**RESTAURANT MANAGEMENT SYSTEM**

1. Objectives
   1. Main Objective

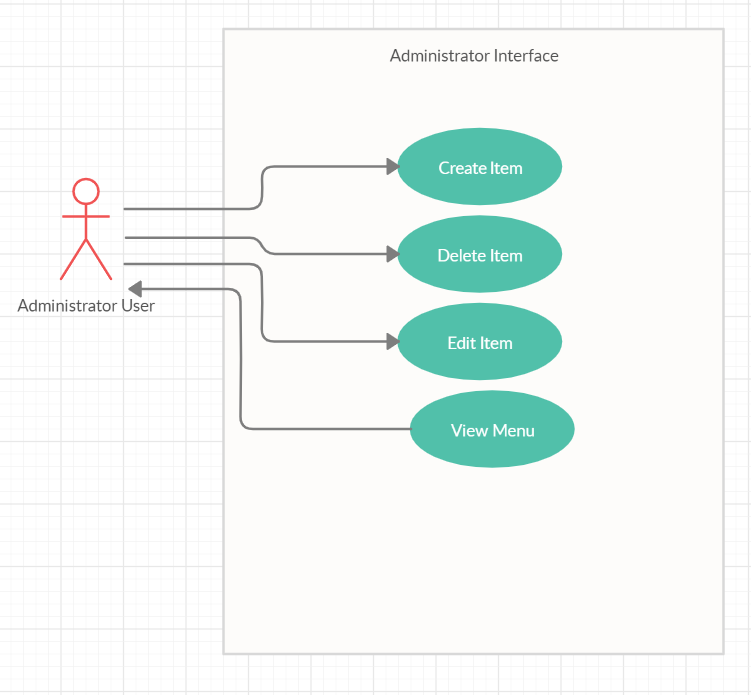
Consider implementing a restaurant management system. The system should have three types of users: administrator, waiter and chef. The administrator can add, delete and modify existing products from the menu. The waiter can create a new order for a table, add elements from the menu, and compute the bill for an order. The chef is notified each time it must cook food that is ordered through a waiter.

To simplify the application, it is assumed that the system is used by only one administrator, one waiter and one chef, and there is no need of a login process.

* 1. Secondary Objectives
* Implement the classes that define the products using the Composite Design Pattern. (Chapter 3)
* Define the Order class. (Chapter 3, 4)
* Implement the interface IRestaurantProcessing which defines the main functionalities of the Restaurant class. (Chapter 3, 4)
* Implement the FileWrite which provides the functionality for generating bills. (Chapter 3)
* Implement the RestaurantSerializator class. (Chapter 3)
* Implement the view classes for each of the three interfaces. (Chapter 3, 4)
* Implement the controller classes for each of the three interfaces. (Chapter 3, 4)
* Implement the Observer Design Pattern on the restaurant and ChefController classes. (Chapter 3)

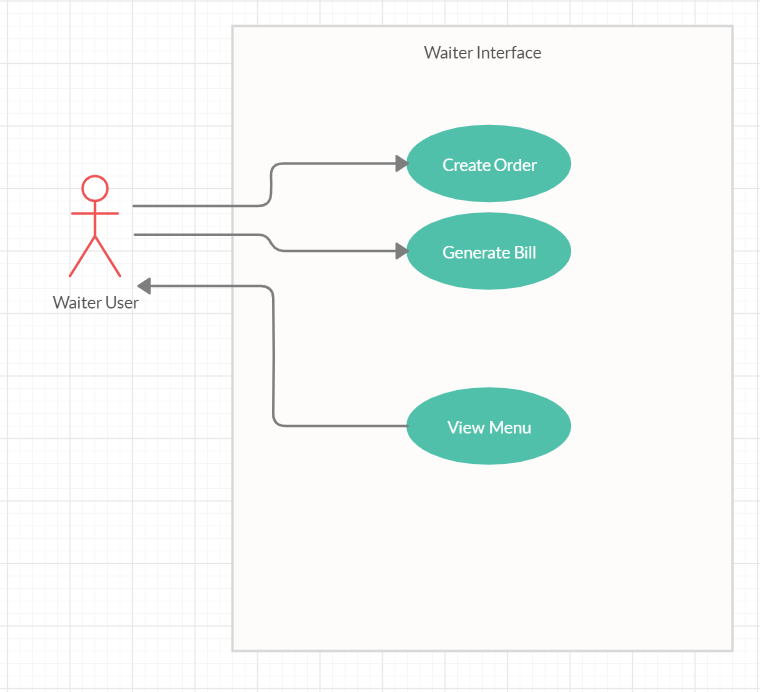
1. Task analysis, Modeling, Scenarios, Uses Cases

