**Documentație**

Tehnici de Programare

Tema 3 : RESTAURANT MANAGEMENT SYSTEM

A realizat: Cozma Victoria,

Grupa 30225, Automatica si Calculatoare

1. Obiectivele lucrarii

Obiectivul lucrarii este simularea unui sistem de management a unui restaurant cu ajutorul limbajului orientat pe obiect Java si a threadurilor. Pasii care au fost urmati pentru atingerea obiectivului au fost:

1. Analiza scenariului si identificarea use-case-ului
2. Identificarea cerintelor functionale si maparea acestora pe module
3. Realizarea unei diagrame de clase
4. Definirea claselor si obiectelor
5. Abstractizarea
6. Testarea lucrarii
7. Analiza problemei, modelare, scenarii, cazuri de utilizare

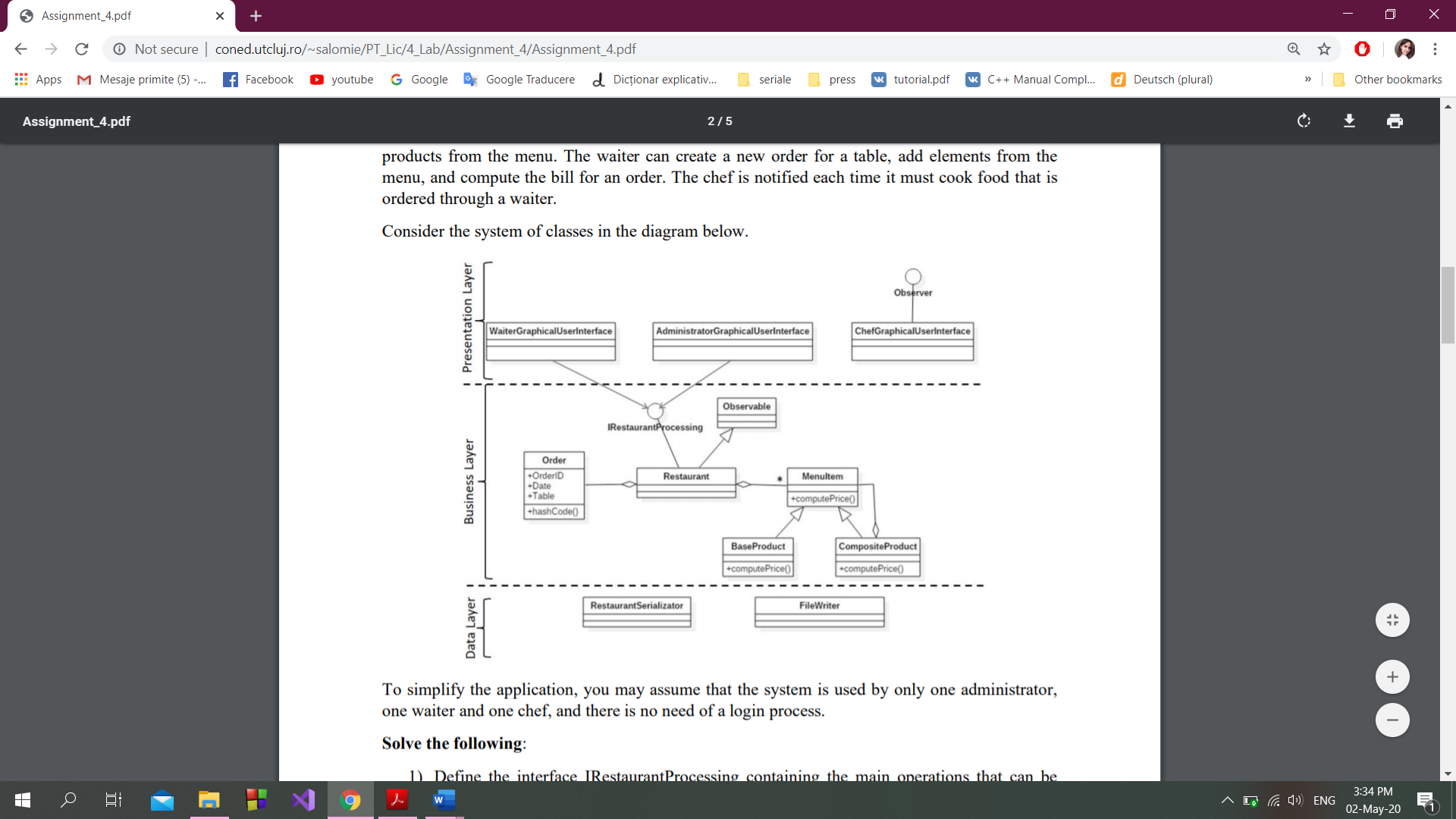
In procesul de realizare a temei am analizat cerintele problemei si am identificat urmatoarele use-case-uri si cerinte fundamentale:

* Alegerea unor structuri de date potrivite pentru stocarea comenzilor si a MenuItem-urilor
* Implementarea operatiilor de insert, delete, update, generate bill
* Implementarea Composite Design Pattern-ului
* Implementare interfetei Observable
* Implementare serializarii si crearea fisierului .ser

Pentru aceasta, am definit clasele generice MenuItem , Order , Restaurant si cele de interfata grafica: WaiterGUI, AdministratorGui de maxim 200 linii cu metodele aferente de maxim 30 din linii. De asemenea, am implementat interfata Observable si IrestaurantProcessing.

1. Proiectare

Prima data, am construit diagrama de clase si relatiile dintre clase. Acesta este un prim pas, conform caruia ne vom ghida in continuare. Diagrama de clase contine 4 clase principale: Restaurant, Order, MenuItem si MainClass. De asemenea, diagrama contine clasele corespunzatoare interfetelor grafice: WaiterGUI, AdministratorGUI si ChefGUI. Pentru a introduce conceptul de serializare in proiect, am creat si clasa RestaurantSerializator, care implementeaza interfata Serializable (deprecated) si are rolul de a anunta chef-ului aparitia unei noi comenzi. Din aceste clase au fost extrase use-case-urile determinate anterior.



1. Implementare

In continuare vom descrie fiecare clasa, cu atributele aferente si metodele de baza.

1. *Clasa Restaurant* – are drept scop simularea managemenului unui restaurant in cadrul caruia exista chlnerul, administratorul si bucatarul. Are atributele ArrayList<MenuItem> menu, folosit cu scop de stocare a produselor din menu si ArrayList<Order> orderList, care stocheaza toate comenzile. Toate atributele au gradul de vizibilitate private. Am folosit fenomenul de incapsulare pentru a limita accesul obiectelor de alta clasa la atributele clasei Restaurant, Astefel, am implementat settere si gettere pentru fiecare atribut.Clasa contine metode pentru adaugarea noilor produse in menu, editare pretului acestora, stergerea item-urilor din menu si informare a bucatarului ca s-a facut o noua comanda.

