DOKUZ EYLÜL UNIVERSITY

ENGINEERING FACULTY

DEPARTMENT OF COMPUTER ENGINEERING

CME 2210

Object Oriented Analysis and Design

RESTAURANT INFORMATION SYSTEM

By

Sefa Çelik 2020510078

Hasan Balıkçi 2021510012

Ozan Kalkan 2020510044

Yağız Gökdeli 2020510036

CHAPTER ONE

INTRODUCTION

Target of this report is to explain the features of our project "Restaurant Information System" to the lab supervisors of the course CME 2210 Object Oriented Analysis and Design.

Project Description:

The "Restaurant Information System" (will be mentioned as RIS in the following parts of the report). RIS provides a basic management system with some additions (which are going to be mentioned in the following part) for restaurants and also a remote ordering platform for customers of the restaurants. Instead of having a personnel-controlled inventory, RIS uses a software-supported inventory tracker to improve restaurant productivity by reducing employee workload and stress. In addition to that, the software will prevent human error and work more accurately since it will work with exact numbers instead of hypothetical calculations.

Objectives of The Project:

The "Restaurant Information System" (will be mentioned as RIS in the following parts of the report) aims to track and arrange the orders, inventory, staff, and reservations for the restaurants and create a remote system for the customers to place their orders. RIS keeps track of orders and reservations that are placed by customers, inform the restaurant about them and keeps track of used food ingredients after each order and warns the restaurant for the materials that requires a refill. In addition to that, the system stores the fundamental data about the establishment, such as available and busy tables, employees, their jobs and salaries, menu with prices.

The objectives can be summarised as follows:

- Recording and viewing products with their prices as menu, staff of restaurant and their datas, availability of tables of restaurant.
- Inventory tracking and warning for diminishing materials.
- Avoiding running out of stocks.
- A remote reservation and order system for customers.

Project Success Criterias:

- An easily useable interface for both restaurants as customers.
- Precise automatic inventory tracking.
- Information system for management that records fundamental data.
- Recording reservations that are placed by customers.

Technical Requirements:

- JAVA and OOP knowledge.
- File operations.
- Library knowledge for GUI.
- Data structures and algorithms knowledge.

CHAPTER TWO

REQUIREMENTS

In this part, we will try to explain classes and their roles, their attributes, their functions and their methods. Also, we will write details about important functions.

- **Restaurant:** This class will keep all the information about restaurant. For example menu, inventory and staff list. We will create a restaurant object and will get informations from this object when needed.

```
private String password
private Menu restaurantMenu
private List<Inventory> restaurantInventories
private List<Staff> restaurantStaff
private List<Table>  restaurantTables
private PriorityQueue<Order> restaurantOrders
private List<Feedback> feedbacks
private Manager manager
private int totalMoney
```

Attributes

- public Restaurant(String password, Menu restaurantMenu, List<Inventory> restaurantInventories, List<Staff> restaurantStaff, List<Table> restaurantTables, PriorityQueue<Order> restaurantOrders, List<Feedback> feedbacks, Manager manager, int totalMoney)
- set/get methods for these attributes

- **Menu:** This class will get all prodcuts' list of the restaurant and will create a menu. All changes about menu will be controled by this class.

  private List <Product> products } **Attribute**

  - public Menu(List <Product> products)
  - public void updateMenu():
    - *This function grants the ability to operate in the system menu according to the inputs of the user by using the following functions*
  - private double updatePrice(String elementName, int newPrice)
  - private void addNewElement(String elementName, int price, String type, String trait, String contents)
  - private void removeAnElement(String elementName)
  - private Product findProduct(String elementName)
  - private boolean isProductExists(String elementName)
  - private void displayMenu()

- **OrderFactory:**
  - public Order getOrder( )

- **Order <interface>:** Customer's informations, what customer wants from the menu, which table customer wants to reserve and the time of the order will be stored in this class' objects.

  private Customer customer
  private List <Product> productList
  private boolean isOrderReady
  private boolean isForFuture
  private int orderTime
  } **Attributes**

  - public Order(Customer customer, List <Product> productList, boolean isForFuture, int orderTime)
  - set/get methods for these attributes

- **InRestaurantOrder implements Order:** Customer's informations, what customer wants from the menu, which table customer wants to reserve and the time of the order will be stored in this class' objects.

```
private Customer customer
private List <Product>  productList
private boolean isOrderReady
private boolean isForFuture
private int orderTime
private Table orderTable
```

Attributes

  - public InRestaurantOrder(Customer customer, List <Product> productList, Table orderTable, boolean isForFuture, int orderTime)
  - public InRestaurantOrder(Customer customer, List <Product> productList, boolean isForFuture, int orderTime)
  - set/get methods for these attributes

- **AddressDeliveryOrder implements Order:** Customer's informations, what customer wants from the menu, which table customer wants to reserve and the time of the order will be stored in this class' objects.

```
private Customer customer
private List <Product>  productList
private boolean isOrderReady
private boolean isForFuture
private int orderTime
private Address address
```

Attributes

  - public AddressDeliveryOrder(Customer customer, List <Product> productList, boolean isForFuture, int orderTime)
  - public AddressDeliveryOrder (Customer customer, List <Product> productList, boolean isForFuture, int orderTime, Address address)
  - set/get methods for these attributes

- **Address:**

```
private String city
private String district
private String neighborhood
private int aptNo
```

Attributes

  - public Address(String city, String district String neighborhood, int aptNo)
  - set/get methods for these attributes

- **Product <abstract class> :** This abstract class will be the keystone of the menu and superclass of Food and Beverage classes.

private String elementName
private int price
private int productId
private String type
private String trait
private String contents

Attributes

- public Product(String elementName, int price, int productId, String type, String trait, String contents)
- public String getElementName()
- public int getPrice()
- public int getProductId
- public String getType()
- public String getTrait()
- public abstract void display()

