UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

PROGRAMMING TECHNIQUES

FOOD DELIVERY MANAGEMENT SYSTEM

STUDENT: ELEKES PÉTER
UNIVERSITY: UNIVERSITY OF TECHNICAL ENGINEERING CLUJ-NAPOCA
FACULTY: COMPUTER SCIENCE
YEAR: II.
GROUP: 30422

Table of Contents

Assignment Task 3
Projection 6
Implementation 7
Business Logic 7
Data Access 8
Presentation (GUI) 8
Start 9
JavaDocs 9
Possible further development 9
Conclusion 10
Bibliography 10

Assignment Task

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

The administrator can:

- 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:
    - o 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.
    - o the products ordered more than a specified number of times so far.
    - o 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.
    - o the products ordered within a specified day with the number of times they have been ordered.

The client can:

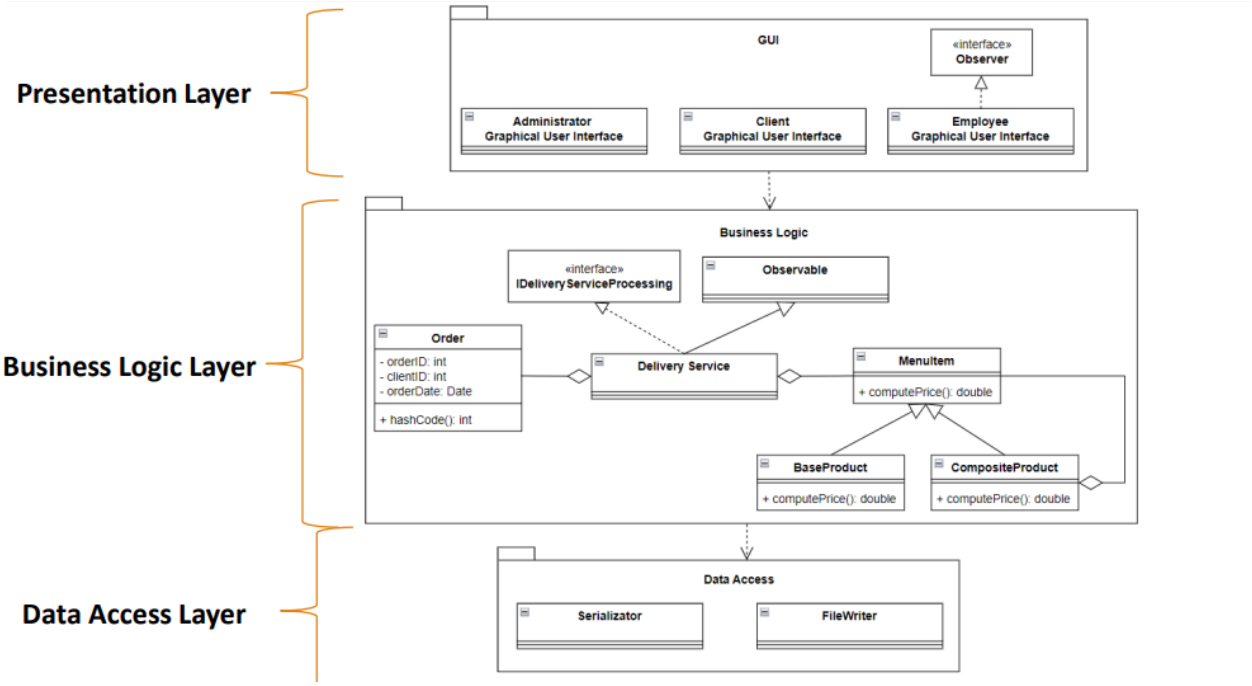
- Register and use the registered username and password to log in within the system.
- View the list of products from the menu.
- Search for products based on one or multiple criteria such as keyword (e.g., "soup"), rating, number of calories/proteins/fats/sodium/price.
- Create an order consisting of 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.