**import java.util.ArrayList;**

**import java.util.Collection;**

**import java.util.HashMap;**

**import java.util.Iterator;**

**import java.util.Map;**

**import java.util.Observable;**

**import javax.swing.JFrame;**

**import javax.swing.JOptionPane;**

**public class Restaurant extends Observable{**

**private ArrayList<MenuItem> menu = new ArrayList<MenuItem>();**

**// private HashMap<Order, Collection<MenuItem>> orderList = new HashMap<Order,**

**// Collection<MenuItem>>();**

**private ArrayList<Order> orderList = new ArrayList<Order>();**

**public void addItemToMenu(MenuItem m) {**

**menu.add(m);**

**}**

**public void deleteItemToMenu(String name) {**

**Iterator itr = menu.iterator();**

**while (itr.hasNext()) {**

**MenuItem x = (MenuItem) itr.next();**

**if (x.name.equals(name)) {**

**itr.remove();**

**return;**

**}**

**}**

**JOptionPane.showMessageDialog(new JFrame(), "This item doesn't exist");**

**}**

**public void editItemToMenu(String name, String price) {**

**int newPrice = Integer.parseInt(price);**

**Iterator itr = menu.iterator();**

**while (itr.hasNext()) {**

**MenuItem x = (MenuItem) itr.next();**

**if (x.name.equals(name)) {**

**x.price = newPrice;**

**return;**

**}**

**}**

**JOptionPane.showMessageDialog(new JFrame(), "This item doesn't exist");**

**}**

**public int computePrice(String s) {**

**int tableNumber = Integer.parseInt(s);**

**int totalPrice = 0;**

**for (Order i : orderList) {**

**if (i.table == tableNumber) {**

**for (MenuItem j : i.comanda) {**

**totalPrice += j.price;**

**}**

**}**

**}**

**return totalPrice;**

**}**

**public ArrayList<MenuItem> getMenu() {**

**return menu;**

**}**

**public ArrayList<Order> getorderList() {**

**return orderList;**

**}**

**public void addOrder(Order b) {**

**orderList.add(b);**

**notifyObservers(b);**

**}**

**}**

* 1. *Clasa MenuItem* – pentru implementarea acestei clase am folosit Composite Design Pattern-ul. Aceasta clasa are drept atribute name si price. Aceasta clasa poate fi asociata cu produsele din meniu, care pot fi comandate de catre clienti. Aceasta clasa are atributul “abstract”, astfel obiectele de acest tip nu pot fi instantiate. Pentru a rezolva aceasta problema, am creat 2 clase : BaseProduct si CompositeProduct care mostenesc din MenuItem. Design pattern-ul folosit se poate observa in cadrul clasei Composite Pattern. Aceasta mosteneste din MenuItem, dar in acelasi timp contine o lista alcatuita din mai multe MenuItem-uri. Acest tip de mostenire mai poate fi numit si “recursiv”. Toate atributele au fost incapsulate pentru a avea un control mai bun asupra eventualelor erori.

**public** **abstract** **class** MenuItem {

String name;

**int** price;

**public** MenuItem(String s, **int** p) {

name = s;

price = p;

}

**public** MenuItem(String s) {

name = s;

}

}

**public** **class** CompositeProduct **extends** MenuItem{

ArrayList<MenuItem> compProduct = **new** ArrayList<MenuItem>();

**public** CompositeProduct(String s, **int** p) {

**super**(s, p);

}

}

**public** **class** BaseProduct **extends** MenuItem {

**public** BaseProduct(String s, **int** p) {

**super**(s, p);

}

**public** BaseProduct(String s) {

**super**(s);

}

}

1. *Clasa Order*– este responsabila pentru gestionarea tuturor comenzilor din program. Am incercat sa implementez aceasta clasa cat mai user-firendly. Atributele aferente clasei sunt: orderId, date, table si ArrayList<MenuItem> comanda. Vom folosi structura de date ArrayList pentru stocarea denumirii produselor comandata de o anumita masa(specificata de int table).De asemenea, am dat nume cat mai sugestive variabilelor , pentru ca user-ul sa stie instinctiv ce reprezinta fiecare.

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

**int** orderId;

Date date;

**int** table;

ArrayList<MenuItem> comanda = **new** ArrayList<MenuItem>();

**public** Order(**int** id, Date d, **int** t, ArrayList<MenuItem> c) {

orderId = id;

date = d;

table = t;

comanda = c;

}

}

1. *Clasa LogInGUI*

Aceasta clasa este responsabila pentru creare ferestrei de intrare (LogIn) aplicatia noastra Java.Interfata ca contine 2 butoane: “Log in as Waiter” si “Log In as Administrator”. De asemenea, fereastra va contine o imagine de background . In cod vor fi implementate si metodele de AddListener pentru a adauga functionalitate butoanelor.

**public** **class** LogInGUI {

**public** JTextField[][] index = **new** JTextField[9][9];

**public** JFrame mainFrame;

**private** JButton waiterBtn = **new** JButton("Log in as Waiter");

**private** JButton administratorBtn = **new** JButton("Log in as Administrator");

//private JButton chefBtn = new JButton("Log in as Chef");

**public** JPanel board;

**private** JLabel background;

LogInGUI() {

mainFrame = **new** JFrame();

mainFrame.setTitle("My restaurant");

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

mainFrame.setBounds(300, 300, 400, 400);

JPanel butoane = **new** JPanel();

butoane.setLayout(**new** BoxLayout(butoane, BoxLayout.***Y\_AXIS***));

butoane.add(Box.*createRigidArea*(**new** Dimension(0, 100)));

butoane.add(Box.*createRigidArea*(**new** Dimension(200, 0)));

butoane.add(waiterBtn);

butoane.add(Box.*createRigidArea*(**new** Dimension(0, 10)));

butoane.add(administratorBtn);

butoane.add(Box.*createRigidArea*(**new** Dimension(0, 10)));

//butoane.add(chefBtn);

butoane.setOpaque(**false**);

mainFrame.setLocationRelativeTo(**null**);

ImageIcon img = **new** ImageIcon("C:\\Users\\Victoria Cozma\\OneDrive\\Рабочий стол\\restaurant.jpg");

background = **new** JLabel();

background.setIcon(img);

background.setLayout(**new** SpringLayout());

mainFrame.setContentPane(background);

mainFrame.add(butoane, "East");

mainFrame.setResizable(**false**);

mainFrame.setVisible(**true**);

}

**public** **void** setIconImage(Image image) {

// **TODO** Auto-generated method stub

mainFrame.setIconImage(image);