- **Food extends Product :** This class contains the informations for the foods which are name, price, type (food/beverage), traits(starter, main course…) and the special Id defined by the restaurant.

private String elementName
private int price
private int productId
private String type
private String trait
private String contents

Attributes

- public Food(String elementName, int price, int productId, String type, String trait, String contents)
- public String getElementName()
- public int getPrice()
- public int getProductId
- public String getType()
- public String getTrait()
- public void display()
- public String getContents()

○ **Beverage extends Product :** This class contains the informations for the beverages which are name, price, type (food/beverage), traits(hot, cold…) and the special Id defined by the restaurant.

private String elementName
private int price
private int productId
private String type
private String trait
private String contents

} **Attributes**

- public Beverage(String elementName, int price, int productId, String type, String trait, String contents)
- public String getElementName()
- public int getPrice()
- public int getProductId
- public String getType()
- public String getTrait()
- public void display()

- **Inventory :** This class keeps an inventory for each material that tracks the increase and decrease as orders are placed by customers. If a material's amount falls below a certain limit the system returns a warning message for restaurant to refill the stock.

private Ingredient ingredient
private int amount
private static int maxAmount
private static int minPercentage

} **Attributes**

- public Inventory(Ingredient ingredient, int amount, int maxAmount, minPercentage)
- public int increaseAmount()
- public int decreaseAmount()
- public int percentageCalculator()
- public void changeMinPercentage(int newPercentage)
- public void changeMaxAmount(int newMaxAmount)
- public void warning()
  - *Returns a warning message if any inventory item falls below min percentage*
- public void display()
- public boolean checkAmount(int requiredAmount)
  - *Checks if the inventory items can provide for the customer's order requirements*

- **Table :**This class will hold tables' id, capacity and if the table is empty or not.

private int tableId
private int tableCapacity
private boolean isEmpty

} **Attributes**

- public Table(int tableId, int tableCapacity, boolean isEmpty)
- set/get methods for these attributes
- public void display()

- **Ingredient:** This class will be the keystone of the inventory and this class' objects will be used by Product objects.

  private String ingredientName
  private String type } **Attributes**

  - public Ingredient(String ingredientName, String type)
  - set/get methods for these attributes

- **Control :** In order to control the organization of the system, each option in the menu calls for related functions in this class which keeps the system in check all the time.

  private Restaurant restaurant } **Attribute**

  - Select Action Methods*: When selected from the user interface, these methods will run according to user's selection and control the selected object (Inventory, Menu, Staff, Order, Table).*
  - Display Control Methods: *Displays the choices that managers can select to operate through the system and update data.*

  - public Control(Restaurant restaurant)
  - public void displayControl()
  - public void selectActionForControl()
  - private void displayControlMenu(Menu restaurantMenu)
  - private void selectActionForMenu()
  - private void displayControlInventory(List<Inventory> restaurantInventories)
  - private void selectActionForInventory()
  - private void displayAllInventory(List<Inventory> restaurantInventories)
  - private void increaseInventoryAmount(String ingredientName)
  - private void decreaseInventoryAmount(String ingredientName)

- private void restaurantInventoriesAdd(Inventory newInventory)
- private void restaurantInventoriesRemove(Inventory inventory)
- private void displayControlStaff (List<Staff> restaurantStaff)
- private void selectActionForStaff ()
- private void restaurantStaffAdd(Staff newStaff)
- private void restaurantStaffRemove (Staff staff)
- private void displayControlTable(List<Table> restaurantTables)
- private void selectActionForTable()
- private void restaurantTablesAdd(Table newTable)
- private void restaurantTablesRemove (Table table)
- private void displayControlOrder(PriorityQueue<Order> restaurantOrders)
- private void selectActionForOrder()
- private void restaurantOrdersAdd(Order newOrder)
- private void restaurantOrdersRemove (Order order)

- **Manager :** This class will hold manager's informations.

private String managerName
private String managerSurname
private int managerAge

} **Attributes**

- public Manager (String managerName, String managerSurname, int managerAge)
- set/get methods for these attributes

- **Person <abstract class> :** This class will be superclass of Customer and Staff and holds informations which are name, surname, age, etc.

private String name
private String surname
private int age
private String gender
private int salary
private String job

Attributes

  - public Person (String name, String surname, int age, String gender, int salary, String job)
  - public abstract void display()
  - set/get methods for these attributes

- **Staff <interface>:** Objects of this class will hold the employee's name, surname, age, occupation, salary, and gender.

private String name
private String surname
private int age
private String gender
private int salary
private String job

Attributes

  - public Staff(String name, String surname, int age, String gender, int salary, String job)
  - public void display()
  - set/get methods for these attributes

- **Chef implements Staff:**

  private String name
  private String surname
  private int age
  private String gender
  private int salary
  private String job

  Attributes

  - public Chef(String name, String surname, int age, String gender, int salary, String job)
  - public void display()
  - public void prepareOrder(Order order)
  - public void listProducts(Order order)
  - public void decreaseInventory(List<Inventory>, Order order)
  - set/get methods for these attributes

- **Server implements Staff:**

  private String name
  private String surname
  private int age
  private String gender
  private int salary
  private String job

  Attributes

  - public Server(String name, String surname, int age, String gender, int salary, String job)
  - public void display()
  - public void takeOrder(Order order)
  - public boolean confirmOrder(Order order)
  - set/get methods for these attributes

- **Courier implements Staff:**

```
private String name
private String surname
private int age
private String gender
private int salary
private String job
```

Attributes

- public Courier (String name, String surname, int age, String gender, int salary, String job)
- public void display()
- public void deliverOrder(Order order)
- set/get methods for these attributes

- **Cashier implements Staff:**

```
private String name
private String surname
private int age
private String gender
private int salary
private String job
```

Attributes

- public Cashier(String name, String surname, int age, String gender, int salary, String job)
- public void display()
- public void takePayment(Payment payment)
- public boolean controlPayment(Payment payment)
- set/get methods for these attributes

- **Customer extends Person :** Objects of this class will hold the customer's name and surname.

  private String name
  private String surname
  private int age
  private String gender

  Attributes

  other person attributes will be assigned null by the system.

