Fundamental Programming Techniques

Assignment 4:

Restaurant Management System

Name: Filimon Adelin

Group: 30424

1. Objective

The objective of this assignment is to design a java application capable of processing commands specific to a restaurant system. The design of this application followed the next steps:

1. Create and implement the provided UML diagram;
2. Use Composite pattern for menu items;
3. Design the user interface following MVC design pattern;
4. Use Observer pattern to notify the chef;
5. Create pre / post conditions and invariants for model class.

First, the application is created based on a given UML diagram containing the next layers: Presentation Layer, Business Layer, and Data Layer. In the Presentation Layer resides the MVC-related classes distributed in controller and view packages, after their usage. The Business Layer provides all business-related classes such as the MenuItem or Restaurant class. The output saves through serialization and also the bill generator resides in the Data Layer.

The restaurant menu represents a list of menu items. This menu items follow the composite design pattern such that a product can contain another product. In the second step this behavior is accomplished using inheritance and aggregation.

In this stage of the development, the application is unable to interact with clients. The clients of the application are Administrator, Waiter and Chef. The first two must manage the menu and the orders of the restaurant and therefore the application must provide a user interface for those two. Using model-view-controller design pattern and with help of two dependencies, MigLayout and Flatlaf, the UI task is accomplished.

The Chef client have no so much interaction with our app but to get notified each time a composite product is needed. That behavior is completed using Observer pattern.

The pre / post conditions are accomplished using assertions and the invariants are checked using a “well-formed” method.

2. Problem analysis

The application should accomplish basic operations of a restaurant: create menu item, delete menu item, edit menu item, create order, compute price for an order and generate a bill in text format. Also, the application should be able to load/save restaurant data using serialization.

It is pretty obvious that the main actors of this app are the administrator and the waiter. In Figure 1 it is shown the use case diagram of the administrator.

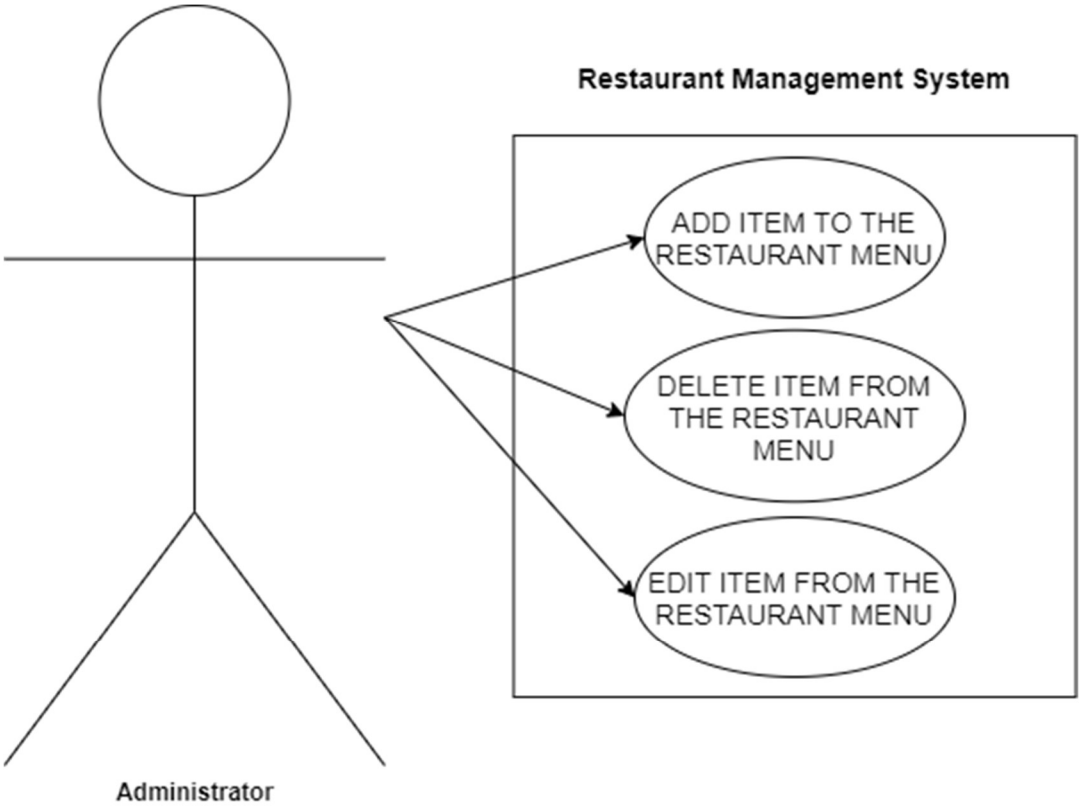

Figure 1

Use case: Add Item to the Restaurant Menu;

Preconditions: -

Main success scenario:

1. The administrator presses the “ADD PRODUCT” button;
2. The application opens a new dialog asking for input;
3. The administrator selects what type of item to add (Base or composite);
4. The administrator writes a product name and a price if the base product is selected;
5. The administrator writes a product name and selects the composite of the product from the menu list;
6. The administrator presses the “OK” button;
7. The application validates the input. If the input is incorrect an error message is displayed;
8. If the input is valid the application will create a new menu item which should be visible in the Table.

