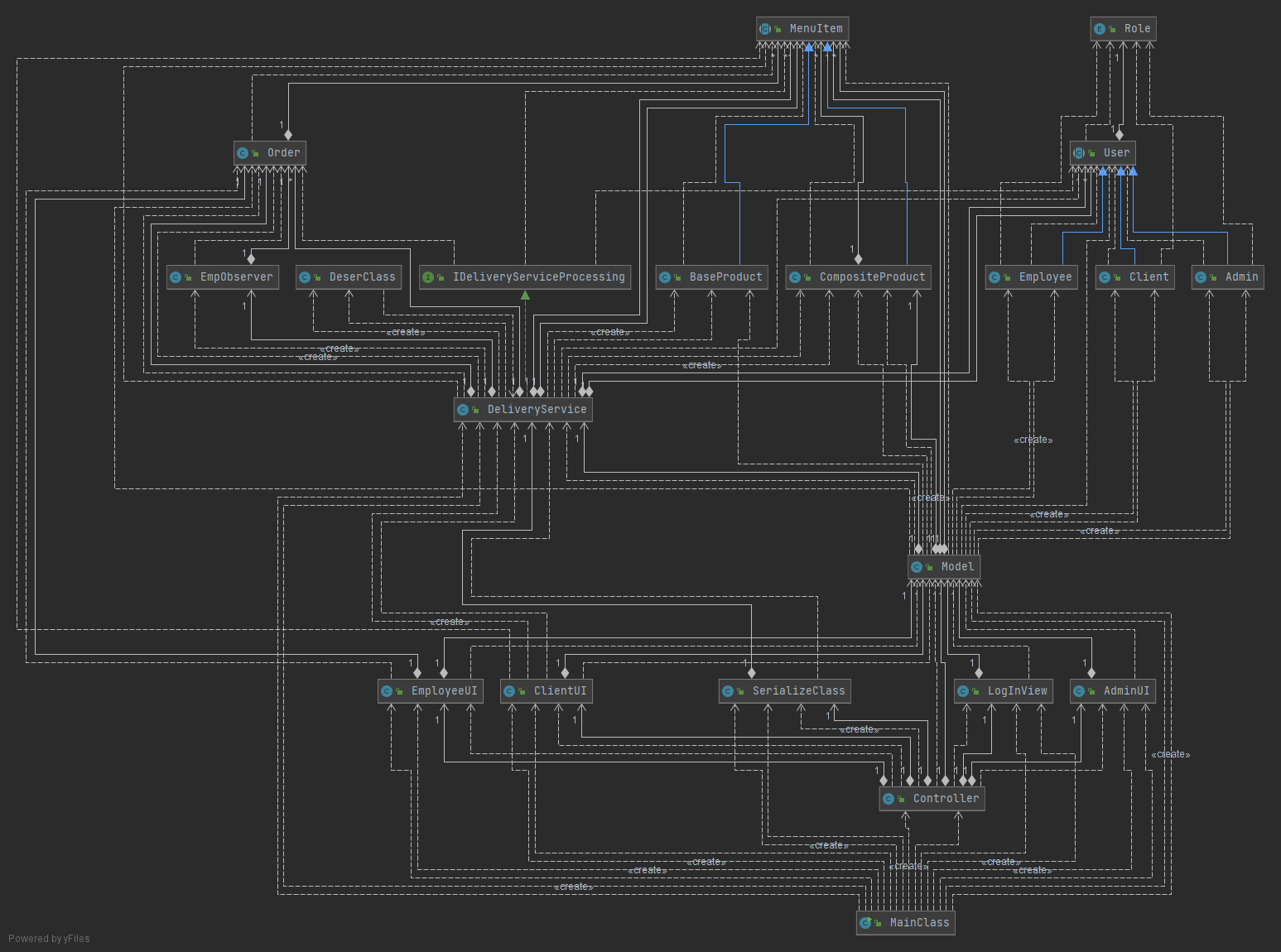
În plus, sper deosebire de acea diagramă UML, voi avea nevoie de o clasă pentru serializare și una pentru deserializare. Iar, pentru a stoca informațiile legate de orders, voi folosi o structură de tipul HashMap<Order, <ArrayList<MenuItems>> și una de tipul List<MenuItems> pentru a stoca toate produsele din meniu.

