

**Programming Techniques**

**Food Delivery Management System**

*Course Teacher*: prof. Ioan Salomie

*Laboratory Assistant*: ing. Cristian Stan

*Student*: Ivan Alexandru-Ionut

Contents:

1. Assignment objective
2. Problem analysis, scenario, modeling, use-cases
3. Design
4. Implementation
5. Results
6. Conclusions
7. Bibliography
8. Assignment objective

The objective of this assignment was to develop an Order Management application that simply manages the orders created by the user and the creation, deletion and editing of users or products from/into the database. As secondary objectives, there are:

The objective of this assignment was to develop a Food Delivery Management System that manages three types of users: administrator, user and employee and handles all the requests to create orders, import and manage products and also notifies the employee about the created orders to be managed. Some of the secondary objectives were to:

* To create an UI interface for every type of user and handle all kind of requests, from import or editing products to creating orders and notify employees.
* To use Javadoc for documenting classes and generate corresponding Javadoc files.
* To use lambda expressions and streams for importing and serializing/deserializing products.
* To generate certain reports based on different criteria inputted by the administrator.
* To filter the products according to the criteria inputted by the user.
* To clearly make use of good OOP concepts

1. Problem analysis, modeling, scenario and use-cases
   1. Problem analysis

We were required to manage all the products by importing them from the initial .csv, not keeping the duplicates and also give the administrator the opportunity to create custom products composed of multiple base products. The user has the ability to login/register if it has/doesn’t have an account and the account is saved by serialization and written in the .csv file.

We were also required to save in a database/.csv/json file all the orders made by the user, all the custom products created by the administrator and to generate a bill for every order that is made.

The administrator also has the ability to edit/add/delete a base product, as well as, mentioned above also, to create a custom product composed of base products.

* 1. Modeling

We will use a Layered Architecture to make the project more readable and also because it was suggested by the class diagram.

Both Order class, Base Product class and also User class will be mapped to the .csv files using @JsonMapping in order for the Jackson to easily serialize and deserialize the data. The BaseProduct, MenuItem and CompositeProduct classes were created using Composite Design Patter, as stated in the requirements of the assignment, while the DeliveryService class was made implemented using Design by Contract methods.

The DeliveryService was also implemented using the IDeliveryService interface, that is composed of the basic methods that our Delivery Service must implement.

Instead of using a Serializer class, my project uses the Jackson dependency and its methods and Mappers in order to more simply serialize the data, using a CsvMapper and a CsvSchema that was automatically generated from my model classes.

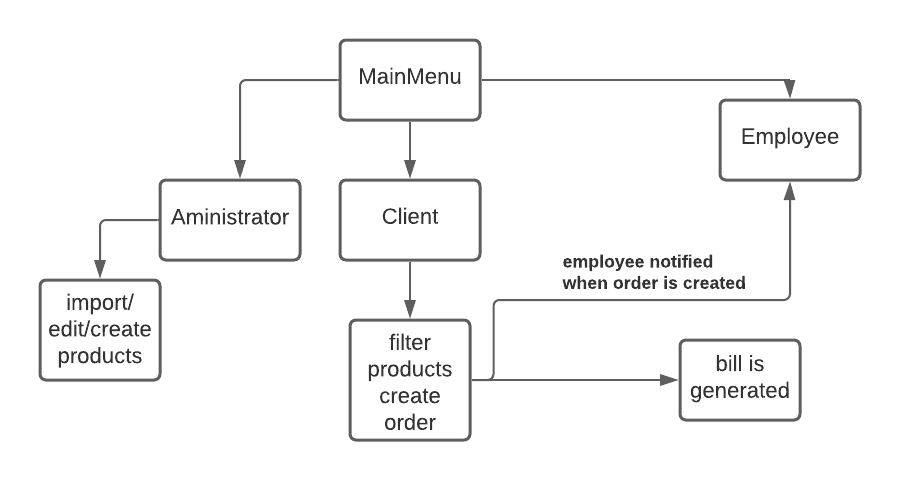
The presentationLayer contains just the controller classes that handles all the FXML files and their request, having a super class, called SceneController that contains a DeliveryService instance in order for them to be able to use only one instance in the whole project. Also, the SceneController class contains the scene and stage of the project in order to use only one window for the administrator or the client view at once, the employee having a separate view.

