MINISTRY OF EDUCATION AND RESEARCH
TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

PROGRAMMING TECHNIQUES

ASSIGNMENT 4

FOOD DELIVERY MANAGEMENT SYSTEM

Reckerth Daniel-Peter

30444/2

Contents

I. Assignment Objective

The fourth assignment aims to create and implement a catering company's food delivery management system. The client can select items from the company's menu. The system should have three different types of users who log in with a username and password: administrators, regular employees, and clients.

II. Analysis, Modelling, Scenarios, Use Cases

1. Problem Analysis

The real-life problem which arises is how to efficiently manage some food delivery's company catering services. We must take into consideration that this implies we have a menu which can be worked upon, like adding, creating, modifying, and deleting products from it and that we have clients which can make orders based on the menu. Regarding employees, they should be notified when a new order is made for the catering firm to process it, start it, and deliver it.

2. Modelling the Problem

When we talk about modelling, we talk about high-level abstractions, devoid of the complexity and low-level details.

To model the problem, we must think of many things like users, menu, and their actions. To model the user, we have thought of a class representing it with username, password, and a role in order to discriminate between and administrator, client and employee.

The actions per see that a user may perform are found in a so-called `DeliveryServiceProcessing` interface, which will be explained later.

The products can be of two types: base products or composite products made of smaller base products. To model this situation, we have followed the composite design pattern used for modelling them.

The menu is initially populated from a .csv file, removing duplicates.

3. Use Case Scenarios

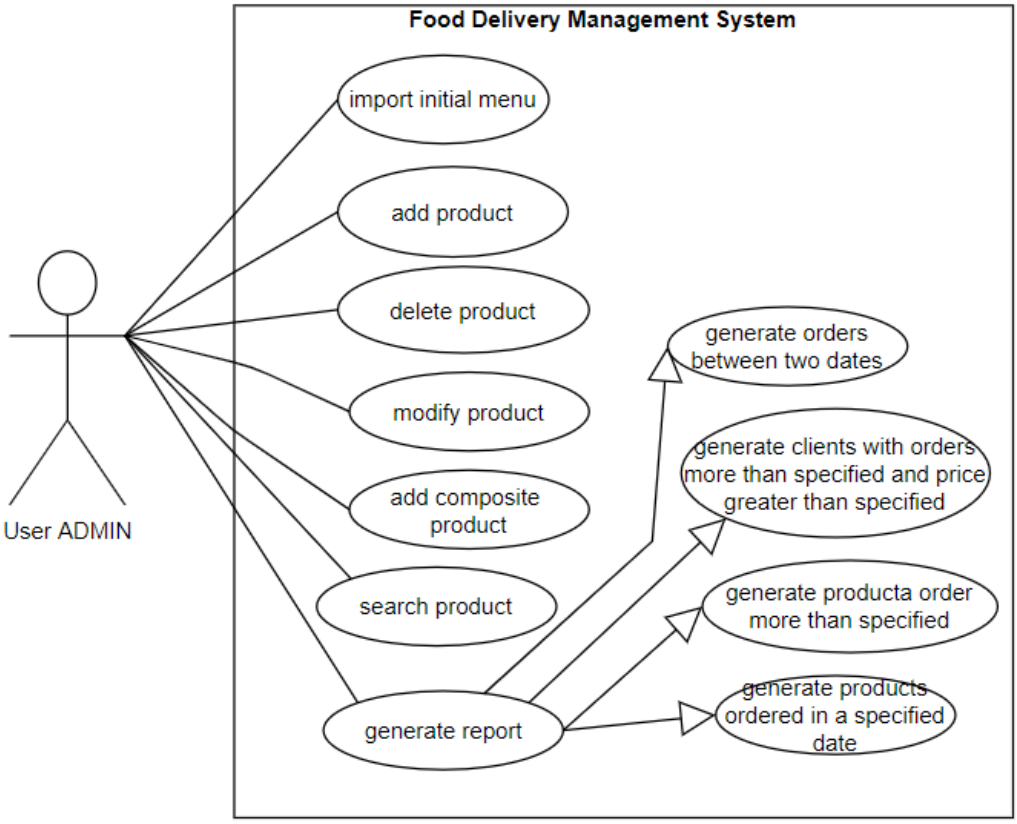
Scenarios represent different actions that can be taken by the user of the application. Together with a use case, they represent a list of actions or event steps typically defining the interactions between a role (known in UML as an actor) and the system to achieve a goal. [Wikipedia]

Therefore, a use case is an important and indispensable requirement analysis technique. Some actions the admin can take which we talked before are:

- Import the initial set of products which will populate the menu from a .csv file.

- Manage the products from the menu: add/delete/modify products and create new products composed of several products from the menu (an example of composed product could be named “daily menu 1” composed of a soup, a steak, a garnish, and a dessert).

- Generate reports about the performed orders considering the following criteria:
    - time interval of the orders – a report should be generated with the orders performed between a given start hour and a given end hour regardless the date.
    - the products ordered more than a specified number of times so far
    - the clients that have ordered more than a specified number of times so far and the value of the order was higher than a specified amount.
    - the products ordered within a specified day with the number of times they have been ordered.

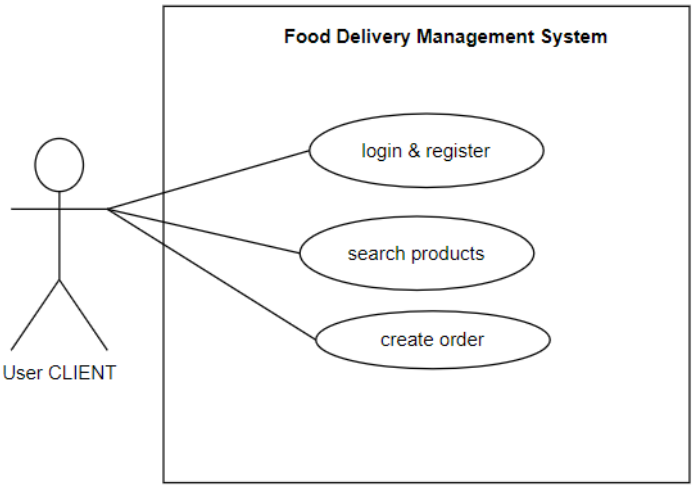


The client can:

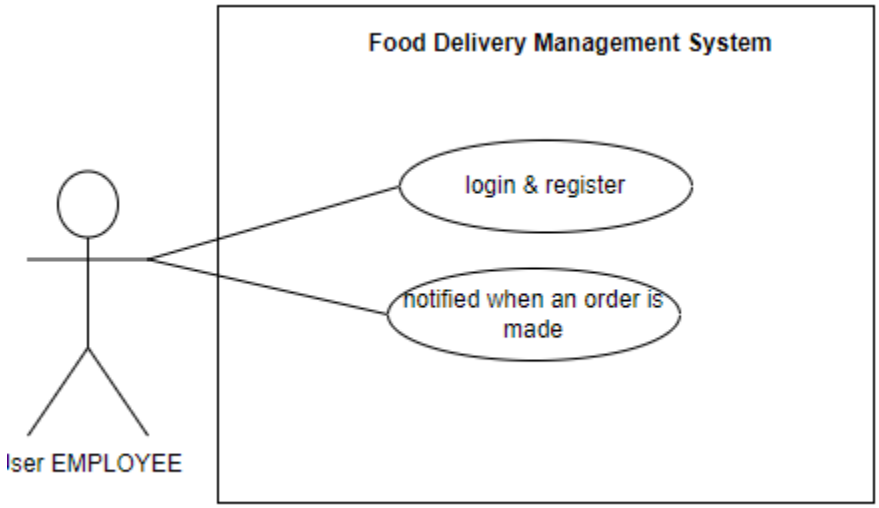
- Register and use the registered username and password to log in within the system.
- View the list of products from the menu.
- Search for products based on one or multiple criteria such as keyword (e.g., “soup”),

rating, number of calories/proteins/fats/sodium/prices.

- Create an order consisting of several products – for each order the date and time will be persisted, and a bill will be generated that will list the ordered products and the total price of the order.



The employee user is notified each time a new order is performed by a client so that it can prepare the delivery of the ordered dishes.



III. Design

1. Design Decisions

Regarding the design decisions we had to take into account that some constraints were imposed by the assignment.

Firstly, we had to define an interface, as a contract, which contains the main operations that can be executed by the administrator and client. The administrator can, as specified in the use case above, import products, manage the products from the menu and generate reports about orders, products, and clients. The client can create orders which implies computing the price, and when an order is made than a bill is generated.