The application should receive as argument the path to a .ser file from which the state of the restaurant will be read or no argument case in which a new restaurant object will be created and the application start in an initial state.

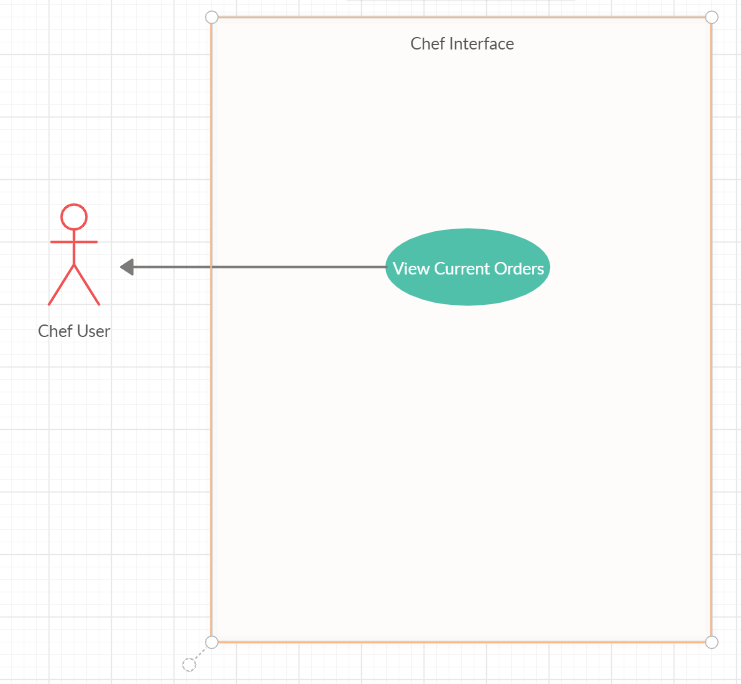


* Use Case: Create Item
* Primary Actor: Administrator User
* Main Success Scenario:
  + The user selects the create item option.
  + The user selects the type of the product (Base of Composite).
  + The user defines the product.
  + The user presses the button to end the operation.
* Alternative Sequences:
  + The user does not provide a correct definition of the product.
    - The item will not be added to the menu.

* Use Case: Delete Item
* Primary Actor: Administrator User
* Main Success Scenario:
  + The user selects the delete item option.
  + The user selects the product from the checkbox list.
  + The user presses the button to end the operation.
* Use Case: Edit Item
* Primary Actor: Administrator User
* Main Success Scenario:
  + The user selects the edit item option.
  + The user selects the product from the checkbox list.
  + The user inserts the updated data.
  + The user presses the button to end the operation.
* Alternative Sequences:
  + The user does not provide a correct definition of the product.
    - The item will not be edited.
* Use Case: View Menu
* Primary Actor: Administrator User
* Main Success Scenario:
  + The user selects the view menu option.



* Use Case: Create Order
* Primary Actor: Waiter User
* Main Success Scenario:
  + The user selects the create order option.
  + The user selects the table.
  + The user selects the products.
  + The user presses the button to end the operation.
* Use Case: Generate Bill
* Primary Actor: Waiter User
* Main Success Scenario:
  + The user selects the generate bill option.
  + The user selects the table.
  + The user presses the button to end the operation.
* Use Case: View Menu
* Primary Actor: Waiter User
* Main Success Scenario:
  + The user selects the view menu option.



* Use Case: View Orders
* Primary Actor: Chef User
* Main Success Scenario:
  + The current processed orders are displayed.