* 1. Scenario and use-cases

The following flow-chart resumes all the scenarios the user can go through.

The application starts in the MainMenu window and the user is able to go either in the administrator view or in the Client view, depending on the needs.

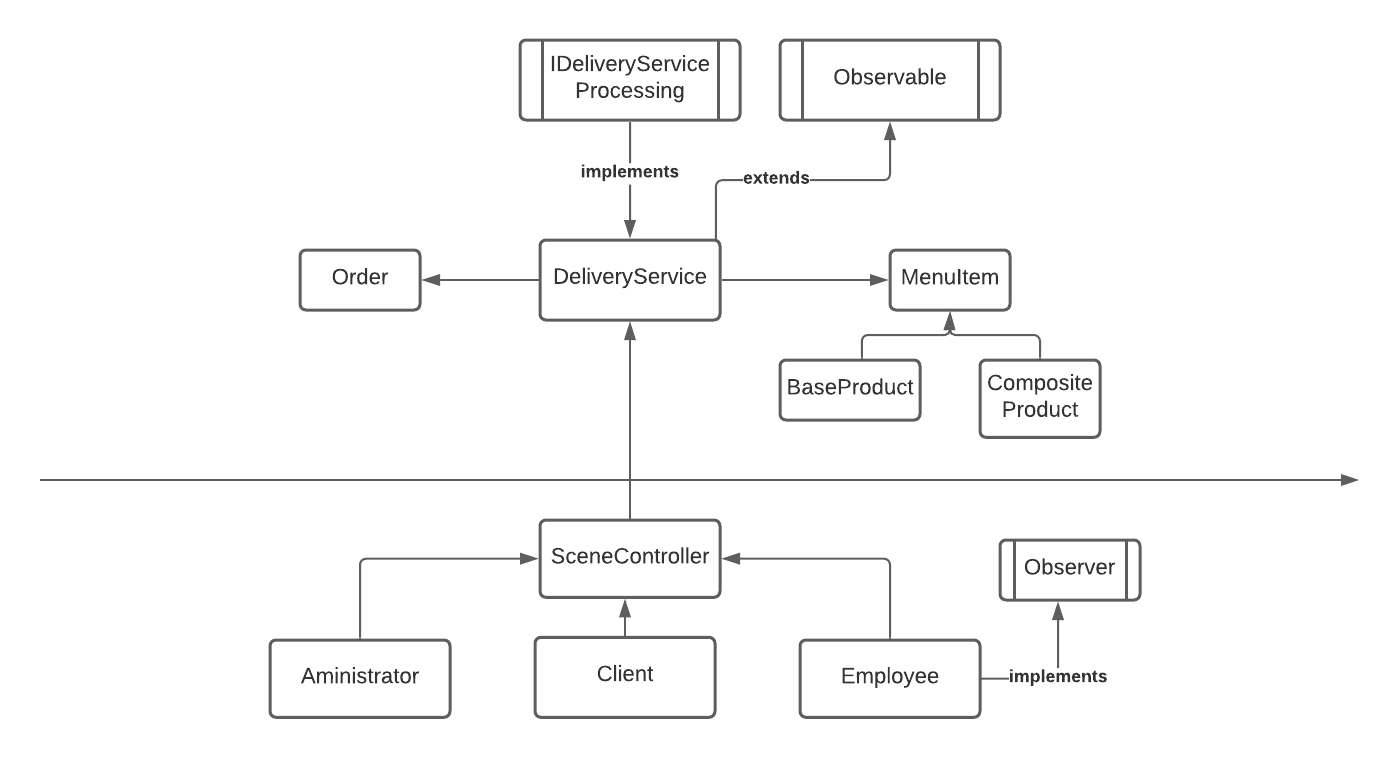
In the administrator View, the user can import, edit or create the products and the products are then written after being saved in the .csv file, as well as the composite products, if any are created. The Client is able to filter the products, to add some products into the current Order and create the order, after which a bill is generated and the employee is notified about the order that was created.