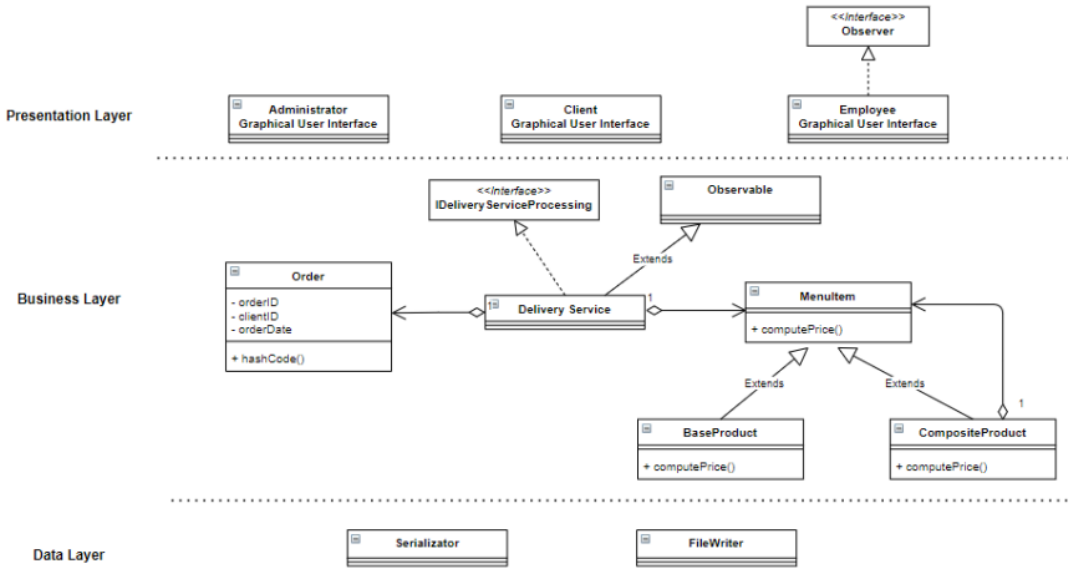
For defining the classes representing the products we had to follow the Composite Design Pattern, which is a structural design pattern that enables us to compose objects into tree structures and then work with these structures as if they were individual objects. Therefore, following the pattern we work with products and composite products through a common abstract class which declares methods for computing the price, the protein, the calories, etc. which is implemented at the level of the classes implementing it. The greatest benefit of this approach is that we don't need to care about the concrete classes of objects that compose the tree. We don't need to know whether an object is a simple product or a sophisticated box. We can treat them all the same via the common interface. When we call a method, the objects themselves pass the request down the tree.

Then for employee notification we had to use the Observer Design Pattern, which is a behavioral design pattern that lets us define a subscription mechanism to notify multiple objects about any events that happened to the object they are observing. This pattern suggests that we add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher. Implementation details will be provided in the next chapter

2. UML Diagrams

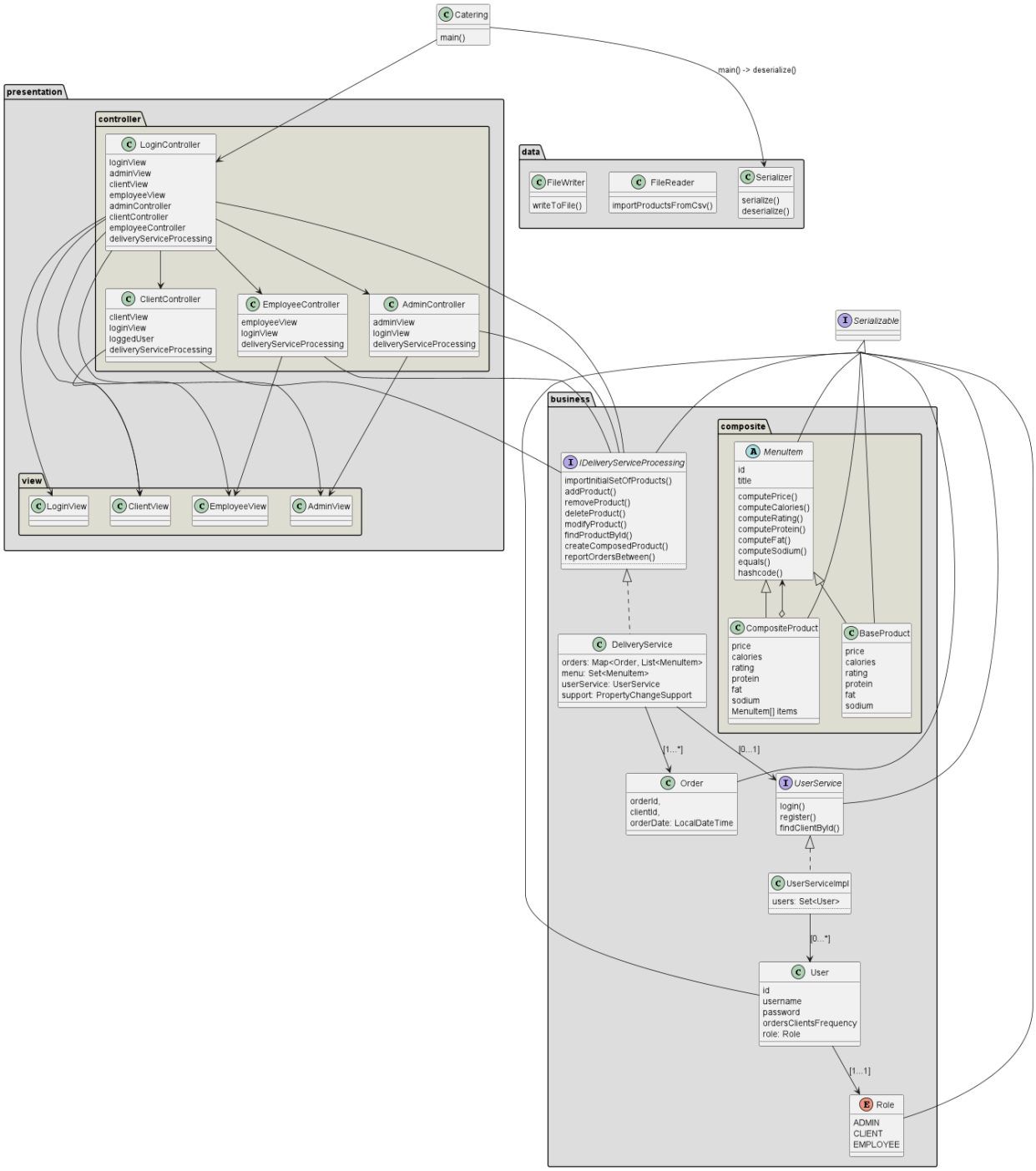
2.1. Class Diagram

We present next the class diagram of the system. To be noted that we also showcase here the packages they are into. Also, we have been provided with an initial class diagram as a starting design of the system. The initial diagram showcases three layers: presentation, data and business which we have followed, the composite pattern and the observer pattern and also additional classes that we should include in the implementation mostly concerning the view classes and specialized classes used for serialization and writing to a file, for generating the bill.



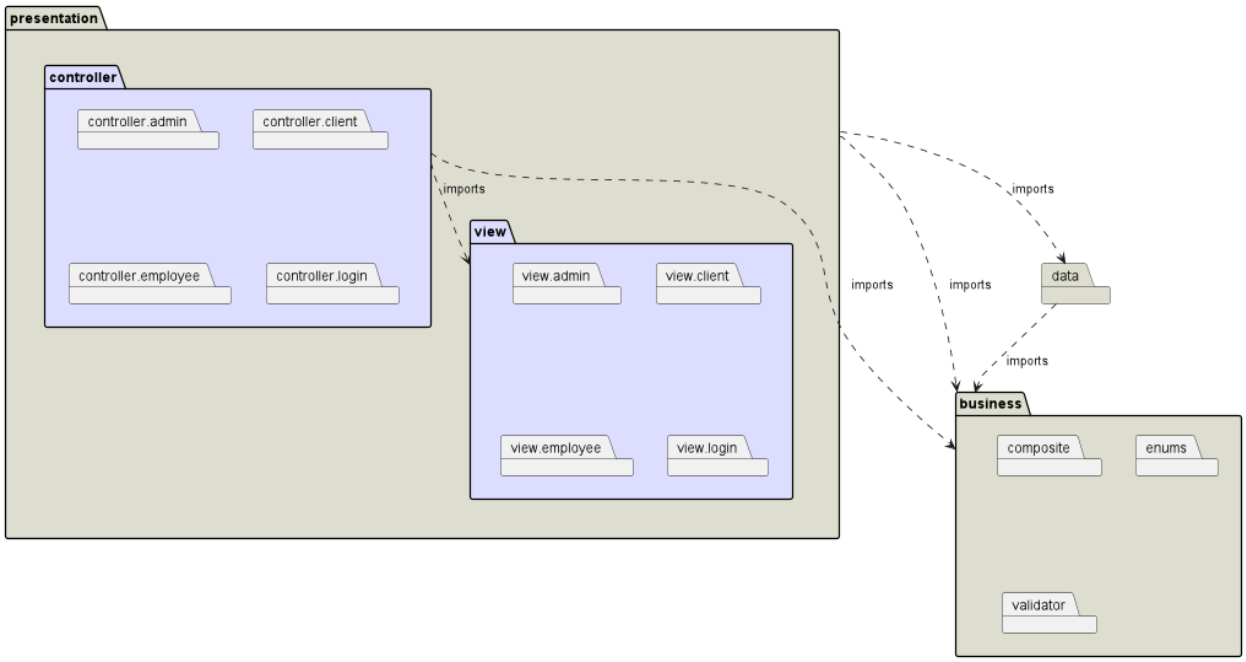
The complete diagram of our class after having the previously as starting point is presented below.

As it can be seen it generally follows the initial diagram however additional classes are included like the ones used for user concerns (user modeled entity, user service) and the controllers, whose role is to bind the view and the business data for the user's actions to take effect.



2.2. Package Diagram

Below is presented the package diagram. As it can be seen most of the imports and usages are within the presentation package where the controllers reside. They import the business package and the sub-package view as they have to work both with the delivery service processing interface and the views of the users based on their role: admin, client or employee. Also, a login view is provide where the main logic of the application starts.



