**Cristian Flaviu - Remus**

**Group 30423**

**Homework4**

1. **Objectives :**

The main purpose of this program is to implement 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 ordered through a waiter.

Second Objectives:

1. **Designing Model:** represent the data models of the application (BaseProduct, CompositeProduct, Restaurant , etc.)

1. **Implementing the Graphic User Interface (GUI).** Final step where we make a simple GUI for the user to interact with.
2. **Problem analysis, scenarios and use cases :** The program expects interaction with a user in order to introduce date and specify the flow of instruction. The user can choose what operation she/he desire to do, firstly, the user has to chose how to use the application (as Administrator, as Waiter or as a Chef).After that she / he can some specific operation depending on what type of user he is .

**Use case “Administrator”:**

1. Use case “Add a Base Product ”

The user have to open the program and choose to login as administrator by pressing on the button Administrator. After that she / he has to introduce the name of the BaseProduct and the Price . The user can choose any name for the product except those names who already exists on the restaurant menu and also can introduce any positive number in the textField for price;

1. Use case “Add a Composite Product”

The user have to open the program and choose to login as administrator by pressing on the button Administrator. After that she / he has to introduce a name for the composite product and choose which are the products contained by that composite product. In order to do that the user has to select from the comboBox and press the button add. After selecting the desired list of products the user has to press the button AddCompositeProduct.

1. Use case “Edit a Product”

The user have to open the program and choose to login as administrator by pressing on the button Administrator. After that she / he has to introduce the name of the product she /he wants to delete and after that the user have to choose new products which will compose the

edited product. In order to save the modification the use has to press the button edit.

1. Use case “Delete a Product”

The user have to open the program and choose to login as administrator by pressing on the button Administrator. After that she / he has to choose what product he/she wants to delete and press the button Delete.

1. Use case “View the Menu”

The user have to open the program and choose to login as administrator by pressing on the button Administrator. After that she / he has to press the ViewMenuITems button and a new window will be opened showing a list with all the MenuItems existing in the restaurant.

**Use case Waiter**

1. Use case “New Order ”

The user have to open the program and choose to login as waiter by pressing on the button waiter. After that the user has to introduce the value of the table from where the order is taken and also the user has to choose the list of products ordered by selecting from the comboBox the product and pressing the button add. After selecting all products

he / she has to press the button Order .

1. Use case “View All Orders”

The user have to open the program and choose to login as waiter by pressing on the button waiter. After that the user has to press the button View All Orders and a new window will be opened.

Alternative scenarios :

The user does not respect the specified format (he has introduced other characters than numbers,etc). In this case the system won’t produce any output and an error message will be displayed.

1. **Design (design decisions, UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interface)**

Firstly, we must design a design all model classes that we need to store the information(BaseProduct, CompositeProduct, Restaurant ,etc).