* 1. **Împărțirea în clase și pachete**

Pasul următor după stabilirea tipurilor de date pe care le voie folosi, este împărțirea în clase, respectiv în pachete. Clasele care sunt prezentate în diagrama UML sunt: „Order”, pentru lucrul cu comenzile, „DeliveryService” care conține logica programului, implementând interfața „IDeliveryService”, clasa abstractă „MenuItem” implementată de „BaseProduct” și de „CompositeProduct”, clasa pentru serializare și pentru interfețele grafice. În plus, în urma analizei de la pasul precedent, am stabilit de faptul că mă voi folosi de clasa abstractă „User”, implementată de „Employee”, „Client” și „Admin”. Ceea ce diferă la cele trei tipuri de clienți va fi valoarea obiectului de tip „Role”, acesta fiind un enum. Deoarece interfața „Observable” este considerată „deprecated”, am folosit o clasă „EmpObserver” care va îndeplini funcția acesteia.

De asemenea, mă folosesc de arhitectura de tip MVC, deci voi avea nevoie de clasele „QueueView”, „QueueController”, „QueueModel”. În cele din urmă, am nevoie de o clasă main, „MainClass”.

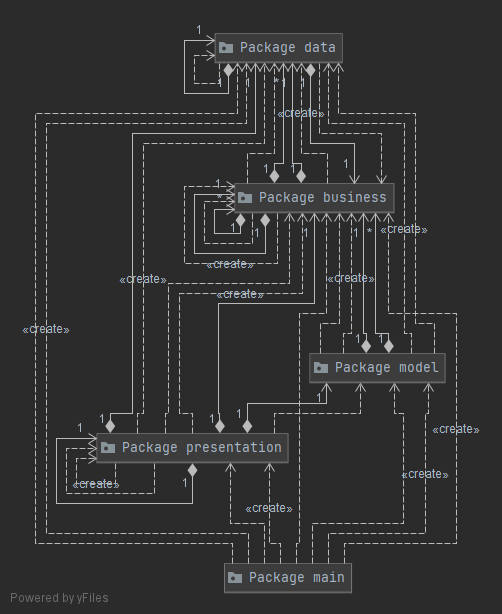
**Diagrama UML pentru clase:**



Pentru împărțirea în pachete am folosit ca și criteriu funcționalitatea claselor, ajungând la:

* main: conține clasa „MainClass”
* business: conține logica aplicației, respectiv clasele „BaseProduct”, „CompositeProduct”, „Order”, „EmpObserver”, „MenuItem”, „DeliveryService”, respectiv interfața „IDeliveryService”
* data: conține clasele „Admin”, „Client” , „Employee” , „User” , „DeserClass” , „SerializeClass” , și enum-ul „Role”
* model: care conține clasa „Model”
* presentation: conține clasele legate de GUI, „AdminUI”, „ClientUI”, „EmployeeUI”, „LogInView”, „Controller”

**Diagrama UML pentru pachete:**



* 1. **Algoritmi**

**isWellFormed:** este o metodă invariant, care trebuie să fie adevarată indiferent de ce se întamplă pe parcursul execuției. Astfel, această metodă verifica existența unui admin în lista de useri, deoarece fără un admin, aplicația nu ar putea funcționa corespunzător.

**importProducts:** în fișierul products.csv se află toate produsele de tip MenuItem, cu proprietățiile separate cu o virgulă. Așadar, parcurgând fișierul linie cu linie și căutând ”,” după care facem split, putem salva toate produsele.

