

Food Delivery Management System

Miu Daria

Assignment Objectives

Main objective

The main objective of the assignment is to design and implement a food delivery management system for a catering company. The system should have three types of users that log in using a username and a password: administrator, regular employee, and client. The client can order products from the company's menu.

Sub-objectives

- Analyze the problem and identify the functional and non-functional requirements
- Create the design of order management system
- Implement the order management system
- Test the order management system in multiple scenarios

Problem Analysis, Modeling, Scenarios, Use-cases

Problem Analysis

In real world circumstances, such a system is mandatory for a catering company to work normally, it helps to store and organize data. In order to fulfil all the needs for a good and complete food delivery system, the application will accept 3 types of users : admin, employee and client, each of them having the possibility to perform a range of specific actions. The administrator should have the permission to administer the base information for the food delivery system, the client should be able to make orders and the employee should prepare the orders it receives. Each type of user should be able to register and use the registered username and password to log in within the system.

Modeling

Each user will have a set of possible actions according to the following plan:

The administrator can:

- Import products
- Manage the products from the menu: add/delete/modify products and create new products composed of several products
- Generate reports about the performed orders

The client can:

- 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/price.
- 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 is notified each time a new order is performed by a client so that it can prepare the delivery of the ordered dishes.

The data of the app will be stored using serialization in multiple files. To retrieve the data, at each run of the application the files will be deserialized.

To reach its purpose, such an application should meet certain requirements like:

Functional requirements:

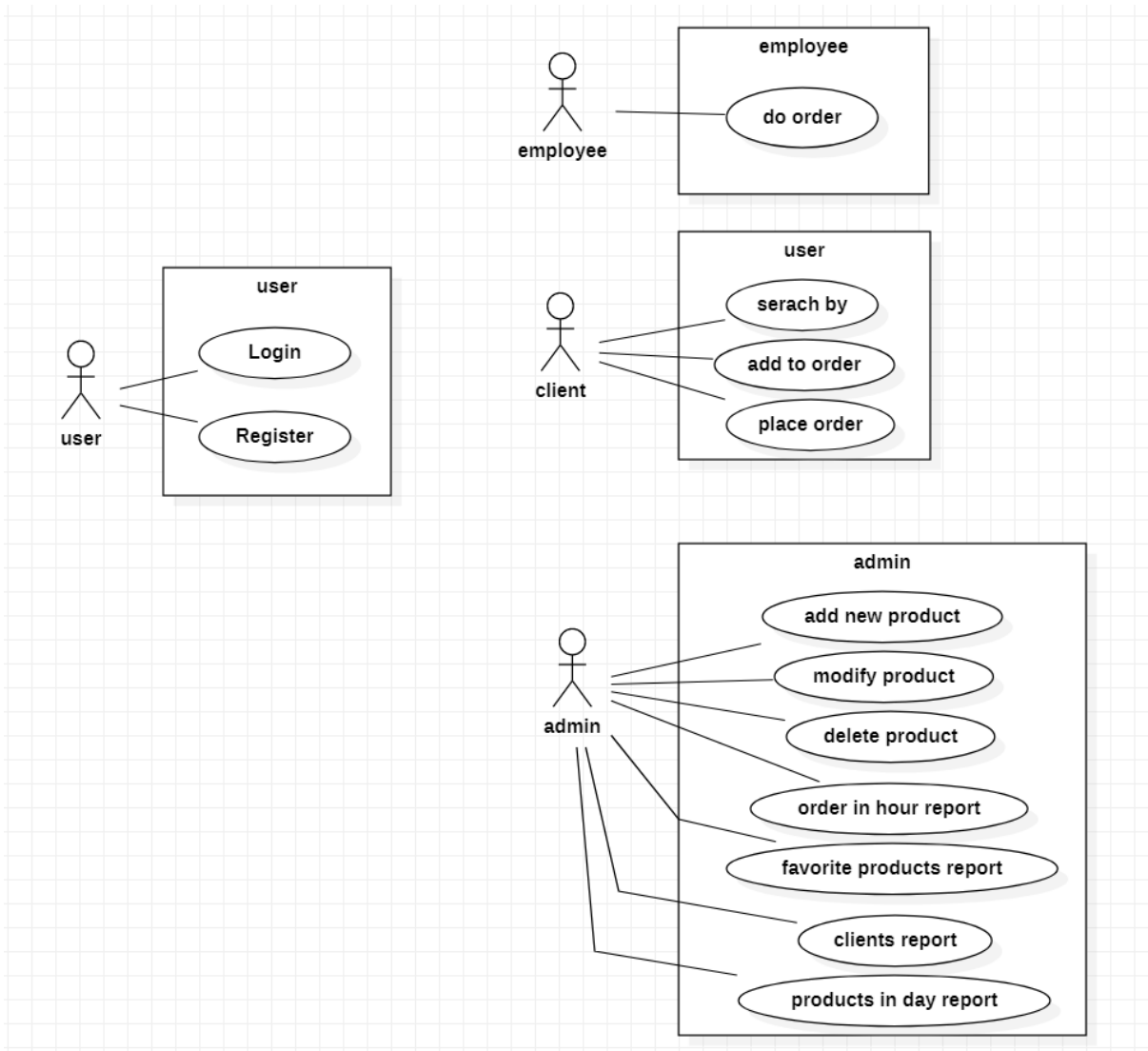
- The application should allow users to login
- The application should provide the administrator the functionalities for adding products, deleting products, updating products and creating reports
- The application should provide the user the functionalities for filtering the menu and making orders
- The application should notify the employee each time an order is created

Non-functional requirements:

- The simulator should be intuitive and easy to use.