Requirements	Points
• Object-oriented programming design, classes with maximum 300 lines, methods with maximum 30 lines, Java naming conventions • Implement the class diagram from Section 1. Choose appropriate data structures for saving the Orders and the MenuItems. • Define the class BaseProduct with the following fields: title, rating, calories, proteins, fats, sodium, price. Read the data from the file products.csv using streams and split each line in 7 parts: title, rating, calories, protein, fat, sodium, price, and create a list of objects of type BaseProduct. • Graphical interface: o Log in window o Window for Administrator operations o Window for Client operations • Use lambda expressions and stream processing for generating the administrator specific reports • Use lambda expressions and stream processing to implement the search functionalities available to the client • Good quality documentation covering the sections from the documentation template	minimal requirement
Use the Composite Design Pattern for modelling the classes MenuItem, BaseProduct, CompositeProduct.	1 point
Create bill in .TXT format.	0.5 point
Design by contract: preconditions and postconditions in the IDeliveryServiceProcessing interface. Implement them in the DeliveryService class using the assert instruction. Define an invariant for the class DeliveryService. Generate the corresponding JavaDoc files which should include the custom tags and descriptions associated to the defined pre, post conditions and invariants.	1.5 points

Window for the employee user: use Observer Design Pattern to notify each time a new Order is added.	1 point
Save the information from the DeliveryService class in a file (i.e., file.txt) using serialization. Load the information when the application starts.	1 point

Assignment analysis, scenario, approach, and use cases



I used a layered architecture to ease the workflow and to have a logical overview of the problem. In the Business Logic Layer I used the Composite Design Pattern to create the MenuItem, BaseProduct and CompositeProduct, and I used the Observer Design Pattern to notify the employee each time an order is created. The Presentation Layer is used to create the user interface, it contains and describes the way a user can interact with the application. The Data Access Layer contains the necessary classes to create bills/reports or to serialize the files.

Use Case

The application can be used as an admin, an employee or a customer.

An admin can view, edit, delete, add, import products, create composite products and generate reports.

A customer can view products, search for them following various criteria, and place an order.

An employee gets notified each time a new order is placed, and can view the new order, along with the existing ones

A general scenario:

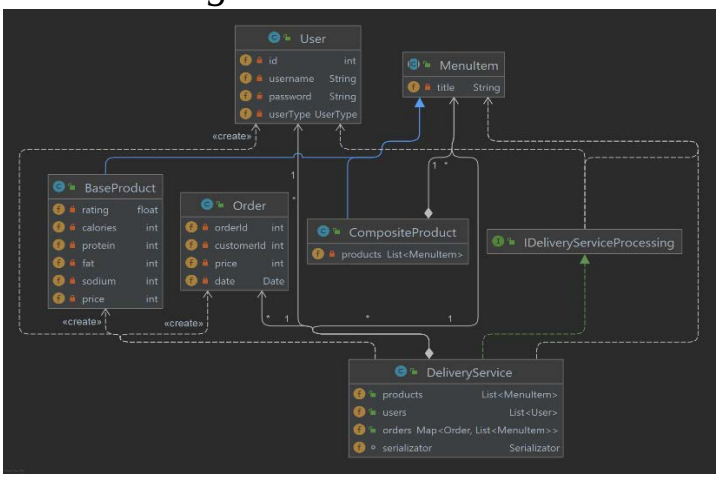
1) Open the log in window
   a) If the user is an admin, employee or a customer with an existing account they can log in
   b) If the user is a customer without an account, they have to register
2) User specific window pops up
   a) If the user is an admin, they can view the products in the table
      i) Press the add button, fill in the textfields and add a new product
      ii) Select a product from the table, press edit and edit the properties
      iii) Select a product from the table, press delete and delete the product
      iv) Select multiple products from the table, press Composite Product and provide a name for the composite product
      v) Press generate report, select the wanted report and fill in the data
      vi) Import products from csv, which overwrites all the existing products
   b) If the user is a customer, they can view the products in the table
      i) Search for products by various criteria
      ii) Select products from the table and add them to the ordering cart
      iii) Order the cart
   c) If the user is an employee they can view all the orders

Projection

I approached the problem using a layered architecture pattern and used classes that were grouped into the following packages: BusinessLogic, DataAccess, GUI. This solution is an efficient approach on creating an application that is logically constructed and easy to improve. Some sub-components of the program were designed with other patterns in mind. The products trio were created with the composite design pattern, and the employee class implemented the observer interface in order to be notified whenever a new order was placed.

