

Documentație firmă catering

Fariseu Adriana-Teodora

Facultatea de Automatică și Calculatoare, Grupa 30225

Anul 2020-2021

# Obiectivul principal

Se cere realizarea unei aplicații pentru o firmă de catering, având un sistem de role based access control și o modalitate de procesare a informațiilor din fișiere. Rolurile constau în client, angajat și administrator, angajatul având un sistem de notificare pentru când un client plasează o comandă. De asemenea, administratorul poate să genereze rapoarte și să modifice produsele din fișiere

Această aplicație este foarte complexă, având nevoie în realizarea ei de următoarele obiective secundare:

|  |  |  |
| --- | --- | --- |
| Obiectiv | Descriere | Capitol în care este detaliat |
| Analiza problemei | Pentru a transpune problema în cod, trebuie mai întâi să se extragă din enunț informații relevante în legătură cu funcționalitățile și cazurile de utilizare. | [2](#_Analiza_problemei) |
| Proiectarea | Odată extrase informațiile necesare, se poate începe maparea acestora în paradigma de programare pusă la dispoziție. | [3](#_Proiectare) |
| Implementarea | În acest pas se realizează scrierea codului pentru rezolvarea problemei cerute, pe baza mapării realizate anterior. | [4](#_Implementare) |
| Testarea | Pentru a ne asigura că produsul final sau chiar rezultate parțiale necesare pentru o etapă următoare funcționează corect, codul implementat trebuie la rândul lor supus unor teste ce simulează scenarii de utilizare identificate în etapa de analiză. | [5](#_Testare_și_rezultate) |

# Analiza problemei

*Cerinte funcționale*

**Operații pe fișiere**. Principala modalitate de populare cu produse a aplicației este prin citirea dintr-un fișier .csv. De asemenea, se poate crea adițional o chitanță într-un fișier .txt

**Interfață grafică**. Un astfel de tip de aplicație este mai natural de folosit sub forma unei interfețe grafice, în special pentru o fereastră de comandă de produse sau editare de date

**Persistența informațiilor după îchiderea aplicației**. În ciuda faptului că acest lucru e trecut la cerințe non-funcționale, este un lucru crucial într-un astfel de context. Aici însă se optează spre serializare decât spre baze de date

**Role based access control**. Aplicația necesită nu numai operații pe bază de conturi, însă și permisiuni pe bază de roluri, acestea fiind administrator, angajat, și client

|  |  |  |
| --- | --- | --- |
| Admin | Angajat | Client |
| * Poate să opereze pe produse și meniuri(adăugare, ștergere, modificare) * Poate să creeze meniuri ce conțin mai multe produse de bază * Poate să importe o listă de produse inițiale dintr-un fișier .csv * Poate să genereze diferite rapoarte legate de statistici | * Primească notificări când un client comandă ceva | * Își poate face cont pe platformă * Vedea produsele dintr-un meniu * Căuta produse pe baza mai multor criterii simultan * Să comande mai multe produse și să descarce chitanța în format .txt |

**Generarea de rapoarte pentru administrator**. După cum s-a menționat, administratorul poate genera anumite rapoarte precum:

* Lista comenzilor dintr-un interval orar
* Lista produselor care au fost comandate de mai multe ori decât un anumit număr
* Lista clienților care au comandat de mai multe ori decât o valoare dată, având comenzi de un total mai mare decât unul dat
* Toate produsele comandate într-o anumită zi și de câte ori au fost comandate