}

**void** addWaiterListener(ActionListener cal) {

waiterBtn.addActionListener(cal);

}

**void** addAdminListener(ActionListener cal) {

administratorBtn.addActionListener(cal);

}

}

1. *Clasele AdministratorGUI si WaiterGUI*

Aceste clase sunt asemanatoare. Ambele raspund de crearea ferestrelor de intersactiune cu user-ul (waiter-ul sau administratorul). Daca la etapa de logare a fost apasat butonul de “Log in as Waiter”, se va deschide fereastra care contine optiunile disponibile chelnerului: New Order, Compute Price, View Orders si Generate Bill. In mod analog, Daca s-a logat administratorul, atunci se va deschide o alta fereastra care va contine urmatoarele optiuni: Add new item, Edit item, delete item, view items. De asemenea, daca se doreste revenirea la fereastra de LogIn, se va pasa butonul “Back”. Am incercat sa implementez acese interfete cat mai user-friendly. In acest scop am adaugat imagini butoanelor, pentru ca user-ul sa-si dea seama la nivel insinctiv ce rol are fiecare buton.

**public** **class** AdministratorGUI **implements** IRestaurantProcessing {

**public** JFrame mainFrame;

**private** ImageIcon addImage = **new** ImageIcon("C:\\Users\\Victoria Cozma\\OneDrive\\Рабочий стол\\add.png");

**private** ImageIcon deleteImage = **new** ImageIcon("C:\\Users\\Victoria Cozma\\OneDrive\\Рабочий стол\\delete.png");

**private** ImageIcon viewImage = **new** ImageIcon("C:\\Users\\Victoria Cozma\\OneDrive\\Рабочий стол\\view.png");

**private** ImageIcon editImage = **new** ImageIcon("C:\\Users\\Victoria Cozma\\OneDrive\\Рабочий стол\\edit.png");

**private** ImageIcon backImage = **new** ImageIcon("C:\\Users\\Victoria Cozma\\OneDrive\\Рабочий стол\\back.png");

**private** JButton addBtn = **new** JButton("Add item", addImage);

**private** JButton deleteBtn = **new** JButton("Delete Item", deleteImage);

**private** JButton viewBtn = **new** JButton("View Items", viewImage);

**private** JButton editBtn = **new** JButton("Edit Item", editImage);

**private** JButton backBtn = **new** JButton("Back", backImage);

**public** JPanel board;

**private** JLabel background;

AdministratorGUI() {

mainFrame = **new** JFrame();

mainFrame.setTitle("My restaurant");

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

mainFrame.setBounds(500, 500, 500, 500);

JPanel butoane = **new** JPanel();

butoane.setLayout(**new** BoxLayout(butoane, BoxLayout.***Y\_AXIS***));

butoane.add(addBtn);

butoane.add(Box.*createRigidArea*(**new** Dimension(0, 5)));

butoane.add(deleteBtn);

butoane.add(Box.*createRigidArea*(**new** Dimension(0, 5)));

butoane.add(editBtn);

butoane.add(Box.*createRigidArea*(**new** Dimension(0, 5)));

butoane.add(viewBtn);

butoane.add(Box.*createRigidArea*(**new** Dimension(0, 50)));

butoane.add(backBtn);

butoane.setOpaque(**false**);

mainFrame.setLocationRelativeTo(**null**);

ImageIcon img = **new** ImageIcon("C:\\Users\\Victoria Cozma\\OneDrive\\Рабочий стол\\restaurant.jpg");

background = **new** JLabel();

background.setIcon(img);

background.setLayout(**new** SpringLayout());

mainFrame.setContentPane(background);

mainFrame.add(butoane, "East");

mainFrame.setResizable(**false**);

mainFrame.setVisible(**true**);

}

**public** **void** setIconImage(Image image) {

// **TODO** Auto-generated method stub

mainFrame.setIconImage(image);

}

**void** addBackListener(ActionListener cal) {

backBtn.addActionListener(cal);

}

**public** **void** addCreateItemListener(ActionListener cal) {

addBtn.addActionListener(cal);

}

**public** **void** deleteMenuItemListener(ActionListener cal) {

deleteBtn.addActionListener(cal);

}

**public** **void** editMenuItemListener(ActionListener cal) {

editBtn.addActionListener(cal);

}

**public** **void** viewMenuItemListener(ActionListener cal) {

viewBtn.addActionListener(cal);

}

@Override

**public** **void** addOrderListener(ActionListener cal) {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** computePriceListener(ActionListener cal) {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** createBillListener(ActionListener cal) {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** viewOrderListener(ActionListener cal) {

// **TODO** Auto-generated method stub

}

}

1. *Clasele AdministratorController si WaiterController*

Aceste clase sunt asemanatoare. Ambele raspund de crearea ferestrelor de interactiune cu user-ul (waiter-ul sau administratorul). Aceste clase au fost introduse pentru a urma pattern-ul Model View Controller.

**public** **class** AdminController {

**private** Restaurant myRest;

**private** RestaurantController resContr;

**public** AdminController(Restaurant a, RestaurantController b) {

myRest = a;

resContr = b;

resContr.createMenuItemListener(**new** CreateMenuItemListener());

resContr.deleteMenuItemListener(**new** DeleteMenuItemListener());

resContr.editMenuItemListener(**new** EditMenuItemListener());

}

**class** CreateMenuItemListener **implements** ActionListener {

**public** **void** actionPerformed(ActionEvent e) {

String prodName = resContr.getNewProductName();

String prodPriceString = resContr.getNewProductPrice();

**int** prodPrice = Integer.*parseInt*(prodPriceString);

MenuItem newMenuItem = **new** BaseProduct(prodName, prodPrice);

myRest.addItemToMenu(newMenuItem);

**for** (MenuItem i : myRest.getMenu()) {

System.***out***.println(i.name + " " + i.price);

}

}

}

**class** DeleteMenuItemListener **implements** ActionListener {

**public** **void** actionPerformed(ActionEvent e) {

String prodName = resContr.getProdToDelete();

myRest.deleteItemToMenu(prodName);

**for** (MenuItem i : myRest.getMenu()) {

System.***out***.println(i.name + " " + i.price);

}

}

}

**class** EditMenuItemListener **implements** ActionListener {

**public** **void** actionPerformed(ActionEvent e) {

String prodName = resContr.getProdToEdit();

String newPrice = resContr.getNewProductNewPrice();

myRest.editItemToMenu(prodName, newPrice);

**for** (MenuItem i : myRest.getMenu()) {

System.***out***.println(i.name + " " + i.price);