3. Data Structures

Regarding the used data structures, we had to take into account the assignment's suggestions. It is suggested that we use a structure of type Map<Order, Collection<MenuItem>> for storing the order related information in the DeliveryService class. The key of the map will be formed of objects of type Order, for which the hash-code method has been overwritten to compute the hash within the map from attributes of the order class.

For this purpose, we have decided to use the following map: Map<Order, List<? Extends MenuItem>> because an order can contain duplicated products or menu items ordered. However the next requirement suggested we choose a Collection<MenuItem> for storing the actual menu for the clients. For this collection I have decided to use a Set<MenuItem> as each product in the collection should be unique. Parsing the initial csv. file returns already a set so for this reason no duplicates are present. Moreover, the actual map is a HashMap while the set is a LinkedHashSet so that we also preserve the ordering in which menu items are introduced in the set collection from the initial csv and further additions.

In addition, for users we have decided to keep them also in a set so that we do not have duplicated users although checks for the uniqueness of the username are made.

We also use a Set<MenuItems> as the collection part of the composite product for the same reasons of no duplicates as a composite product should be made of base products which are unique.

4. Designing of Classes

With regard to the design of the classes some notable information should be made. The Composite Design Pattern is designed with three main classes: an abstract class MenuItem which defines the id taken and incremented from a static field and the title and which defines the abstract methods of computing the price, rating, calories, protein, fat and sodium for which the implementation

should occur in the classes extending it, namely BaseProduct and CompositeProduct. The pattern is also shown in the fact that a base product simply return its price, calories, rating, etc. while a composite product makes the sum, or the average in case of the rating of its menu items present in its collection.

We have defined two interfaces: IDeliveryServiceProcessing which deals with actions taken by the administrator and client user but also considers the Observer Pattern and the UserService which defines action of login and authorization for all the users. The delivery interface follows the design by contract techniques meaning that Javadoc tags are provided for preconditions, postconditions and the interface's implementation is realized with assert instructions. Also, an invariant for the class implementation is and. This delivery interface implementation also acts as an observable as it has a PropertyChangeSupport registered on it which, when a new order is created fires the property change for the observers (the employee controller) with the newly created order.

5. Algorithms

No special algorithms have been developed in this assignment, but there are some interesting implementations.

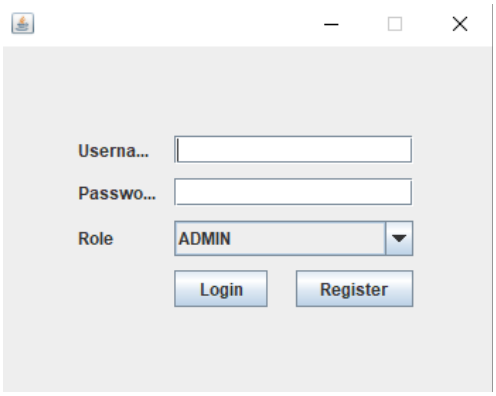
Most of the methods in the delivery processing service are implemented using streams and lambda processing. The reports that the admin cand make (to find products ordered more than a specified amount of time, to report orders between two given local date times or to report the clients with orders more than a specified frequency and price or finding products ordered in a given date) are examples of methods where only streams have been used with lambda predicates.

The csv file with the initial menu items must have been parsed such that duplicates should be removed. The simplicity of the method consists of the fact that the parsed product is added in a set, so we already have the guarantee, due to the set's principle, of no duplicates.

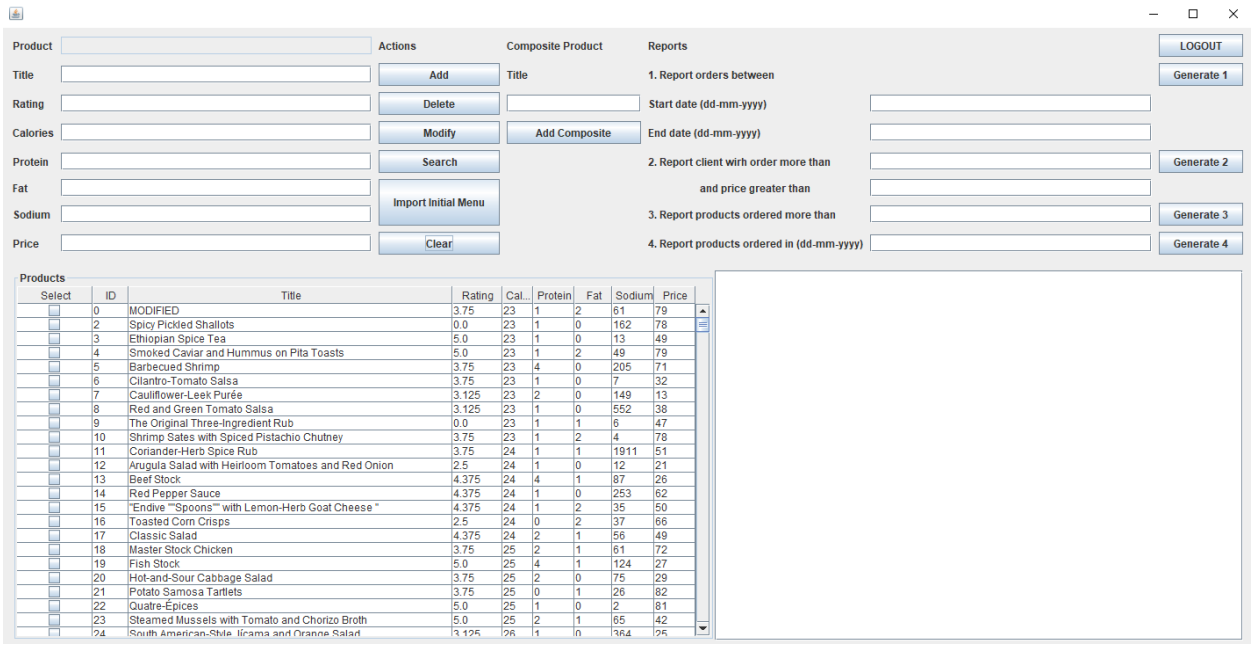
Also worth mentioning is the serializer class in which we serialize the delivery interface and write it to a output file. When the application starts again, we deserialize that file and if some input-output exception occurs (for example if no such file exists at the moment) a new service is created with preregistered users present (one admin, three clients, one employee) and serialized.

6. Interface

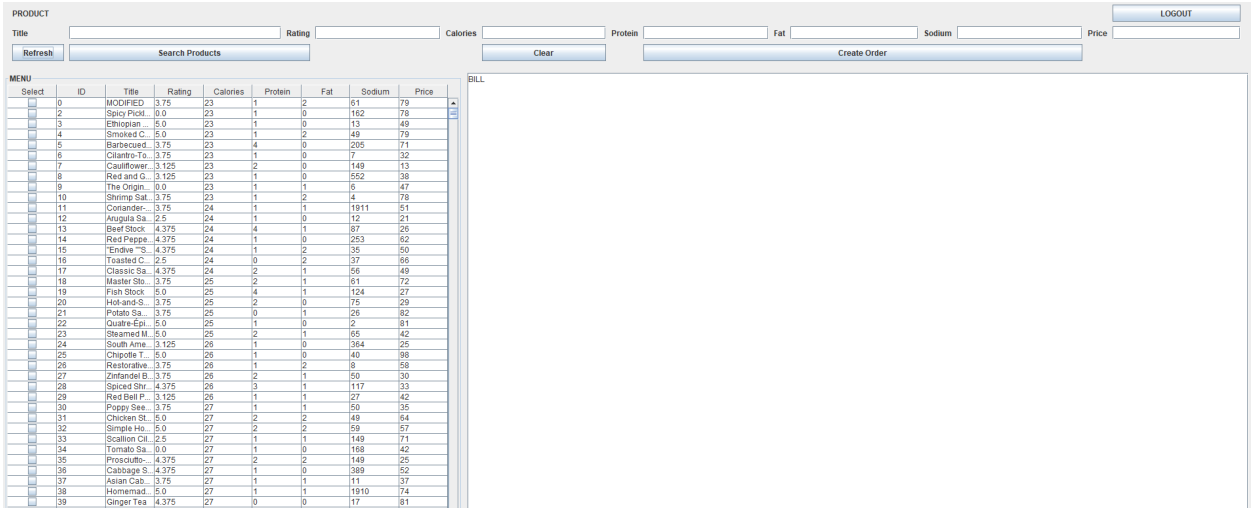
In this chapter we will present some snippets of the user interface. Below we can see the login view. A new user can register by completing the fields, however if the username already exists an error message will appear. The same happens with the login also when a wrong username or password are introduced.