Pentru generarea acestora trebuie să se folosească expresii lambda și stream-uri

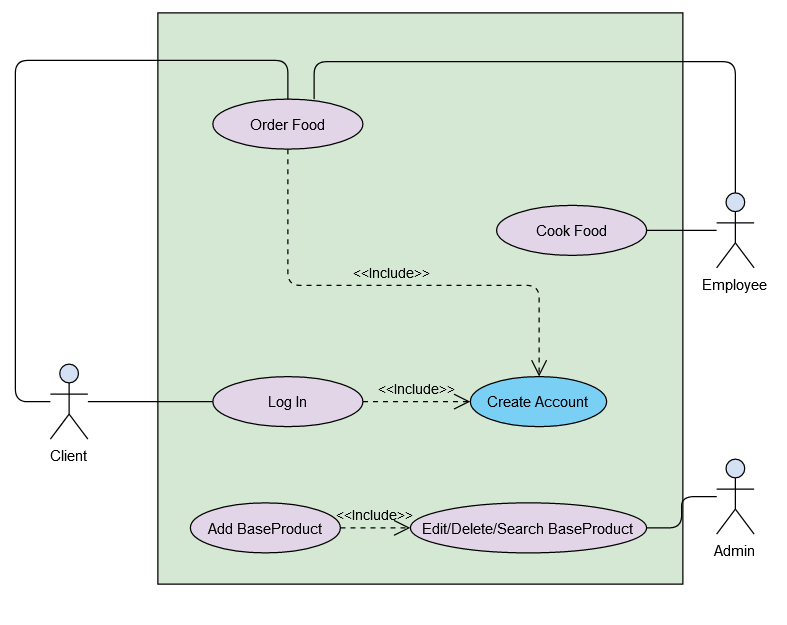
*Cerințe non-funcționale*

**Sistem de notificare**. De fiecare dată când un client plasează o comandă, un angajat ar trebui notificat pentru a știi să-i pregătească mâncarea pentru comandă

**Design by Contract**. Pentru mentenanța mai ușoară a codului, este mai bine să se cunoască informații despre precondiții, postcondiții și axiome ale datelor care se folosesc în clasă

*Scenarii de utilizare*

Chiar dacă există un punct comun în logarea în aplicație, fiecare rol are cazuri de utilizare total unice



Use case: Editarea unui produs

Actor principal: adminul

Cazul de utilizare principal:

1. Adminul se loghează în aplicație
2. Adminului i se deschide o fereastră unde apar casete cu produsele existente
3. Adminul dă click pe butonul de editare, care transformă fereastra în una editabilă
4. Se salvează modificările în fișierul de serializare

Scenariu alternativ: Adminul greșește parola sau numele de utilizator

* Se va afișa un mesaj de eroare și adminul își va reintroduce datele de autentificare

Scenariu alternativ: Adminul se răzgândește și nu mai vrea să editeze.

* Adminul va ieși din modul de editare al ferestrei

# Proiectare

Diagrama UML

Pornind de la diagrama UML impusă, am fost nevoită să adaug mai multe clase pentru o funcționare mai bună, precum o colecție și clase pentru stocarea conturilor din DeliveryService. De asemenea, nici nu se pune problema la categoria claselor de GUI. De obicei, când vine vorba de interfețe grafce, ajung să-mi fac mai multe clase utilitare pentru a păstra o tematică cât de cât consistentă. Ca dovadă, diagrama asta nu atinge nici măcar cea mai mică parte din interfața grafică (vă rog nu mă puneți să fac manual diagrama a 40 de clase că o să plâng). Am generat diagrama UML pentru presentation layer daca va interesează, dar n-am putut sa-l pun în documentație pentru că standardele umane pentru foi de documente nu coincid cu așa ceva

Graphical user interface

Description automatically generated

Observer Pattern

Este un pattern arhitectural care conține un Observable (resurse care se pot schimba) și unul sau mai mulți Observatori care sunt notificați în privința modificărilor efectuate asupra obiectului Observable. În acest caz, angajații serviciului de catering trebuie să fie anunțați atunci când un client comandă ceva, pentru a ști să pregătească mâncarea pentru comandă

Composite Pattern

Este un pattern arhitectural care permite tratarea unui grup de obiecte la fel ca o singură instanță. Ea se bazează pe subclase care moștenesc o clasă de bază care este fie clasă abstractă, fie interfață. Acestea se pot împărți în următoarele categori:

* Component: clasa rădăcină de la care vor pleca toate subclasele. În cazul acesta este reprezentată de clasa abstractă MenuItem.
* Frunză: Implementează funcționalitățile din clasa Component, fără a avea referințe la alte obiecte printre variabilele sale instanță (BaseProduct în cazul acesta)
* Compozit: Conține frunze ca variabile instanță. Spre exemplu, CompositeProduct și OrderItems conțin câte o listă de referințe la obiecte de tipul BaseProduct

Prin tratarea unei liste de obiecte ca o unitate, se poate calcula ușor totalul unei comenzi sau al unui meniu

Structuri de date folosite

Set-uri

Set-urile sunt colecții care nu acceptă obiecte egale (care egalitate e definită de metoda .equals() sau compareTo()). În acest proiect a trebuit să fie asigurată unicitatea numelui produselor din stoc și dintr-o comandă, dar și al numelui utilizatorilor pentru a nu se crea confuzii la logare