Use-cases and Scenarios

Use Case Diagram



Use Case Scenarios

User - Login Button

Successful scenario

step 1: introduces username, password, checks type of account

step 2: user presses the login button

step 3: user is logged in

Unsuccessful scenario

step 1: introduces username, password, checks type of account
step 2: user presses the login button
step 3: the username and the password don't match a valid registered account and an error message is being displayed
step 4: back to step 1

User - Register Button

Successful scenario

step 1 : introduces username, password, checks type of account
step 2 : user presses the create account button
step3 : account is created, an information message is displayed

Unsuccessful scenario

step 1 : introduces username, password, checks type of account
step 2 : user presses the create account button
step3 : the username already exists in the application data and an error message is displayed
step 4: back to step 1

Administrator - Add product

Successful scenario

step 1: administrator introduces fields of product
step 2: administrator presses the add button
step 3: the product is created and an information message is displayed

Unsuccessful scenario

step 1: administrator introduces fields of product
step 2: administrator presses the add button
step 3: administrator missed some of the fields or and an error is displayed
step4 : back to step 1

Administrator - Delete product

Successful scenario

step 1: administrator selects the item to delete
step 2: administrator selects the delete button
step 3: the product is deleted

Unsuccessful scenario

step 1: administrator selects the item to delete
step 2: administrator selects the delete button
step 3: something goes wrong in the application and nothing happens

Administrator - Update product

Successful scenario

step 1: administrator selects the item to update
step 2: administrator updates the fields

step 3: administrator selects the update button

step 4: the product is updated

Unsuccessful scenario

step 1: administrator selects the item to update

step 2: administrator updates the fields

step 3: administrator selects the update button

step 4: something goes wrong in the application and nothing happens

Administrator - Create report 1

Successful scenario

step 1: administrator introduces the parameters for the report

step2: administrator clicks the generate report button

step 3: a report is generated

Unsuccessful scenario

step 1: administrator introduces the parameter for the report

step2: administrator clicks the generate report button

step 3: some data is invalid and an error message is displayed

step 4:back to step 1

Employee- do order

Successful scenario

step 1: employee selects an entry from the table

step 2 : employee presses the done button

step 3: the order is deleted from the table

Unsuccessful scenario

step 1: employee selects an entry from the table

step 2 : employee presses the done button

step 3: something goes wrong in the application and nothing happens

Client - search button

Successful scenario

step 1: client introduces data in wanted fields

step 2: client presses the search button

step 3: the table is filtered

Unsuccessful scenario

step 1: client introduces data in wanted fields

step 2: client presses the search button

step 3: some data is invalid and an error message is displayed

step 4: back to step 1

Client - add button

Successful scenario

step 1: client selects item

step 2: client presses the add button

step 3: the product is added to the order

Unsuccessful scenario

step 1: client selects item

step 2: client presses the add button

step 3: something goes wrong in the application and nothing happens

Client - finalize order button

Successful scenario

step 1: client presses the finalize order button

step 2: the order is created and a message appears informing that a bill was generated

Unsuccessful scenario

step 1: client presses the finalize order button

step 2: something goes wrong and nothing happens

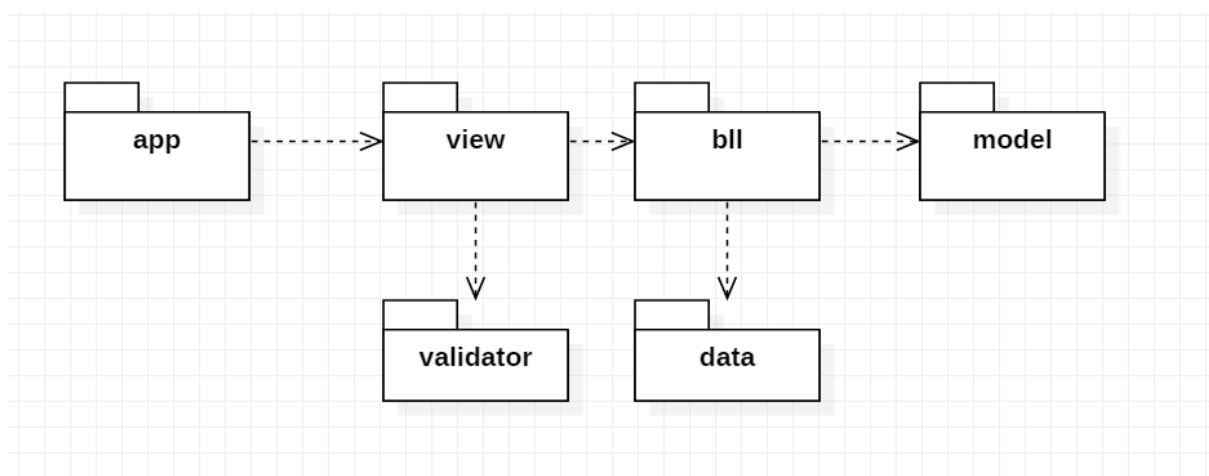
Design

The application is structured in packages using a layered architecture. The purpose of using a layered architecture is to split the application into different layers. Each layer has a special purpose and calls functions of the layers below it.

To clearly represent this pattern, the application is divided into multiple packages.

Packages

Package Diagram



The app Package: contains only the **Main** class, which is run to start the application.