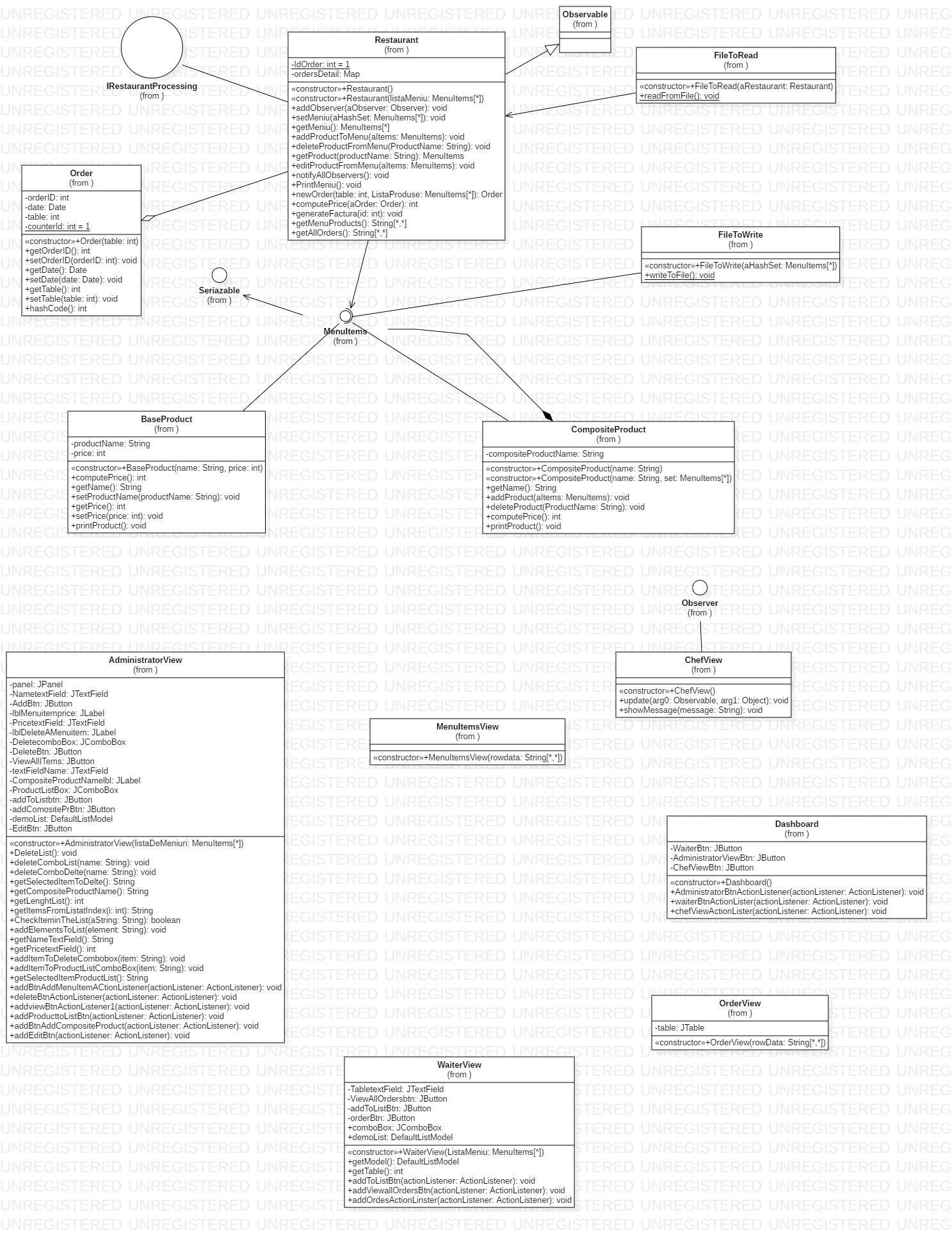
The final step will represent designing the GUI. It simply displays the required information in different Windows.

There are 3 packages :

* **bussinesLayer** which store the following classes: BaseProduct ,

CompositeProduct, MenuItems , FilteToRead, FileToWrite ,MenuItems, IrestaurantProcessing , Order and Restaurant

* **presentation** which hold the following classes : AdministratorView, ChefView, Controller ,Dashboard , MenuItemesView, OrderView ,WaiterView
* **dataLayer** which store the following classes: FilteToRead, FileToWrite



1. **Implementation**
2. BaseProduct

**public** **class** BaseProduct **implements** MenuItems {

**private** String productName;

**private** **int** price;

ProductName is the name of the Product

Price represent the price of the product

1. Composite Product

**public** **class** CompositeProduct **implements** MenuItems {

**private** String compositeProductName;

**private** HashSet <MenuItems> listOfProduct=**new** HashSet<MenuItems>();

CompositeProductName represent the name of the compositeProductName

ListOfProduct represent the list of products from the Restaurant

1. MenuItems

**public** **interface** MenuItems **extends** Serializable{

**public** **int** computePrice();

**public** **void** printProduct();

**public** String getName();

The method computePrice will compute the price of the product.