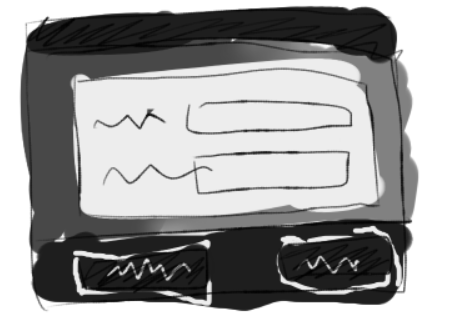
Hashing și Mapping

În acest proiect s-a cerut folosirea tabelelor de dispersie și a mapării anumitor obiecte. Un prim exemplu ar fi chiar păstrarea în memorie a comenzilor care, în loc să aibă o listă de produse ca variabilă instanță, au fost mapate la un obiect de tipul Order.

Maparea a ajutat de asemenea și la realizarea corespondenței între produse și numărul lor de apariții în toate comenzile

Interfața grafică

Am încercat să am un design cât mai facil și culori care să nu fie orbitoare. Am făcut inițial o mapare de contrast și componente pentru fereastra de Login și am pornit de la asta pentru restul ferestrelor



Opțiuni pentru utilizator în interfață. Chiar dacă unele lucruri nu sunt precizate în mod explicit, următoarele funcționalități ar trebui luate în considerare pentru a aranja interfața grafică:

|  |  |  |
| --- | --- | --- |
| Admin | Angajat | Client |
| * Gestionare produse simple * Gestionare meniuri(composite product) * Gestionare useri | * Notificări * Comenzi active (comenzi văzute deja la notificări) | * Comandă * Comenzile mele |

Astfel, putem face o corespondență pentru acestea la un Jcomponent

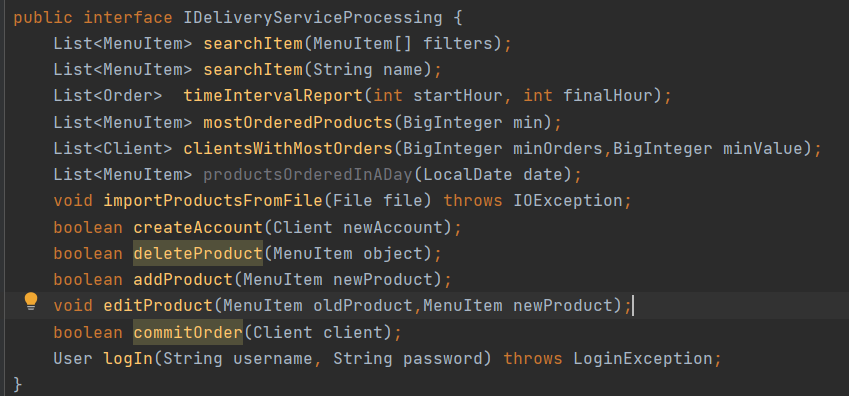
Inițial voiam să fac o fereastră intermediară cu toate opțiunile disponibile la fel ca la proiectul cu OrderManagement, însă cumva am reușit să comprim majoritatea lucrurilor într-o singură fereastră

# Implementare

Organizate în pachete, clasele sunt următoarele

Business

Acolo unde se păstrează logica din spatele aplicației



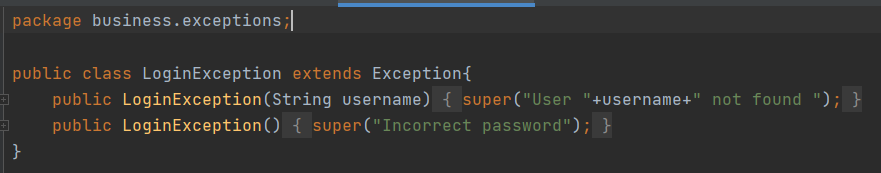
Interfața care ține cont de operațiile necesare pentru operarea în aplicație. Acestea sunt implementate ulterior în clasa DeliveryService, unde se rețin și toate informațiile actuale, ca într-o bază de date.

Business.exceptions

Excepții pentru validarea datelor. Din moment ce acest lucru se realizează mai mult prin aserții, acestea au fost folosite mai mult la logarea în aplicație

Text

Description automatically generated



Business.items

Sunt tiputile de produse din aplicație (BaseProducts, Meniuri etc.)