The model Package: contains the **User, Order, MenuItem, BaseProduct, CompositeProduct** classes, the fundamental objects of the app, that help organize the data with its specific properties.

The data Package: contains the classes **FileWriter, FileReader, Serializer** used for accessing and updating the data of the application.

The validator Package: contains a class **InputValidator** that contains methods for validating the inputs of the application

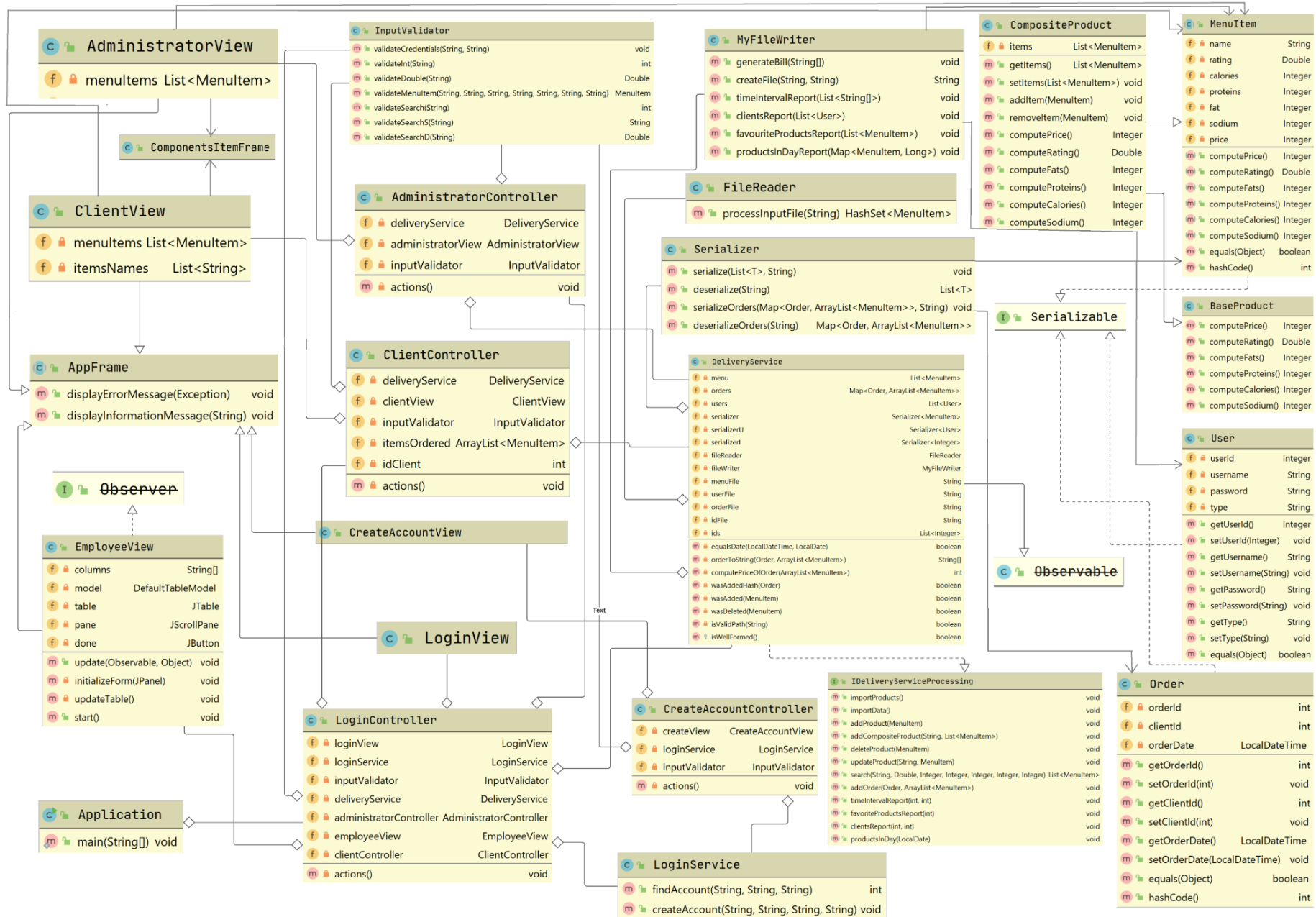
The blt Package: contains the classes that implement the application logic : **DeliveryService, LoginService.**

The view Package: contains the graphical user interface classes: **AppFrame, ClientView, AdministratorView, EmployeeView, CreateAccountView, LoginView, ComponentsItemFrame** and the controllers for each graphical user interface: **ClientController, AdministratorController, Employee Controller, CreateAccountController, LoginController.** They help to create a clear connection between the view displayed to the user and the business logic of the application.

Classes

Class Diagram

The relationships between classes are represented in the following diagram on the next page.



Design decisions

For implementing the functionality of the system some design patterns were used.

The Observer Design Pattern : this pattern was used for sending data from the DeliveryService(Observable) to the EmployeeView(Observer) in order to notify the employee about a new order.

The Design By Contract Pattern : this pattern was used for implementing the DeliveryService class, its most important methods contain pre and post conditions. The class also contains an invariant.