**modifyMenuItems:** pentru a putea modifica produsele în funcție de anumite criterii, fără a le introduce din nou pe cele pe care dorim să le lăsăm la fel, voi folosi un MenuItem „dummy”, care conține valoarea „-1” pentru toate atributele pe care nu doresc să le modific. Folosindu-mă de acest „dummy” și de numele produsului pe care doresc să îl editez, îl voi căuta în lista de produse, urmând să modific toate câmpurile care nu au în „dummy” -1.

**logIntoAccount:** în cazul în care datele de intrare nu sunt nule, voi căuta în lista de utilizatori numele de utilizator introdus. În cazul în care este găsit si parola nu este invalidă voi returna valori care mă vor ajuta să determin rolul utilizatorului pentru a deschide fereastra respectivă, sau numele de utilizator în cazul clientului.

**searchBy:** folosindu-mă din nou de un „dummy”, care reprezinîa de această data un vector de tipul String, având pe pozitii consecutive proprietățiile unui obiect de tip menuItem, dacă am o valoarea diferită de „-1”, voi aplica un filtru de căutare, în caz contrat trecând pur și simplu la următorul element.

**generateReportOne:** caut toate comenzile plasate în intervalul orar dat și le adaug într-o listă, pe care o voi returna ca rezultat.

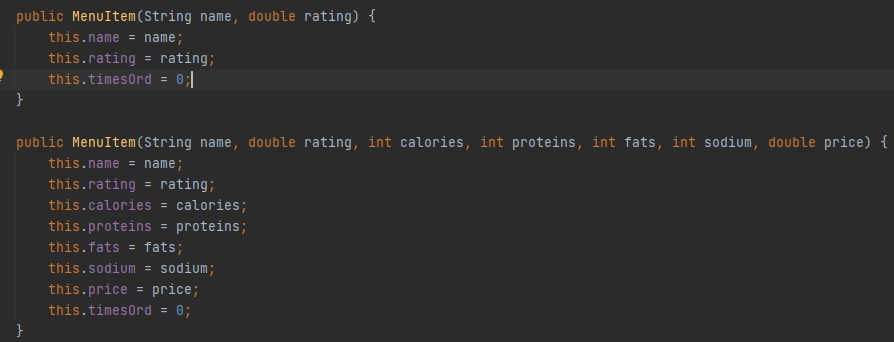
**generateReportTwo:** adaug într-o listă toate produsele care au fost comandate, luându-le din HashMap. Salvez apoi într-o altă listă produsele care au fost comandate de cel putțin numărul dat de ori, având această informație salvat în MenuItem.

**generateReportThree:** salvez, pentru început, toate comenzile care au o valoare mai mare sau egală cu cea dată, într-o listă. Parcurg apoi simultan lista de useri și cea salvată anterior. Dacă username-ul din liastă corespunde cu cel care a plasat comanda și numărul de comenzi plasate de clientul respectiv este mai mare sau egal cu numărul dat de comenzi, atunci user-ul este adăugat în lista rezultat.

**generateReportFour:** adaug într-un Map toate comezile plasate în ziua dată. Adaug apoi toate valorile din Map, care reprezintă array-uri de MenuItems într-o listă, din care scot duplicatele, rezultând o listă cu toate produsele comandate în ziua respectivă.

**importUsers:** citesc fișierul „users.txt” în care se află toți utilizatorii. În funcție de rolul, specificat în fișier, creez utilizatorii respectivi.