1. Design

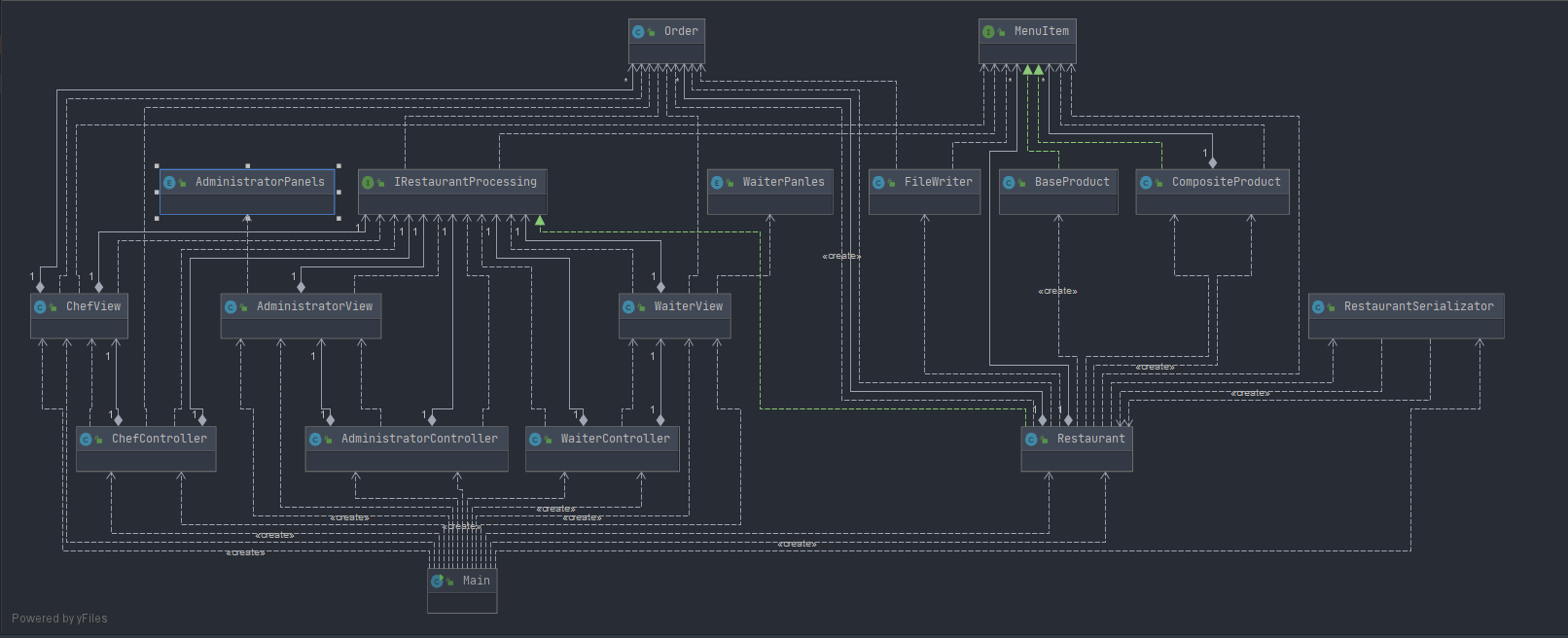
We start off the implementation by creating the classes that define the business layer, the rest of the project will be built around them. The first group of classes from the package are the ones that define the products. They are built using the Composite Product Design Pattern. We define the interface MenuItem based on which the other two classes BaseProduct and CompositeProduct are created. This kind of approach offers the possibility to create CompositeProduct objects that are composed of BaseProduct objects or even other CompositeProduct objects which is a needed feature in this case. Next, the Order class is implemented to define the characteristics of an order (eg table number, order id). It will also play an important role in the restaurant class being the key for identifying the products that are associated to it. Last thing to be done is implementing the IRestaurantProcessing interface having the abstract methods needed to provide the functionality for its users and the class Restaurant which implements this interface.

