Restaurant Management

Documentation

**Vlad Lujerdeanu**

Group 30226 | Year 2 | Semester 2

# Contents

1. [The Objective](#_The_Objective) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2. [Analysis, Modeling and Use Cases](#_Analysis,_Modeling,_Scenarios) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
3. [Designing Phase](#_DesignING_Phase) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
4. [Implementation](#_Implementation) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
5. [Results](#_Results) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
6. [Conclusion](#_Conclusions) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
7. [References](#_References) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

# The Objective

The Objective of this project is to design and implement a management system for a restaurant. The system needs to be used by three types of users: the Administrator, the Waiter and the Chef. The Administrator can add menu items, delete menu items or modify the existing menu items. The Waiter can add new orders, see the orders already placed and compute the bill for each order. The Chef can only see the orders without being able to modify or add orders or menu items.

The app is structured in three packages:

* Business logic – It deals with the functionalities of the management system.
* Data Layer – It deals with the Bill generation and Serialization of data.
* Graphical User Interface Layer – It deals with the GUI displayed on the screen and the functionalities associated with it.

# [Analysis](#_Analysis), [Modeling](#_Modeling) and [Use Cases](#_Use_Cases)

### Analysis

A management system is a collection of functionalities working together used by an organization to ensure that it can fulfill the tasks required to achieve its objectives. These objectives cover many aspects of the organization's operations including safe operation, product management, client relationships and worker management.

### Modeling

The application will be receiving data from an input .ser file given as the first argument when running the jar file. The Serialization file be generated automatically and updated every time the application is closed. Note: Force closing the application will result in serialization errors.

The menu item classes are based on the Composite Design Pattern.

The functionalities of the order viewing system use the Observer Design Pattern and are updated every time there is a change in Orders.

The Order storing system is implemented using a Hash Map with the hash code being the id of the order modulo the number of tables in the restaurant.

The restaurant class implements the functionalities described in the IRestaurantProcessing interface and the data from this class is stored in the file “restaurant.ser”.

### Use Cases

* Administrator:
  1. Running in terminal the command:

java -jar Restaurant\_Management.jar <input File>

* 1. Use the GUI tab for Administrator to Add, Edit or Remove menu items.
* Waiter:
  1. Running in terminal the command:

java -jar Restaurant\_Management.jar <input File>

* 1. Use the GUI tab for Waiter to Add Orders, Generate de Bill or View all the orders.
* Chef:
  1. Running in terminal the command:

java -jar Restaurant\_Management.jar <input File>

* 1. Use the GUI tab for Waiter to View all the orders.