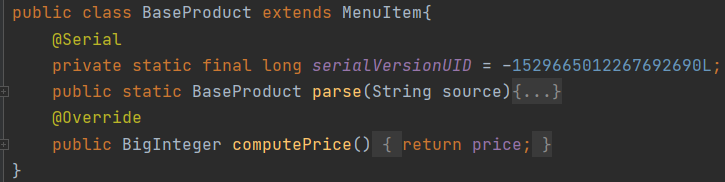
Text

Description automatically generated

Text

Description automatically generated

Clasa componentă absrtactă care implementează cerințele comune tuturorsubclaselor și stabilește metodele abstracte



Produsele simple (nu fac parte dintr-un produs compus, fiind ”frunze”)

Text

Description automatically generated

Composite items sunt clase compuse din mai multe frunze care implementează operații specifice pentru șiruri

Text

Description automatically generated

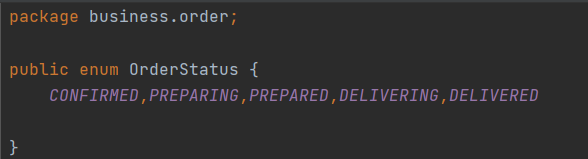
Clasa pentru stocarea produselor dintr-un order. La rândul său este un composite object.

Business.order

Text

Description automatically generated

Clasa pentru comenzi, folosită în maparea obiectelor din comandă



(nefolosit) Un enum care aratp în ce stadiu se află comanda. Plănuiam să mă mai joc puțin cu Observer și să notific și clientul când se schimbă statusul comenzii, însă nu am mai apucat. Am lăsat acest enum pentru eventuale îmbunătățiri

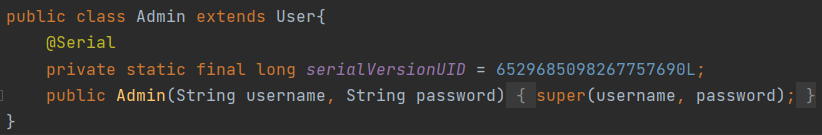
Business.users

Clasele pentru tipurile de utilizatori

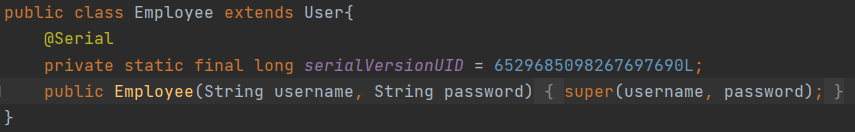
Text

Description automatically generated

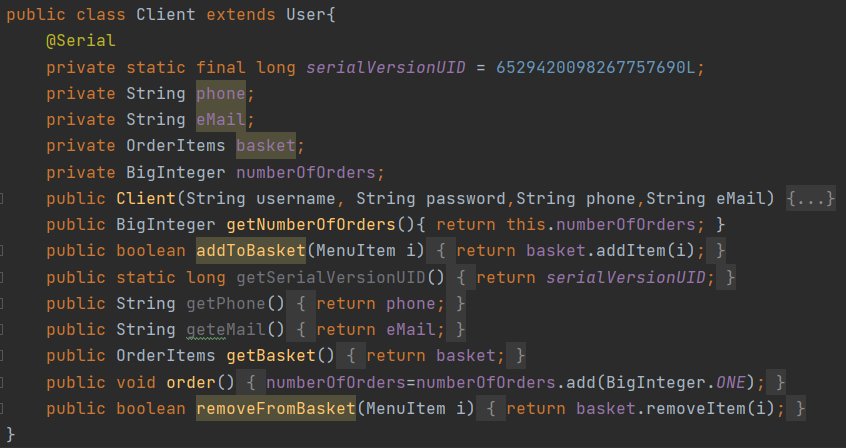
O clasă generală care implementează funcționalități comune tuturor tipurilor de utilizator



Clasa pentru Admin. Nu are funcționalități în plus, însă pot fi potențial adăugate



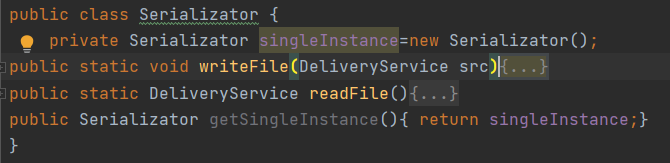
Clasa pentru angajat. La fel, se poate extinde pe viitor cu mai multe funcționalități