1. **Implementare**
   1. **Clase și metode**

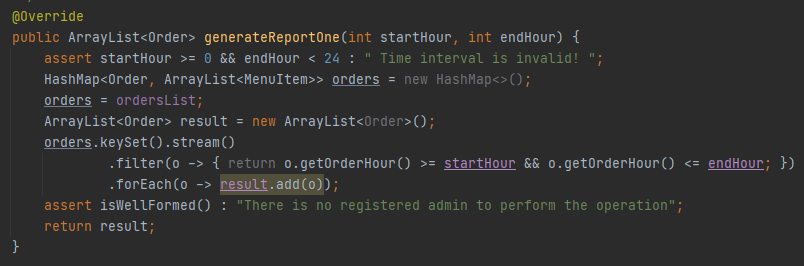
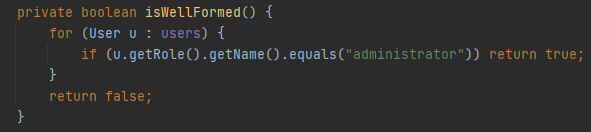


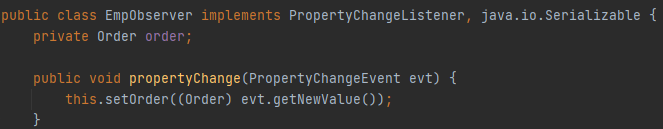
**MenuItem:** clasă abstractă, care conține atributele menționate în suportul pentru assignment, plus o variabilă care în care este stocat de câte ori a fost cumpărat produsul respectiv. Conține metode de get, set și metoda abstractă „computePrice”. Are, de asemenea, doi constructori.

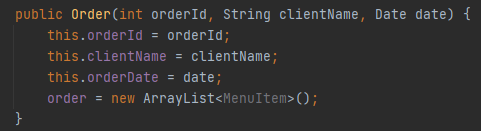
**BaseProduct:** implementează clasa abstractă „MenuItem”, metoda „computePrice” reprezentând o metodă de „get” a prețului. Am implementat, de asemenea, și „toString”, „equals” și „hashCode”.

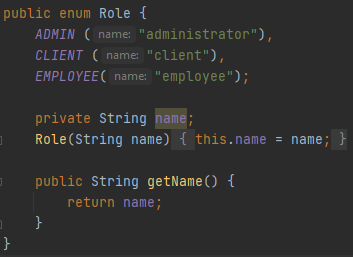
**CompositeProduct:** și aceasta implementează MenuItem și metodele menționate anterior. Conține, în plus, un ArrayList de MenuItem, care pot fi adăugate sau eliminate.

**IDeliveryService:** reprezintă o interfață, în care sunt declarate operațiile care pot fi efectuate de cele trei tipuri de utilizatori. Toți utilizatorii se folosesc de „register” și „logIntoAccount”, restul fiind operațiile specifice pentru fiecare.

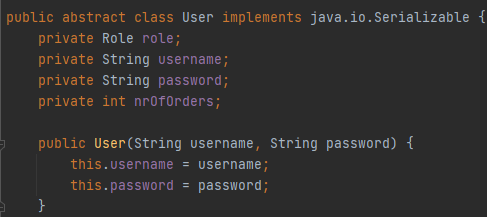
**DeliveryService:** conține logica de bază a aplicației. La prima rulare, înainte ca datele să fie serializate, acestea vor fi create și inițializate. De asemenea, utilizatorii vor fi importați din fișierul în care au fost stocate aceste date. „addPropertyChangeListener” și „setOrder” sunt metodele care înlocuiesc interfața „Observable”. Clasa implementează metodele declarate în interfață, care au fost menționate în secțiunea de algoritmi. În plus, metoda invariant „isWellFormed” este verificată în fiecare dintre celelalte metode, alături de precondiții și postcondiții, cu folosind „assert”.

**EmpObserver:** este observer-ul, în timp ce clasa DeliveryService reprezintă observable. Clasa conține metode de get si set pentru order, deoarece aceasta este proprietatea despre care vrem sa știm daca s-a modificat, pentru a putea notifica angajatul. De asemenea, metoda „propertyChange”, folosindu-se de metoda de set, modifica vechea valoare în cea nouă.