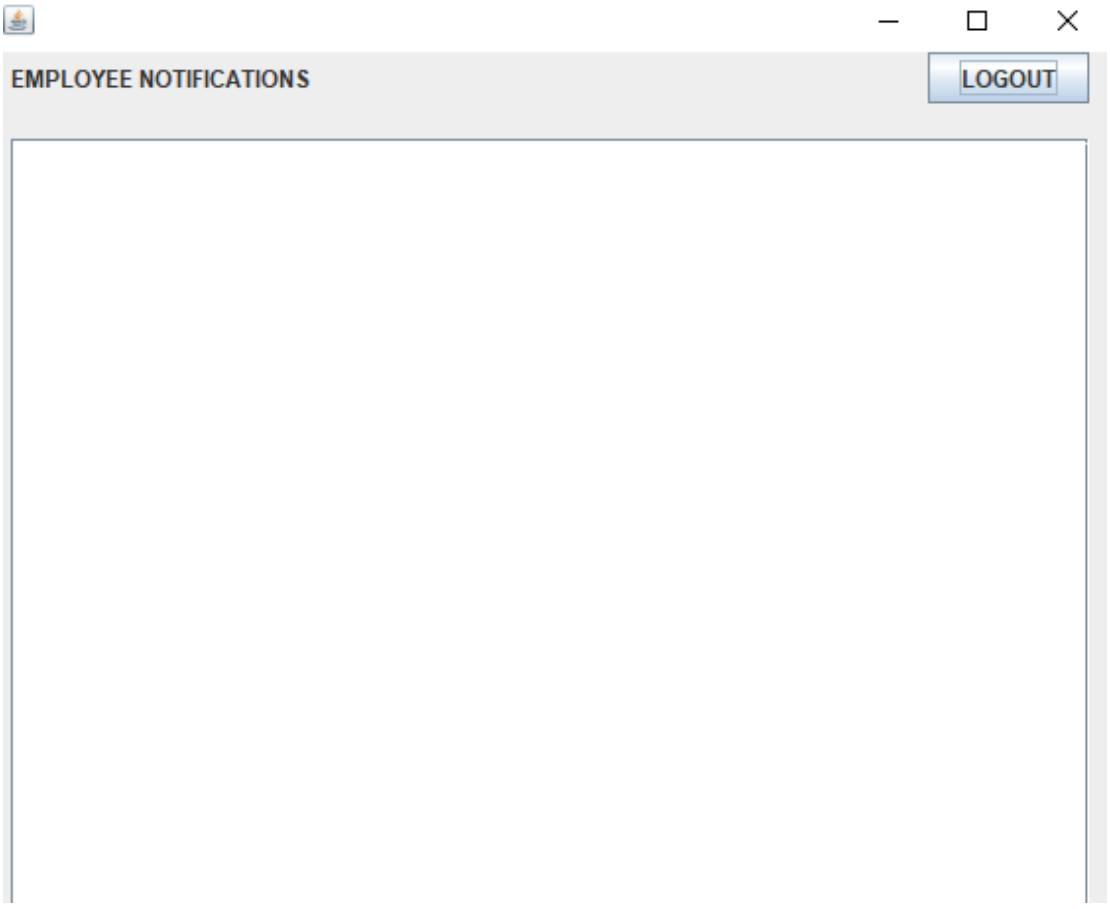
Following we can see the admin view. The admin can perform actions on the menu as it can be seen, or even create composite products by selectin the checkbox of the items it wants the composite product to be made of. Reports can also be generated and will appear in the right text box. All the input fields are validated to not be empty and to be valid (i.e., be numbers or real numbers). The product field contains the id and is not editable.



The client can search for certain products based on the given criteria. To be noted that at least one criterion should be inputted otherwise a warning message will appear, but it can be whatever combinations the client wants. The client can also make an order if he wishes so, by selecting the menu items which he or she wants and pressing the create order button. This will print the order in the text box area on the right but will also generate the order details in a txt following this convention name: BILL + logged client id. Further orders will be appended to this txt file shown as a bill.



The employee view is quite simple as it provides only the created order details via the observer pattern for the employee to prepare.



IV. Implementation

1. Business Logic

In the business logic we have implemented the composite design pattern, the interfaces implementation for the delivery processing and the users service, the order class. Snippets of this implementations will be presented below.

1.1 Composite Menu Item

```java
public abstract class MenuItem implements Serializable {
  private int id = Utils.menuItemId++;

  private String title;

  public MenuItem(String title) { this.title = title; }

  public abstract int computePrice();

  public abstract double computeRating();

  public abstract int computeCalories();

  public abstract int computeProtein();

  public abstract int computeFat();

  public abstract int computeSodium();

  public String getTitle() { return title; }

  public void setTitle(String title) { this.title = title; }

  public int getId() { return id; }

  public void setId(int id) { this.id = id; }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    MenuItem menuItem = (MenuItem) o;
    return title.equals(menuItem.title);
  }

  @Override
  public int hashCode() { return Objects.hash(title); }
}
```

This class represent the abstract menu item defining abstract methods for computing the price, calories, etc.

```java
public class BaseProduct extends MenuItem implements Serializable {

  private double rating;
  private int calories;
  private int protein;
  private int fat;
  private int sodium;
  private int price;

  public BaseProduct(String title, double rating, int calories, int protein, int fat, int sodium

  @Override
  public int computePrice() {
    return this.price;
  }

  @Override
  public double computeRating() {
    return this.rating;
  }

  @Override
  public int computeCalories() { return this.calories; }

  @Override
  public int computeProtein() { return this.protein; }

  @Override
  public int computeFat() { return this.fat; }

  @Override
  public int computeSodium() { return this.sodium; }

  public double getRating() { return rating; }
```

This is the implementation of the base product. As it can be seen the method implemented from the abstract class just return the price of the base product.

The CompositeProduct class is shown below with the implementation of the abstract methods defined in MenuItem. We must iterate in the collection and compute the sum of the price for each base product of it. Same goes for the other compute methods.

```java
public class CompositeProduct extends MenuItem implements Serializable {

  private Set<MenuItem> menuItems;

  public CompositeProduct(String title) {
    super(title);
    this.menuItems = new HashSet<>();
  }

  @Override
  public int computePrice() {
    return menuItems.stream() Stream<MenuItem>
            .map(MenuItem::computePrice) Stream<Integer>
            .mapToInt(Integer::intValue) IntStream
            .sum();
  }

  @Override
  public double computeRating() {
    final double rating = menuItems.stream() Stream<MenuItem>
            .map(MenuItem::computeRating) Stream<Double>
            .mapToDouble(Double::doubleValue) DoubleStream
            .average() OptionalDouble
            .orElse( other: 0.0);
    final DecimalFormat df = new DecimalFormat( pattern: "#.##");
    return Double.parseDouble(df.format(rating));
  }

  @Override
  public int computeCalories() {
    return menuItems.stream() Stream<MenuItem>
            .map(MenuItem::computeCalories) Stream<Integer>
            .mapToInt(Integer::intValue) IntStream
```

1.2. IDeliveryServiceProcessing and implementation

Next, we will show the interface of the delivery system.

```java
public interface IDeliveryServiceProcessing extends Serializable {

  void importInitialSetOfProducts(String filePath) throws IOException;

  /**
   * Add a new base product to the menu.
   *
   * @param product the product to add
   * @pre product != null
   */
  void addProduct(BaseProduct product);

  /**
   * Delete a product from the menu by id;
   *
   * @param id the id of the product to delete
   * @pre productId >= 0
   * @post menu.size() = menu.size@pre - 1
   */
  void deleteProduct(int id);

  /**
   * Modify the product, i.e. updating it.
   *
   * @param productId       the id of the product to update
   * @param updatingProduct the product to update
   * @pre orderId >= 0
   * @pre updatingProduct != null
   */
  void modifyProduct(int productId, BaseProduct updatingProduct);

  /**
   * Get the product with the given id.
   *
   * @param id the id of the product to find
   * @return the product with the given id
   * @pre orderId >= 0
   */
  MenuItem findProductById(int id);
```

```java
/**
 * Create a composed product based on the given base products.
 *
 * @param title      the title of the new product
 * @param productIds the ids of the products to add to the new product
 * @pre title != null
 * @pre productIds != null
 * @post menu.size() = menu.size@pre + 1
 */
void createComposedProduct(String title, List<Integer> productIds);

/**
 * Get orders between two dates.
 *
 * @param startDate the start date
 * @param endDate   the end date
 * @return the orders between the given dates
 * @pre startDate != null
 * @pre endDate != null
 */
List<Order> reportOrdersBetween(LocalDateTime startDate, LocalDateTime endDate);

/**
 * Get the menu items order more than specified number of times.
 *
 * @param numberOfTimes the number of times
 * @return the menu items ordered more than specified number of times
 * @pre numberOfTimes >= 0
 */
List<MenuItem> reportProductsOrderedMoreThan(int numberOfTimes);
```