1. Design
   1. OOP design

Regarding the OOP design, we have used Composite Patter, Design by Contract methods and also Layered architecture, so the project follows the imposed OOP design. I have still used Lombok to generate the Getters, Setters and the Constructors. The models classes only have their fields, which are private, so good OOP design was used here as well. The only interface created was the IDeliveryService and its use can be easily seen in the DeliveryService class.

* 1. Class diagram / Application diagram

1. 

Above is a simplified application diagram that shows all the relation between classes and how the application works. We can now easily see the Layered Architecture that we were required to make and the relation between the Scene Controller and the DeliveryService, giving the presentation classes a way to manipulate the data through that instance.

* 1. Data Structures

Besides the simple data structures data I have used – integers, Booleans, floats or other primitives, I had to use Map and HashSet data structures, the Map because it was a requirement and the HashSet in order to have more control over the products, as there was supposed to be no duplicates and so the HashSet simplifies this for me.

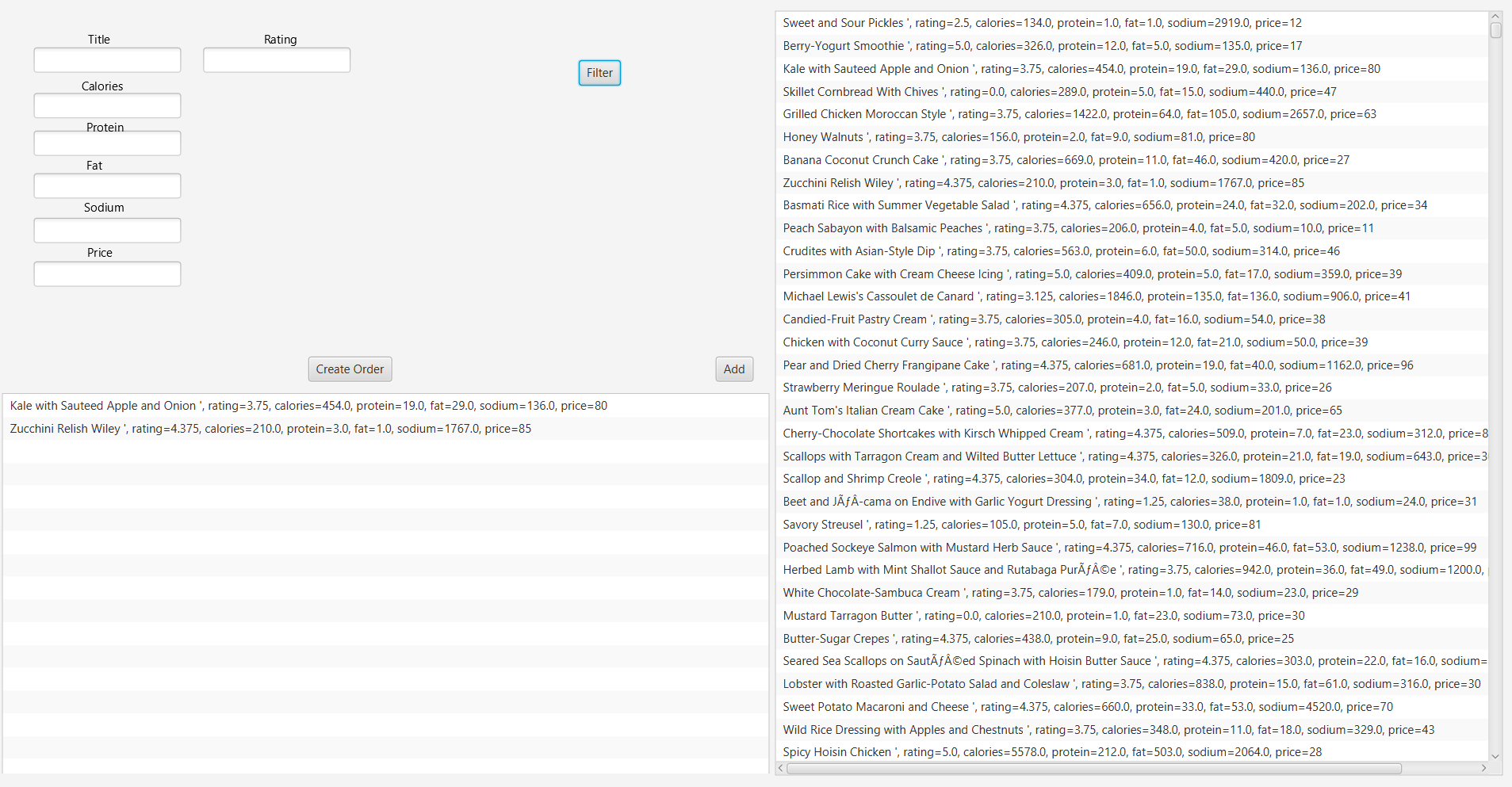
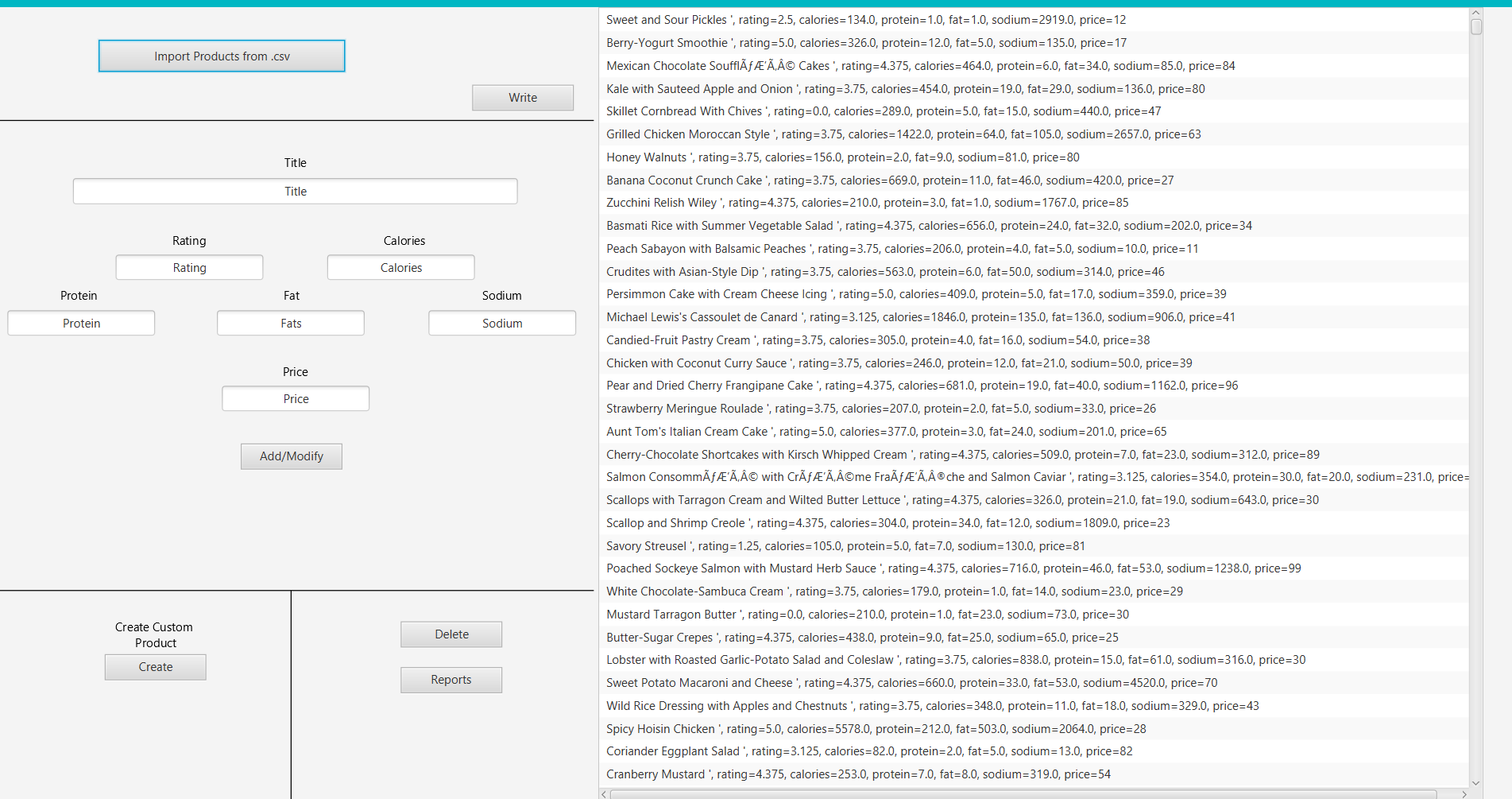
* 1. Class design