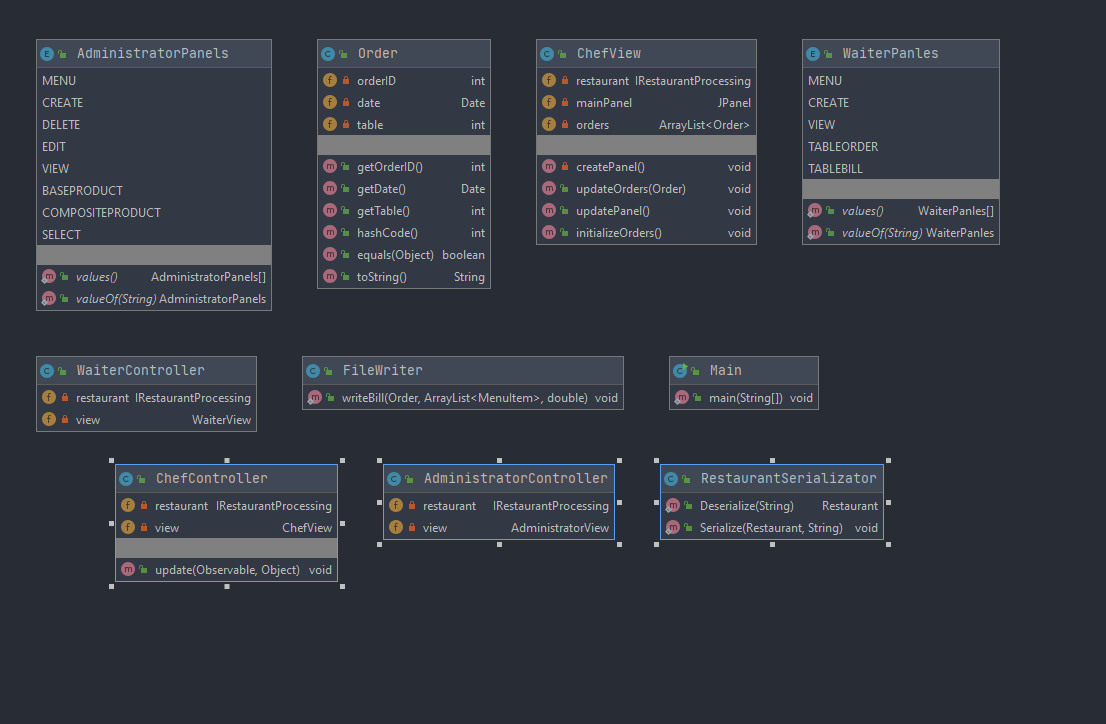
The next step is to implement the classes that reside in the data layer. There are two of them and they have well defined objectives. The FileWriter class is meant for creating a .txt file that will contain bill information regarding a specific order. The other class is RestaurantSerializator which will implement two static methods serialize and deserialize. These two methods will be used for saving the state of the restaurant object as well as retrieving the last state when the application is lunched.