Alternative sequence:

- At any moment of time the administrator can cancel the operation pressing “Cancel” or “X”.

Use case: Delete Item from the Restaurant Menu;

Preconditions: -

Main success scenario:

1. The administrator selects from the table the item/items which wants to delete;
2. The administrator presses the “DELETE PRODUCT” button;
3. The application removes the specified items and the composite products containing those items.

Use case: Edit Item from the Restaurant Menu

Preconditions: -

Main success scenario:

1. The administrator selects from the table the item which wants to modify;
2. The administrator presses the “EDIT PRODUCT” button;
3. The application opens a new dialog asking for input;
4. The administrator can change the product name and price in case of a base product and product name and the composition of the product in case of composite products;
5. The administrator presses the “OK” button;
6. The application validates the input. If the input is incorrect an error message is displayed;
7. If the input is valid the application will modify the item according to the specified input.

Alternative sequence:

- At any moment of time the administrator can cancel the operation pressing “Cancel” or “X”.

*This operation could be performed directly in the table for some fields:

1. The administrator double-clicks the field of an item which wants to modify;
2. If that field is not a price of a composite product or the composition of a product then the application permits to change that field;
3. The administrator changes the field and presses “Enter”;
4. The application validates the input. If the input is incorrect the application will not change the field;
5. If the input is valid the application will modify the item accordingly;

In the Figure 2 it is presented the use case diagram of the waiter.

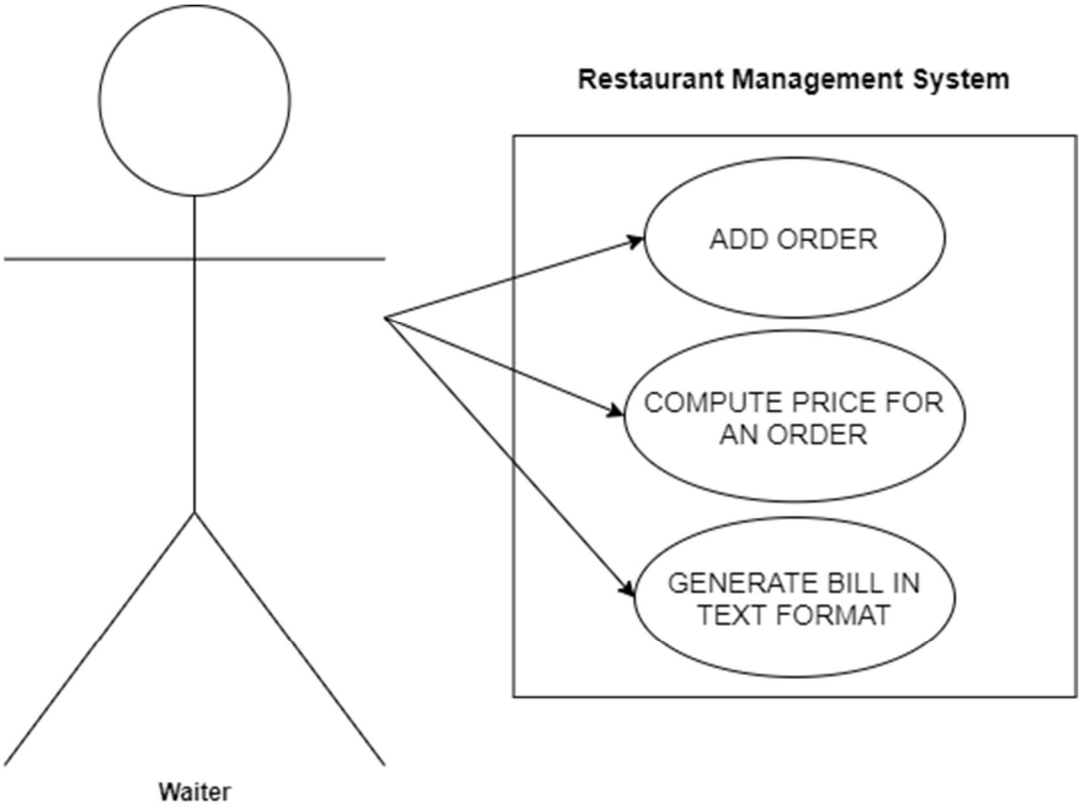

Figure 2

Use case: Add Order to the Restaurant Orders List;

Preconditions: -

Main success scenario:

1. The waiter presses the “ADD ORDER” button;
2. The application opens a new dialog asking for input;
3. The waiter writes a date and a table number for the order;
4. The waiter double-clicks the items from the menu list which want to be part of the order;
5. The waiter presses the “OK” button;
6. The application validates the input. If the input is incorrect an error message is displayed;
7. If the input is valid the application will create a new order which should be visible in the Table.