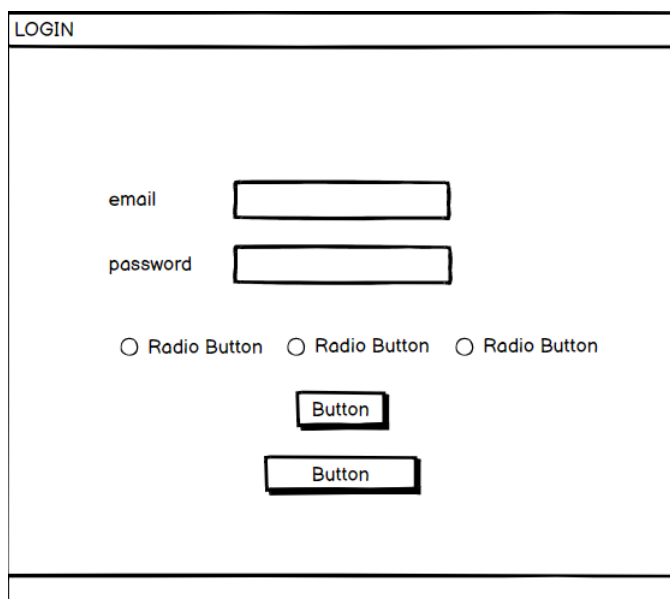
The Composite Design Pattern : the pattern was used for the MenuItem, CompositeProduct classes. The CompositeProduct extends the MenuItem and also contains a List of MenuItems.

The IDeliveryService interface was created to describe the functionalities of the DeliveryService class.

Graphical User Interface - design

The graphical user interface should be intuitive and easy to understand and to use. The idea of the design is the following. When started the graphical user interface will present a login page to lead users to enter the system or to create an account.

Mockup for LoginView



A mockup of a login window titled "LOGIN". The window contains two text input fields labeled "email" and "password". Below these fields are three radio buttons, each followed by the text "Radio Button". At the bottom of the window are two buttons, both labeled "Button".

Mockup for AdministratorView

[illegible]

Mockup for ClientView

[illegible]

Mockup for EmployeeView

[illegible]

Data Structures

Data Structure represents a way in which the data can be stored and organized so that it can be used efficiently and easily. The data structures used in this project are Lists, Maps(HashMaps), HashSets.

The choice for the List data structure, more precisely, ArrayList, was made based on the fact that it comes with multiple helpful features and methods that make the process of writing code simpler and faster. Specific methods such as add, isEmpty or get(index) were used. For iterating through lists the foreach was used. Also lists are a great choice when working with a lot of streams is involved.

The choice for the HashMaps was made based on the need for a way of storing an ArrayList of<MenuItem> for each order. So the HashMaps solve this problem by allowing us to use the Order as a key, and the ArrayList<MenuItem> as the value, also providing unicity for the entries.

HashSets were used when reading the initial list of products, because there were duplicates, so a hash set would guarantee that there are no duplicates. The hashing was made based on the name of the menu items.

Implementation

User Class

The class contains fields needed to identify a user such as password and username, also id and type for storing the data more organized. Besides the attributes the class contains a constructor (with parameters and without parameters) and getters and setters for all its attributes and an overriding of method equals().

Order Class

The class contains fields needed to identify an order such as orderId, clientId, date. Besides the attributes the class contains a constructor (with parameters and without parameters) and getters and setters for all its attributes and an overriding of method equals() and hashCode().

MenuItem, BaseProduct, CompositeProduct Class

MenuItem class is an abstract class extended by BaseProduct and CompositeProduct. The class contains fields needed to identify an item from a menu such as name, rating, number of calories/proteins/fats/sodium/price. Besides the attributes the class contains a constructor (with parameters and without parameters) and getters and setters for all its attributes and an

overriding of method equals(). And also abstract methods created especially for the CompositeProduct which extends this class and contains a list of MenuItems so its values need to be recalculated.

DeliveryService Class

The class implements the base logic of the application. It implements all the methods from the IDeliveryService interface. It contains the following very important fields regarding the data stored:

```
private List<MenuItem> menu = new ArrayList<>();
private Map<Order, ArrayList<MenuItem>> orders = new HashMap<>();
private List<User> users = new ArrayList<>();
private Serializer<MenuItem> serializer = new Serializer();
private Serializer<User> serializerU = new Serializer();
private Serializer<Integer> serializerI = new Serializer();
private FileReader fileReader = new FileReader();
private MyFileWriter fileWriter = new MyFileWriter();
private String menuFile = "menuItems3.ser";
private String userFile = "users2.ser";
private String orderFile = "orders3.ser";
private String idFile = "currentIds.ser";
private List<Integer> ids = new ArrayList<>();
```

Another important aspect of the class is that it contains an invariant to check for its stability.

```
protected boolean isWellFormed() {
    if(menu == null) return false;
    if(users == null) return false;
    if(orders == null) return false;
    if(ids == null) return false;
    if(fileReader == null) return false;
    if(isValidPath(userFile)) return false;
    if(isValidPath(orderFile)) return false;
    if(isValidPath(menuFile)) return false;
    if(isValidPath(idFile)) return false;
    return true;
}
```

It also contains a method for computing the price of orders

```
private int computePriceOfOrder(ArrayList<MenuItem> items){
    int sum = items.stream().reduce( identity: 0, (partialAgeResult, user) -> partialAgeResult + user.getPrice(),
        Integer::sum);
    return sum;
}
```

It contains a method for parsing an order in a writable format for creating the reports

```
private String[] orderToString(Order order, ArrayList<MenuItem> items){
    String[] orderDetails = new String[5];
    orderDetails[0] = String.valueOf(order.getOrderID());
    orderDetails[1] =String.valueOf(order.getClientId());

    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy hh:mm");
    String orderDate = order.getOrderDate().format(formatter);
    orderDetails[2] = orderDate;

    StringBuilder sb = new StringBuilder();
    items.forEach(menuItem -> sb.append(menuItem.getName()).append(", "));
    orderDetails[3] = StringUtils.chop(sb.toString());

    orderDetails[4] = String.valueOf(computePriceOfOrder(items));
    return orderDetails;
}
```