}

}

}

}

1. Interfata IRestaurantProcessing

**public** **interface** IRestaurantProcessing {

**void** deleteMenuItemListener(ActionListener cal);

**void** editMenuItemListener(ActionListener cal);

**void** viewMenuItemListener(ActionListener cal);

**void** addCreateItemListener(ActionListener cal);

**void** addOrderListener(ActionListener cal);

**void** computePriceListener(ActionListener cal);

**void** createBillListener(ActionListener cal);

**void** viewOrderListener(ActionListener cal);

}

1. Serializarea proiectului – folosita pentru salvarea datelor si crearea fisierului restaurant.ser

SerializableClass object = **new** SerializableClass();

FileOutputStream file = **new** FileOutputStream("restaurant.ser");

ObjectOutputStream out = **new** ObjectOutputStream(file);

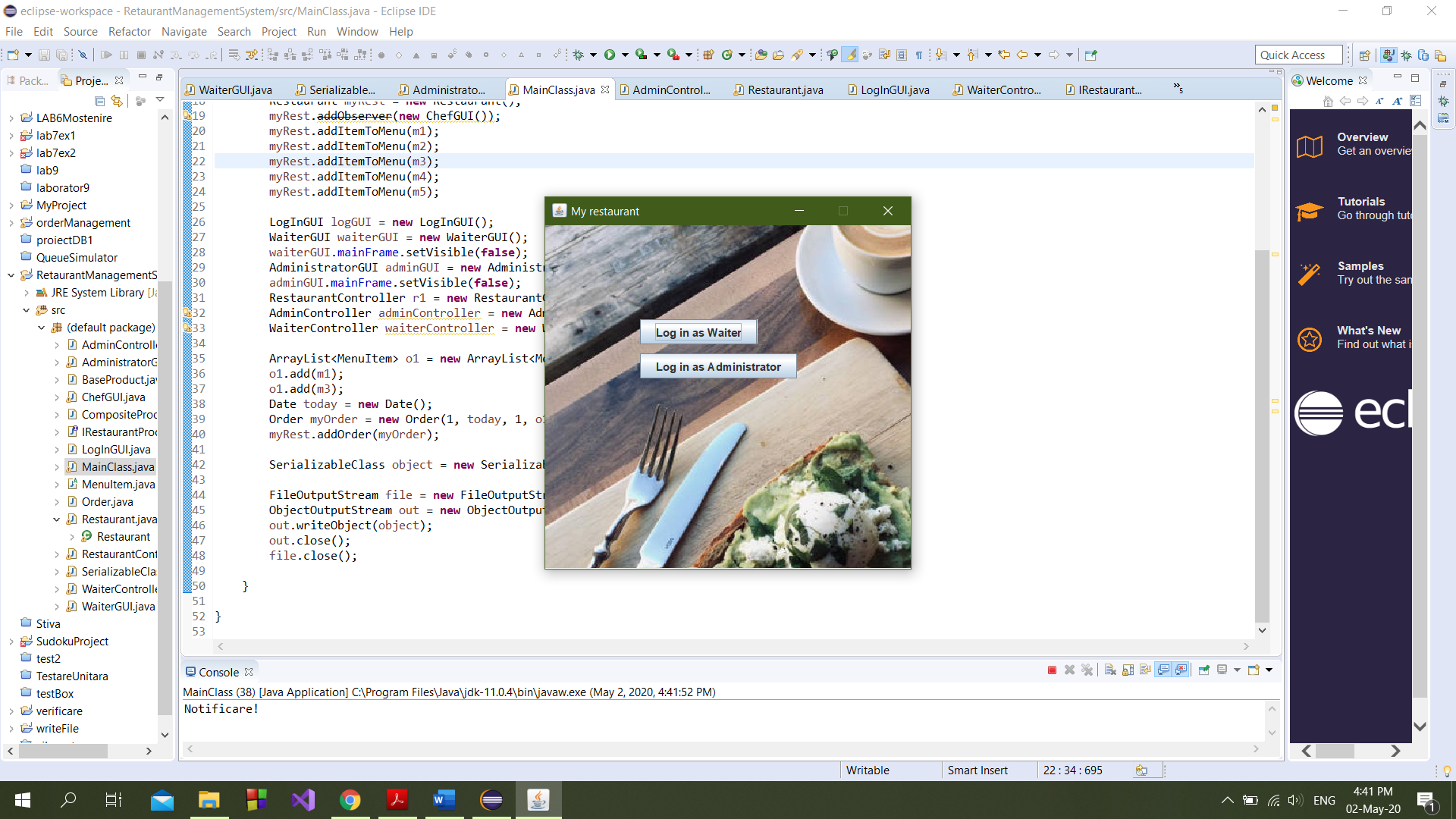
out.writeObject(object);

out.close();