Alternative sequence:

- At any moment of time the waiter can cancel the operation pressing “Cancel” or “X”.

Use case: Compute price for an Order;

Preconditions: -

Main success scenario:

1. The waiter selects from the table the order which total wants to be computed;
2. The application computes the total price for the order;
3. The application opens a dialog specifying the total price for that order.

Use case: Generate bill in text format

1. The waiter selects from the table the order which bill wants to be generated;
2. The application tries to generate a bill in text format with information about the order;
3. The application opens a dialog specifying the state of the operation;
4. If the operation succeeds, an output file with name bill#.txt will appear.

The chef is notified each time when a new order with composite products is being added to the orders list. This fact is observed in the UI of the Chef where the application prints a message for Chef with products to cook.

*The application can take one parameter: the .ser file of the restaurant. If the parameter is not specified the app will create a new instance of Restaurant. The serialization is done after pressing "Yes" on the dialog which appear when the application is closing. The name of the generated file is "restaurant.ser".

3. Design of the application

The application is split in three main packages: business, data, and presentation. The last one is also containing another two packages: controller and view. It is considered that the model of the MVC architecture is the Restaurant class.

Starting from data package, I considered to use Singleton pattern on FileWriter and RestaurantSerializator since they work with IO connections and there is no need for multiple instances of those classes.

The business package contains all model related classes: BaseProduct, CompositeProduct, MenuItem, Order, and Restaurant. The CompositeProduct class acts as a container for another products, it can contain another base or composite products. Its price is calculated from that list. The list must not contain duplicates what leads to a Set data structure but such structures are not good enough for displaying in a table because it is not iterable. This disadvantage of Set structure makes me choose an ArrayList which is more suitable to iteration operations, and for ensuring the unicity of the composition a validation must be performed before a new element is inserted. The Restaurant class make evidence of orders in a HashMap structure. The key of the structure is an Order, with hashCode and equals methods implemented and the value represents a list of MenuItems; in that case the unicity doesnt matter. In this class resides the menu which is, again, an ArrayList with some validation. In this package can be found the IRestaurantProcessing interface which defines the restaurant operations that can be performed by actors described above.

In the presentation package, in the view section, it is implemented swing related classes such as AdministratorGUI, WaiterGUI, and ChefGUI. Those classes are extending JPanel so that they could be used in a JTabbedPane, the application frame content. The rest of the classes from the view package represents helper classes in order to improve the UI. They are: AddProductPanel, EditProductPanel, and AddOrderPanel, used for displaying the dialogs in one of those cases; MenuTableModel and OrderTableModel, used for improving the JTables with the desired functionality. They inherit the AbstractTableModel class and implement specific behavior of those methods.

The presentation package also presents the controller section, in which we can found the AdministratorController, WaiterController, the controllers used for implementing the needed listeners, and

ApplicationFrame, the root frame of the application. I decided to place this class in the controller package since it represents the starting point of the application and use the necessary components to initialize the system.

The UML diagrams for classes and packages are shown below:

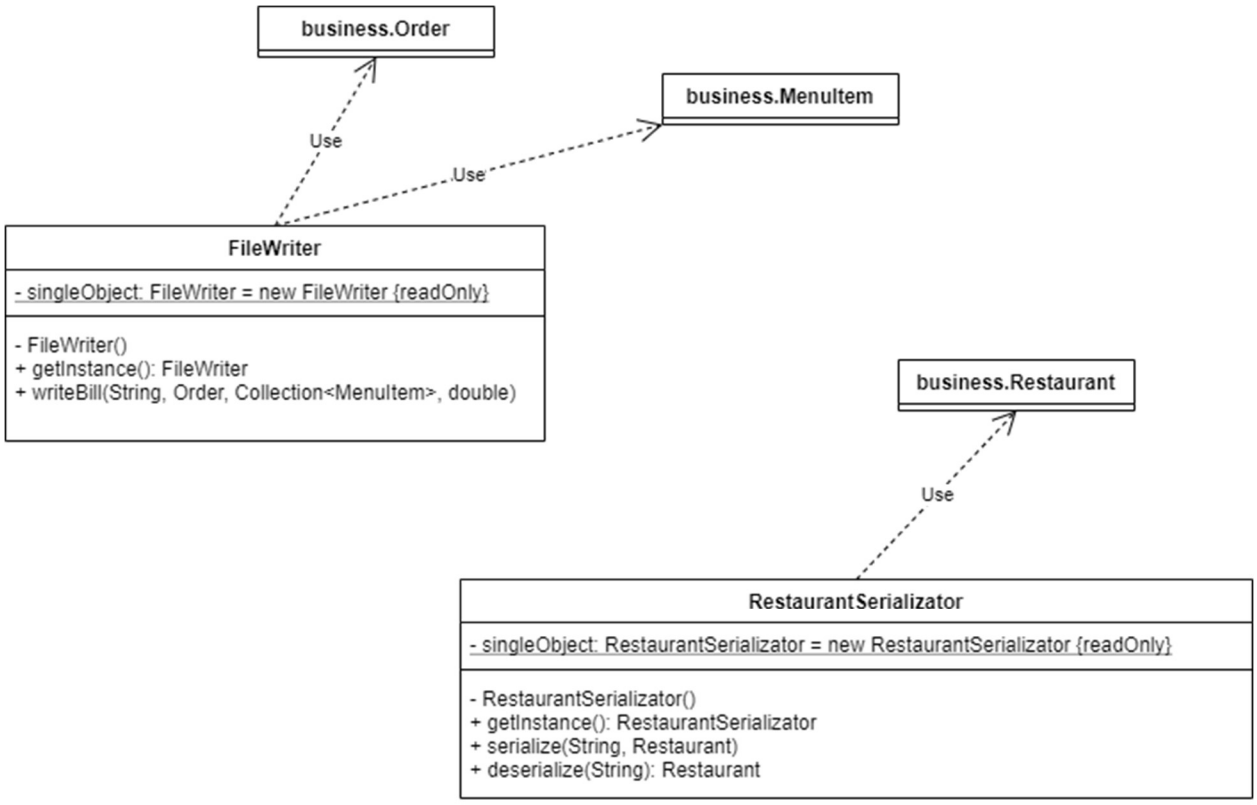

Figure 3. data package

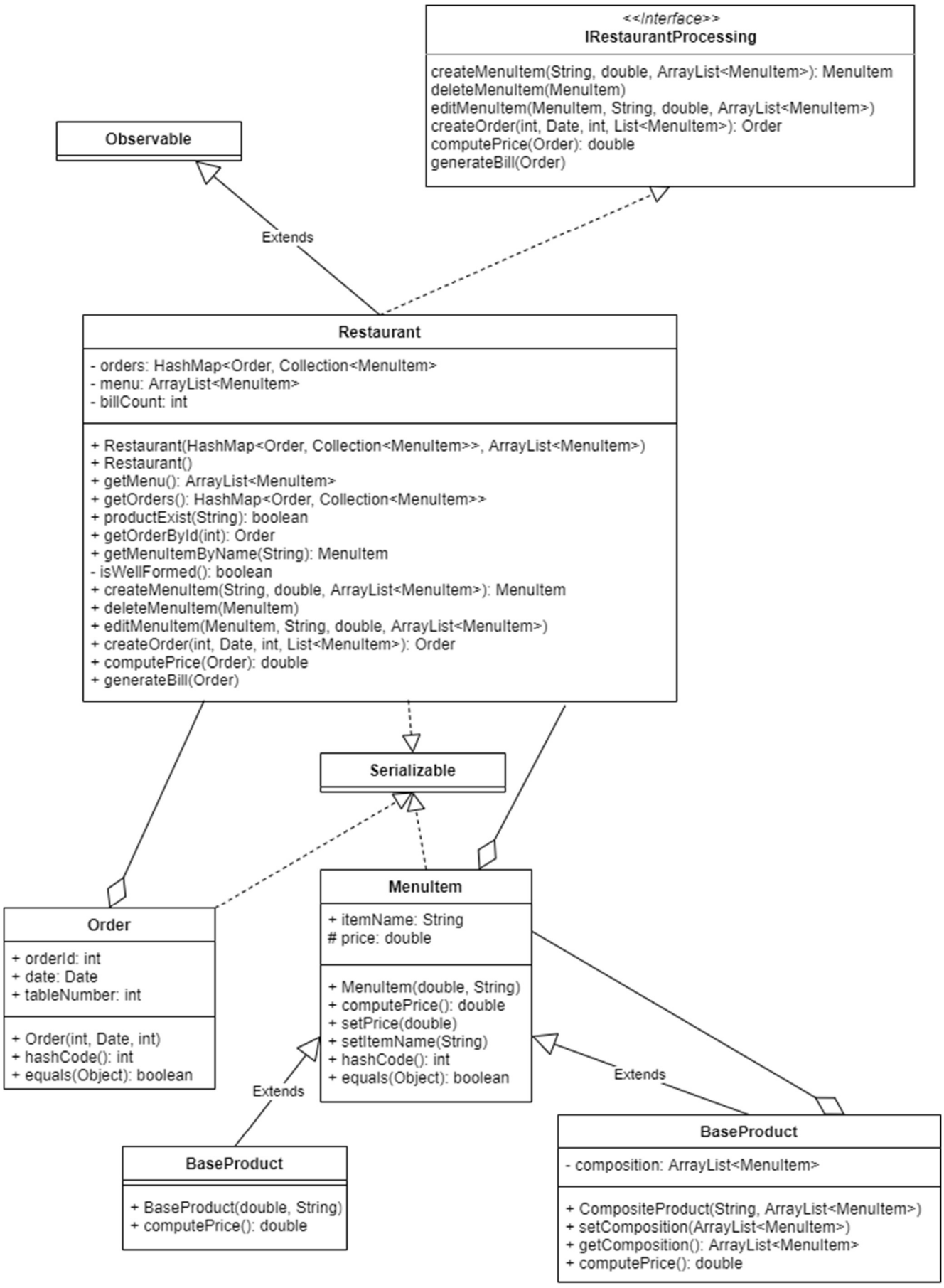

Figure 4. business package

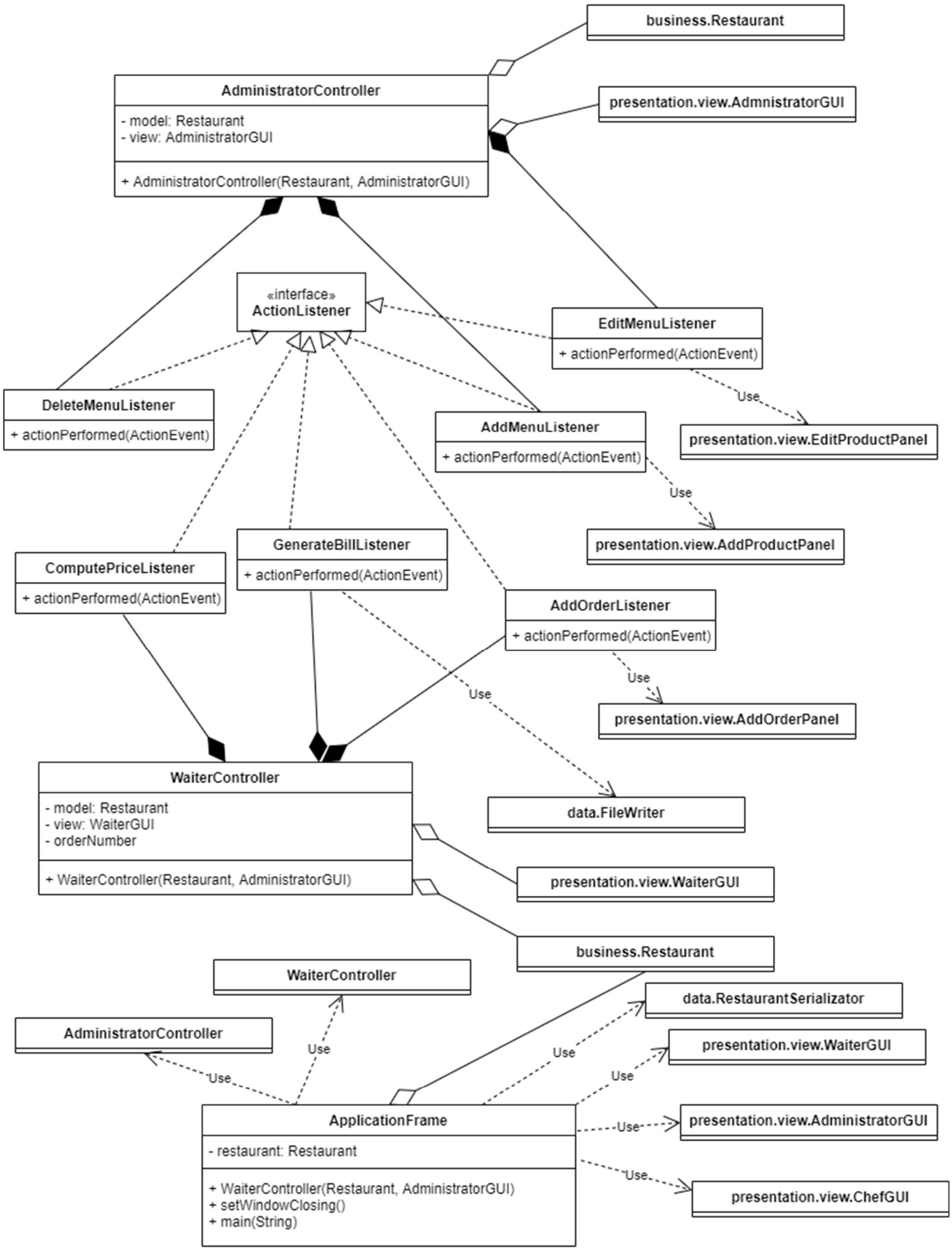

Figure 5. presentation.controller package

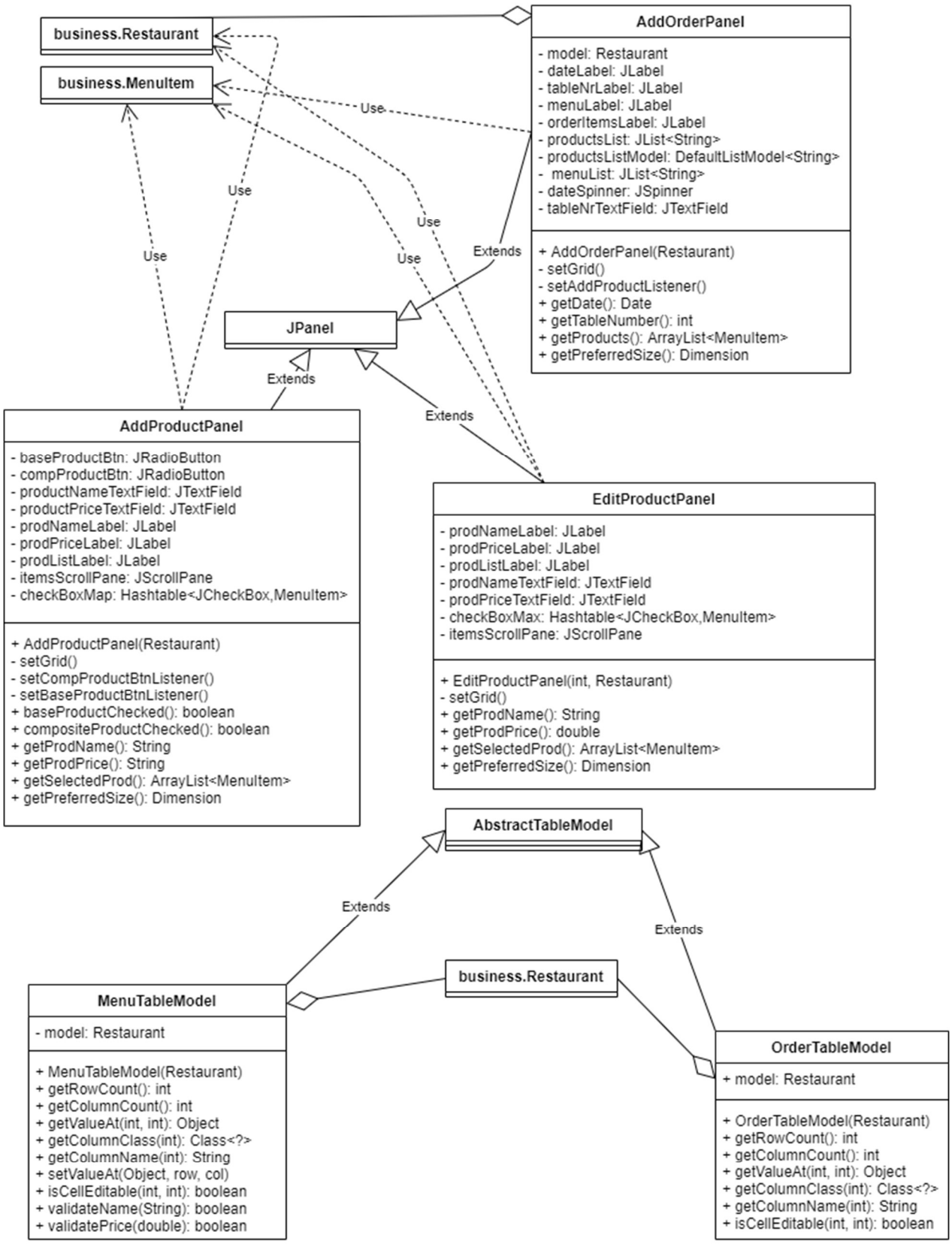

Figure 6.1 presentation.view package

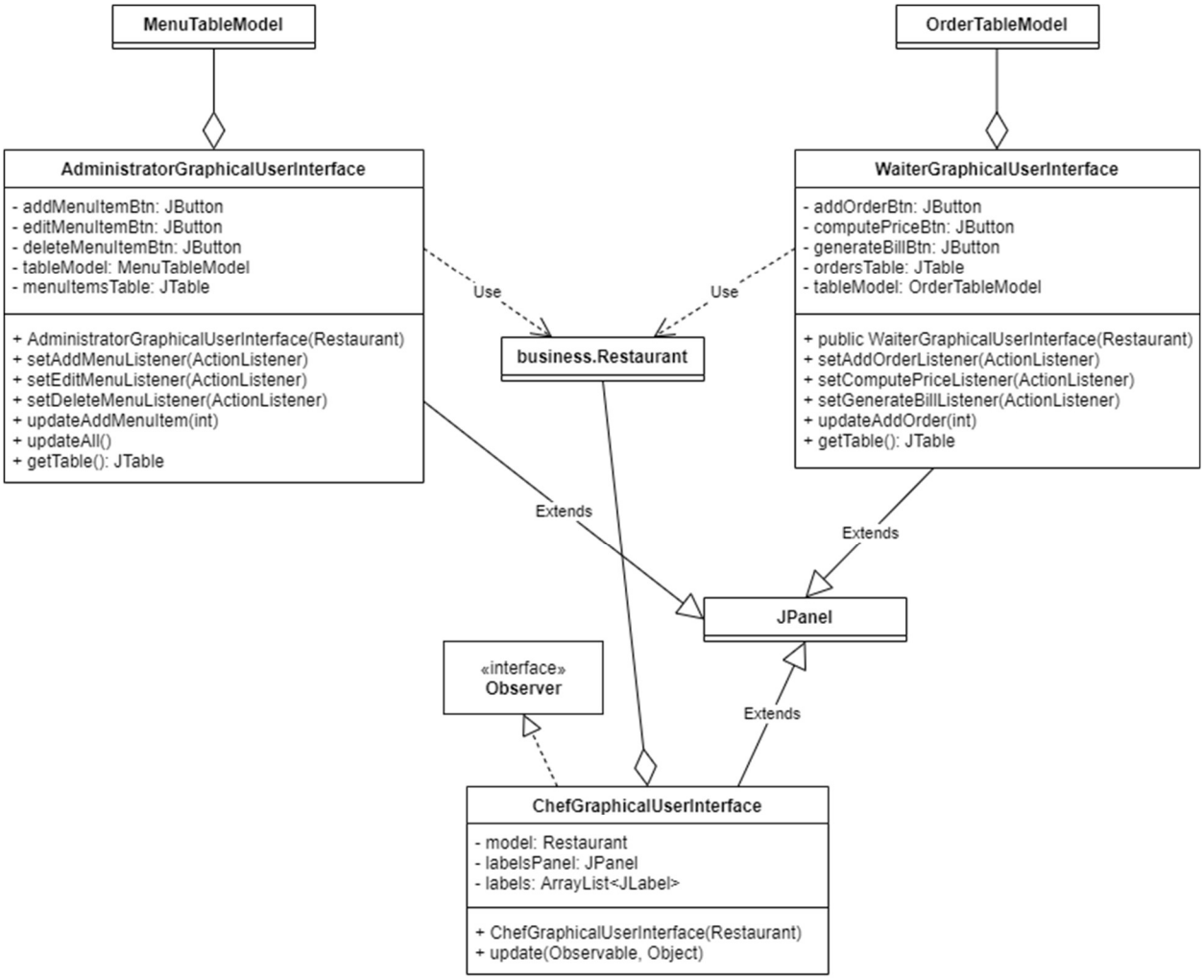

Figure 6.2 presentation.view package

4. Implementation

FileWriter class:

- Field: private static final FileWriter singleObject: the only object of this class.
- Constructor: private FileWriter(): the private constructor doesnt allow new instantiations of the FileWriter class.
- Method: public static FileWriter getInstance(): returns the singleObject.
- Method: public void writeBill(String filename, Order order, Collection<MenuItem> items, double total): writes a bill in text format with specified name which contain order information specified in the parameters list. This method throws IOException and must be handled.