Updating a product

```
public void updateProduct(String productToModify, MenuItem menuItem) {
    assert isWellFormed();
    assert !productToModify.isBlank();
    assert menuItem != null;
    assert menu.stream().anyMatch(m -> m.getName().equals(productToModify));
    menu.stream().forEach(menuItem1 -> {
        if (menuItem1.getName().equals(productToModify)){
            menuItem1.setName(menuItem.getName());
            menuItem1.setPrice(menuItem.getPrice());
            menuItem1.setRating(menuItem.getRating());
            menuItem1.setCalories(menuItem.getCalories());
            menuItem1.setFat(menuItem.getFat());
            menuItem1.setSodium(menuItem.getSodium());
        }
    });
    menu.stream().forEach(menuItem1 -> {
        if(menuItem1 instanceof CompositeProduct){
            if(((CompositeProduct) menuItem1).getItems().stream().anyMatch(menuItem2 -> menuItem2.equals(menuItem))){
                menuItem1.computeCalories();
                menuItem1.computeFats();
                menuItem1.computePrice();
                menuItem1.computeProteins();
                menuItem1.computeRating();
                menuItem1.computeSodium();
            }
        }
    });
    assert (menu.stream().anyMatch(m -> m.equals(menuItem)));
    serializer.serialize(menu, menuFile);
}
```

An example of how a report is created using streams

```
public void timeIntervalReport(int minH, int maxH) {
    assert minH >= 0 && maxH <= 24;
    Set<Order> result = orders.keySet().stream().filter(order -> order.getOrderDate().getHour() >= minH &&
        order.getOrderDate().getHour() <= maxH).collect(Collectors.toSet());
    List<String[]> strings = result.stream().map(order -> orderToString(order, orders.get(order))).
        collect(Collectors.toList());
    fileWriter.timeIntervalReport(strings);
}
```

How the functionality of creating a new order is implemented

```
public void addOrder(Order order, ArrayList<MenuItem> items) {
    assert isWellFormed();
    assert order.getOrderDate() != null && order.getClientId() > 0 && order.getOrderId() > 0;
    assert !items.isEmpty();
    orders.put(order, items);
    String[] orderDetails = orderToString(order, items);
    fileWriter.generateBill(orderDetails);
    setChanged();
    notifyObservers(orderDetails);
    assert wasAddedHash(order);
    serializer.serializeOrders(orders, orderFile);
    serializerI.serialize(ids, idFile);
}
```

Adding a composite product to the app data

```
public void addCompositeProduct(String name, List<MenuItem> components){
    assert isWellFormed();
    assert !components.isEmpty();
    assert !name.isEmpty();
    assert menu.stream().noneMatch(m -> m.getName().equals(name));

    CompositeProduct newCompositeProduct = new CompositeProduct(name, (double)0, calories: 0,
        proteins: 0, fat: 0, sodium: 0, price: 0);
    newCompositeProduct.setItems(components);
    newCompositeProduct.computeCalories();
    newCompositeProduct.computeFats();
    newCompositeProduct.computePrice();
    newCompositeProduct.computeProteins();
    newCompositeProduct.computeRating();
    newCompositeProduct.computeSodium();
    menu.add(newCompositeProduct);

    assert newCompositeProduct.getCalories() != 0 && newCompositeProduct.getFat() != 0 &&
        newCompositeProduct.getPrice() != 0 && newCompositeProduct.getRating() != 0 &&
        newCompositeProduct.getProteins() != 0 && newCompositeProduct.getSodium() != 0 ;
    assert !newCompositeProduct.getItems().isEmpty();
    assert wasAdded(newCompositeProduct);
    serializer.serialize(menu, menuFile);
}
```

Importing the data at each new run of the app

```
public void importData(){
    assert isWellFormed();
    menu = serializer.deserialize(menuFile);
    users = serializerU.deserialize(userFile);
    ids = serializerI.deserialize(idFile);
    orders = serializer.deserializeOrders(orderFile);
    assert !menu.isEmpty() && !users.isEmpty() && !ids.isEmpty() && !orders.isEmpty();
}
```

Searching for objects meeting certain criteria, it is done using streams. each field that does not have to be taken into account is either -1 if it is a number or "###" if it is the name of the item.

```
public List<MenuItem> search(String name, Double rating, Integer calories, Integer proteins,
                             Integer fat, Integer sodium, Integer price){

    assert isWellFormed();
    assert name != null && rating != null && calories != null && proteins != null && fat != null && sodium != null
           && price != null;

    List<MenuItem> result = new ArrayList<>();
    result = menu.stream().filter(!name.equals("###") ? m-> m.getName().contains(name) : m -> true ).
        filter(!price.equals(-1)? m -> m.getPrice().equals(price) : m -> true).
        filter(!calories.equals(-1)? m -> m.getCalories().equals(calories) : m -> true).
        filter(!proteins.equals(-1)? m -> m.getProteins().equals(proteins) : m -> true).
        filter(!fat.equals(-1)? m -> m.getFat().equals(fat) : m -> true).
        filter(!sodium.equals(-1)? m -> m.getSodium().equals(sodium) : m -> true).
        filter(!rating.equals(-1.0)? m -> m.getRating().equals(rating) : m -> true).collect(Collectors.toList());

    return result;
}
```

Login Service Class

The class implements methods for logging in and creating a new account.