**Order:** conține structura pentru comenzi. Pentru a putea face diferența între acestea, pe lângă numele clientului, dată și preț, am adăugat atributul id, care este unic pentru fiecare comandă. Iar pentru a genera mai ușor rapoartele, fiecare comandă conține și o listă de MenuItems. În plus, folosind o structură de tipul HashMap, am implementat atât metode de get si set, cat și „equals” și „hashCode”.



**Role:** este un enum, care conține rolurile posibile pe care le poatea avea un un utilizator: administrator, client, respectiv employee. Pentru a avea un cod mai ordonat, am adăugat proprietatea „name”, referindu-mă la acest nume în cod, și nu la elementele scrise cu majuscule.



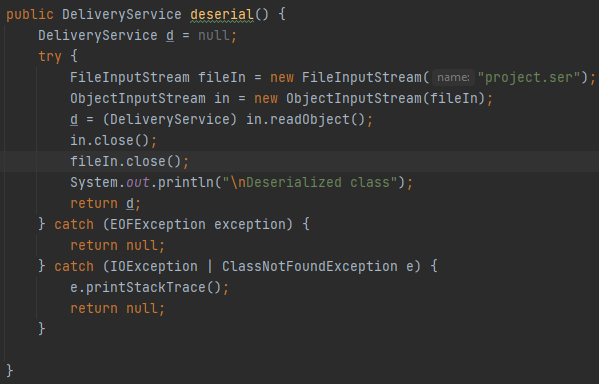
**User:** este o clasă abstractă, care conține atributele pe care le poate avea orice utilizator, singurul lucru care este particular pentru fiecare fiind rolul. De asemenea, pentru ușurință în generarea rapoartelor, am adăugat numărul de comenzi efectuat de fiecare, însă voi folosi această proprietatea doar pentru clienți.

**Admin:** implementează clasa abstractă „User”. Rolul este în constructor ca „Role.ADMIN”, fiind clasa care definește administratorii.

**Client:** implementează clasa abstractă „User”. Rolul este în constructor ca „Role.CLIENT”, fiind clasa care definește clienții.

**Employee:** implementează clasa abstractă „User”. Rolul este în constructor ca „Role.EMPLOYEE”, fiind clasa care definește angajații.

**SerializeClass:** este clasa care se ocupă de serializare. Am ales să salvez întreaga clasă „DeliveryService”, în fișierul „project.ser”. Singura metodă a clasei, „ser”, deschide fișierul, în care scrie datele cu ajutorul unui „ObjectOutputStream”, iar apoi în închide.

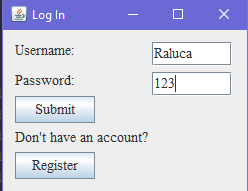
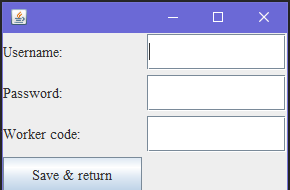
**DeserClass:** această clasă este cea care se ocupă de deserializare, obiectul pe care îl extrag din fișier fiind. Din nou, de tipul „DeliveryService”. De această dată, preiau din fișier cu ajutorul unui „ObjectInputStream” obiectul, pe care îl returnez.

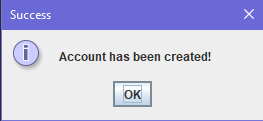
**MainClass:** reprezintă clasa „main” a programului, în care are loc execuția propriu-zisă. Conține un obiect de tipul DeliveryService, iarpentru ca ma folosesc de arhitectura „MVC”, conține și obiecte de tipul claselor „Model”, „Controller” și cele patru clase de view.