  - public Customer(String name, String surname)
  - public Customer(String name, String surname, int age)
  - public Customer(String name, String surname, int age, String gender)
  - public void display()
  - set/get methods for these attributes

- **Feedback :** This class is for receiving feedback from customers and display them on screen.

  private String comment
  private Customer customer
  private Order order

  Attributes

  - public Feedback(String comment, Customer customer, Order order)
  - public void display()

- **FileOperations:** This class will work as a pseudo database. Reading from files and writing the changes on them will be held on that class.

  - public FileOperations()
  - private List<Inventory> createRestaurantInventories()
  - private List<Staff> createRestaurantStaff()

- private List<Table> createRestaurantTables()
- private PriorityQueue<Order> createRestaurantOrders()
- private List<Feedback> createFeedbacks()
- private Manager readRestaurantManagerInfo()
- private Menu createRestaurantMenu()
- public Restaurant createRestaurant ()
- public void writeRestaurantInformations(Restaurant restaurant)
  - *Writes the current data into text files to store the data of the restaurant*

- **Screen:** This class will present two seperate interfaces (one for restaurant and one for customer) and operate in menu according to the key inputs given by users.

  private Restaurant restaurant } **Attribute**

  - public Screen(Restaurant restaurant)
  - public void chooseUser()
  - public boolean getPassword(String password)
  - public void displayScreenForCustomer()
  - public void displayAllTables
  - public void displayEmptyTables()
  - public void getOrder()
  - public void displayFeedbacks(List<Feedback> feedbacks)
  - public void askForFeedback()
  - public void getFeedback()
  - public void displayScreenForManager(Control controlRestaurant)
  - public void showManagerInfo()

- **PaymentFactory:**
  - public Order getOrder( )

- **Payment <interface> :**

  private int cost
  private Order order } **Attributes**

  - public Payment(int cost, Order order)
  - set/get methods for these attributes

- **PayWithCash :**

  private int cost
  private Order order } **Attributes**

  - public PayWithCash(int cost, Order order)
  - set/get methods for these attributes

- **PayWithCard :**

  private int cost
  private Order order
  private int cardNo } **Attributes**

  - public PayWithCard(int cost, Order order, int cardNo)
  - public boolean checkCardNo()
  - set/get methods for these attributes

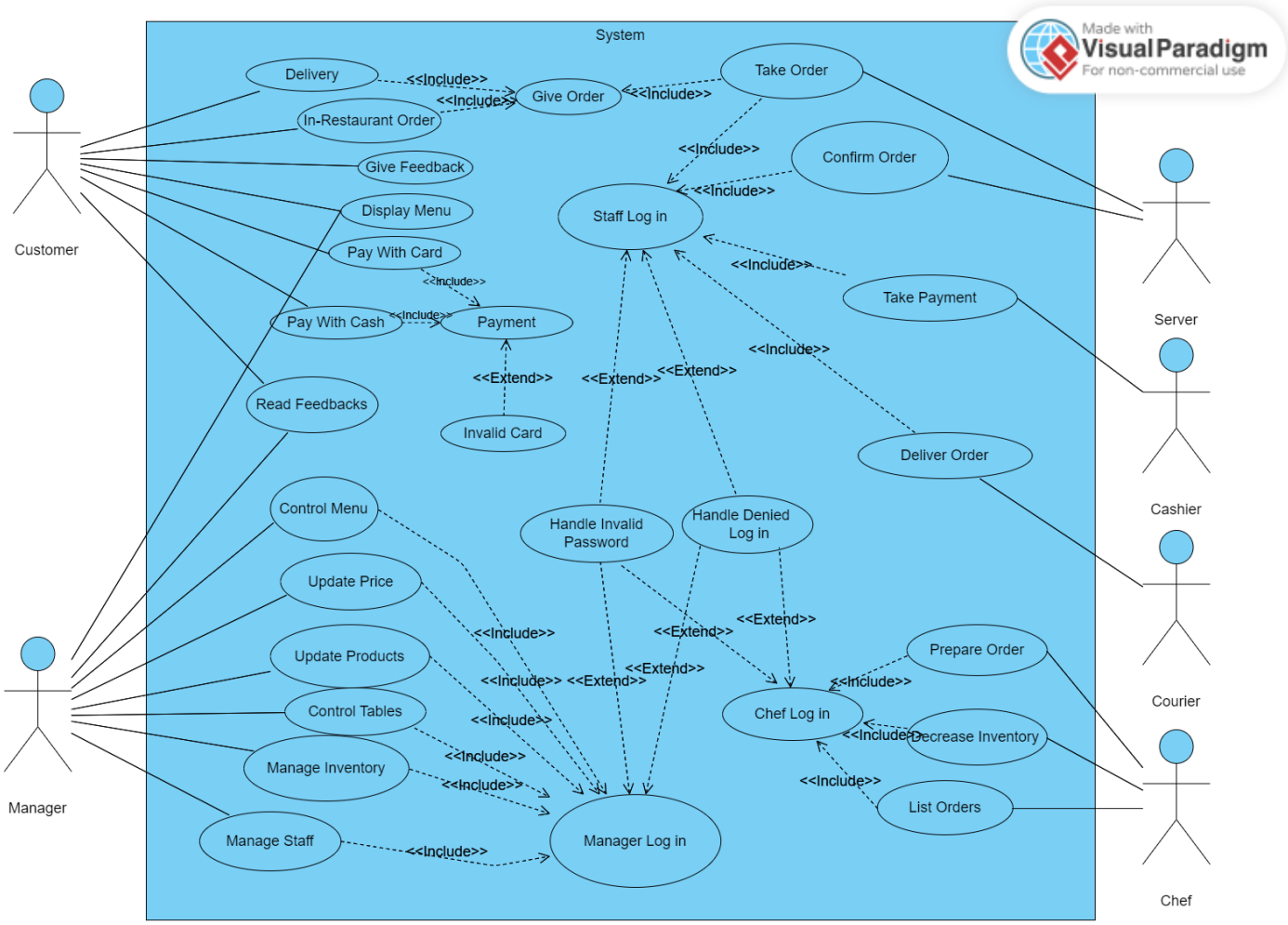
- **Test**
- **List implements ListInterface**
- **PriorityQueue implements PriorityQueueInterface**
- **ListInterface**
- **PriorityQueueInterface**