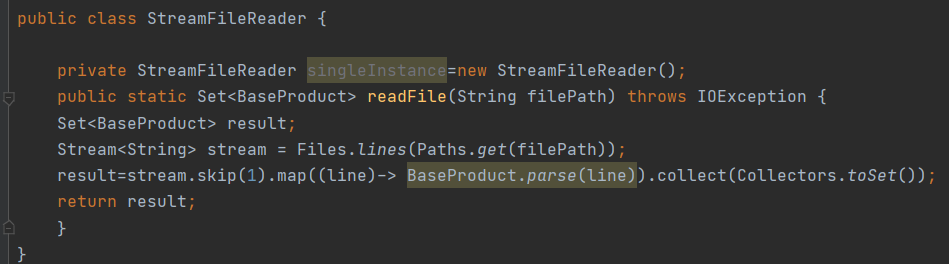
Clasa pentru client. Singura clasă care are funcționalități în plus, însă tot nu au fost folosite momentan. Cu toate acestea, se pot folosi în viitor

Data

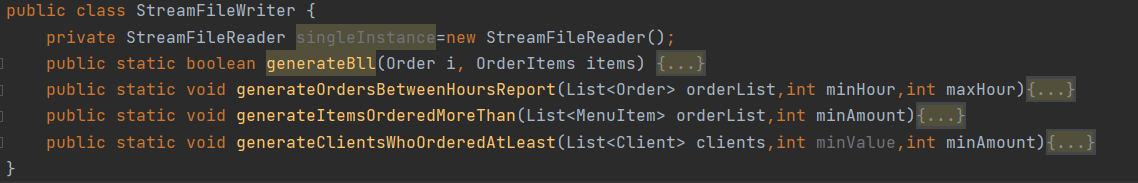
Conține clase pentru citire și scriere și, în general, lucrul pe informații. Acestea sunt organizate după singleton pattern



Este clasa care se ocupă de serializarea și deserializarea obiectului de tip DeliveryService. Din cauză că nu se mai folosește o sursă externă de stocare precum o bază de date, trebuie să se recurgă la metode care scriu informațiile din aplicație ca șir de biți, urmând să fie ulterior reîncărcate la rulare



Se ocupă cu citirea de BaseProducts dintr-un fișier .csv



Se ocupă cu emiterea chitanței și de rapoarte

Presentation

Aici sunt cuprinse pachetul pentru view și controller. De asementea, pentru view mai există și elemente utilitare pentru a formata mai ușor interfața

În principiu, acestea pot fi separate pe 3 categorii:

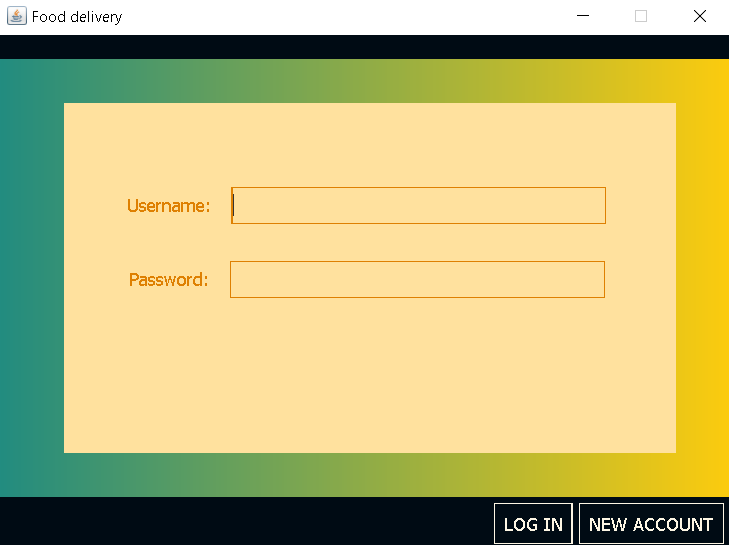
Ferestre + Controllere pentru useri: Sunt ferestre care apar după logarea în aplicație, personalizate după tipul de user care se loghează(de exemplu: UserWindow + Controller, ClientWindow + Controller, EmployeeWindow + Controller, AdminWindow + Controller, ShoppingCart + Controller etc.