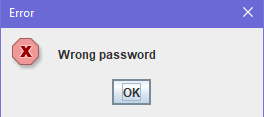
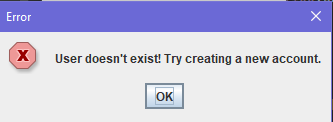
**Model:** conform arhitecturii de tip MVC, aceasta conține logica din spatele aplicației. Conține metode utile, precum „findAll”, cu ajutorul căreia creez tabelul cu produsele, metoda de importare si adăugare de utilizatori în fișierul .txt, precum și patru metode, una pentru fiecare dintre cele pentru rapoarte care are rolul de a scrie rezultatul acestora în fișier.

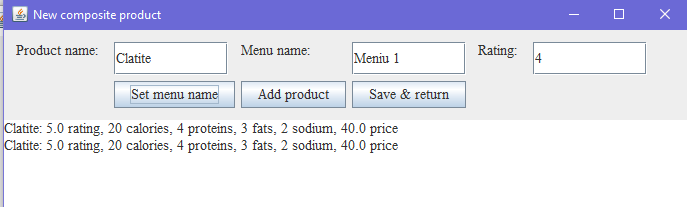
**4.2. GUI**

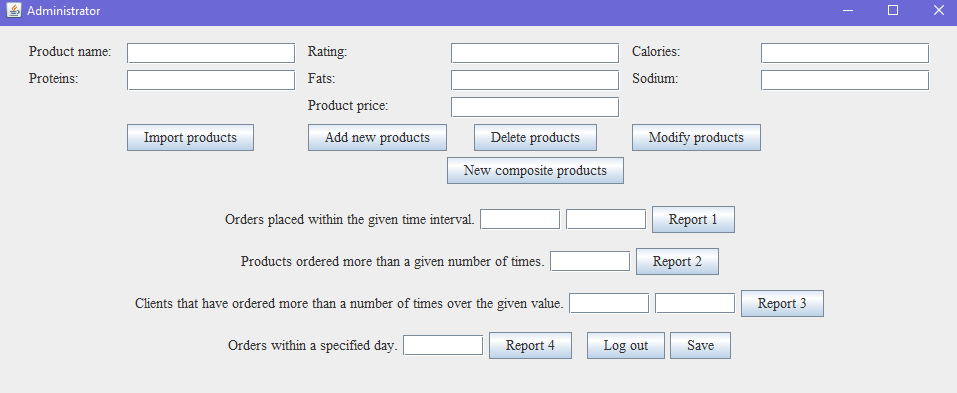
GUI (Graphical User Interface) din Java reprezintă mecanismul cu ajutorul căruia utilizatorul poate interacționa cu un program. Acesta conține componente grafice, cum ar fi butoane, etichete, ferestre etc., pe care utilizatorul le poate folosi pentru a interacționa cu ușurință cu aplicația. GUI are un rol important în construirea cu ușurință a unei interfețe pentru aplicațiile Java.