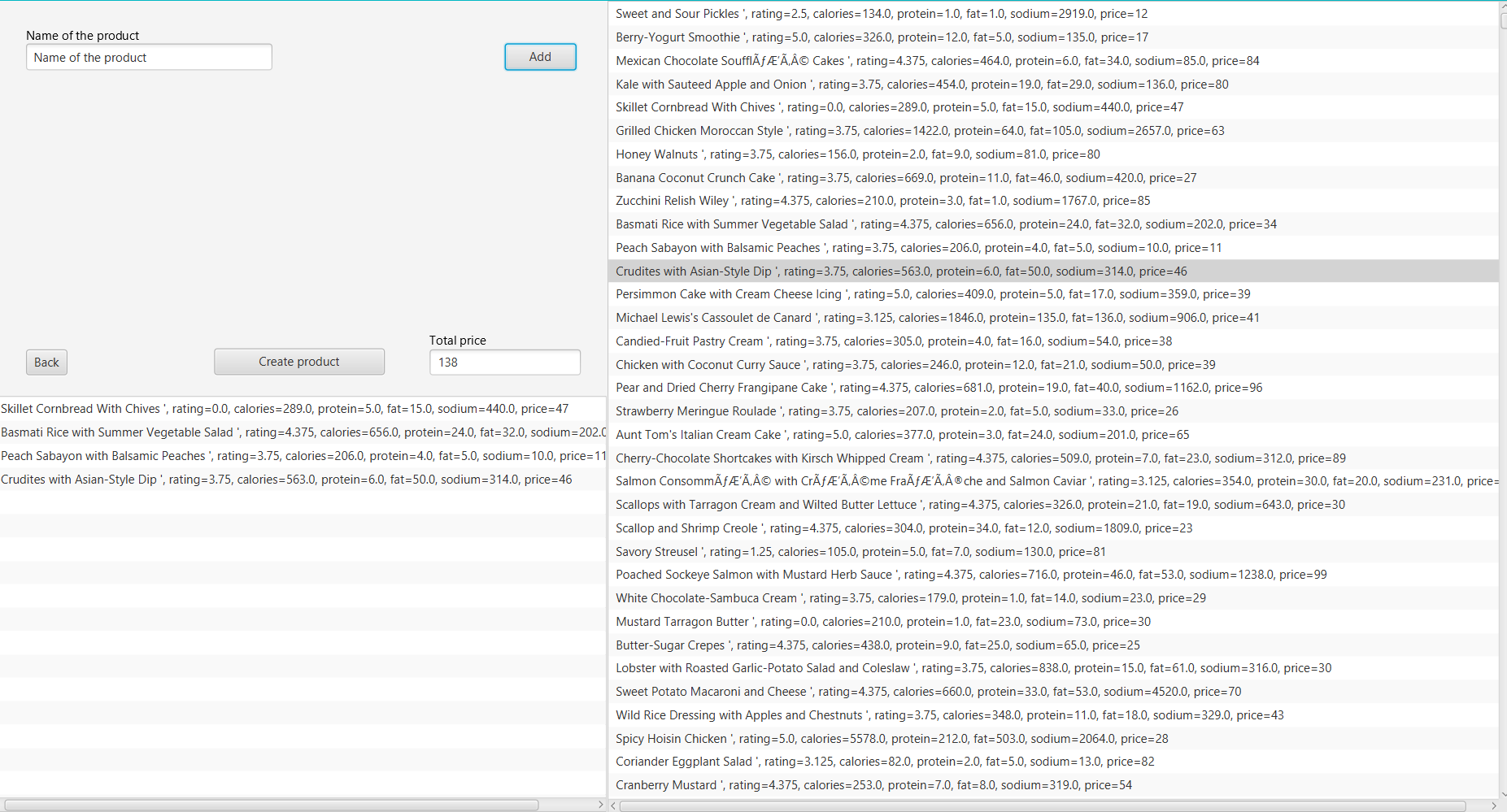
In this part I will briefly explain the class design regarding the division in the packages.