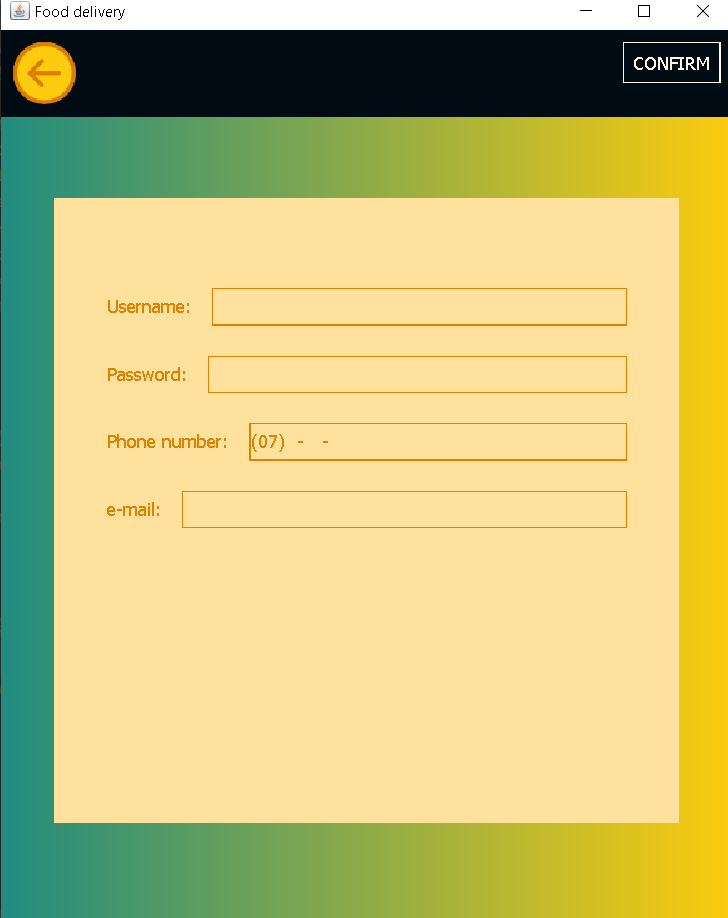
Ferestre + controllere de dialog: Sunt ferestre care imită comportamentul unei ferestre de dialog: Apar overlayed cu fereastra principală și doar returnează anumite input-uri necesare pentru fereastra principală

Alte ferestre + controllere: Fereastra + controller de logare, fereastra abstractă de aplicație care conține constante de cunori și inițializarea tematicii