Find account looks for the entered credentials in the file of data.

```
public int findAccount(String user, String password, String type){
    Serializer<User> userSerializer = new Serializer<>();
    List<User> users = new ArrayList<>();
    String userFile = "./users2.ser";

    users = userSerializer.deserialize(userFile);

    if(users.stream().noneMatch(user2 -> user2.getUsername().equals(user) && user2.getPassword().equals(password)
        && user2.getType().equals(type))){
        throw new RuntimeException("No "+type+" account with these credentials");
    }

    return users.stream().filter(user1 -> user1.getUsername().equals(user)).
        collect(Collectors.toList()).get(0).getUserId();
}
```


FileReader Class

The class contains a method for reading from a .csv file using streams. It is used to load the initial set of products. The products are stored in a hashset to ensure unicity.

```
inputList = br.lines().skip(1).map((line) -> {  
    String[] p = line.split(regex: ",");  
    MenuItem item = new BaseProduct(p[0], Double.parseDouble(p[1]), Integer.parseInt(p[2]),  
        Integer.parseInt(p[3]), Integer.parseInt(p[4]), Integer.parseInt(p[5]), Integer.parseInt(p[6]));  
    return item;  
}).collect(Collectors.toCollection(HashSet::new));
```

FileWriter Class

The class contains methods used for writing the reports and the bill.

How a new file is generated

```
public String createFile(String orderId, String clientId){  
    StringBuilder fileName = new StringBuilder();  
    fileName.append("./bill");  
    fileName.append(orderId + "_");  
    fileName.append(clientId);  
    fileName.append(".txt");  
    File file = new File(fileName.toString());  
    boolean result;  
    try  
    {  
        result = file.createNewFile();  
        if(result)  
        {  
            System.out.println("file created "+file.getCanonicalPath());  
        }  
        else  
        {  
            System.out.println("File already exist at location: "+file.getCanonicalPath());  
        }  
    }  
    catch (IOException e)  
    {  
        e.printStackTrace();  
    }  
    return fileName.toString();  
}
```

How the bill is written

```
public void generateBill(String[] strings){
    StringBuilder sb = new StringBuilder();
    sb.append("ORDER : ").append(strings[0]).append("\n");
    sb.append("CLIENT : ").append(strings[1]).append("\n");
    sb.append("DATE&TIME : ").append(strings[2]).append("\n");
    String products = strings[3].replaceAll(regex: ",", replacement: "\\n");
    sb.append(products).append("\n");
    sb.append("PRICE : ").append(strings[4]);
    System.out.println(sb.toString());
    try {
        FileWriter fw = new FileWriter(createFile(strings[0], strings[1]));
        fw.write(sb.toString());
        fw.close();
    } catch (IOException exception) {
        exception.printStackTrace();
    }
}
```

How a report is generated

```
public void clientsReport(List<User> result){
    StringBuilder sb = new StringBuilder();
    sb.append(String.format("%20s %20s \r\n", "id", "username"));
    result.forEach(user -> {
        sb.append(String.format("%20s %20s \r\n",
            user.getUserId(), user.getUsername()));
        sb.append("\n");
    });
    System.out.println(sb.toString());
    try {
        FileWriter fw = new FileWriter(fileName: "clients_report.txt");
        fw.write(sb.toString());
        fw.close();
    } catch (IOException exception) {
        exception.printStackTrace();
    }
}
```

Serializer Class

Class contains methods for serializing and deserializing lists of objects and maps.

Serializing a list

```
public void serialize(List<T> entries, String file){
    try {
        FileOutputStream fileOut =
            new FileOutputStream(file);
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(entries);
        out.close();
        fileOut.close();
        System.out.printf("Serialized data is saved\n");
    } catch (IOException i) {
        i.printStackTrace();
    }
}
```

Deserializing a list

```
public List<T> deserialize(String file){
    List<T> entries = new ArrayList<>();
    try {
        FileInputStream fileIn = new FileInputStream(file);
        ObjectInputStream in = new ObjectInputStream(fileIn);
        entries = (List<T>) in.readObject();
        in.close();
        fileIn.close();
    } catch (IOException i) {
        i.printStackTrace();
        return null;
    } catch (ClassNotFoundException c) {
        System.out.println("class not found");
        c.printStackTrace();
        return null;
    }
    return entries;
}
```

InputValidator Class

The class contains methods for validating the inputs in the application.

This is how a new added product is validated

```
public MenuItem validateMenuItem(String name, String rating, String calories, String proteins, String fat, String sodium,
                                String price) {
    if (name.isBlank() || rating.isBlank() || calories.isBlank() || proteins.isBlank() || fat.isBlank() ||
        sodium.isBlank() || price.isBlank()) {
        throw new RuntimeException("One of the fields is empty");
    }

    MenuItem menuItem = new BaseProduct(name, validateDouble(rating), validateInt(calories), validateInt(proteins),
        validateInt(fat), validateInt(sodium), validateInt(price));

    return menuItem;
}
```

Checking if an input is an Integer

```
public int validateInt(String s) {
    int number;
    try {
        number = Integer.parseInt(s);
    } catch (NumberFormatException e) {
        throw new RuntimeException("field must be an integer");
    }
    return number;
}
```

Graphical User Interface Implementation . The View Package

The graphical user interface was implemented using Java Swing. The interface was designed according to the previously designed mockups.