* **model**
  + - User
      * This is simply the User Model class, that contains its fields: id, username and password
    - Order
      * This is the Order Model class that has besides the fields, the override of the equals method and also of the hashCode.
    - BaseProduct
      * The BaseProduct is the class that contains the fields for a base product that is imported from the .csv file and also the equals method that verifies only the title field.
    - CompositeProduct
      * This is a special type of Product, as it is composed of multiple MenuItems
    - MenuItem
      * This is the super or parent class of the two classes mentioned above.
* **businessLayer**
  + - **IDeliveryServiceProcessing**
      * The interface that has the basic methods that the DeliveryService must implement. Among others, are:
        + **importProducts/Orders/Clients**
        + **exportProducts/Orders/Clients**
        + **modify/add/delete Product**
        + **createCustomProduct**
        + **createOrder**
    - **DeliveryService**
      * This is the mot important class of the project as it does most of the things our application needs to do, from importing and exporting, to filtering and creating bills. It defines all the methods inherited from the interface mentioned above and implements them.
      * The class has the fields that save the orders with their corresponding products in a Map<Order, HashSet<MenuItem>> field and it contains the field where the products are stored: HashSet<MenuItem>, as well as a Collection for the Users: ArrayList<User>.
      * The importing/exporting of the products is done using JacksonMapper and CsvSchema/Map, as well as an ObjectWriter and MappingIterator, doing this eases in a very fancy manner the serialization and the deserialization of the data.
      * Notifying the employee is done using the Observer class and it notifies the employee class using the notifyObservers() and setChanged() methods.
* **presentationLayer**
  + - SceneController
      * The class contains the DeliveryService instance and also the user, scene and stage, while also having two important methods: changeScene() which simply changes the scene to one specified as argument and showAlert() which displays and alert message with the parameters given to the function.
    - Aministrator
      * The Administrator controller handles the administrator view window, getting and transmitting requests to/from the delivery service and the interface.
    - Client
      * The Client controller helps the User to add products and create a certain order.
    - ClientLogin, AdministratorReport, AdministratorComposite, ProductInformantion
      * There are also other controllers that handle different kinds of requests
    - MainMenu
      * There is also a controller that behaves as a mainMenu and the User can go to Administrator or to the User

There also exists an **App** class that contains the main function and it has the responsibility of starting the application, which can also be started by using ***gradlew run.***

* 1. UI part





Above we have examples for the 3 UI windows, corresponding to the Administrator, the Client and the AdministratorComposite view windows.