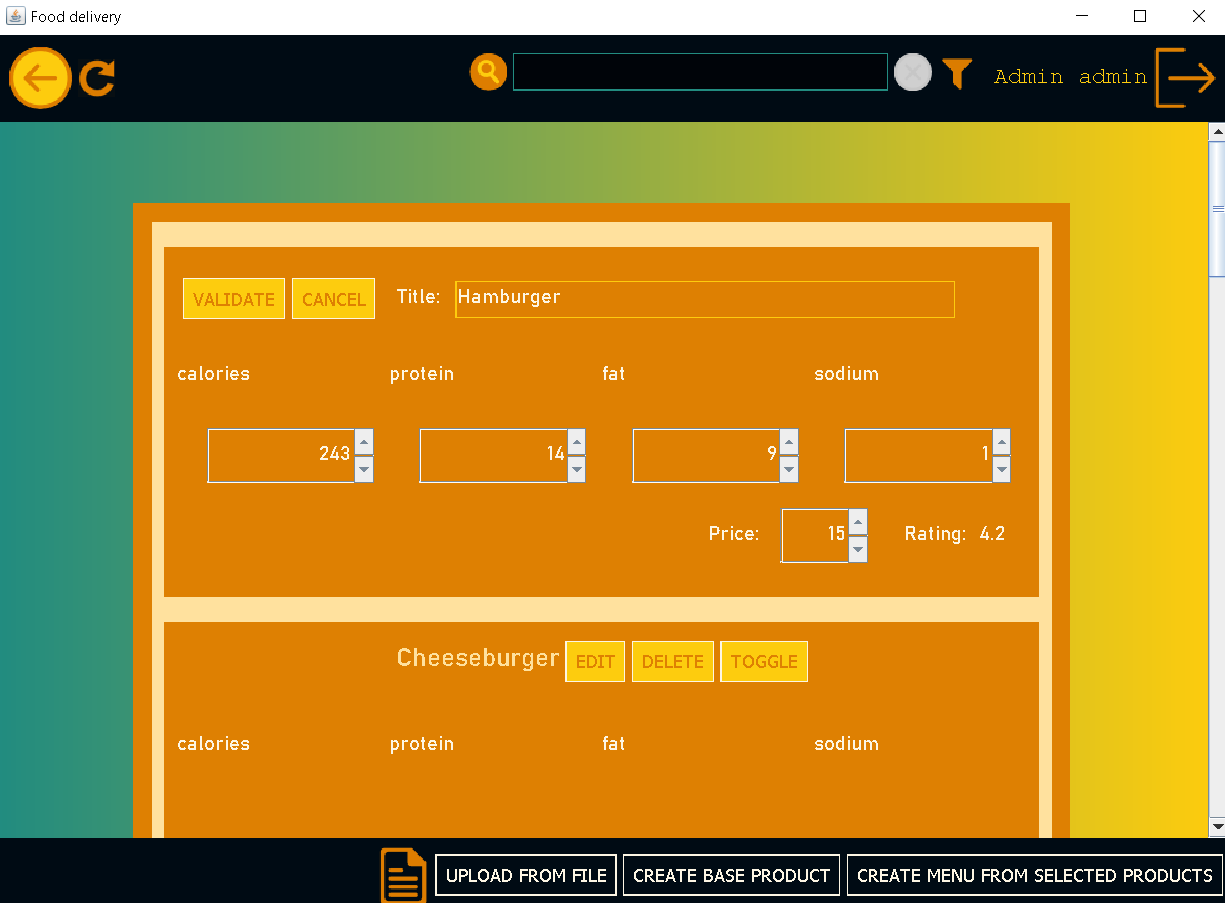
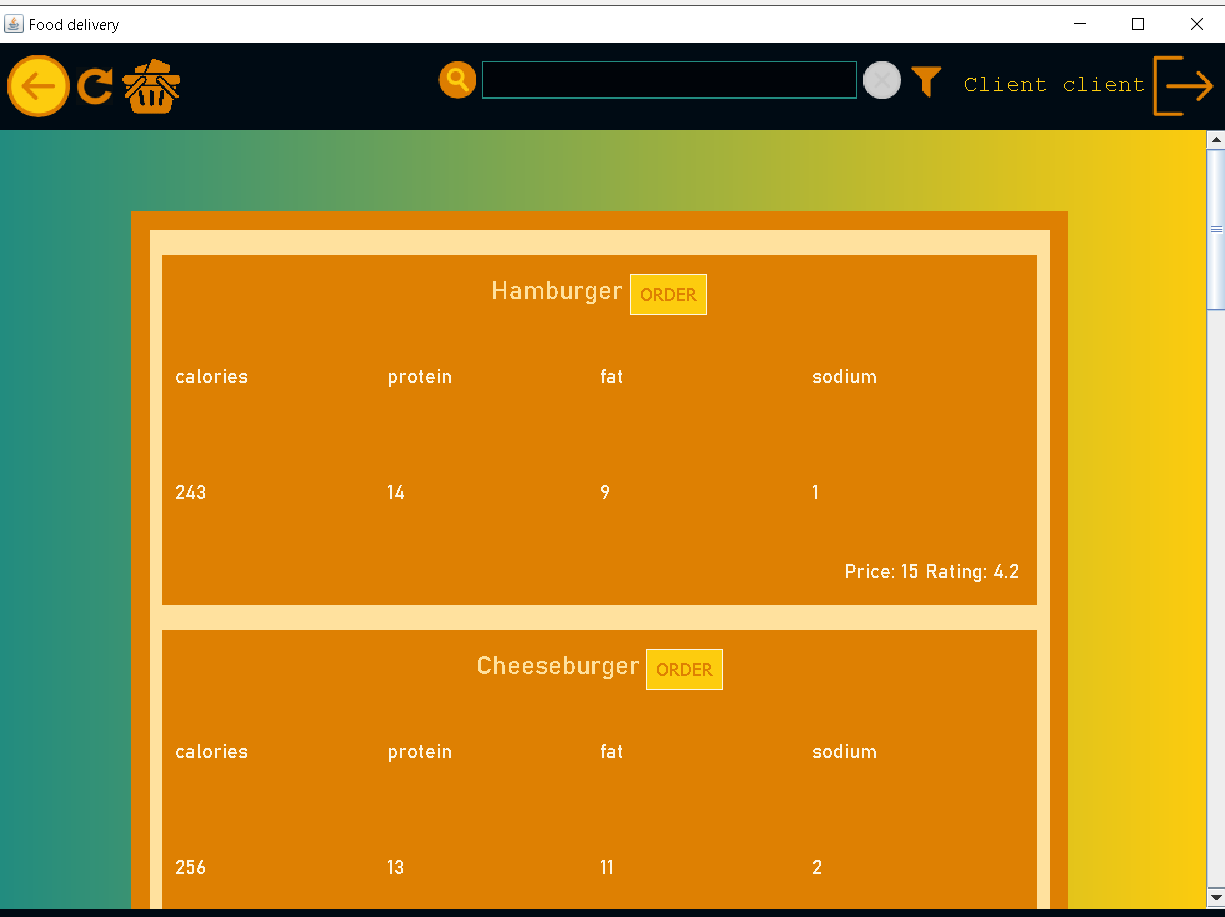
# Testare și rezultate