CHAPTER THREE

UML DIAGRAMS

1 - Use Case Diagram:

Use Case Diagrams are used to show all the functions needed during the management of project, the actors that will trigger these functions, the actors that will be affected by the functions, and the relationships between the functions.



In this use case diagram, there are six actors and there are actions that these actors can access:

1-) Customer: After login, customer enters a menu which offers them different actions to do. In this screen, customer can view restaurant's menu, place a delivery or in restaurant order, choose payment methods, give feedback, and read feedbacks.

2-) Manager: After manager login done, manager views a menu that can monitor and update basically everything in the restaurant. Manager actor is the most extensive actor of all. It can view and make updates on menu, monitor restaurant data, customer feedbacks, manage and update tables, staff, and inventory.

3-) Server: After customer places an order, server takes order, checks if ingredients of that order is enough or not, and if it is enough server confirms order.

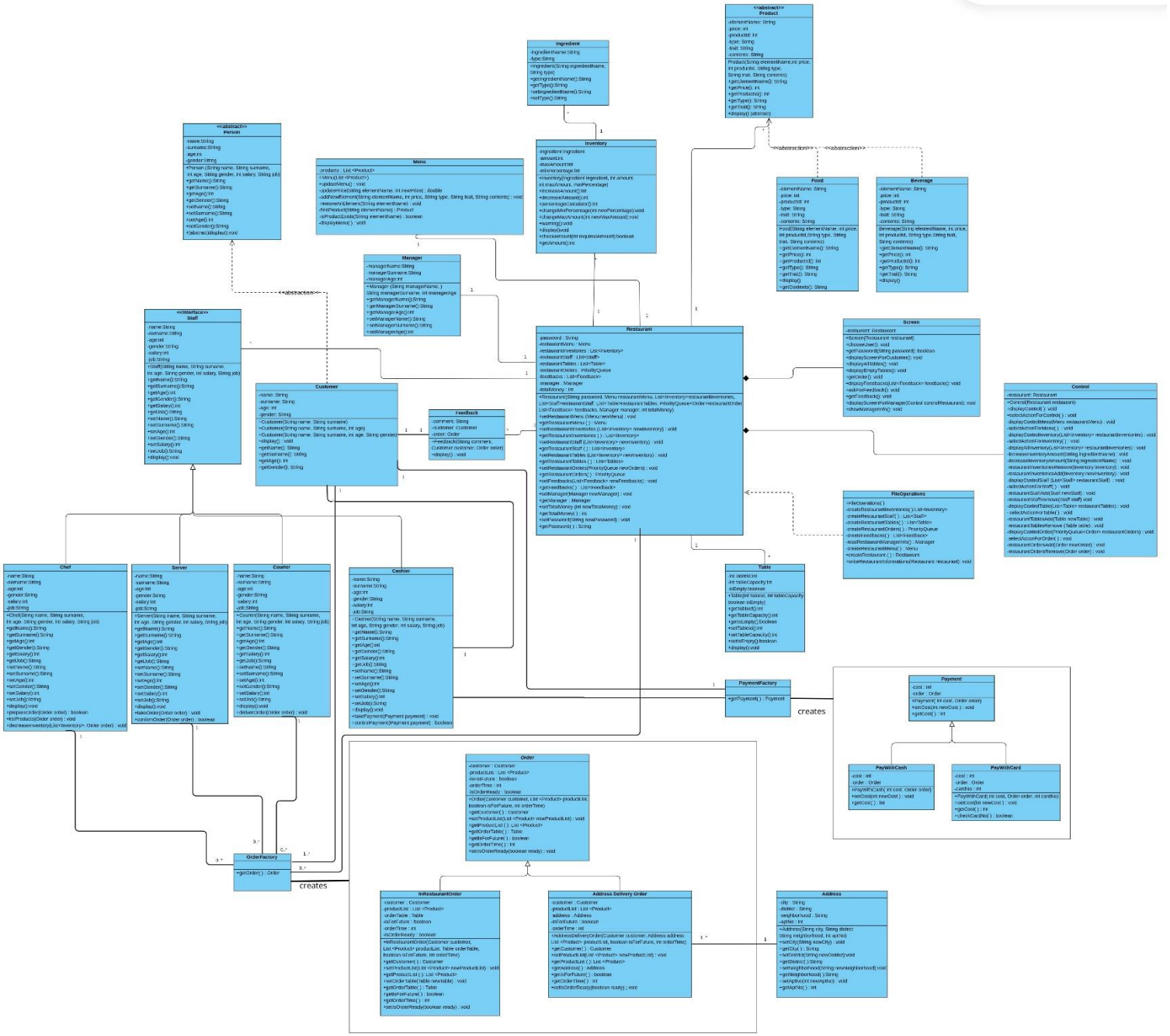
4-) Cashier: Cashier has access to restaurant’s balance. When a customer makes payment, cashier’s duty is to receive the payment and check the validation of the payment.