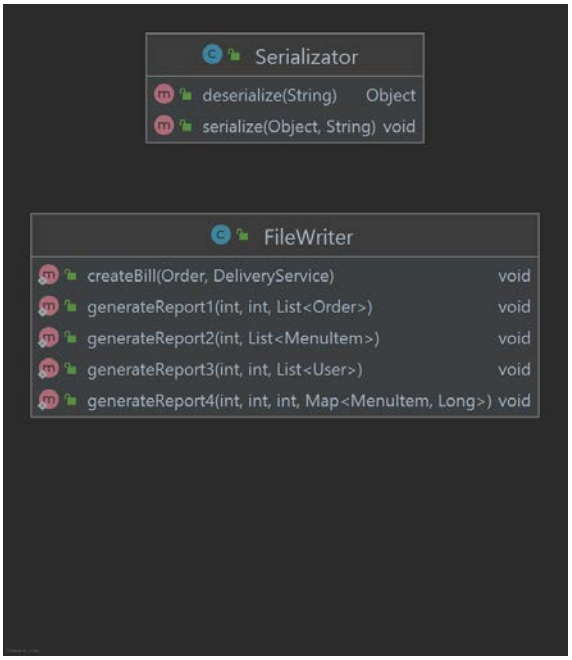
Implementation

Business Logic



The classes implemented in this package are: User, MenuItem, BaseProduct, Order, CompositeProduct, DeliveryService. There is also an interface, IDeliveryServiceProcessing, which describes all the methods implemented in the DeliveryService class, along with their pre- and postconditions. The MenuItem, BaseProduct and CompositeProduct trio were created with the Composite Design Pattern in mind, which eases the way they can be handled. The main functionality of the program depends on the DeliveryService class. Some crucial methods implemented are: importFromFile, which reads the products.csv and imports all of the products without duplicates; register, which registers a new user; login, which logs in a user; the four generateReport methods and the search methods; the CRUD methods used by the admin. Every method that implements the ones described in the interface also assert the pre- and postconditions. In order to generate the reports for the admin and provide the results for the user I used lambda expressions, which seemed quite hard to understand. In order to preserve the data, the list of products, users and orders are serialized with the help of the serializator implemented in the DataAccess package. This way, all of the data is stored in a .ser file, and every time a new entry is added, this file is updated.

Data Access

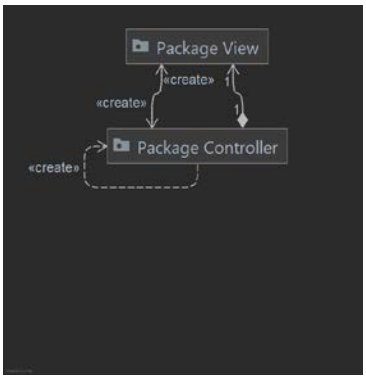


This package implements two classes, namely Serializator and FileWriter, both of which communicate with the file system. The serializator class has two methods, namely serialize and deserialize. When we serialize an object we take its data and output it into a file, and when we deserialize a file, we take the file's data and put it into an object. It's a straightforward mechanism, yet immensely useful to create an application like this.

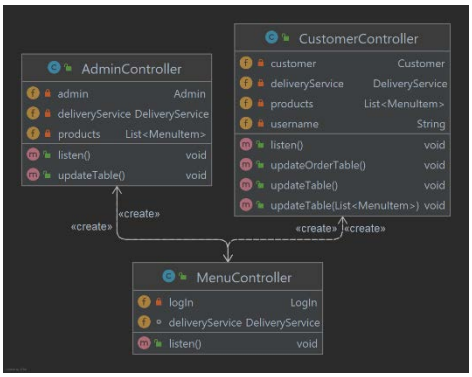
TheFileWriter class implements the methods used to create the bill or create various reports. the createBill method gets the order and the instance of the service, and outputs a bill where the order id, client id, price, date, and all the products are specified. The file is saved in the format "bill_unixtime.txt".

All the generateReport methods work following the same philosophy, having all of the required information displayed and saved in a "reportnumber_unixtime.txt" format.