# DesignING Phase

### Design Decisions

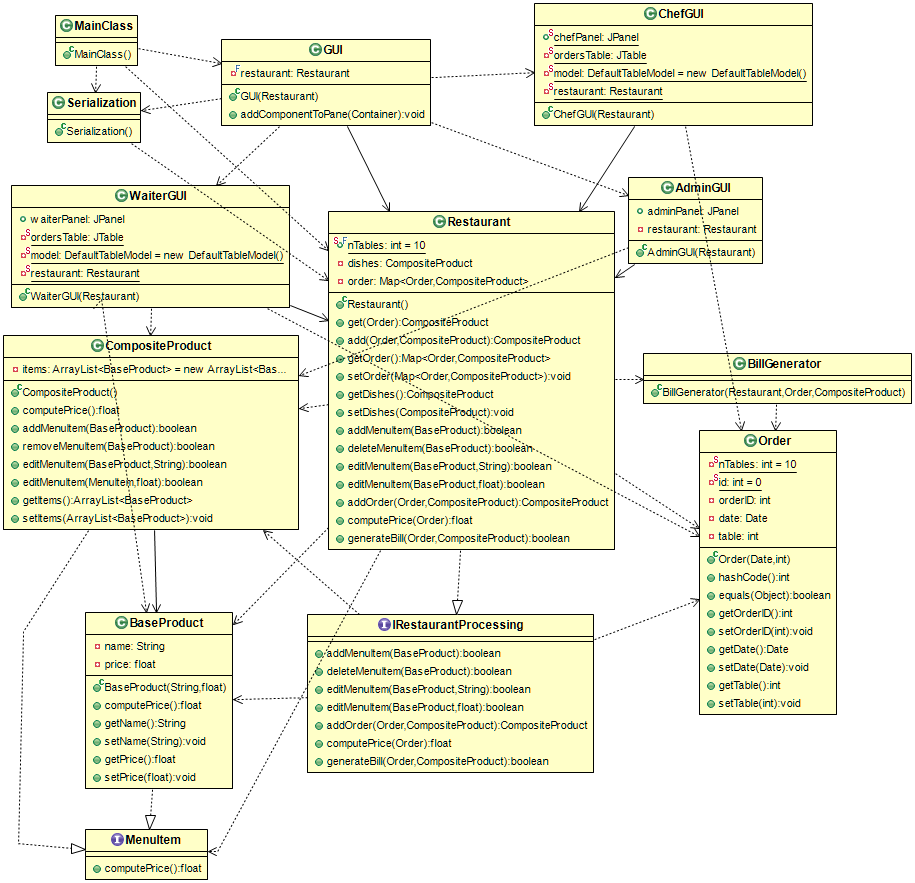
Designing process for this project was a top-down one, and the implementation was bottom-up, firstly implementing the Menu Items using the Composite Design Pattern, then implementing the Order class and the Restaurant class. Then, adding the GUI and all its functionalities. Lastly, adding the data layer including the Serialization class and the Bill Generator.

Firstly, the application expects the input to be a ser file given as the first argument when running it which contains the information stored in the restaurant class.

Another design decision is following the structure of the Data Layer, the Business Layer and the Presentation Layer for separation between the functionalities.

A very important design decision is implementing Menu items using the Composite Design Pattern and using the Observer Design Pattern to update the table of Orders in the GUI.

### UML Diagram



# Implementation

#### Restaurant:

*/\*\*  
 \* This class manages all the restaurant functionalities.  
 \*/*public class Restaurant implements IRestaurantProcessing, Serializable {  
 public static final int *nTables* = 10;  
 private CompositeProduct dishes;  
 private Map<Order, CompositeProduct> order;  
  
 */\*\*  
 \* The Class constructor  
 \*/* public Restaurant() {  
 this.order = new HashMap<Order, CompositeProduct>();  
 dishes = new CompositeProduct();  
 Order.*setNTables*(*nTables*);  
 }  
  
 */\*\*  
 \* This method returns the menu items corresponding to the order given as parameter.  
 \*  
 \** ***@param*** *theOrder the order for which the menu items are returned  
 \** ***@return*** *the menu items from the order given as parameter.  
 \** ***@pre*** *theOrder != null  
 \** ***@post*** *@return == Collection<MenuItem> from the order  
 \*/* public CompositeProduct get(Order theOrder) {  
 return this.order.get(theOrder);  
 }  
  
 */\*\*  
 \* This method adds a items to the hash map and return the items given as parameter 'm'.  
 \*  
 \** ***@param*** *o the order  
 \** ***@param*** *m items corresponding to the order  
 \** ***@return*** *the collection of items given as parameter 'm'  
 \** ***@pre*** *order != null  
 \** ***@pre*** *o != null  
 \** ***@pre*** *m != null  
 \** ***@post*** *@return == Collection<MenuItem> from the order  
 \*/* public CompositeProduct add(Order o, CompositeProduct m) {  
  
 CompositeProduct cp = this.order.put(o, m);  
 WaiterGUI.*update*();  
 ChefGUI.*update*();  
 return cp;  
 }  
  
 */\*\*  
 \* This method returns the order hash map.  
 \*  
 \** ***@return*** *the order hash map  
 \** ***@pre*** *order != null  
 \** ***@post*** *@return == order  
 \*/* public Map<Order, CompositeProduct> getOrder() {  
 return order;  
 }  
  
 */\*\*  
 \* This method sets the order hash map.  
 \*  
 \** ***@pre*** *order != null  
 \*/* public void setOrder(Map<Order, CompositeProduct> order) {  
 this.order = order;  
 }  
  