5-) Courier: If order was placed as a delivery, courier delivers the order to the given address.

6-) Chef: When customer places and order, chef receives the order and prepares it. Chef is able to view all given orders. Meanwhile, chef controls the inventory and decreases the ingredients used in the prepared orders.

2- Class Diagram:

Class diagram is a type of diagram that describes the type of classes and their objects existing in the system and the relationship between them. It is drawn for convenience when creating and implementing the project.



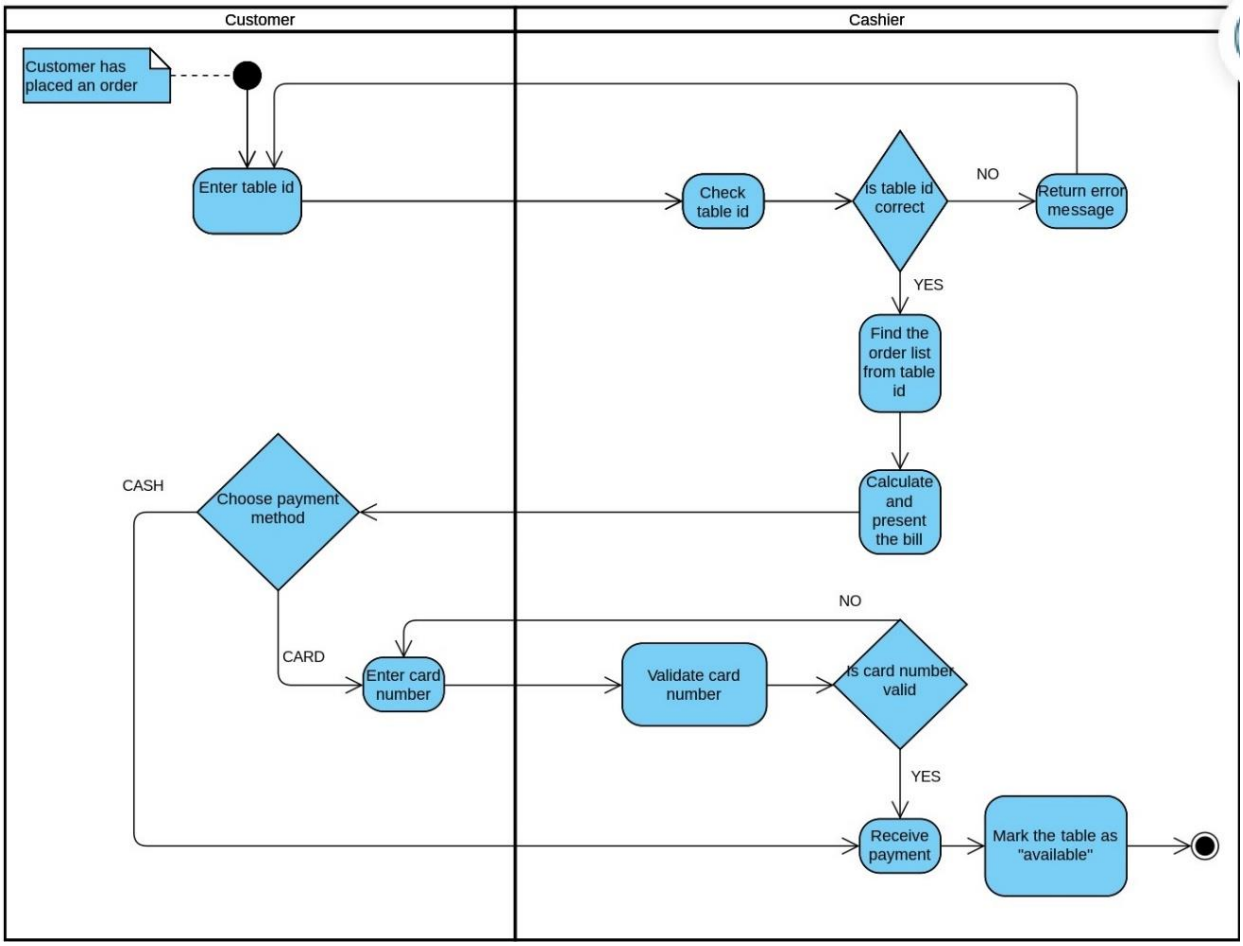
In this class diagram we have 29 classes. The Restaurant class is related with nearly all classes and these classes are as follows: Menu, Manager, Inventory, Product, Order Factory, Table, Screen, Control, File Operation. Customer Class is related with Cashier, Order Factory, and Payment Factory. Order Factory class is related with Chef, Server, Courier, Customer, and Restaurant classes. These three classes form the fundamental structure of the Restaurant Information System.

There are two abstract classes and one interface. Chef, Server, Courier, and Cashier classes implements Staff interface. Food and Beverage classes extends to Product abstract class. Customer class extends to Person abstract class.

Finally, there are two factory classes: Order Factory and Payment Factory. We designed these factory classes so that it will be easier to use when new additions are made in the future.