1. Implementation

* **User, Order, MenuItem, CompositeProduct, BaseProduct**
  + Nothing much to be said about the model classes, as they were created using Lombok, they just have the required fields
* **IDeliveryServiceProcessing**
  + The interface defines the methods that must be implemented by the DeliveryService class, this being the only thing it must do.
* **DeliveryService**
  + The class handles most of the requests and saves the data in the fields declared in it.
  + The import functions use the JacksonMapper to build a CsvSchema generated from the classes (BaseProduct, Client, User) and then simply read from the .csv files that already exist in the resource folder.
  + The export functions work similar to the ones that import, but they write using the writeAll method and an ObjectWriter object.
  + Creating a custom product, adding and modifying a product is done simply using the Map or the HashSet and adding/deleting the needed object that is passed as an argument to our function.
  + The creation of the order is done differently, as the Date and the title of the Order must be created in the createOrder() function.
* **Client**
  + The client controller also filters the products using the filter() function besides showing the data and handling the requests from the user. It uses lambda expressions and stream in order to easily filter the products accordingly to the user’s input.
  + There is also a special functionality that the Client controller has: it shows a product whenever the user presses on the item on the ListView or the Composite Product if there is one.
* **Administrator, Administrator Report**
  + The administratorReport has also the function of creating some reporst using lambda expressions and stream that will be generated and showed on the ListView.

1. Results

It was not a requirement to test our methods and it was truly not necessary to test them.

1. Conclusions and further implementation

The conclusion of the project is that we make use of strong OOP concepts and clearly divide it into small sub-problems, the project becomes simpler in no time. Another conclusion is that, when you start documenting your project, you realize that you might have not understood some ideas or your might have not been so clear in your implementation, so it is always better to start your project documenting and stating your ideas and your plan.

I have also learnt new things, like using Composite Design Patter and Design by Contract and a layered Architecture and methods, as well as using Lombok to generate methods that I need without wasting time, especially if my models have a lot of fields. I have also made my fourth project using the MCV pattern, so I have consolidated a more in-depth understanding of the pattern. I have learnt how to create a good documentation and good code documentation comments, while IntelliJ helped me with the Collections and Arrays functions. The other important thing I have got better at is working with git, git bash and GitLab, which is very important for the future. Also, I have learnt how to use streams, which seem to be very important and useful.

This is also my second project with Gradle and I find it very useful, just as useful as Maven is, but simpler and easier to follow **build.gradle** than **pom.xml**.

The other thing that I have learnt and I think it is very important is JavaDoc and using it gave me the opportunity to generate .html files of java documentation of my project.

A further implementation for the project might be to create a more user-friendly interface and give the user more control over the data and over his orders/products, while also giving more control for the administrator and giving more responsibilities for the employee.

1. Bibliography

* [**https://www.baeldung.com/java-jdbc**](https://www.baeldung.com/java-jdbc)
* [**https://app.diagrams.net/**](https://app.diagrams.net/)
* **https://lucid.app/**
* [**https://dzone.com/articles/layers-standard-enterprise**](https://dzone.com/articles/layers-standard-enterprise)
* [**https://www.atlassian.com/git/tutorials/setting-up-a-repository**](https://www.atlassian.com/git/tutorials/setting-up-a-repository)
* [**https://www.baeldung.com/javadoc**](https://www.baeldung.com/javadoc)
* [**https://en.wikipedia.org/wiki/Design\_by\_contract**](https://en.wikipedia.org/wiki/Design_by_contract)
* [**https://www.geeksforgeeks.org/composite-design-pattern/**](https://www.geeksforgeeks.org/composite-design-pattern/)