 */\*\*  
 \* This method is a getter for 'dishes' Composite Product.  
 \*  
 \** ***@return*** *MenuItem dishes  
 \*/* public CompositeProduct getDishes() {  
 return dishes;  
 }  
  
 */\*\*  
 \* This is a setter method for 'dishes' Composite Product.  
 \*  
 \** ***@param*** *dishes Composite Product with all dishes  
 \*/* public void setDishes(CompositeProduct dishes) {  
 this.dishes = dishes;  
 }  
  
 @Override  
 public boolean addMenuItem(BaseProduct m) {  
 return getDishes().addMenuItem(m);  
 }  
  
 @Override  
 public boolean deleteMenuItem(BaseProduct m) {  
 return getDishes().removeMenuItem(m);  
 }  
  
 @Override  
 public boolean editMenuItem(BaseProduct m, String name) {  
 return getDishes().editMenuItem(m, name);  
 }  
  
 @Override  
 public boolean editMenuItem(BaseProduct m, float price) {  
 return getDishes().editMenuItem(m, price);  
 }  
  
 @Override  
 public CompositeProduct addOrder(Order o, CompositeProduct items) {  
 return add(o, items);  
 }  
  
 @Override  
 public float computePrice(Order o) {  
 float total = 0f;  
 ArrayList<BaseProduct> orderItems = get(o).getItems();  
 for (MenuItem bp : orderItems) {  
 total += bp.computePrice();  
 }  
 return total;  
 }  
  
 @Override  
 public boolean generateBill(Order o, CompositeProduct cp) {  
 new BillGenerator(this, o, cp);  
 return true;  
 }  
}

#### Order:

*/\*\*  
 \* This class creates an object that defines the details about an order  
 \*/*public class Order implements Serializable {  
 private static int *nTables* = 10;  
 private static int *id* = 0;  
 private int orderID;  
 private Date date;  
 private int table;  
  
 */\*\*  
 \* Constructor for the Order object  
 \*  
 \** ***@param*** *date order date  
 \** ***@param*** *table order table  
 \*/* public Order(Date date, int table) {  
 this.orderID = ++*id*;  
 this.date = date;  
 this.table = table;  
 }  
  
 @Override  
 public int hashCode() {  
 return orderID % *nTables*;  
 }  
  
 @Override  
 public boolean equals(Object obj) {  
 return this.orderID == ((Order) obj).getOrderID() && this.date == ((Order) obj).getDate() && this.table == ((Order) obj).getTable();  
 }  
  
 /// GETTERS AND SETTERS  
  
 public int getOrderID() {  
 return orderID;  
 }  
  
 public void setOrderID(int orderID) {  
 this.orderID = orderID;  
 }  
  
 public Date getDate() {  
 return date;  
 }  
  
 public void setDate(Date date) {  
 this.date = date;  
 }  
  
 public int getTable() {  
 return table;  
 }  
  
 public void setTable(int table) {  
 this.table = table;  
 }  
  
 public static void setNTables(int nTables) {  
 Order.*nTables* = nTables;  
 }  
}

#### Base Product:

public class BaseProduct implements MenuItem, Serializable {  
 private String name;  
 private float price;  
  
 public BaseProduct(String name, float price) {  
 this.name = name;  
 this.price = price;  
 }  
  
 public float computePrice() {  
 return price;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public float getPrice() {  
 return price;  
 }  
  
 public void setPrice(float price) {  
 this.price = price;  
 }  
}

#### Composite Product:

public class CompositeProduct implements MenuItem, Serializable {  
 private ArrayList<BaseProduct> items = new ArrayList<BaseProduct>();  
  
 public float computePrice() {  
 int total = 0;  
 for (MenuItem m : items) {  
 total += m.computePrice();  
 }  
 return total;  
 }  
  
 public boolean addMenuItem(BaseProduct m) {  
 if (m != null) {  
 items.add(m);  
 return true;  
 }  
 return false;  
 }  
  