The whole user interface is composed of 11 classes.

The Controllers

The controllers of the application implement the action listeners for all the buttons/tables/radio buttons of the user interface. They help to create a clear connection between the view displayed to the user and the business logic of the application.

Login Controller

example of action listener for a radio button (logging in as an employee)

```

    if(loginView.getRb2().isSelected()){
        try{
            loginService.findAccount(loginView.getUsername(),String.valueOf(loginView.getPasswordText()), type: "employee");
            deliveryService.addObserver(employeeView);
            employeeView = new EmployeeView();
        }catch (RuntimeException exception){
            loginView.displayErrorMessage(exception);
        }
    }
}

```

CreateAccount Controller

example of action listener for a radio button (registering as an employee)

```

    if(createView.getRb2().isSelected()){
        try{
            loginService.createAccount(createView.getUsername(),String.valueOf(createView.getPasswordText()),
                type: "employee", createView.getSecret());
            createView.displayInformationMessage("Account created");
        }catch (RuntimeException exception){
            createView.displayErrorMessage(exception);
        }
    }
}

```

Administrator Controller

The action listener for creating a menu(composed product) by iterating the selected rows.

```

administratorView.addMenuActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        List<MenuItem> itemList = new ArrayList<>();
        administratorView.setRows(administratorView.getProducts().getSelectedRows());
        int[] rows = administratorView.getRows();
        for(int i : rows){
            String[] row = administratorView.getRowAt(i,administratorView.getProducts());
            MenuItem menuItem = deliveryService.getMenu().stream().filter(m -> m.getName().equals(row[0])).
                collect(Collectors.toList()).get(0);
            itemList.add(menuItem);
        }

        deliveryService.addCompositeProduct(administratorView.getMenuName(),itemList);
        administratorView.setMenuItems(deliveryService.getMenu());
        administratorView.initializeTable();
    }
});

```

The action performed when updating an item

```
administratorView.updateMenuActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {

        administratorView.setRows(administratorView.getProducts().getSelectedRows());

        String[] row = administratorView.getRowAt(administratorView.getRows()[0], administratorView.getProducts());
        if(row[7].equals(" ")){
            MenuItem menuItem = new BaseProduct(row[0], Double.valueOf(row[1]), Integer.parseInt(row[2]),
                Integer.parseInt(row[3]), Integer.parseInt(row[4]), Integer.parseInt(row[5]),
                Integer.parseInt(row[6]));
            deliveryService.updateProduct(administratorView.getItemsNames().get(administratorView.getRows()[0]),
                menuItem);
            administratorView.setMenuItems(deliveryService.getMenu());
            administratorView.initializeTable();
        }
    }
});
```

The action performed when deleting an item

```
administratorView.deleteActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        List<MenuItem> itemList = new ArrayList<>();
        administratorView.setRows(administratorView.getProducts().getSelectedRows());
        int[] rows = administratorView.getRows();
        for(int i : rows){
            String[] row = administratorView.getRowAt(i, administratorView.getProducts());
            MenuItem menuItem = deliveryService.getMenu().stream().filter(m -> m.getName().equals(row[0])).
                collect(Collectors.toList()).get(0);
            itemList.add(menuItem);
        }
        itemList.forEach(m -> deliveryService.deleteProduct(m));
        administratorView.setMenuItems(deliveryService.getMenu());
        administratorView.initializeTable();
    }
});
```

The action performed when creating a report

```
administratorView.ordersInTimeReportActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try{
            deliveryService.timeIntervalReport(inputValidator.validateInt(administratorView.getMin()),
                inputValidator.validateInt(administratorView.getMax()));
            System.out.println(administratorView.getMin() + " " + administratorView.getMax());
        }catch (RuntimeException ex){
            administratorView.displayErrorMessage(ex);
        }
    }
});
```

Client Controller

The action performed when adding a product to the order

```
clientView.addToOrderActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        clientView.setRows(clientView.getProducts().getSelectedRows());  
        int[] rows = clientView.getRows();  
        for(int i : rows){  
            String[] row = clientView.getRowAt(i,clientView.getProducts());  
            MenuItem menuItem = deliveryService.getMenu().stream().filter(m -> m.getName().equals(row[0])).  
                collect(Collectors.toList()).get(0);  
            itemsOrdered.add(menuItem);  
        }  
    }  
});
```

The action performed when searching for an item

```
clientView.viewSortedActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        try {  
            List<MenuItem> menuItems = deliveryService.search(inputValidator.validateSearchS(clientView.getName()),  
                inputValidator.validateSearchD(clientView.getRating()),  
                inputValidator.validateSearch(clientView.getCalories()),  
                inputValidator.validateSearch(clientView.getProteins()),  
                inputValidator.validateSearch(clientView.getFat()),  
                inputValidator.validateSearch(clientView.getSodium()),  
                inputValidator.validateSearch(clientView.getPrice())  
            );  
            menuItems.forEach(m -> System.out.println(m.getName()));  
            clientView.initializeTable(menuItems);  
        } catch (RuntimeException exception) {  
            clientView.displayErrorMessage(exception);  
        }  
    }  
});
```

The View

These classes use components from java.awt and javax.swing to implement the aspect of the interfaces. Components used are : JTextFields, JButtons, JLabel, JPanels, JRadioButtons, JScrollPane, JTable.