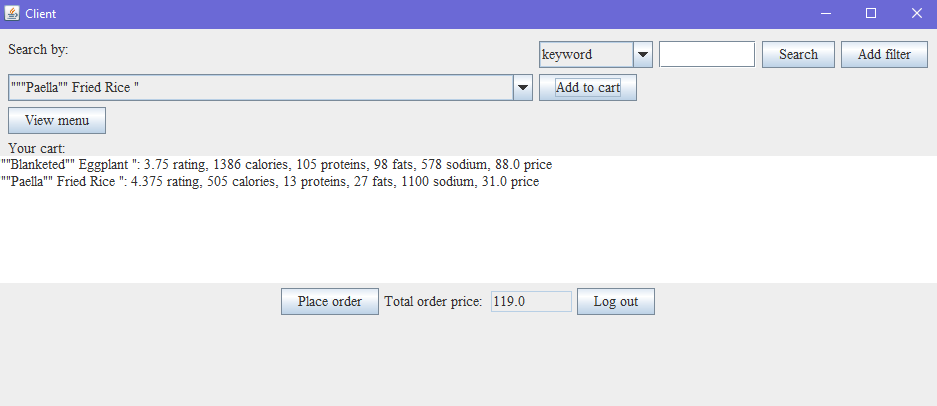


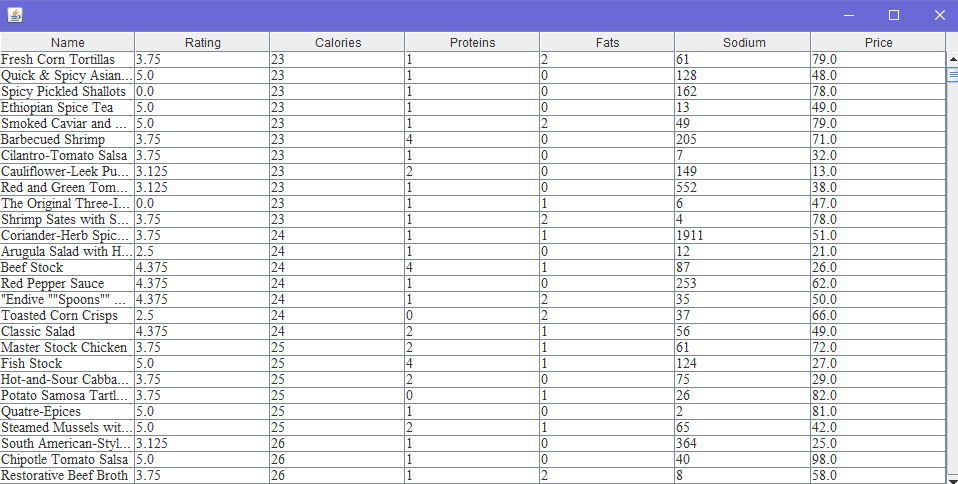
**LogInView:** cu ajutorul acestei clase am realizat interfața grafică pentru fereastra de log-in și pentru crearea unui nou cont. În cazul în care nu are deja un cont, utilizatorul poate apăsa pe butonul „Register”, care va deschide o nouă fereastră, în care își poate introduce datele. Am adăugat câmpul „Woker code” pentru a avea posibilitatea de a crea un nou cont și pentru celelalte tipuri de utilizatori. În cazul în care contul a fost creat, un „dialogPane” va afișa un mesaj de succes. Iar, dacă atunci când intră în cont persoana respectivă greșește numele de utilizator sau parola, un mesaj de eroare corespunzător va fi afișat.

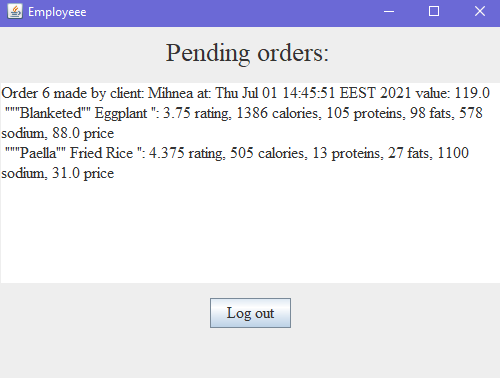


**AdminUI:** clasa care formează interfața grafică pentru fereastra adminului. Aceasta conține câmpuri pentru datele de intrare pentru fiecare dintre operațiile care pot fi efectuate de acesta, și butoanele corespunzătoare. Pentru crearea unui noi CompositeProduct, am creat o nouă fereastră, în care poate fi urmărit conținutul acestui produs.





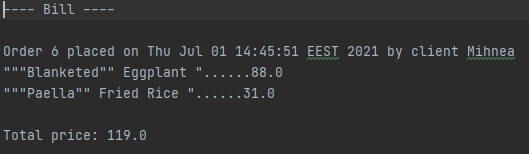
**ClientUI:** clasa care se ocupă de interfața clientului. Am decis să adaug un combo-box pentru a selecta produsele care formează conținutul comenzii. Acesta este actualizat în timp real și poate fi vizualizat cu ajutorul unui text-area, la fel și prețul. Pentru a putea căuta după mai multe criterii, am adăugat, din nou, un combo-box, si un text-field, utilizatorul aplicând filtrele dorite cu butonul „Add filter” și vizualizând rezultalele cu „Search”. De asemenea, atunci când alege să vizualizeze meniul, o nouă fereastră se va deschide, cu un tabel cu toate produsele existente.