PrintProduct will print all baseProduct contained on the menu of the restaurant

(A compositeProduct is composed by BaseProducts)

GetName will return the name of the product;

1. **public** **class** Restaurant **extends** Observable **implements** IRestaurantProcessing{

**private** HashSet<Order> orders;

**private** HashSet<MenuItems> meniuri;

**private** Map<Order, HashSet<MenuItems>> ordersDetail;

**private** List<Observer> observatorii;

The attribute meniuri is a list with all the products contained in the restaurant

The attribute ordersDetail represent the map where the orders and the products ordered will be stored.

The attribute observatorii represent a list with all observers. In our case it will be only one, ChefView.

1. Order

**public** **class** Order {

**private** **int** orderID;

**private** Date date;

**private** **int** table;

**private** **static** **int** *counterId*=1;

The attribute *counterid* represent the id of the Order and will be unique for any order.It will be incremented with 1 at each order.

OrderID represents the id of the order

Date represents the data when the order has been placed

Table represents the table from where the order has been placed

1. **public** **class** FileToRead {

**static** Restaurant *aRestaurant*;

**public** FileToRead(Restaurant aRestaurant) {

**this**.*aRestaurant*=aRestaurant;

}

**public** **static** **void** readFromFile() {

ObjectInputStream inputStream;

**try** {

inputStream=**new** ObjectInputStream(**new** FileInputStream("Meniu.bin"));

HashSet<MenuItems> meniu=(HashSet<MenuItems>)inputStream.readObject();

*aRestaurant*.setMeniu(meniu);

} **catch** (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

This Method will read from the file “Menu.bin” and will add all the MenuItems to the restaurant

1. **public** **class** FileToWrite {

**private** **static** HashSet<MenuItems> *aHashSet*=**new** HashSet<>();

**public** FileToWrite(HashSet<MenuItems> aHashSet) {

**this**.*aHashSet*=aHashSet;

}

**public** **static** **void** writeToFile( ) {

**try** {

ObjectOutputStream outStream=**new** ObjectOutputStream(**new** FileOutputStream("Meniu.bin"));

outStream.writeObject(*aHashSet*);

} **catch** (IOException e) {

e.printStackTrace();

}

}

This method will serialize into the file Meniu.bin all the MenuItems from the HashSet.

1. AdminstratorView

**public** **class** AdministratorView **extends** JFrame {

**private** JPanel panel;

**private** JTextField NametextField;

**private** JButton AddBtn;

**private** JLabel lblMenuitemprice;

**private** JTextField PricetextField;

**private** JLabel lblDeleteAMenuitem;

**private** JComboBox DeletecomboBox;

**private** JButton DeleteBtn;

**private** JButton ViewAllITems;

**private** JTextField textFieldName;

**private** JLabel CompositeProductNamelbl;

**private** JComboBox ProductListBox;

**private** JButton addToListbtn;

**private** JButton addComositePrBtn;

**private** DefaultListModel demoList;

**private** JButton EditBtn;

This class represents the View of the Administrator where the user can add, delete, modify and also view all the products from the Menu.

1. **public** **class** WaiterView **extends** JFrame {

**private** JTextField TabletextField;

**private** JButton ViewAllOrdersbtn;

**private** JButton addToListBtn;

**private** JButton orderBtn ;

**public** JComboBox comboBox;

**public** DefaultListModel demoList;

This class represents the view of the Waiter where the user can make new orders and also view all the previous orders

1. Dashboard

**public** **class** Dashboard **extends** JFrame {

**private** JButton WaiterBtn;

**private** JButton AdministratorViewBtn;

**private** JButton ChefViewBtn;

**public** Dashboard() {

**this**.setBounds(100, 100, 615, 387);

setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

getContentPane().setLayout(**null**);

AdministratorViewBtn = **new** JButton("Administrator");

AdministratorViewBtn.setBounds(229, 114, 140, 29);

getContentPane().add(AdministratorViewBtn);

WaiterBtn = **new** JButton("Waiter");

WaiterBtn.setBounds(229, 177, 140, 34);

getContentPane().add(WaiterBtn);

ChefViewBtn = **new** JButton("Chef");

ChefViewBtn.setBounds(229, 242, 140, 29);

getContentPane().add(ChefViewBtn);