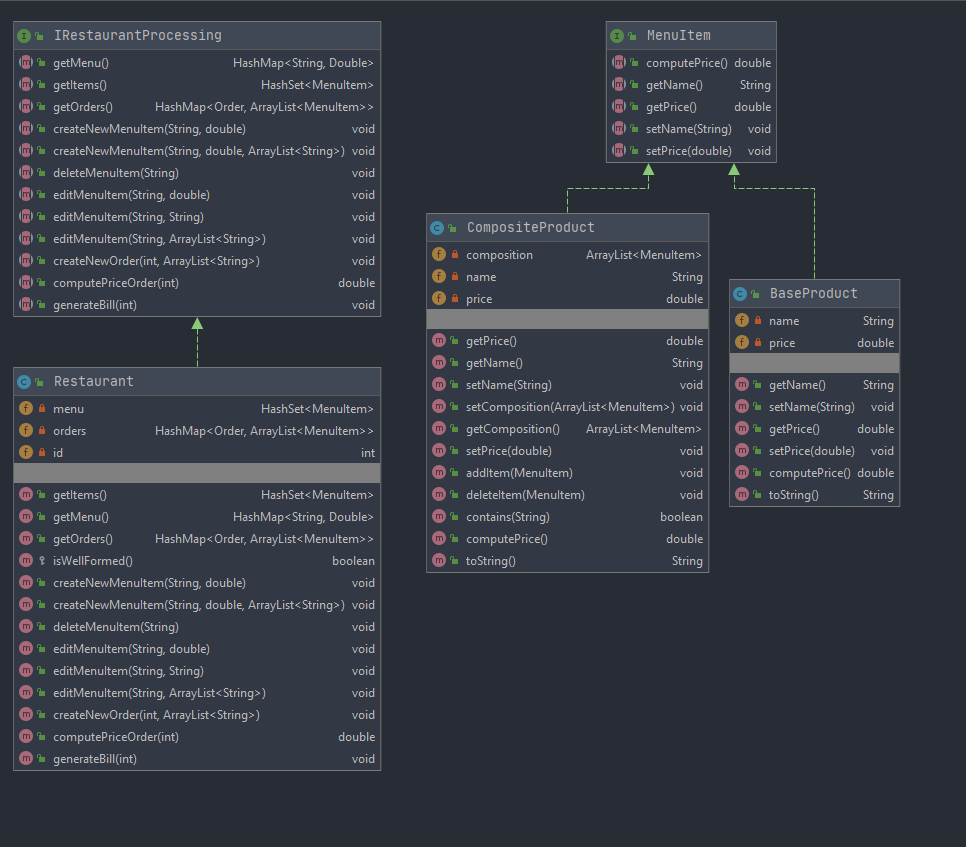
Moving on, we now must take care of the graphical user interface of the application. We identify three users: Administrator, Waiter and Chef and this means that each one of them will have its specific interface providing the functionalities associated to its role. The implementation will be done accordingly to the Model View Controller design. The model is common to all of them being the restaurant. Considering all these aspects, it is clear the need of implementing for each interface a view class and a controller one. The Administrator interface will provide the user features like adding a new item to the menu, deleting or editing an existing one and also the possibility of displaying the menu. The Waiter interface will provide the user features like creating a new order for a table as well as generating a bill for an existing order. The Chef interface will only display the current processing orders.

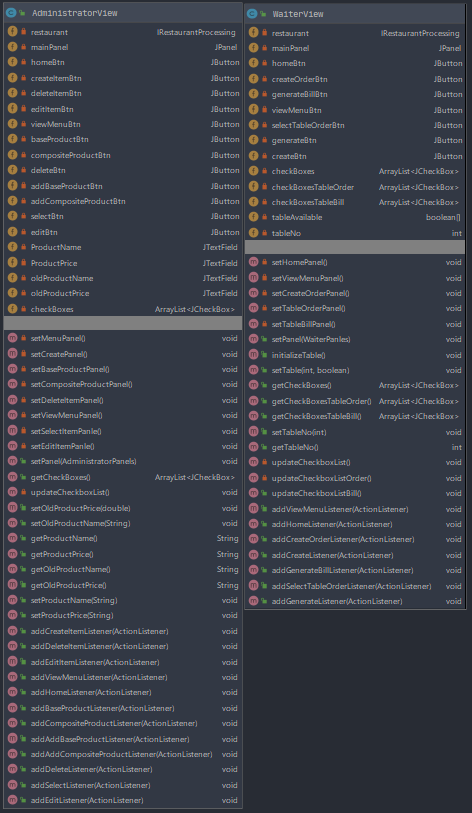
In order to automatized the process of coordinating the data, the Observer Design Pattern is used having as observer the Chef controller class and the observable being the restaurant. In this way, every time a new order is created by the Waiter controller class, the Chef controller class will be notified and will update its view accordingly.

The whole design of the application will be better understood from the below diagrams that contain all the classes along with their dependencies.









1. Implementation

* MenuItem Interface

Creates the abstract view of a menu item which can take many concrete forms but all of them must obey the given functionality that is to be able to evaluate itself to a price. There are other four abstract methods easy access to the main field of the objects.

* BaseProduct Class

This class defines the products that are at the bottom of the hierarchy. The class implements the MenuItem interface having the computePrice method simply returning its price.

* CompositeProduct Class

This class defines the products that constiue the higher levels of the hierarchy. We are talking about products that contain base products and my also contain other composite products as their composition. The class implements the MenuItem interface having the computePrice method returning the sum of al the prices of the products from its composition plus the price of the product itself.

* Order Class

This class simply defines an order having as characteristics an id (orderID), a table (table) and a date (date). It plays an important role in identifying the menu items associated with it stored in a HashMap structure. In order to provide this, the class overrides the hashCode method as well as the euals method (the equality is given by their orderID fields).