As it can be seen Javadoc comments have also been included. The implementation of the class is shown below.

```java
/**
 * This class is responsible for implementing the delivery interface.
 *
 * @author Daniel Reckerth
 * @invariant wellFormed()
 */
public class DeliveryService implements IDeliveryServiceProcessing {
  private final Map<Order, List<? extends MenuItem>> orders;
  private Set<MenuItem> menu;
  private final UserService userService;
  private final PropertyChangeSupport support = new PropertyChangeSupport( sourceBean: this);

  public DeliveryService(UserService userService) {
    this.orders = new HashMap<>();
    this.userService = userService;
  }


  @Override
  public void importInitialSetOfProducts(String filePath) throws IOException {
    assert filePath != null;
    menu = FileReader.importProductsFromCsv(filePath);
  }

  @Override
  public void addProduct(BaseProduct product) {
    assert product != null;
    assert wellFormed();
    menu.add(product);
  }

  @Override
  public void deleteProduct(int id) {
    assert id >= 0;
    assert wellFormed();
    menu.removeIf(menuItem -> menuItem.getId() == id);

  }
```

```java
@Override
public List<Order> reportOrdersBetween(LocalDateTime startDate, LocalDateTime endDate) {
  assert startDate != null;
  assert endDate != null;
  assert startDate.isBefore(endDate);
  assert wellFormed();
  return orders.keySet().stream()
          .filter(menuItems -> menuItems.getOrderDate().isAfter(startDate) && menuItems.getOrderDate().isBefore(endDate))
          .collect(Collectors.toList());
}

@Override
public List<MenuItem> reportProductsOrderedMoreThan(int numberOfTimes) {
  assert numberOfTimes > 0;
  assert wellFormed();
  List<MenuItem> popularProducts = new ArrayList<>();
  menu.stream()
          .filter(menuItem -> orders.entrySet().stream()
                  .filter(entry -> entry.getValue().contains(menuItem)).count() > numberOfTimes)
          .forEach(popularProducts::add);
  return popularProducts;
}

@Override
public boolean createOrder(Order order, List<MenuItem> orderProducts) {
  assert wellFormed();
  Map.Entry<Order, List<? extends MenuItem>> entry = Map.entry(order, orderProducts);
  try {
    orders.put(order, orderProducts);
    String text = order.toString() + orderProducts + "\nTotal price: " + getOrderTotalPrice(order) + "\n";
    support.firePropertyChange( propertyName: "orders", this.orders, text);
    return true;
  } catch (Exception e) {
    return false;
  }
}
```

As it can be seen we can see the implementation of the reports use stream and lambdas.

1.2 Order class

```java
public class Order implements Serializable {
  private final int orderId = Utils.orderId++;
  private final int clientId;
  private final LocalDateTime orderDate;

  public Order(int clientId, LocalDateTime orderDate) {
    this.clientId = clientId;
    this.orderDate = orderDate;
  }

  public int getOrderId() { return orderId; }

  public int getClientId() { return clientId; }

  public LocalDateTime getOrderDate() { return orderDate; }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Order order = (Order) o;
    return orderId == order.orderId && clientId == order.clientId && orderDate.equals(order.orderDate);
  }

  @Override
  public int hashCode() { return Objects.hash(orderId, clientId, orderDate); }

  @Override
  public String toString() {
    DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

    return "Order " + orderId +
            ", clientId=" + clientId +
            ", orderDate=" + dateTimeFormatter.format(orderDate) + "\n";
  }
}
```

1.3. Concerning Users

```java
public class User implements Serializable {
  private final int id = Utils.userId++;
  private String username;
  private String password;
  private Role role;
  private int ordersClientsFrequency = 0;

  public User(String username, String password, Role role) {
    this.username = username;
    this.password = password;
    this.role = role;
  }

  public int getId() { return id; }

  public String getUsername() { return username; }

  public void setUsername(String username) { this.username = username; }

  public String getPassword() { return password; }

  public void setPassword(String password) { this.password = password; }

  public Role getRole() { return role; }

  public void setRole(Role role) { this.role = role; }

  public int getOrdersClientsFrequency() { return ordersClientsFrequency; }

  public void incrementOrdersFrequency() { this.ordersClientsFrequency++; }
```

```java
public class UserServiceImpl implements UserService {
  Set<User> users;

  public UserServiceImpl() { this.users = new LinkedHashSet<>(); }

  @Override
  public User login(String username, String password) throws IllegalArgumentException {
    final Optional<User> first = users.stream()
            .filter(user -> user.getUsername().equals(username) && user.getPassword().equals(encodePassword(password)))
            .findFirst();

    return first.orElseThrow(() -> new IllegalArgumentException("User not found"));
  }

  @Override
  public boolean register(User user) {
    assert user != null;
    assert user.getUsername() != null;
    assert user.getPassword() != null;
    assert user.getRole() != null;
    if(user.getUsername().isBlank() || user.getPassword().isBlank()) {
      return false;
    }
    user.setPassword(encodePassword(user.getPassword()));
    return users.add(user);
  }

  @Override
  public User findClientById(int id) throws IllegalArgumentException {
    final Optional<User> byId = users.stream()
            .filter(user -> user.getId() == id && user.getRole() == Role.CLIENT)
            .findFirst();
```

2. Data Logic

2.1 File Reader

```java
public class FileReader {

  public static Set<MenuItem> importProductsFromCsv(String filePath) throws IOException {
    Set<MenuItem> menuItems = new LinkedHashSet<>();
    try (BufferedReader br = new BufferedReader(new java.io.FileReader(filePath))) {
      String line = br.readLine(); // skip header
      while ((line = br.readLine()) != null) {
        String[] values = line.split( regex: ",");
        String title = values[0];
        double rating = Double.parseDouble(values[1]);
        int calories = Integer.parseInt(values[2]);
        int protein = Integer.parseInt(values[3]);
        int fat = Integer.parseInt(values[4]);
        int sodium = Integer.parseInt(values[5]);
        int price = Integer.parseInt(values[6]);
        MenuItem product = new BaseProduct(title, rating, calories, protein, fat, sodium, price);
        if (!menuItems.add(product)) {
          Utils.menuItemId--;
        }
      }
      return menuItems;
    }
  }

}
```

2.2 File Writer

```java
public class FileWriter {

  public static void writeToFile(String fileName, String content) {
    try (BufferedWriter br = new BufferedWriter(new java.io.FileWriter(fileName,  append: true))) {
      br.write(content);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

2.3 Serializer

```java
public class Serializer {

  public static void serialize(IDeliveryServiceProcessing deliveryServiceProcessing) {
    // Serialization
    try {
      FileOutputStream file = new FileOutputStream( name: "file.ser");
      ObjectOutputStream out = new ObjectOutputStream(file);
      out.writeObject(deliveryServiceProcessing);
      out.close();
      file.close();
      System.out.println("Object has been serialized");
    } catch (IOException ex) {
      System.out.println("IOException is caught");
    }
  }

  public static IDeliveryServiceProcessing deserialize() {
    IDeliveryServiceProcessing deliveryService;
    try {
      FileInputStream file = new FileInputStream( name: "file.ser");
      ObjectInputStream in = new ObjectInputStream(file);
      deliveryService = (IDeliveryServiceProcessing) in.readObject();
      System.out.println("Object has been deserialized");
      in.close();
      file.close();
      return deliveryService;
    } catch (IOException ex) {
      System.out.println("IOException is caught");
      UserService userService = new UserServiceImpl();
      userService.register(new User( username: "admin",  password: "admin", Role.ADMIN));
      userService.register(new User( username: "client",  password: "client", Role.CLIENT));
      userService.register(new User( username: "client1",  password: "client1", Role.CLIENT));
      userService.register(new User( username: "client2",  password: "client2", Role.CLIENT));
      userService.register(new User( username: "employee",  password: "employee", Role.EMPLOYEE));
      deliveryService = new DeliveryService(userService);
      serialize(deliveryService);
      return deliveryService;
    } catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException is caught");
```

3. Presentation

We will present only the controller class as the view ones are standard Swing class and the GUI view has been presented before.

3.1. Login Controller

```
public class LoginController {

  private final IDeliveryServiceProcessing iDeliveryServiceProcessing;

  private final LoginView loginView;
  private final AdminView adminView;
  private final ClientView clientView;
  private final EmployeeView employeeView;

  private final AdminController adminController;
  private final ClientController clientController;
  private final EmployeeController employeeController;

  public LoginController(LoginView loginView, AdminView adminView, ClientView clientView, EmployeeView employeeView, IDelive
    this.iDeliveryServiceProcessing = iDeliveryServiceProcessing;

    this.loginView = loginView;
    this.adminView = adminView;
    this.clientView = clientView;
    this.employeeView = employeeView;

    this.adminController = new AdminController(adminView, loginView, iDeliveryServiceProcessing);
    this.clientController = new ClientController(clientView, loginView, iDeliveryServiceProcessing);
    this.employeeController = new EmployeeController(employeeView, loginView, iDeliveryServiceProcessing);

    loginView.setLoginButtonListener(new LoginButtonListener());
    loginView.setRegisterButtonListener(new RegisterButtonListener());
  }

  private class LoginButtonListener implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
      String username = loginView.getUsername();
      String password = loginView.getPassword();
      try {
        User loggedUser = iDeliveryServiceProcessing.getUserService().login(username, password);
        switch (loggedUser.getRole()) {
          case ADMIN -> {
```

```
        case EMPLOYEE -> {
          loginView.setVisible(false);
          employeeView.setVisible(true);
        }
        case CLIENT -> {
          loginView.setVisible(false);
          clientView.setVisible(true);
          clientController.setLoggedUser(loggedUser);
        }

      }
    } catch (IllegalArgumentException ex) {
      loginView.showMessage(ex.getMessage(), JOptionPane.ERROR_MESSAGE);
    }
  }
}

private class RegisterButtonListener implements ActionListener {

  @Override
  public void actionPerformed(ActionEvent e) {
    String username = loginView.getUsername();
    String password = loginView.getPassword();
    Role role = loginView.getRole();
    User user = new User(username, password, role);
    if (iDeliveryServiceProcessing.getUserService().register(user)) {
      loginView.showMessage( message: "User created successfully", JOptionPane.INFORMATION_MESSAGE);
    } else {
      loginView.showMessage( message: "Username taken or empty fields!", JOptionPane.ERROR_MESSAGE);
    }
    loginView.clearFields();
  }
}
```

3.2. Admin Controller