3- Activity Diagram:

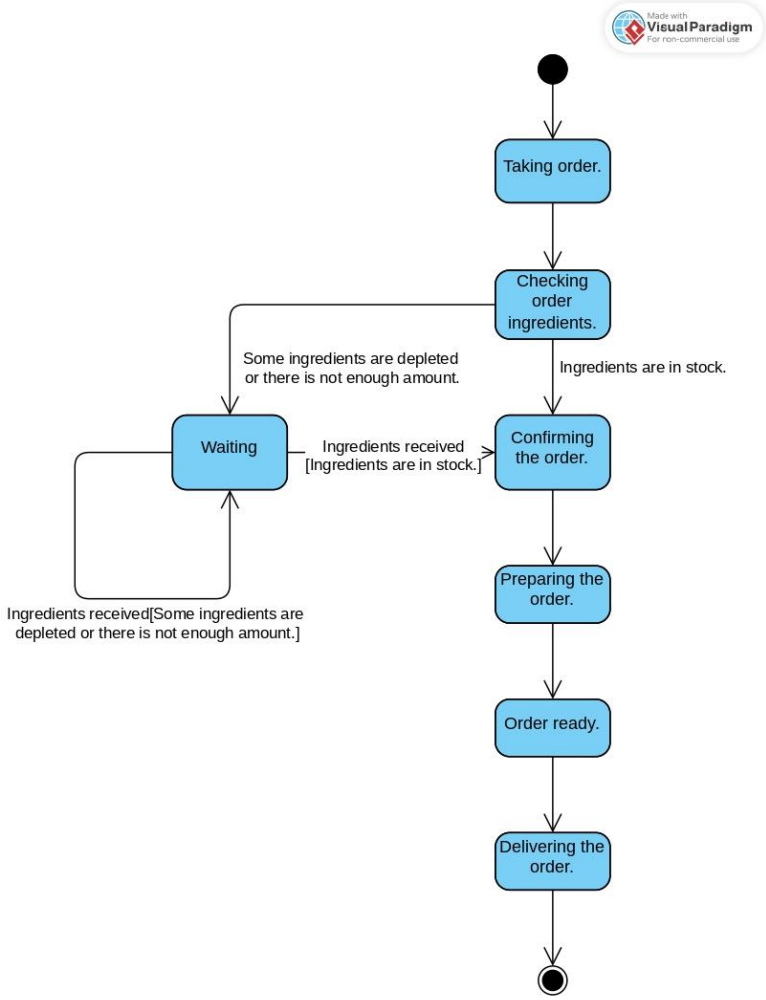
An activity diagram is a type of UML (Unified Modeling Language) diagram that represents the workflow or activity of a system or a process. It is used to visualize the steps and flow of a process or a business workflow.



The precondition for this diagram is that the customer must place an order before. When customer wants to pay his/her bill first he/she should enter his/her table id. Then the system which controlled by cashier checks table id and looks if the table id is correct or not. If it is not correct cashier wants another table id and if it is correct the system finds the orders of that table, calculates, and presents the bill. The payment method must be chosen by the customer. If it is cash cashier's system will receive payment and marks the table as available. If the payment method is chosen as card, first, the system validates the given card number. If it is valid the system receives the payment and marks the table as available. If the card number is not valid, system asks for another card number to be entered from customer.

4- State Diagram:

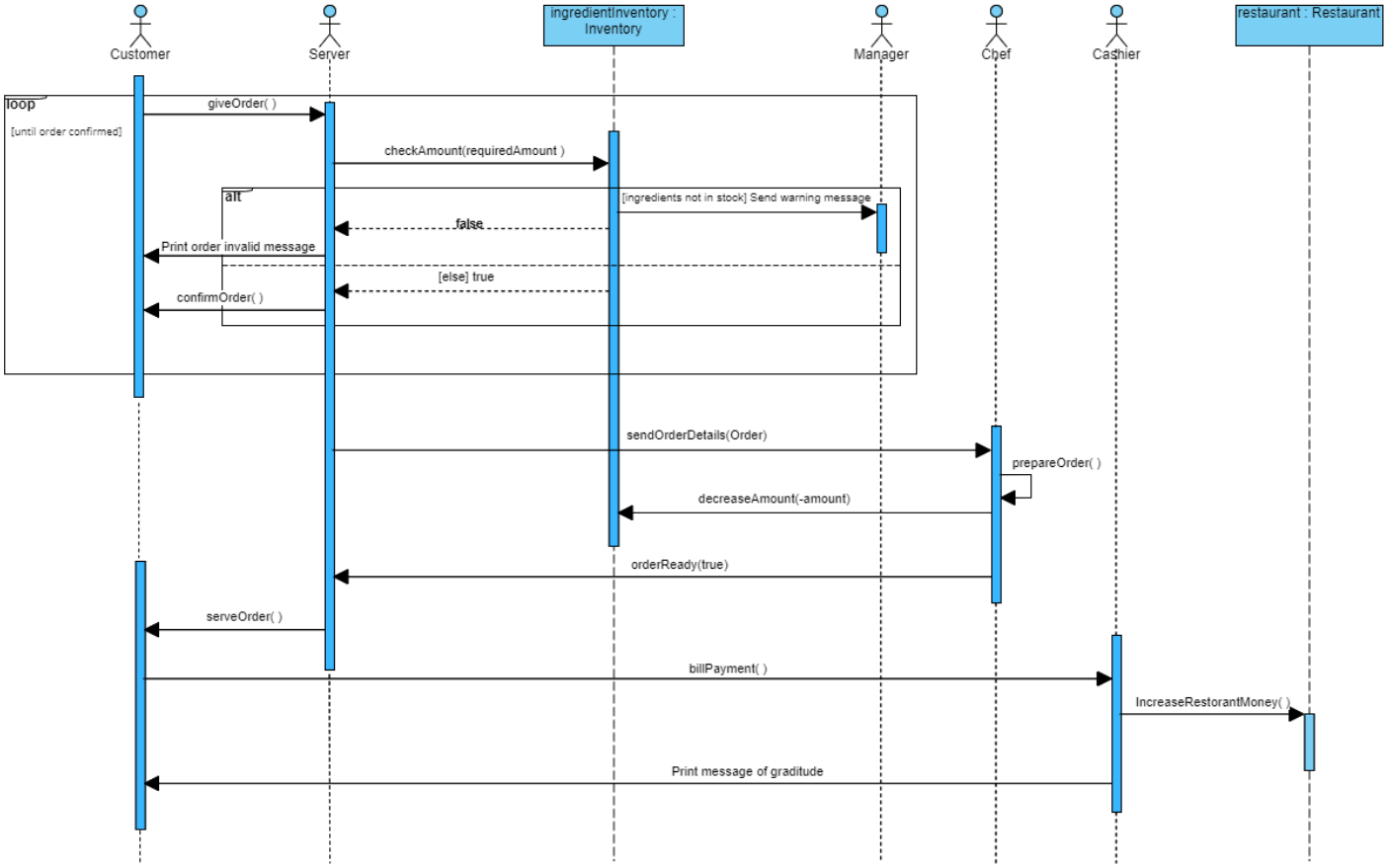
A state diagram is a type of UML (Unified Modeling Language) diagram that models the behavior of a system or an object in response to external stimuli. It is used to visualize the different states that an object can be in and the transitions between those states.