La rularea aplicației se va deschide o fereastră de logare cu un câmp de username și parolă



De la aceast pas există două variante: Se crează un cont nou de client prin completarea câmpurilor

sau se loghează cu un cont deja existent (admin, angajat sau client)



Din moment ce clientul și adminul au mai multe funcționalități, ferestrele pentru aceste tipuri de cont sunt mai dezvoltate. Fereastra pentru angajat există doar pentru a primi notificări

Clientul și adminul pot căuta după titlu sau avansat după mai multe filtre prin bara de căutare din header. Clientul își poate vedea obiectele din căruciorul de cumpărături din iconița pentru coș. Adminul poate gestiona rapoartele și crearea de produse din subsol

# Concluzii

Acest proiect a împins exersarea a multor noțiuni noi, chiar atât de multe încât abia dacă poți sta să le aplici după cum le înțelegi, în loc de ce este dat mură-n gură în prezentări. Implementarea nu este neapărat extrem de dificilă, însă trebuie stat foarte mult timp pe documentarea în legătură cu noțiuni noi sau chiar necunoscute deoarece nu au fost predate la curs (cough cough ... Serializarea). Chiar și cu vacanța, noțiunile sunt noi, ample și multe și nu prea ajunge timpul să le înțelegi calumea. Deși probabil mi s-a părut asta pentru ca am avut o săptămână în minus și săptămâni foarte încărcate și de alte materii

Pe de altă parte, m-a ajutat să dezvolt și să abstractizez și mai mult anumite clase de interfață grafică pe care le-am folosit în trecut.

# Dezvoltări ulterioare

**Pagină de date personale și mai multe atribute pentru angajat, admin, client.** Ca în aplicații care deja există pe piață, ar fi frumos să existe o fereastră pentru vizualizarea acestor date. Ar putea exista chiar și imagini de profil

**Operarea pe date personale pentru admin**. În general, adminul este cineva care are un control foarte mare în întreținerea sistemului informatic. De asemenea, există și cazuri în care se doresc modificări de date personale. De aceea, ar fi bine ca adminul să aibă și o pagină de operare pe utilizatori și datele lor personale.

**Interfață de notificări mai bună pentru angajat.** Din cauza timpului limitat, am implementat interfața pentru admin doar pentru a se observa pattern-ul Observer. În viitor ar fi mai bine ca notificările să fie panouri care durează puțin mai mult timp, nu doar ferestre de dialog care se închid imediat

**Vizualizarea comenzilor**. Ar fi foarte util pentru clienți să-și vadă comenzile active (care nu au fost încă livrate). Dacă făceam o casetă de interfață pentru comenzi, puteam să completez și punctul anterior

**Dezvoltarea unui sistem mai complex de notificări**. Acest lucru implică primirea de notificări pentur mai multe evenimente și implementarea pattern-ului de observer și pentru clienți. Ar fi foarte util ca angajatul să poată schimba statusul unei comenzi, iar clientul să fie notificat de acest lucru

**Dezvoltarea interfeței pentru formular**. Formularul pentru creare de client, produs și căutare de produs și chiar logare sunt făcute doar ca să fie funcționale, însă ele ar putea fi și mai bune dacă ar avea o opțiune de resetare a tuturor input-urilor. De asemenea, codul lor ar fi mult mai ușor de scris prin reflection

**Thread-uri pentru interfața grafică**. Este un adevărat Coșmar să se încarce toată interfața grafică

**Folosirea clasei de Employee ca observer, în locul ferestrei pentru angajat.** Acest lucru ar face notificările să nu se mai piardă odată cu închiderea ferestrei, iar angajatul ar putea primi notificări și când e offline

# Bibliografie și resurse

<https://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html>

<https://www.geeksforgeeks.org/composite-design-pattern/>

<http://users.csc.calpoly.edu/~jdalbey/205/Resources/customtags>

<http://www.jidesoft.com/javadoc/com/jidesoft/swing/RangeSlider.html> - Clasă pentru un rangeslider