```
public AdminController(AdminView view, LoginView loginView, IDeliveryServiceProcessing deliveryServiceProcessing) {
  this.view = view;
  this.loginView = loginView;
  this.deliveryServiceProcessing = deliveryServiceProcessing;
  view.setAddButtonActionListener(new AddProductListener(view, deliveryServiceProcessing));
  view.setDeleteButtonActionListener(new DeleteProductListener(view, deliveryServiceProcessing));
  view.setModifyButtonActionListener(new ModifyProductListener(view, deliveryServiceProcessing));
  view.setGenerate1ButtonActionListener(new GenerateReport1Listener(view, deliveryServiceProcessing));
  view.setGenerate2ButtonActionListener(new GenerateReport2Listener(view, deliveryServiceProcessing));
  view.setGenerate3ButtonActionListener(new GenerateReport3Listener(view, deliveryServiceProcessing));
  view.setGenerate4ButtonActionListener(new GenerateReport4Listener(view, deliveryServiceProcessing));
  view.setSearchButtonActionListener(new SearchProductsListener(view, deliveryServiceProcessing));
  view.setImportMenuButtonActionListener(new ImportButtonListener(view, deliveryServiceProcessing));
  view.setMenuTableActionListener(new ProductsTableMouseListener(view, deliveryServiceProcessing));
  view.setClearButtonActionListener(new ClearButtonActionListener());
  view.setLOGOUTButtonActionListener(new LogoutButtonActionListener());
  view.setAddCompositeButtonActionListener(new AddCompositeProductListener(view, deliveryServiceProcessing));
}

private class ClearButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(java.awt.event.ActionEvent e) {
    view.clear();
    view.fillMenuTable(deliveryServiceProcessing.getMenu());
    view.setReportTxt("");
  }
}

private class LogoutButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(java.awt.event.ActionEvent e) {
    view.dispose();
    loginView.setVisible(true);
  }
}
```

```java
public class AddCompositeProductListener implements ActionListener {
  private final AdminView view;
  private final IDeliveryServiceProcessing deliveryServiceProcessing;

  public AddCompositeProductListener(AdminView view, IDeliveryServiceProcessing deliveryServiceProcessing) {
    this.view = view;
    this.deliveryServiceProcessing = deliveryServiceProcessing;
  }

  @Override
  public void actionPerformed(ActionEvent e) {
    String title = view.getCompositeTitle();
    List<Integer> productIds = view.getCheckedRowsOfJTable();
    if(productIds == null || title.isBlank() ) {
      view.showMessage( message: "Select at least one base product and check title is not empty!", JOptionPane.WARNING_MESSAGE);
    } else {
      deliveryServiceProcessing.createComposedProduct(title, productIds);
      view.showMessage( message: "Added composed product " + title, JOptionPane.INFORMATION_MESSAGE);
      Serializer.serialize(deliveryServiceProcessing);
    }


  }
}
```

3.3. Client Controller

```java
public class ClientController {
  private final ClientView clientView;
  private final LoginView loginView;
  private User loggedUser;
  private final IDeliveryServiceProcessing deliveryServiceProcessing;

  public ClientController(ClientView clientView, LoginView loginView, IDeliveryServiceProcessing deliveryServiceProcessing) {
    this.clientView = clientView;
    this.loginView = loginView;
    this.deliveryServiceProcessing = deliveryServiceProcessing;
    clientView.addRefreshButtonActionListener(new RefreshButtonListener());
    clientView.addSearchProductsButtonActionListener(new SearchProductsListener(clientView, deliveryServiceProcessing));
    clientView.addLOGOUTButtonActionListener(new LogoutButtonActionListener());
    clientView.addClearButtonActionListener(new ClearButtonActionListener());
    clientView.addCreateOrderButtonActionListener(new CreateOrderButtonActionListener());
  }

  public void setLoggedUser(User loggedUser) { this.loggedUser = loggedUser; }

  private class RefreshButtonListener implements java.awt.event.ActionListener {
    @Override
    public void actionPerformed(java.awt.event.ActionEvent e) {
      clientView.fillMenuTable(deliveryServiceProcessing.getMenu());
    }
  }

  private class LogoutButtonActionListener implements ActionListener {
    @Override
    public void actionPerformed(java.awt.event.ActionEvent e) {
      clientView.clear();
      clientView.dispose();
      loginView.setVisible(true);
    }
```

```
private class ClearButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(java.awt.event.ActionEvent e) {
    clientView.clear();
    clientView.clear();
    clientView.fillMenuTable(deliveryServiceProcessing.getMenu());
  }
}

private class CreateOrderButtonActionListener implements ActionListener {
  @Override
  public void actionPerformed(java.awt.event.ActionEvent e) {
    List<Integer> products = clientView.getCheckedRowsOfJTable();
    if (products.isEmpty()) {
      clientView.showMessage( message: "You must select at least one product", JOptionPane.WARNING_MESSAGE);
    } else {
      final List<MenuItem> collect = products.stream()
              .map(deliveryServiceProcessing::findProductById)
              .collect(Collectors.toList());
      final Order order = new Order(loggedUser.getId(), LocalDateTime.now());
      if(deliveryServiceProcessing.createOrder(order, collect)) {
        loggedUser.incrementOrdersFrequency();
        clientView.showMessage( message: "Ordered added!", JOptionPane.INFORMATION_MESSAGE);
        String text = order.toString() + collect + "\nTotal price: " + deliveryServiceProcessing.getOrderTotalPrice(order) + "\n";
        clientView.setBillTxtArea(text);
        FileWriter.writeToFile( fileName: "BILL-" + loggedUser.getId() + ".txt", text);
        Serializer.serialize(deliveryServiceProcessing);
      }

    }
  }
}
```

3.4. Employee Controller

```
public class EmployeeController implements PropertyChangeListener {
  private final EmployeeView view;
  private final LoginView loginView;
  private final IDeliveryServiceProcessing deliveryServiceProcessing;

  public EmployeeController(EmployeeView view, LoginView loginView, IDeliveryServiceProcessing deliveryServiceProcessin
    this.view = view;
    this.loginView = loginView;
    this.deliveryServiceProcessing = deliveryServiceProcessing;
    this.deliveryServiceProcessing.addPropertyChangeListener( pcl: this);
    view.addLogoutButtonListener(new LogoutButtonActionListener());
  }

  @Override
  public void propertyChange(PropertyChangeEvent evt) {
    view.appendTextToNotificationTxtArea(evt.getNewValue().toString() + "\n");
  }

  private class LogoutButtonActionListener implements ActionListener {
    @Override
    public void actionPerformed(java.awt.event.ActionEvent e) {
      view.dispose();
      loginView.setVisible(true);
    }
  }
}
```

It can be seen this class is the observer and implements PropertyChangeListener, overwriting the propertyChange() method. When the new value is being fired on change from the delivery service is sen as parameter here and taken from the event and appended in the text area.

V. Results

Some results are shown here. For example, the bill generated for each client.

4.1. Client 1 Bill

```
Order 0, clientId=1, orderDate=26-05-2022 22:14:57
[Smoked Caviar and Hummus on Pita Toasts
   id=4
   rating=5.0
   calories=23
   protein=1
   fat=2
   sodium=49
   price=79
, Barbecued Shrimp
   id=5
   rating=3.75
   calories=23
   protein=4
   fat=0
   sodium=205
   price=71
]
Total price: 150
Order 1, clientId=1, orderDate=26-05-2022 22:15:02
[Smoked Caviar and Hummus on Pita Toasts
   id=4
   rating=5.0
   calories=23
   protein=1
   fat=2
   sodium=49
   price=79
, Cilantro-Tomato Salsa
   id=6
   rating=3.75
   calories=23
   protein=1
   fat=0
   sodium=7
   price=32
]
Total price: 111
Order 2, clientId=1, orderDate=26-05-2022 22:15:08
[Smoked Caviar and Hummus on Pita Toasts
   id=4
   rating=5.0
   calories=23
   protein=1
   fat=2
   sodium=49
   price=79
, Red and Green Tomato Salsa
   id=8
   rating=3.125
   calories=23
   protein=1
   fat=0
   sodium=552
   price=38
```

```
, The Original Three-Ingredient Rub
   id=9
   rating=0.0
   calories=23
   protein=1
   fat=1
   sodium=6
   price=47
]
Total price: 164
Order 3, clientId=1, orderDate=26-05-2022 22:15:26
[Shrimp Sates with Spiced Pistachio Chutney
   id=10
   rating=3.75
   calories=23
   protein=1
   fat=2
   sodium=4
   price=78
, Coriander-Herb Spice Rub
   id=11
   rating=3.75
   calories=24
   protein=1
   fat=1
   sodium=1911
   price=51
, Arugula Salad with Heirloom Tomatoes and Red Onion
   id=12
   rating=2.5
   calories=24
   protein=1
   fat=0
   sodium=12
   price=21
]
Total price: 150
```

5.2.Client 2

```
VI.   Order 4, clientId=2, orderDate=26-05-2022 22:15:37
      [Smoked Caviar and Hummus on Pita Toasts
         id=4
         rating=5.0
         calories=23
         protein=1
         fat=2
         sodium=49
         price=79
      , Red Pepper Sauce
         id=14
         rating=4.375
         calories=24
         protein=1
         fat=0
         sodium=253
```