RestaurantSerializator class:

- Field: private static final RestaurantSerializator singleObject: the only object of this class.

- Constructor: private RestaurantSerializator(): the private constructor doesnt allow new instantiations of the RestaurantSerializator class.
- Method: public void serialize(String filename, Restaurant restaurant): realize the serialization of the Restaurant object specified in parameters list. The file is saved with the filename path.
- Method: public Restaurant deserialize(String filename): realize the deserialization of the file with path specified as parameter and returns the Restaurant object obtained.

*The serialize and deserialize methods can print to the standard output if some exception is thrown.

BaseProduct class:

- Field: private static final long serialVersionUID: the field used for serialization.
- Constructor: public BaseProduct(double price, String itemName): makes a call to super constructor initializing the super fields with the values specified as parameters.
- Method: public double computePrice(): this method is inherited from the abstract class MenuItem and implemented such that it returns the price field.

CompositeProduct class:

- Field: private static final long serialVersionUID: the field used for serialization.
- Field: private ArrayList<MenuItem> composition: the list of MenuItems that this product contains.
- Constructor: public CompositeProduct(String itemName, ArrayList<MenuItem> composition): makes a call to super constructor initializing the price and itemName fields and initialize the composite field with the one specified in parameters list.
- Method: public double computePrice(): this method is inherited from the abstract class MenuItem and implemented such that it returns the sum of all prices of menuitems the composite list contains.

MenuItem class:

- Field: private static final long serialVersionUID: the field used for serialization.
- Field: public String itemName: the name of the item.
- Field: protected double price: the price of the item.
- Constructor: public MenuItem(double price, String itemName): initializes the itemName and price fields.
- Method: public abstract double computePrice(): abstract method which needs to be implemented by child classes.

*The class have also the hashCode() and equals() methods implemented.

Order class:

- Field: private static final long serialVersionUID: the field used for serialization.
- Field: public int orderId: the id of the order.
- Field: public Date date: the date this order was created.
- Field: public int tableNumber: the number of the table at which the order was taken.
- Constructor: public Order(int orderId, Date date, int tableNumber): the constructor initialize the fields with the specified values.

*The class have also the hashCode() and equals() methods implemented.