JLabel lblNewLabel = **new** JLabel("Fancy Restaurant");

lblNewLabel.setFont(**new** Font("Century", Font.***BOLD***, 21));

lblNewLabel.setHorizontalAlignment(SwingConstants.***CENTER***);

lblNewLabel.setBounds(178, 47, 230, 34);

getContentPane().add(lblNewLabel);

}

**public** **void** AdministratorBtnActionListener(ActionListener actionListener)

{

AdministratorViewBtn.addActionListener(actionListener);

}

**public** **void** waiterBtnActionLister(ActionListener actionListener)

{

WaiterBtn.addActionListener(actionListener);

}

**public** **void** chefViewActionLister(ActionListener actionListener)

{

ChefViewBtn.addActionListener(actionListener);

}

}

This class represent the first View from where the user has to choose how to use the application, as an administrator, as a waiter or as a chef.

1. Controller

**public** **class** Controller {

**private** Restaurant aRestaurant;

**private** AdministratorView administratorView;

**private** WaiterView waiterView;

**private** FileToRead fileToRead;

**private** Dashboard aDashboard;

**private** ChefView aChefView;

**public** Controller() {

aRestaurant = **new** Restaurant();

fileToRead = **new** FileToRead(aRestaurant);

fileToRead.*readFromFile*();

administratorView = **new** AdministratorView(aRestaurant.getMeniu());

waiterView = **new** WaiterView(aRestaurant.getMeniu());

aDashboard = **new** Dashboard();

aDashboard.setVisible(**true**);

aChefView = **new** ChefView();

aRestaurant.addObserver(aChefView);

initializeButtons();

}

**public** **void** initializeButtons() {

administratorView.addBtnAddMenuItemACtionListener(a -> {

String name = administratorView.getNameTextField();

**int** price = 0;

**if** (name.length() == 0) {

JOptionPane.*showMessageDialog*(**null**, "Null name");

} **else** {

**if** (aRestaurant.getProduct(name) != **null**) {

JOptionPane.*showMessageDialog*(**null**, "The product already exists");

} **else** {

**try** {

price = administratorView.getPricetextField();

MenuItems aBaseProduct = **new** BaseProduct(name, price);

aRestaurant.addProductToMenu(aBaseProduct);

administratorView.addItemToDeleteCombobox(name);

administratorView.addItemToProductListComboBox(name);

waiterView.comboBox.addItem(name);

}**catch** (Exception e) {

JOptionPane.*showMessageDialog*(**null**,"wrong number format");

}

}

}

});

administratorView.addviewBtnActionListener1(e -> {

MenuItemsView aItemsView = **new** MenuItemsView(aRestaurant.getMenuProducts());

aItemsView.setVisible(**true**);

});

administratorView.addProducttoListBtn(a -> {

**if** (!administratorView.CheckIteminTheList(administratorView.getSelectedItemProductList())) {

administratorView.addElementsToList(administratorView.getSelectedItemProductList());

} **else** {

JOptionPane.*showMessageDialog*(**null**, "the product already added to the list");

}

});

administratorView.addBtnAddCompositeProduct(a -> {

HashSet<MenuItems> aHashSet = **new** HashSet<>();

**for** (**int** i = 0; i < administratorView.getLenghtList(); i++) {

aHashSet.add(aRestaurant.getProduct(administratorView.getItemsFromListatIndex(i)));

}

**if** (aHashSet.size() == 0) {

JOptionPane.*showMessageDialog*(**null**, "You have to add an object");

} **else** {

String name = administratorView.getCompositeProductName();

**if** (aRestaurant.getProduct(name) != **null**) {

JOptionPane.*showMessageDialog*(**null**, "the name is already used");

}

**else** {

MenuItems aItems = **new** CompositeProduct(name, aHashSet);

administratorView.addItemToDeleteCombobox(name);

administratorView.addItemToProductListComboBox(name);

waiterView.comboBox.addItem(name);

aRestaurant.addProductToMenu(aItems);

administratorView.DeleteList();

}

}

});

administratorView.addEditBtn(a -> {

String name = administratorView.getCompositeProductName();