State diagram is used to show the states of an object in the program and the transitions between these states. We made the State diagram based on the situations that the Order object can take in case an order is placed outside the restaurant. The Order object is first created when the order is received, then this object must be controlled according to the ingredients amount in its content. As a result of the control, the Order object can be confirmed or it can wait until the ingredients in it are received and then it is confirmed. Order Object is prepared after confirm and prepared Order Object becomes ready and then delivered.

5-Sequence Diagram:

A sequence diagram is a type of UML (Unified Modeling Language) diagram that models the interactions between objects or components within a system. It is used to visualize the flow of messages and method calls between the different components or objects in a system.

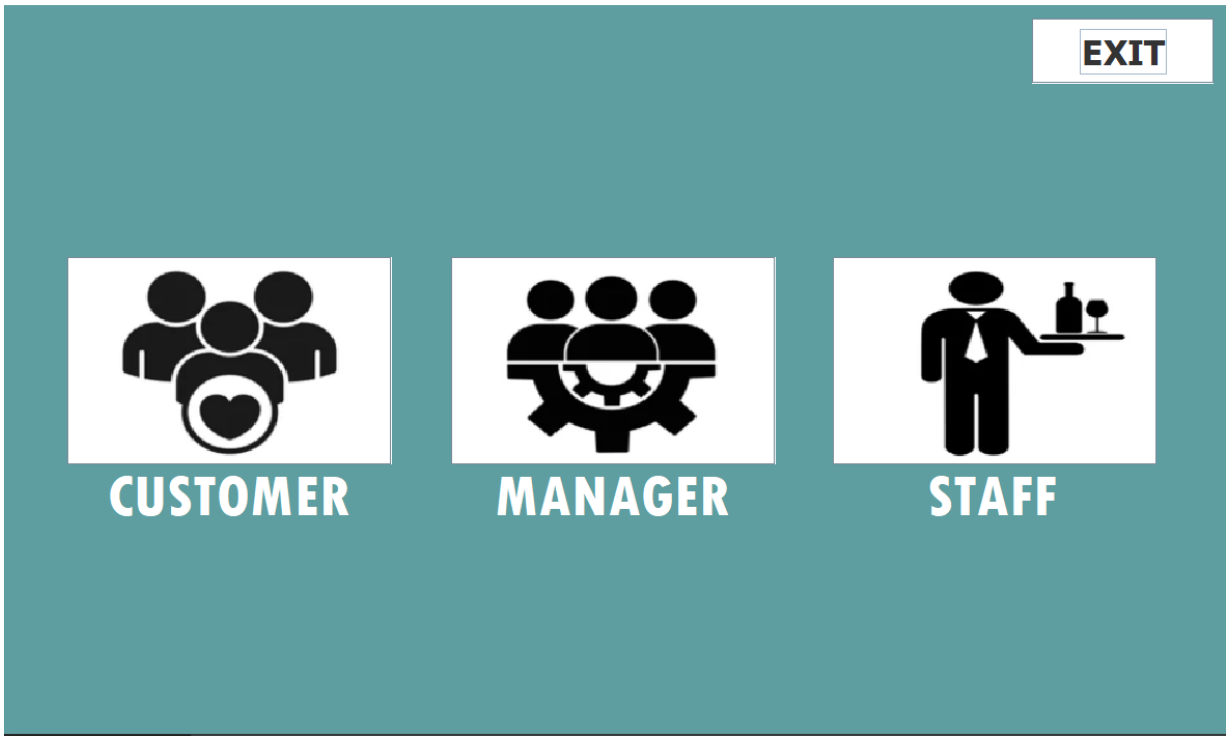


The mechanics of order system is represented in this sequence diagram. When customer places an order, the order goes under some validation checks in server. If any error detected, a warning message is returned. After the order is taken, it is shown on chef's screen and order starts being prepared. Here chef also updates the amount of ingredients used in the order. Then the order is served to the customer, and after that the payment is taken and restaurant's money is increased by cashier.