```
   price=62
]
Total price: 141
Order 5, clientId=2, orderDate=26-05-2022 22:15:43
["Endive ""Spoons"" with Lemon-Herb Goat Cheese "
   id=15
   rating=4.375
   calories=24
   protein=1
   fat=2
   sodium=35
   price=50
, Toasted Corn Crisps
   id=16
   rating=2.5
   calories=24
   protein=0
   fat=2
   sodium=37
   price=66
, Classic Salad
   id=17
   rating=4.375
   calories=24
   protein=2
   fat=1
   sodium=56
   price=49
]
Total price: 165
Order 6, clientId=2, orderDate=26-05-2022 22:15:49
[Master Stock Chicken
   id=18
   rating=3.75
   calories=25
   protein=2
   fat=1
   sodium=61
   price=72
, Fish Stock
   id=19
   rating=5.0
   calories=25
   protein=4
   fat=1
   sodium=124
   price=27
]
Total price: 99
Order 7, clientId=2, orderDate=26-05-2022 22:15:55
[Quatre-Épices
   id=22
   rating=5.0
   calories=25
   protein=1
   fat=0
   sodium=2
```

```
    price=81
  , Steamed Mussels with Tomato and Chorizo Broth
    id=23
    rating=5.0
    calories=25
    protein=2
    fat=1
    sodium=65
    price=42
  , South American-Style Jícama and Orange Salad
    id=24
    rating=3.125
    calories=26
    protein=1
    fat=0
    sodium=364
    price=25
  ]
  Total price: 148
  Order 9, clientId=2, orderDate=26-05-2022 22:20:13
  [Fish Stock
    id=19
    rating=5.0
    calories=25
    protein=4
    fat=1
    sodium=124
    price=27
  , White Fish Stock
    id=300
    rating=0.0
    calories=56
    protein=10
    fat=1
    sodium=216
    price=15
  ]
  Total price: 42
  Order 10, clientId=2, orderDate=26-05-2022 22:20:24
  [Red and Green Tomato Salsa
    id=8
    rating=3.125
    calories=23
    protein=1
    fat=0
    sodium=552
    price=38
  , The Original Three-Ingredient Rub
    id=9
    rating=0.0
    calories=23
    protein=1
    fat=1
    sodium=6
    price=47
  , Shrimp Sates with Spiced Pistachio Chutney
    id=10
```

```
      rating=3.75
      calories=23
      protein=1
      fat=2
      sodium=4
      price=78
   ]
   Total price: 163
   Order 11, clientId=2, orderDate=26-05-2022 22:20:37
   [Green-Peppercorn Cornmeal Crackers
      id=67
      rating=5.0
      calories=32
      protein=1
      fat=2
      sodium=53
      price=42
   , Seared Steak Lettuce Cups
      id=68
      rating=5.0
      calories=32
      protein=3
      fat=2
      sodium=70
      price=41
   , Salsa Criolla
      id=69
      rating=5.0
      calories=32
      protein=1
      fat=1
      sodium=435
      price=28
   ]
   Total price: 111
   Order 12, clientId=2, orderDate=26-05-2022 22:20:42
   [Green-Peppercorn Cornmeal Crackers
      id=67
      rating=5.0
      calories=32
      protein=1
      fat=2
      sodium=53
      price=42
   , Seared Steak Lettuce Cups
      id=68
      rating=5.0
      calories=32
      protein=3
      fat=2
      sodium=70
      price=41
   , Salsa Criolla
      id=69
      rating=5.0
      calories=32
      protein=1
```

```
         fat=1
         sodium=435
         price=28
      , Fresh Green Salsa (Salsa verde cruda)
         id=70
         rating=5.0
         calories=32
         protein=1
         fat=1
         sodium=959
         price=99
      , Phyllo Pecan Crisps
         id=71
         rating=5.0
         calories=32
         protein=0
         fat=2
         sodium=9
         price=56
      ]
      Total price: 266
```

5.3. Client 3

```
Order 8, clientId=3, orderDate=26-05-2022 22:16:06
[Potato Samosa Tartlets
   id=21
   rating=3.75
   calories=25
   protein=0
   fat=1
   sodium=26
   price=82
]
Total price: 82
Order 13, clientId=3, orderDate=26-05-2022 22:21:54
[Fish Stock
   id=19
   rating=5.0
   calories=25
   protein=4
   fat=1
   sodium=124
   price=27
]
Total price: 27
Order 14, clientId=3, orderDate=26-05-2022 22:22:10
[Cilantro-Tomato Salsa
   id=6
   rating=3.75
   calories=23
   protein=1
   fat=0
   sodium=7
   price=32
, Cauliflower-Leek Purée
```

```
    id=7
    rating=3.125
    calories=23
    protein=2
    fat=0
    sodium=149
    price=13
]
Total price: 45
```

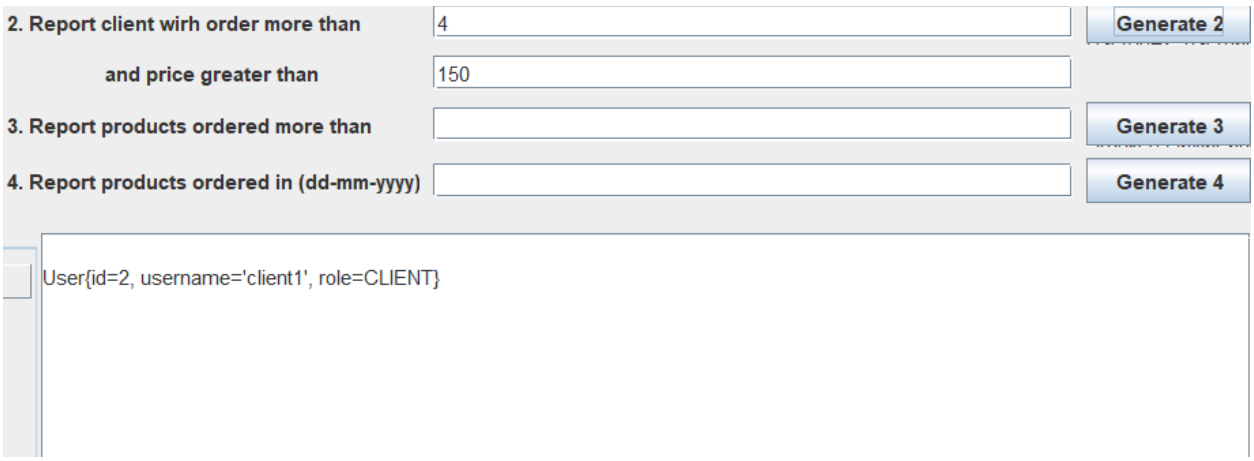
5.4. Reports

These are products ordered strictly more than twice:

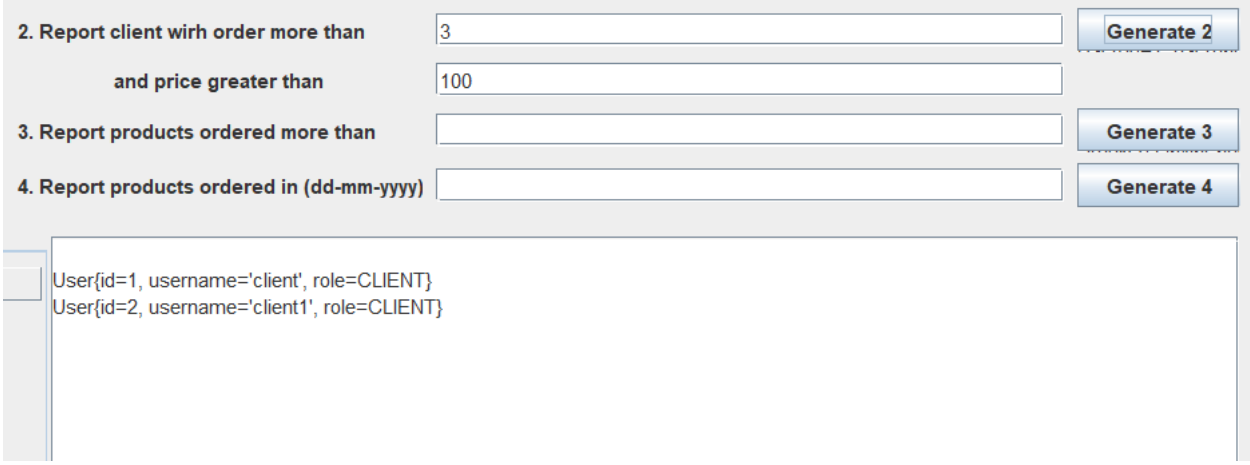
```
Smoked Caviar and Hummus on Pita Toasts
  id=4
  rating=5.0
  calories=23
  protein=1
  fat=2
  sodium=49
  price=79