IRestaurantProcessing interface:

- Method: MenuItem createMenuItem(String menuItemName, double price, ArrayList<MenuItem> menuItems): creates a new MenuItem and returns it.
- Method: void deleteMenuItem(MenuItem menuItem): deletes a MenuItem from the menu list.
- Method: void editMenuItem(MenuItem menuItem, String menuItemName, double price, ArrayList<MenuItem> items): changes the MenuItem specified by the parameter with the fields specified fields.
- Method: Order createOrder(int orderId, Date date, int table, List<MenuItem> menuItems): creates a new Order and returns it.
- Method: double computePrice(Order order): returns the total price of the specified order.
- Method: void generateBill(Order order) throws IOException: generates a bill in text format using the Order specified as parameter.

Restaurant class:

- Field: private static final long serialVersionUID: the field used for serialization.
- Field: private final HashMap<Order, Collection<MenuItem>> orders: the map structure representing the orders of the restaurant.
- Field: private final ArrayList<MenuItem> menu: the menu of the restaurant.
- Field: private int billCount: the field used for generating the name of the bill in text format.
- Constructor: public Restaurant(HashMap<Order,Collection<MenuItem>> orders, ArrayList<MenuItem> menu): initialize the fields with the specified ones in the parameter list.
- Constructor: public Restaurant(): initialize the fields with new objects.
- Method: public MenuItem createMenuItem(String itemName, double price, ArrayList<MenuItem> composition):
  assert isWellFormed();
  assert itemName != null && price > 0;
  int previousSize = menu.size();
  MenuItem menuItem;
  if(composition == null) menuItem = new BaseProduct(price, itemName);
  else menuItem = new CompositeProduct(itemName, composition);
  menu.add(menuItem);
  assert menu.size() == previousSize + 1;
  assert isWellFormed();
  return menuItem;
  This method assumes that the itemName parameter is not null and also the price is greater than 0. It checks the composition parameter, if it is null the method returns a BaseProduct else it will return a CompositeProduct with the specified fields.
- Method: public void deleteMenuItem(MenuItem menuItem):
  assert isWellFormed();
  assert menu.size() > 0 && menuItem != null && menu.contains(menuItem);
  int previousSize = menu.size();
  menu.remove(menuItem);
  assert menu.size() == previousSize - 1;
  assert isWellFormed();
  This method assumes that the size of the menu is greater than zero and the menu contains the specified menu item.
- Method: public void editMenuItem(MenuItem menuItem, String itemName, double price, ArrayList<MenuItem> composition):

```
assert isWellFormed();
assert menuItem != null && menu.contains(menuItem);
assert price > 0 || price == -1;
if(itemName != null) menuItem.setItemName(itemName);
if(price != -1) menuItem.setPrice(price);
if(composition != null && menuItem instanceof CompositeProduct) {
        ((CompositeProduct) menuItem).setComposition(composition);
}
assert isWellFormed();
```

This method can be used to change any field of a menu item. If you need to change just one field you could place null in case of objects for which you dont want to modify the field and -1 in case of the price.

- Method: public Order createOrder(int orderId, Date date, int tableNumber, List<MenuItem> items)

```
assert isWellFormed();
assert orderId >= 0 && date != null && tableNumber >= 0 && items != null && items.size() > 0;
int previousSize = orders.size();
Order order = new Order(orderId, date, tableNumber);
orders.put(order, items);
for(MenuItem menuItem : items) {
        if(menuItem instanceof CompositeProduct) {
        setChanged();
        notifyObservers(order);
        break;
        }

}

assert orders.size() == previousSize + 1;

assert isWellFormed();

return order;
```

The newOrder method will notify the chef if the order presents composite products.

- Method: public double computePrice(Order order):

```
assert isWellFormed();
    assert order != null && orders.containsKey(order);
    Collection<MenuItem> items = orders.get(order);
    double price = 0;
    for(MenuItem menuItem : items) {
       price += menuItem.computePrice();
    }
    assert isWellFormed();
    return price;
```

- Method: public void generateBill(Order order) throws IOException.
- Method: private boolean isWellFormed():

```
for(Order order : orders.keySet()) {
    if(order == null || orders.get(order) == null || orders.get(order).size() == 0) return false;
}
for(MenuItem item : menu) {
   if(item == null || item.price <= 0) return false;
   if(item instanceof  CompositeProduct) {
      CompositeProduct compositeItem = (CompositeProduct) item;
if(compositeItem.getComposition() == null ||compositeItem.getComposition().size()< 1)   return false;
```

```
double checkPrice = 0;
for(MenuItem subItem : compositeItem.getComposition()) {
      checkPrice += subItem.computePrice();}
if(checkPrice != compositeItem.computePrice()) return false;
   }
}
return true;
```

This method will check the invariant of the class and is present at the beginning and at the end of methods which could change the state of the object.

*The presentation classes have pretty obvious behavior that can be identified by the methods and fields names.

5. Conclusions

The application can be improved by implementing more functionality to the Chef user and can use some sort of database with products and orders. Also, a login system is suitable for this kind of applications.

From this assignment I understood the concepts of pre and post conditions and also invariants, to work with them and to work with Observer and Composite design patterns. Also I learned to work with HashMap structure.