* IRestaurantProcessing Interface

Creates the abstract view of a restaurant defining the abstract methods which provide only the needed functionality for the three different classes that manage a restaurant. These are the methods for creating, editing and deleting a menu item and creating an order, computing the price of an order and generating a bill for an order. Through this interface, the user’s classes are isolated from the implementation particularities of the restaurant class.

* Restaurant Class

This is the concrete implementation of the restaurant. It keeps its menu in an HashSet structure composed of MenuItem objects. The orders and the MenuItems associated with them are stored in a HashMap where the keys are Order objects. The class implements the IRestaurantProcessing interface providing an implementation for all the abstract methods defined in the interface. Also, the class uses the Design By Contract method for all the implemented methods. It defines a method isWellFormed that checks the state of the invariants of the class, those being the two private fields mentioned in the beginning. The check implies that all the composite products ar composed of products tha are also in the menu and that no two orders from the keyset of the HashMap have the same table assigned to them.

* FileWriter Class

This class has the objective of writing a bill formatted .txt file containing information related to an order. The class has only one static method which receives as parameters an Order object, a list of MenuItem objects and the total price of the order.

* RestaurantSerializator Class

This class implements two static methods: serialize and deserialize. The serialize method is used for saving the state of the restaurant object in a file called “restaurant.ser” and the deserialize method is used for retreiveing the state of the restaurant object from a .ser file.

* AdministatorController Class

This class works together with the AdministratorView class providing all the action listeners for the buttons. It provides implementation for all the actions to be performed when certain buttons are clicked.

* WaiterController Class

This class works together with the WaiterView class providing all the action listeners for the buttons. It provides implementation for all the actions to be performed when certain buttons are clicked.

* ChefController Class

This works together with the ChefView class implementing the logic behind the interface. In this case the job of the controller is very simple being the observer of the restaurant class, it only has to update the view whenever restaurant notifies it.

* AdministratorView Class

This class consists of all the necessary objects and methods in order to display the interface for the user and to provide the needed IO functionalities. There are 8 different panels that are accessible through the provided buttons. Each panel is meant to display the objects related to one of the operations supported by the administrator interface.

* WaiterView Class

This class consists of all the necessary objects and methods in order to display the interface for the user and to provide the needed IO functionalities. There are 5 different panels that are accessible through the provided buttons. Each panel is meant to display the objects related to one of the operations supported by the waiter interface.

* ChefView Class

This class implement the view of the Chef interface which is simpler than the other two. It consists of only one JTable in which the orders are displayed. There are no other operations to be done using this interface so its only job is to display information in a graphical way.

* AdministratorPanels Enum

The purpose of this enumeration is to define identifiers for each panel of the AdministratorView class. Using the attributes of this enumeration the communication between AdministratorController class and AdniminstratorView is greatly improved.

* WaiterPanels Enum

The purpose of this enumeration is to define identifiers for each panel of the WaiterView class. Using the attributes of this enumeration the communication between WaiterController class and WaiterView is greatly improved.

1. Results

The testing of the application is done on the relation between the classes from the business layer and the operations implemented by the restaurant class. We have to make the sure that the Composite Design Pattern is well implemented. Next we focus on the restaurant class where all the operations have to be tested. Given the fact that the class uses the design by contract model the testing is very easy. Using the method isWellFormed we can check if the state of the objects is as we exepct.

Some of the scenarios for which the application was tested are:

* Trying to add orders with the same table number.
* Trying to add orders with the same orderID.
* Deleting a base product which is part of other composite products.
* Trying to generate a bill for a non-existing order.

1. Conclusions

The main objective of this project was to get used working with design patterns as well as understanding the data structures provided by java, getting to know their advantages and disadvantages such that you always choose the one that fit your requirements better.

In terms of future improvements, we can consider implementing an user identification mechanism such that more users are supported not just three. The data could be stored in a database which provides a better way of organization.

1. Bibliography

* <https://javarevisited.blogspot.com/2012/01/what-is-assertion-in-java-java.html>
* <https://www.tutorialspoint.com/java/java_serialization.htm>
* <https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#tag>
* <https://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html>
* Ppt presentation: Assignment\_4\_Indications