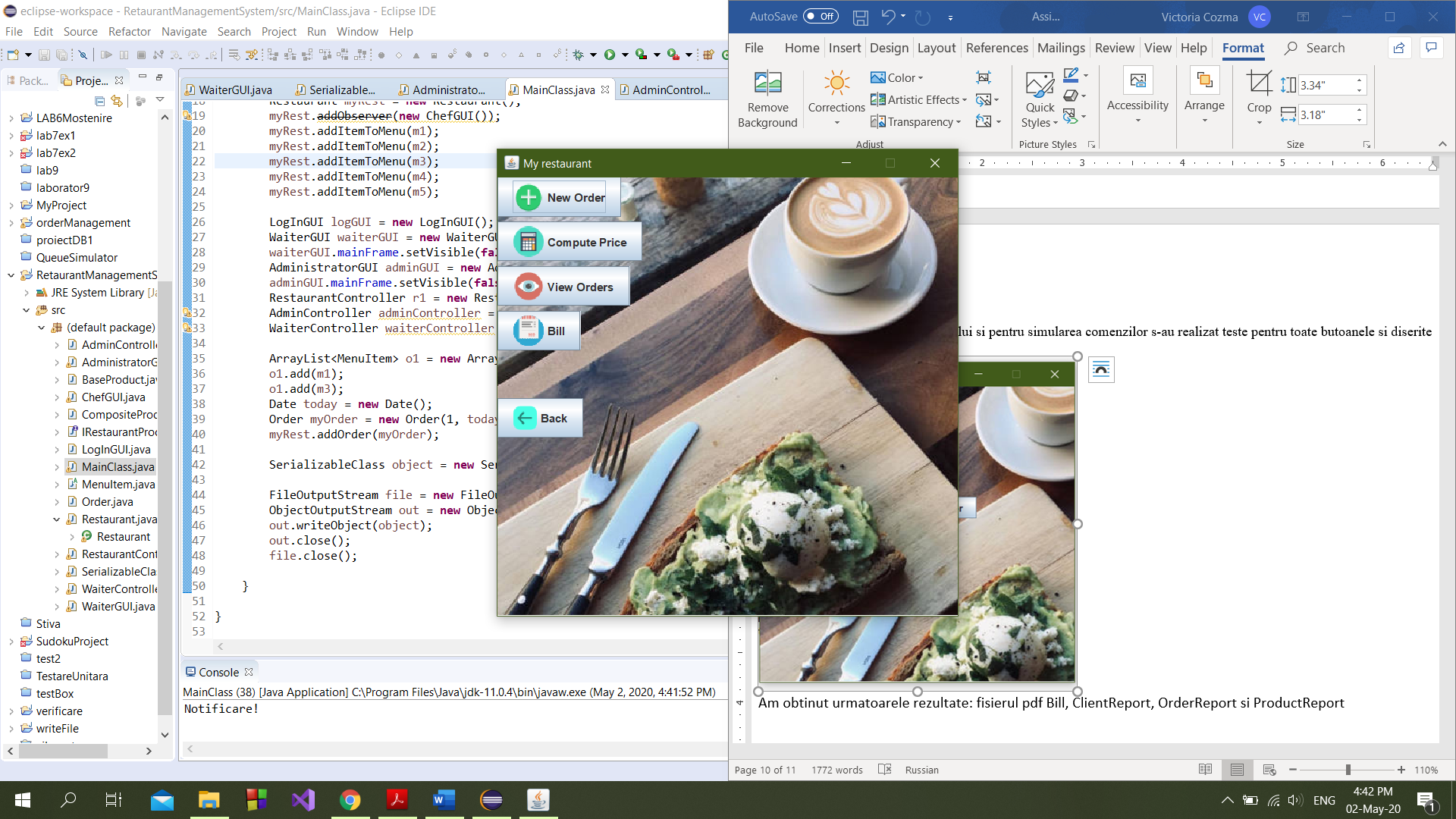
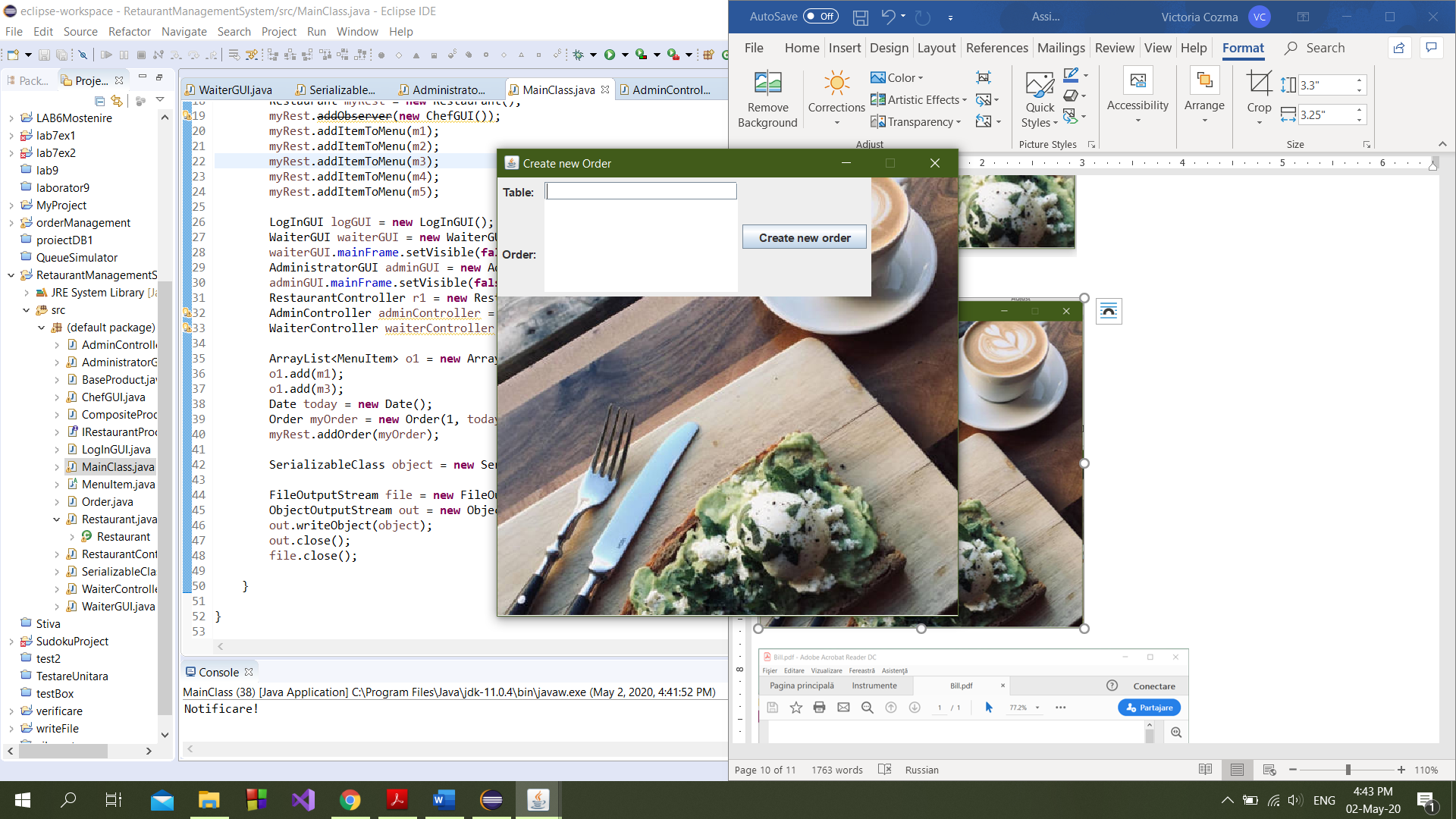
file.close();