Fish Stock
  id=19
  rating=5.0
  calories=25
  protein=4
  fat=1
  sodium=124
  price=27
```

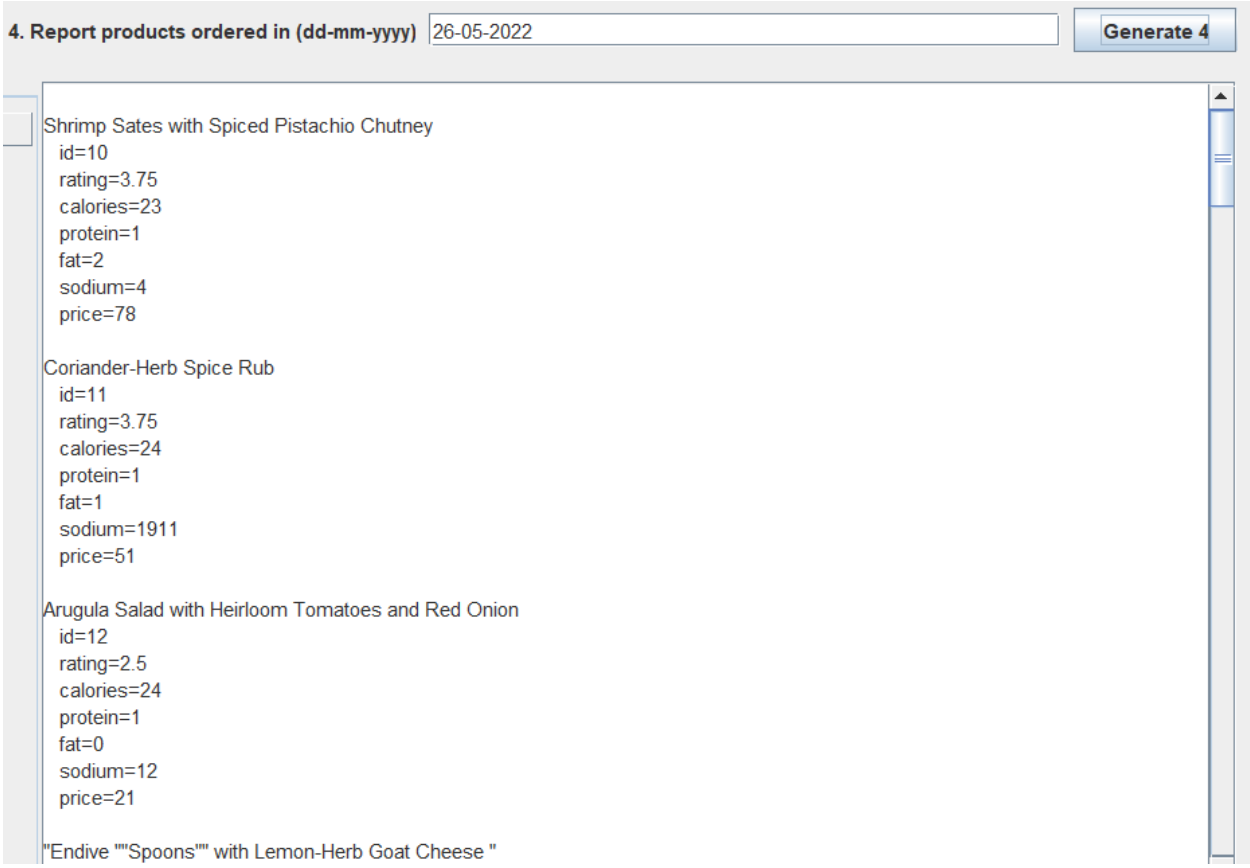
These are the clients with orders more than 4 and price greater than 150.



These are the clients with orders more than 3 and price greater than 100.

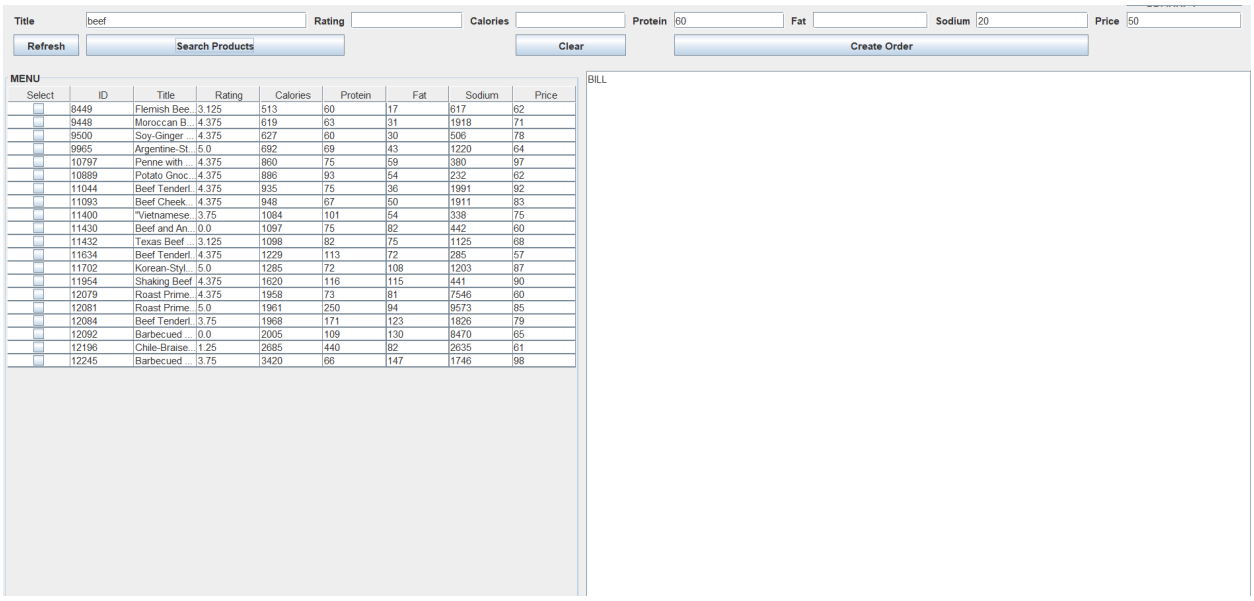


These are some of the products ordered in 26-05-2022.



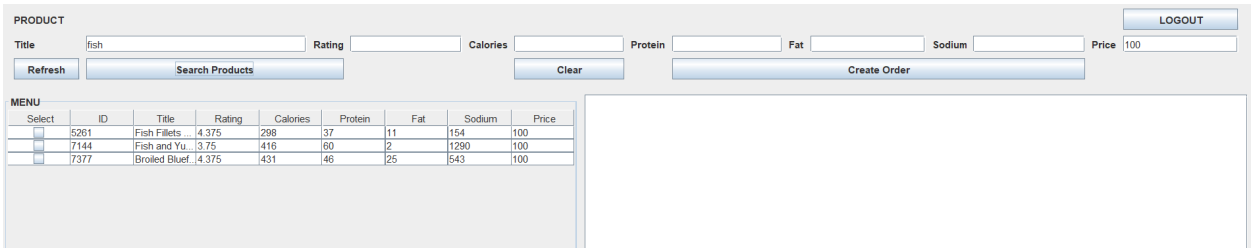
5.5. Search results

These are the results for the following search criteria. Find products containing “beef” in the title with sodium equal or over 20 price over or equal with 50 and protein equal or over with 60.



Select	ID	Title	Rating	Calories	Protein	Fat	Sodium	Price
	8449	Flemish Bee...	3.125	513	60	17	617	62
	9448	Moroccan B...	4.375	619	63	31	1918	71
	9500	Soy-Ginger ...	4.375	627	60	30	506	78
	9965	Argentine-St...	5.0	692	69	43	1220	64
	10797	Penne with ...	4.375	860	75	59	380	97
	10889	Potato Gnoc...	4.375	886	93	54	232	62
	11044	Beef Tenderl...	4.375	935	75	36	1991	92
	11093	Beef Cheek...	4.375	948	67	50	1911	83
	11400	"Vietnamese...	3.75	1084	101	54	338	75
	11430	Beef and An...	0.0	1097	75	82	442	60
	11432	Texas Beef ...	3.125	1098	82	75	1125	68
	11634	Beef Tenderl...	4.375	1229	113	72	285	57
	11702	Korean-Styl...	5.0	1285	72	108	1203	87
	11954	Shaking Beef ...	4.375	1620	116	115	441	90
	12079	Roast Prime...	4.375	1958	73	81	7546	60
	12081	Roast Prime...	5.0	1961	250	94	9573	85
	12084	Beef Tenderl...	3.75	1968	171	123	1826	79
	12092	Barbecued ...	0.0	2005	109	130	8470	65
	12196	Chile-Braise...	1.25	2685	440	82	2635	61
	12245	Barbecued ...	3.75	3420	66	147	1746	98

These are the searches of products of fish equal or over 100 price.



Select	ID	Title	Rating	Calories	Protein	Fat	Sodium	Price
	5261	Fish Fillets ...	4.375	298	37	11	154	100
	7144	Fish and Yu...	3.75	416	60	2	1290	100
	7377	Broiled Bluef...	4.375	431	46	25	543	100

VII. Conclusions

This assignment was very welcomed to understand how to work with streams, lambdas, how to implement some design patterns like composite and observer and how to design an interface by contract taking into consideration preconditions, postconditions, asserts and invariants. It also touched concepts like serialization and deserialization.

Further improvements can be made such as refactoring and better validation of data, although this wasn't the focus of the assignment.

VIII. Bibliography

2022. Programming Techniques - Assignment 4 Support Presentation, Cluj-Napoca: UTCN, AC.

Programming Techniques Course, Cluj-Napoca: UTCN

Composite. Refactoring.Guru. (n.d.). Retrieved May 27, 2022, from https://refactoring.guru/design-patterns/composite

Predrag. (2021, May 15). The observer pattern in java. Baeldung. Retrieved May 27, 2022, from https://www.baeldung.com/java-observer-pattern