The AppFrame Class

This class is a class that extends JFrame and will later be extended by all the other interfaces. The class implements the following methods common to all the interfaces for displaying messages to the user.

```

public void displayErrorMessage(Exception exception) {
    if (exception != null) {
        String message = exception.getMessage();
        JOptionPane.showMessageDialog( parentComponent: this, message, title: "Error", JOptionPane.ERROR_MESSAGE);
    }
}

public void displayInformationMessage(String message) {
    if (!StringUtils.isBlank(message)) {
        JOptionPane.showMessageDialog( parentComponent: this, message, title: "Info", JOptionPane.INFORMATION_MESSAGE);
    }
}
}

```

The Employee View

This class needs to be mentioned separately because it is different from the others because it implements the Observer interface. It contains a method update specific to the Observer pattern in Java. The method is used to notify the employee that a new order was created by adding a new instance in the table of orders.

```

@Override
public void update(Observable o, Object orderDetails) {
    String[] string = (String[]) orderDetails;
    model.addRow(string);
    updateTable();
}

```


Testing and Results

The application was tested for various scenarios while it was implemented by introducing data in the user interface and observing the produced result, and also with the reports.

Report about orders in a time interval

id order	id client	date	products	price
9	2	23/05/2021 06:59	Grilled Chicken Moroccan Style ,Lobster with Roasted Garlic-Potato Salad and Coleslaw	93
15	2	23/05/2021 07:31	Zucchini Relish Wiley ,Candied-Fruit Pastry Cream	123
20	2	24/05/2021 07:18	Chicken with Coconut Curry Sauce ,a,Mussels with Parsley and Garlic	137
7	2	23/05/2021 06:27	Chicken with Coconut Curry Sauce	39
10	2	23/05/2021 06:59	Grilled Chicken Moroccan Style ,Lobster with Roasted Garlic-Potato Salad and Coleslaw	93
19	2	24/05/2021 07:13	menux,a	308
18	2	24/05/2021 06:48	Scallop and Shrimp Creole ,Grilled Pork Sausages with Spiced Figs	81
13	4	23/05/2021 07:01	Red Potato and Green Bean Salad with Dijon Vinaigrette	46
8	2	23/05/2021 06:27	Chicken with Coconut Curry Sauce	39
11	4	23/05/2021 07:01	Red Potato and Green Bean Salad with Dijon Vinaigrette	46
14	2	23/05/2021 07:28	Strawberry Meringue Roulade ,Poached Sockeye Salmon with Mustard Herb Sauce	125
12	4	23/05/2021 07:01	Red Potato and Green Bean Salad with Dijon Vinaigrette	46

Report about products ordered more than a specified number of time

name	calories	price
Chicken with Coconut Curry Sauce	246	39
Red Potato and Green Bean Salad with Dijon Vinaigrette	415	46

Report about the clients

id	username
2	client
4	client1

Report about the products ordered in a day

name	times
a	2
Chicken with Coconut Curry Sauce	1
Pan-Seared Steak with Mushroom-Merlot Sauce	1
Mussels with Parsley and Garlic	2
menux	1
Ecuadoran Tamarillo Salsa	1
Scallops with Tarragon Cream and Wilted Butter Lettuce	1
Grilled Pork Sausages with Spiced Figs	1
Scallop and Shrimp Creole	1
Michael Lewis's Cassoulet de Canard	1
Apple Sorbet	1
Butternut Squash and Sage Orzo	1

How a bill looks like

ORDER : 21
CLIENT : 2
DATE&TIME : 24/05/2021 08:04
Pan-Seared Steak with Mushroom-Merlot Sauce
Ecuadoran Tamarillo Salsa
PRICE : 96

Conclusions

The task of implementing a food delivery management system application was successfully completed, since it can be seen that the application meets all the functional and non-functional requirements.

In conclusion, the food delivery management system application is a useful tool to manage the orders, clients and products from a catering company by implementing create, update, delete and view operations.

The modeling, design and implementation of this software are based on programming techniques and Object-Oriented-Programming principles, so the creation of this app was a

great learning opportunity. I've learned about working with Observer Design Pattern, Composite Design Pattern and Design by contract Pattern . Also, using streams and serialization was something new for me. I think that I have also improved my skills in writing OOP code. Moreover, facing and fixing all sorts of errors and bugs were an important step in learning.

Further Developments

Some improvements that could make the software better would be:

- implementing a more friendly user interface containing pictures of the products.
- implementing the option to delete a product from order when creating the order.
- implementing the option for the administrator to see the orders

Bibliography

- Assignment 4 support presentation, PT2021_Assignment4.pdf
- <https://dzone.com/articles/how-to-read-a-big-csv-file-with-java-8-and-stream>
- <https://xenovation.com/blog/development/java/remove-last-character-from-string-java>
- <https://docs.oracle.com/javase/tutorial/uiswing/components/table.html>
- <https://beginnersbook.com/2014/08/convert-hashset-to-a-list-arraylist/>
- <https://stackoverflow.com/questions/30082555/collectors-toSet-and-hashSet>
- <https://stackoverflow.com/questions/203984/how-do-i-remove-repeated-elements-from-arraylist>
- <https://www.javatpoint.com/java-hashset>
- <https://stackoverflow.com/questions/468789/is-there-a-way-in-java-to-determine-if-a-path-is-valid-without-attempting-to-create>
- <https://www.baeldung.com/java-streams-find-list-items>
- <https://www.baeldung.com/java-observer-pattern>
- <https://xenovation.com/blog/development/java/remove-last-character-from-string-java>
- <https://stackoverflow.com/questions/505928/how-to-count-the-number-of-occurrences-of-an-element-in-a-list/2459753>