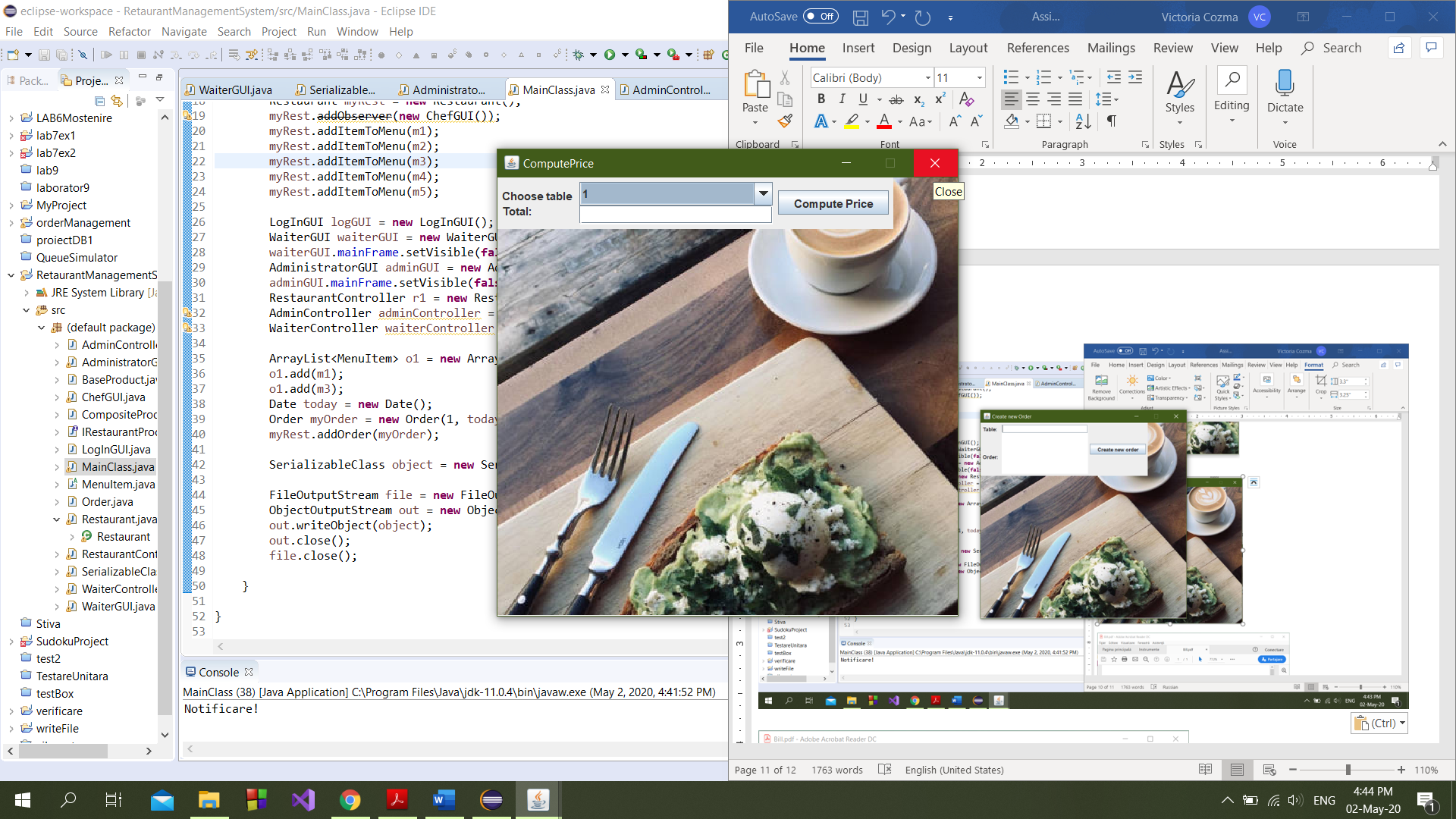
1. Rezultate

Pentru verificarea corectidunii codului si pentru simularea comenzilor s-au realizat teste pentru toate butoanele si diserite variante de rulari.