**if** (aRestaurant.getProduct(name) == **null**) {

JOptionPane.*showMessageDialog*(**null**, "the product is not in the list");

} **else** {

HashSet<MenuItems> aHashSet = **new** HashSet<>();

**for** (**int** i = 0; i < administratorView.getLenghtList(); i++) {

aHashSet.add(aRestaurant.getProduct(administratorView.getItemsFromListatIndex(i)));

}

**if** (aHashSet.size() == 0) {

JOptionPane.*showMessageDialog*(**null**, "you have to add an element to the list");

}

aRestaurant.deleteProductFromMenu(name);

MenuItems aItems = **new** CompositeProduct(name, aHashSet);

aRestaurant.addProductToMenu(aItems);

}

administratorView.DeleteList();

});

administratorView.deleteBtnActionListener(a -> {

String item = administratorView.getSelectedItemToDelte();

administratorView.deleteComboDelte(item);

administratorView.deleteComboList(item);

waiterView.comboBox.removeItem(item);

aRestaurant.deleteProductFromMenu(item);

});

waiterView.addToListBtn(a -> {

String productName = waiterView.comboBox.getSelectedItem().toString();

System.***out***.println(productName);

waiterView.getModel().addElement(productName);

});

waiterView.addOrdesActionLinster(a -> {

**int** table = waiterView.getTable();

HashSet<MenuItems> ListaProduse = **new** HashSet<>();

**for** (**int** i = 0; i < waiterView.getModel().getSize(); i++) {

ListaProduse.add(aRestaurant.getProduct(waiterView.getModel().getElementAt(i).toString()));

}

aRestaurant.newOrder(table, ListaProduse);

waiterView.getModel().removeAllElements();

});

waiterView.addViewallOrdersBtn(a -> {

OrderView aOrderView = **new** OrderView(aRestaurant.getAllOrders());

aOrderView.setVisible(**true**);

});

aDashboard.AdministratorBtnActionListener(a -> {

administratorView.setVisible(**true**);

waiterView.setVisible(**false**);

aChefView.setVisible(**false**);

});

aDashboard.waiterBtnActionLister(a -> {

waiterView.setVisible(**true**);

administratorView.setVisible(**false**);

aChefView.setVisible(**false**);

});

aDashboard.chefViewActionLister(a -> {

aChefView.setVisible(**true**);

administratorView.setVisible(**false**);

waiterView.setVisible(**false**);

});

}

This class gets the entire application moving. This class contains all the action listeners set on the buttons in the user interface and calls all the methods needed to ensure a proper logical development of the system.

1. Chef View

**public** **class** ChefView **extends** JFrame **implements** Observer{

**public** ChefView() {

**this**.setBounds(100, 100, 284, 252);

getContentPane().setLayout(**null**);

Label label = **new** Label("I m a Chef");

label.setAlignment(Label.***CENTER***);

label.setFont(**new** Font("Arial Black", Font.***PLAIN***, 22));

label.setBounds(43, 10, 158, 110);

getContentPane().add(label);

setLocationRelativeTo(**null**);

}

**public** **void** update(Observable arg0, Object arg1) {

**this**.setVisible(**true**);

showMessage("i got an order");

}

**public** **void** showMessage(String message)

{

JOptionPane.*showMessageDialog*(**null**,message);

}

}

This class represents the View of the chef which display a message each time an waiter make a new order.

1. **Results**

There were no tests done on this Project

1. **Conclusion**

This homework brought a nice view of how to work with design patterns, how to serialize an object, how to deserialize , how to compute the hashcode of an object . After developing this application I have learned to understand and plan ahead a medium complexity application that requires data to be structured appropriately in order to increase readability.

I believe I learned more about this by doing this practical homework than only by reading on the subjects. Further improvements of the code can be done making a cleaner GUI and by “cleaning” the code a little bit.

1. **Biography**

Mostly what was presented in Tema4\_HW4\_indication at the lab .

<https://stackoverflow.com>

<http://www.tutorialspoint.com/java/java_serialization.htm>

<https://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html>