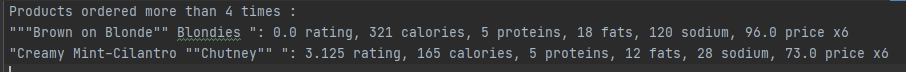
**EmployeeUI:** clasa care se ocupă de interfața angajatului. Acesta nu trebuie sa efectueze nicio operație, doar să fie notificat atunci când este plasată o comandă. Așadar, fereastra conține doar un text-area, în care este afișată cea mai recentă comandă. Pentru ca acest lucru să fie posibil, am implementat metoda „propertyChange”, care modifică conținutul comenzii.

**Controller:** conține logica aflată la baza interfeței grafice, îmbinând view-ul cu modelul, motiv pentru care le conține ca atribute. În această clasă am implementat ActionListener pentru fiecare dintre butoanele vizibile în toate frame-urile prezentate anterior. . Aceastea suprascriu metoda „ActionPerformed” în funcție de acțiunea pe care o execută. Spre exemplu, butonul „Submit” va deschide fereatra corespunzătoare rolului utilizatorului, dacă datele de intrare sunt corecte.

1. **Rezultate**
   1. **Funcționare**

Pentru început, utilizatorul trebuie să intre în contul său. În cazul în care nu are unul deja, poate apăsa pe butonul „Register” pentru a crea unul nou, si pe „Save&Return” odata ce a terminat. Odată cu apăsarea butonului „LogIn” se va deschide o nouă fereastră. Să presupunem că utilizatorul nostru este un admin. Primul lucru pe care va trebui să îl facă este să apese pe butonul „Import products”. După aceasta, el poate efectua orice operații ăși dorelte pe produse. Poate, de asemenea, să genereze rapoarte. În plus, pentru a salva datele de la o execuție a programului la alta, poate apăsa pe butonul „Save”. După ce a terminat, cu ajutorul butonului „LogOut” poate ieși din cont, fără să închidă aplicația, oferindu-i șansa unui alt tip de utilizator să o folosească.

* 1. **Testare**

****Pentru acest proiect nu a trebuit să efectuez teste cu JUnit, în schimb corectitudinea programului reiese din generarea rapoartelor și din conținutul chitanțelor generate la fiecare comandă.

1. **Concluzii**

În cadrul acestui proiect am lucrat pentru prima dată cu lambda expressions și cu java streams, ajungând la concluzia că sunt mai ușor de folosit și mai eficiente, mai ales în cazul în care îmi doresc să filtrez date. De asemenea, am utilizat serializarea si deserializarea pentru prima dată, lucru care poate fi foarte util pentru proiecte viitoare, pentru a salva datele de la o execuție la alta.

1. **Bibliografie**

* Definiție GUI (13.04.2021 21:30): [https://www.guru99.com/java-swing-gui.html definiție GUI 17.03](https://www.guru99.com/java-swing-gui.html%20definiție%20GUI%2017.03)
* Layout (13.04.18:00): <https://docs.oracle.com/javase/tutorial/uiswing/layout/group.html>
* Observer (24.06 19:00): <https://www.baeldung.com/java-observer-pattern>
* Javadoc (30.06 13:00):
  + <https://www.jetbrains.com/help/idea/working-with-code-documentation.html#generate-javadoc>
  + <https://maven.apache.org/plugins-archives/maven-javadoc-plugin-2.8.1/examples/tag-configuration.html>
* Streams & lambda expressions (24.06 12:00): <https://dzone.com/articles/reactive-streams-in-java-9>