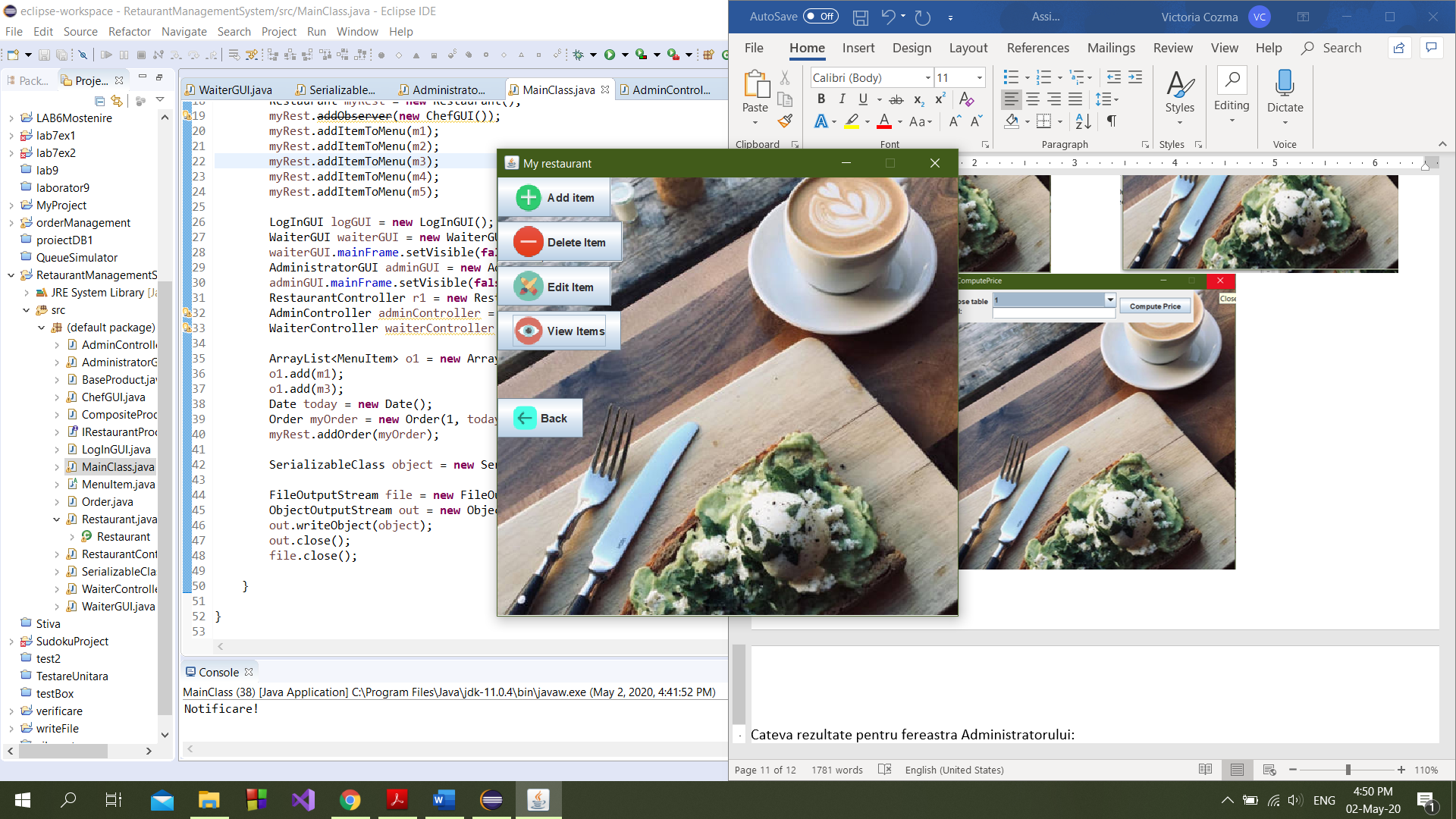
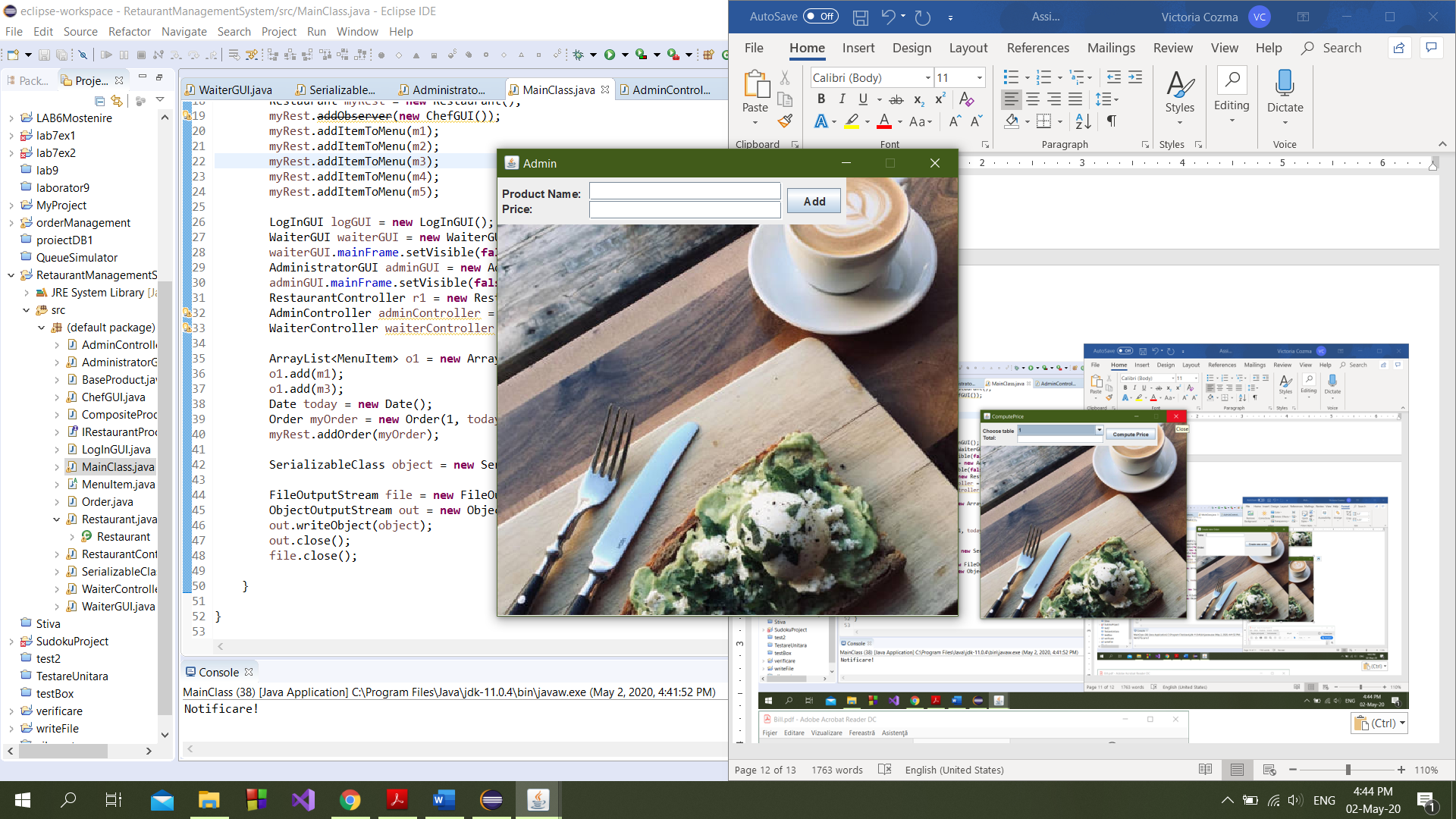


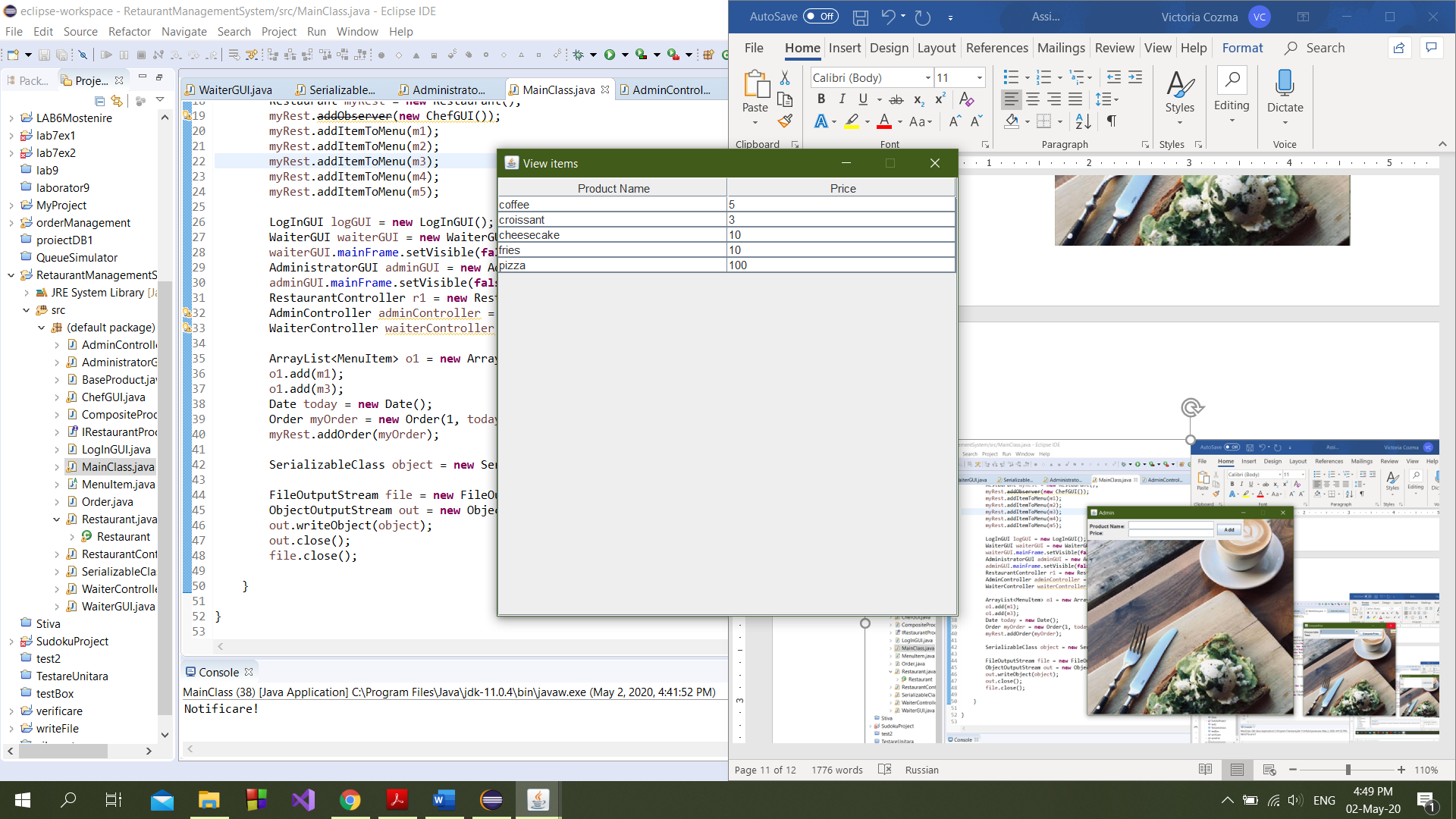
Fereastra Waiter si Fereastra deschisa pentru crearea unei noi comenzi si a calcularii pretului total:

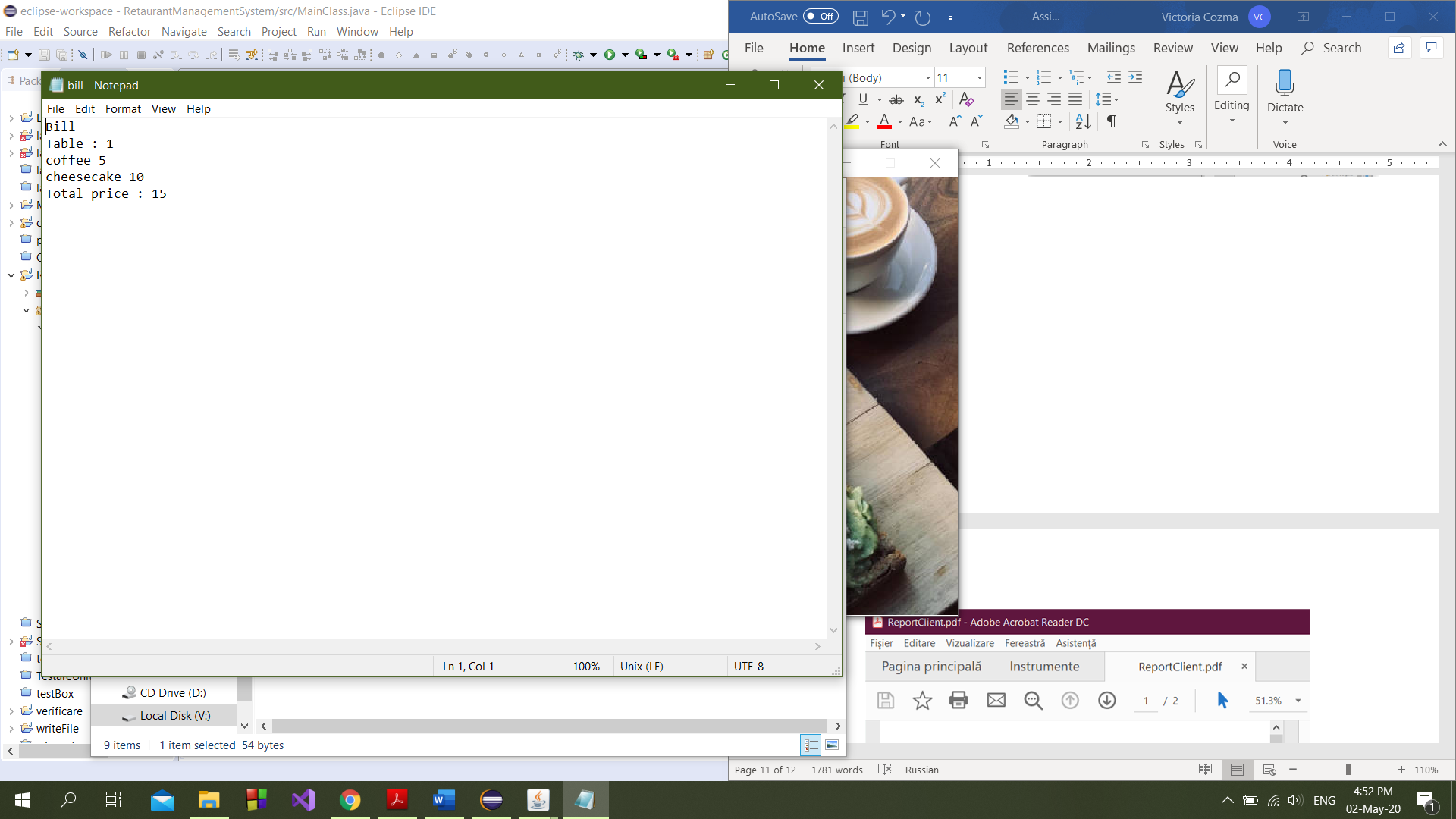


Cateva rezultate pentru fereastra Administratorului:



Crearea notei de plata



1. Concluzii

Obiectivele proiectului au fost atinse. Aceasta tema mi-a consolidat cunostintele de programare orientata pe obiect, intrucat implementarea temei a necesitat structurarea claselor si crearea obiectelor: MenuItem, Restaurant, Order. De asemenea, am invatat sa implementez Composite Design Pattern-ul , ceea ce este o buna tehnica de programare si o voi implementa pe viitor. Am invatat sa folosesc interfata Observable si sa simulez cu ajutorul lor o aplicatie in timp real. Ca posibilitati de dezvoltare ulterioara, ar putea fi implementat un adevarat site al unui restaurant, cu o baza de date si cos de cumparaturi.

1. Bibliografie
2. <http://coned.utcluj.ro/~salomie/PT_Lic/8_Courses/>
3. <http://coned.utcluj.ro/~salomie/PT_Lic/9_Courses/>
4. <https://moodle.cs.utcluj.ro/pluginfile.php/44577/mod_resource/content/3/12-Testarea_Unitara.pdf>
5. <https://moodle.cs.utcluj.ro/pluginfile.php/44568/mod_resource/content/4/POO08.pdf>
6. <http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_4/Java_Concurrency.pdf>