Presentation (GUI)



One of the most necessary packages, the Presentation package includes all the classes which implement the GUI. I have two sub-packages, the View and the Controller. The View package includes all the classes which describe how a window should look like, and in the Controller package the functionality of the windows are implemented. In the Controller package I have an instance of the MenuController, where the user can log in or register. If the user chooses to register a new account, this very same class creates the required window to do so. After the log in, this package either creates a new AdminController, CustomerController, or a new Employee view. All of the subsequent windows that will be opened in these views are taken care of by these classes. The Employee class is an interesting one, because it implements the Observer interface, which is used to notify the employee each time a new order is placed.



Start

This package is quite unremarkable, it simply has a Start class calling a new menuController, which gets the user to the log in window.

JavaDocs

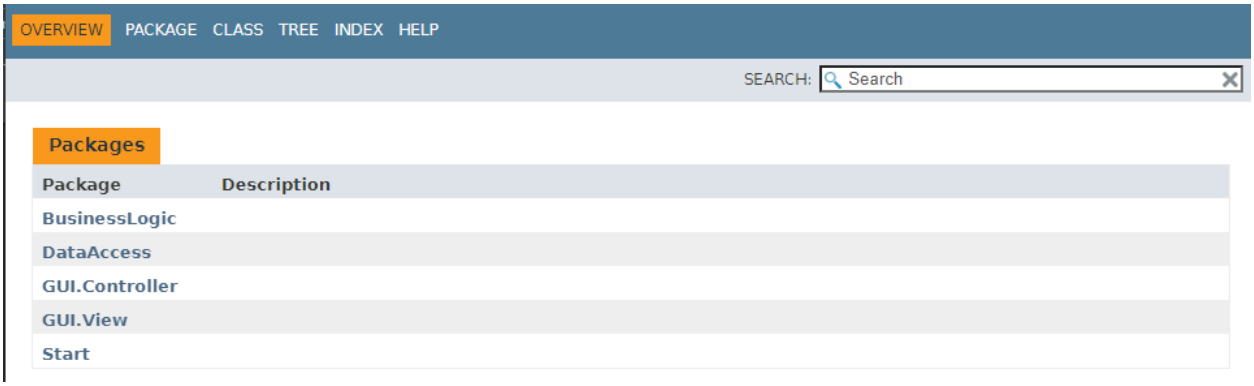
```
/**
 * @pre products.contains(product)
 * @post !products.contains(product)
 * */
```

Another requirement was that we should create JavaDoc files. This is a documentation generator for generating API documentation in HTML format straight from the Java source code. HTML is used for easy navigation between the documents, using hyperlink.

I created my project in IntelliJ IDEA, which can automatically generate a JavaDoc HTML.

A JavaDoc comment differs from a simple comment, having the opening tag /** and ending it with */. The first paragraph should be a general description of whereas the following paragraphs can include descriptive tags such as @param, @return, @throws, or @see.

After generating the JavaDoc in IntelliJ a new folder is created, containing many HTML and script files, and if we open the index.html file, the general overview of the program is presented.



Possible further development

I think a new approach to the GUI would improve on the program. This version is intuitive but does not look great. Unfortunately, I didn't have time to style the GUI, because I'd have quickly run out of time doing so. I also am not pleased with the way much of my methods turned out, they are quite hacky solutions.

Conclusion

At first this final project seemed to be the hardest so far, which it was, but after I sat down, and sketched the layout of the program it became approachable. I didn't use serialization before, but after searching on the internet and looking in the slides I realized that it's a really great solution, and an easy-to-implement one.

Javadoc files still amaze me, and I still enjoy how easy it is to create a state-of-the-art documentation with the help of JavaDocs. A good program already includes well commented code and rendering it in a good-looking way is always a plus.

I could also practice the layered architecture style once again, that proved to be useful too. It is a straightforward approach to creating a program that can be easily enhanced upon in the future.

The two new design solutions, namely the Composite Design Pattern and the Observer Design Pattern were a great addition to the program, and I think I will use these solutions from now on. I was really pleased when I saw how easy it was to create a composite product, and how easy it was to notify the employee each time a new order was placed.

Once again, I learned a great deal during the realization of this project, and I am glad we are challenged by programs like this.

Bibliography

Course slides

Assignment documentation

Support presentation

https://dsrl.eu/courses/pt/

https://www.geeksforgeeks.org/

https://www.youtube.com/

https://stackoverflow.com/