 public boolean removeMenuItem(BaseProduct m) {  
 return items.remove(m);  
 }  
  
 public boolean editMenuItem(BaseProduct m, String name) {  
 for (BaseProduct i : items) {  
 if (i.equals(m)) {  
 i.setName(name);  
 return true;  
 }  
 }  
 return false;  
 }  
  
 public boolean editMenuItem(MenuItem m, float price) {  
 for (BaseProduct i : items) {  
 if (i.equals(m)) {  
 i.setPrice(price);  
 return true;  
 }  
 }  
 return false;  
 }  
  
 public ArrayList<BaseProduct> getItems() {  
 return items;  
 }  
  
 public void setItems(ArrayList<BaseProduct> items) {  
 this.items = items;  
 }  
}

#### Serialization:

public class Serialization {  
  
 public static void serialize(Restaurant restaurant) {  
  
 try {  
 FileOutputStream fout = new FileOutputStream("restaurant.ser");  
 ObjectOutputStream out = new ObjectOutputStream(fout);  
  
 out.writeObject(restaurant);  
 out.close();  
 fout.close();  
 System.*out*.println("Serialized data saved in restaurant.ser");  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public static Restaurant deserialize(String path) {  
 Restaurant rez = null;  
 try {  
 FileInputStream fileIn = new FileInputStream(path);  
 ObjectInputStream in = new ObjectInputStream(fileIn);  
 rez = (Restaurant) in.readObject();  
 in.close();  
 fileIn.close();  
 } catch (IOException | ClassNotFoundException i) {  
 i.printStackTrace();  
 }  
 return rez;  
 }  
}

#### Chef GUI:

*/\*\*  
 \* This class implements the Chef Panel for the GUI  
 \*/*public class ChefGUI extends Component {  
 public static JPanel *chefPanel*;  
 private static JTable *ordersTable*;  
 private static DefaultTableModel *model* = new DefaultTableModel();  
 private static Restaurant *restaurant*;  
  
 public ChefGUI(Restaurant restaurant) {  
 ChefGUI.*restaurant* = restaurant;  
  
 *chefPanel* = new JPanel();  
 *chefPanel*.setLayout(new BoxLayout(*chefPanel*, BoxLayout.*Y\_AXIS*));  
 JLabel ordersTitle = new JLabel("Orders: ");  
 *chefPanel*.add(ordersTitle);  
 *model*.addColumn("ID");  
 *model*.addColumn("Table");  
 *model*.addColumn("Order");  
 *getData*();  
 *ordersTable* = new JTable(*model*);  
  
 *chefPanel*.add(*ordersTable*.getTableHeader());  
 *chefPanel*.add(*ordersTable*);  
 }  
  
 private static void getData() {  
 if(*restaurant* != null) {  
 for (Map.Entry<Order, CompositeProduct> entry : *restaurant*.getOrder().entrySet()) {  
 String id = "" + entry.getKey().getOrderID();  
 String table = "" + entry.getKey().getTable();  
 String order = "";  
 for (BaseProduct m : entry.getValue().getItems()) {  
 order += m.getName() + ", ";  
 }  
 *model*.addRow(new Object[]{id, table, order});  
 }  
 }  
 }  
  
 public static void update() {  
 for(int i = *model*.getRowCount() - 1; i >= 0; i--){  
 *model*.removeRow(i);  
 }  
 *getData*();  
 }  
}

# Results

The project has been tested using “manually” by testing first each classes’ functionalities. The completed project has been tested using different input test files and checking the output files for logical errors.

# Conclusions

To conclude, the project presented above taught me a lot about analyzing a problem and designing a solution to fit a set of strict requirements, storing information in .ser files and also how to use Observer Design pattern, but most importantly, how to search the web for tutorials and help when I’m stuck on a problem.

# References

1. [Laboratory Guide – Ionel Giosan](http://users.utcluj.ro/~igiosan/teaching_poo.html) ( UML Generating in eclipse )
2. [Hash in Java](http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_4/Assignment_4_Indications.pdf) ( Simulation Manager Architecture )
3. [Stackoverflow.com](https://Stackoverflow.com) ( Minor problems with the code )