CHAPTER FOUR

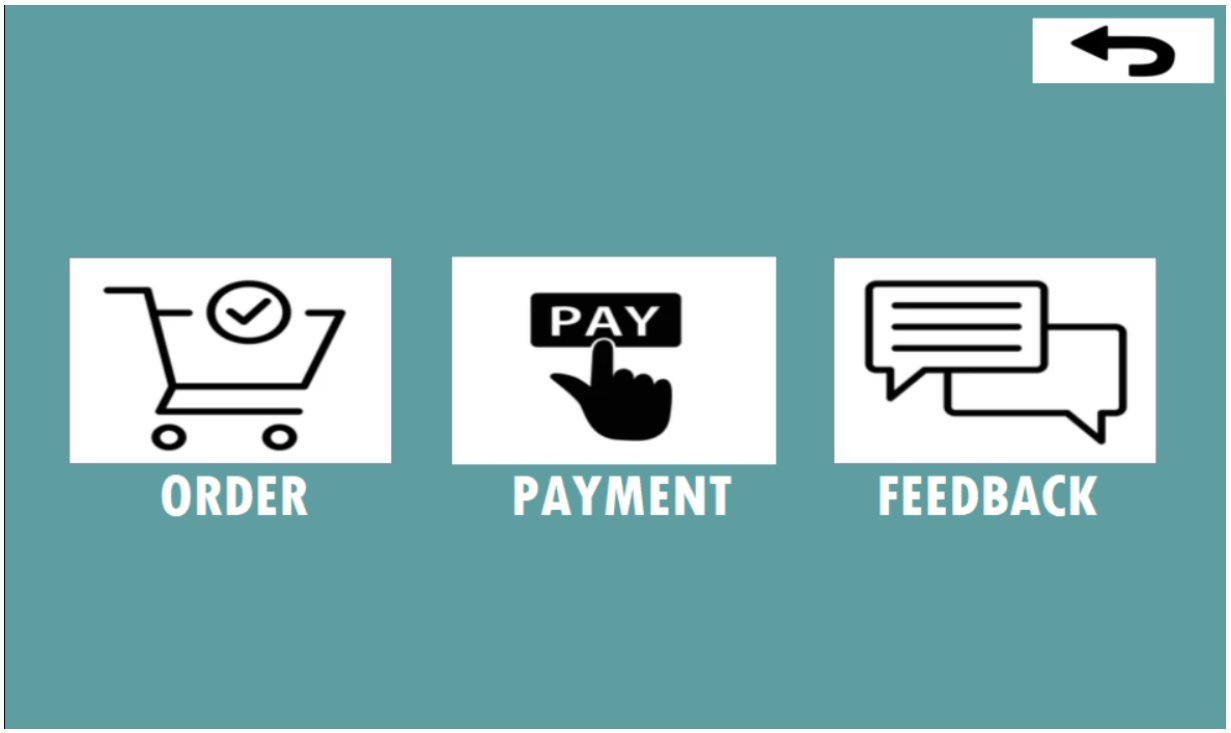
IMPLEMENTATION

This is the opening screen of our GUI. Here you choose who to log in as (customer, manager or staff).

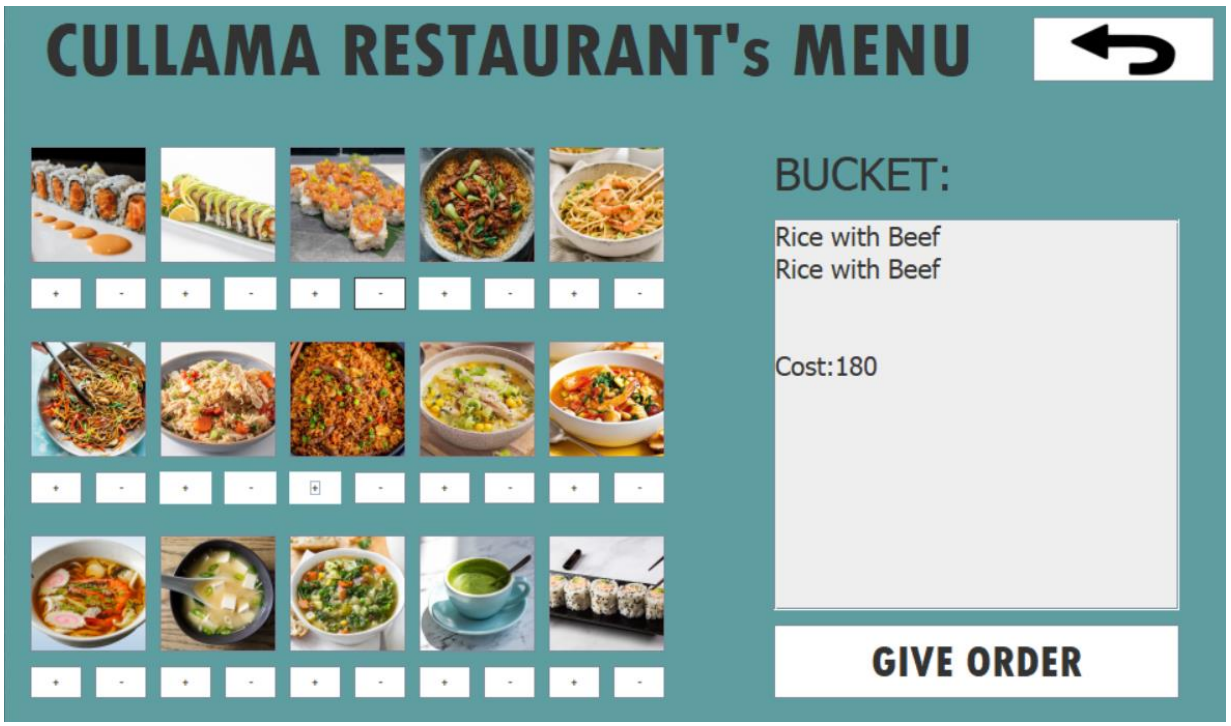


Customer Log In:

After the log in, customer has three options:



The order screen:



For customer's order giving section, the following codes work at the back:

```
food1_increament.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        orderList.add(GUI.cullama.getRestaurantMenu().getMenu().get(0));
        bucketList.setText(writeOrderList(orderList));
    }
});
```

When the items in the menu are selected, the orders are held in an array list and customer's order is added to an orderlist.

```
food1_decreament.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(orderList.contains(GUI.cullama.getRestaurantMenu().getMenu().get(0))) {
            orderList.remove(GUI.cullama.getRestaurantMenu().getMenu().get(0));
            bucketList.setText(writeOrderList(orderList));
        }
    }
});
```

And when clicked on "-" button, the item is deleted from the orderlist.

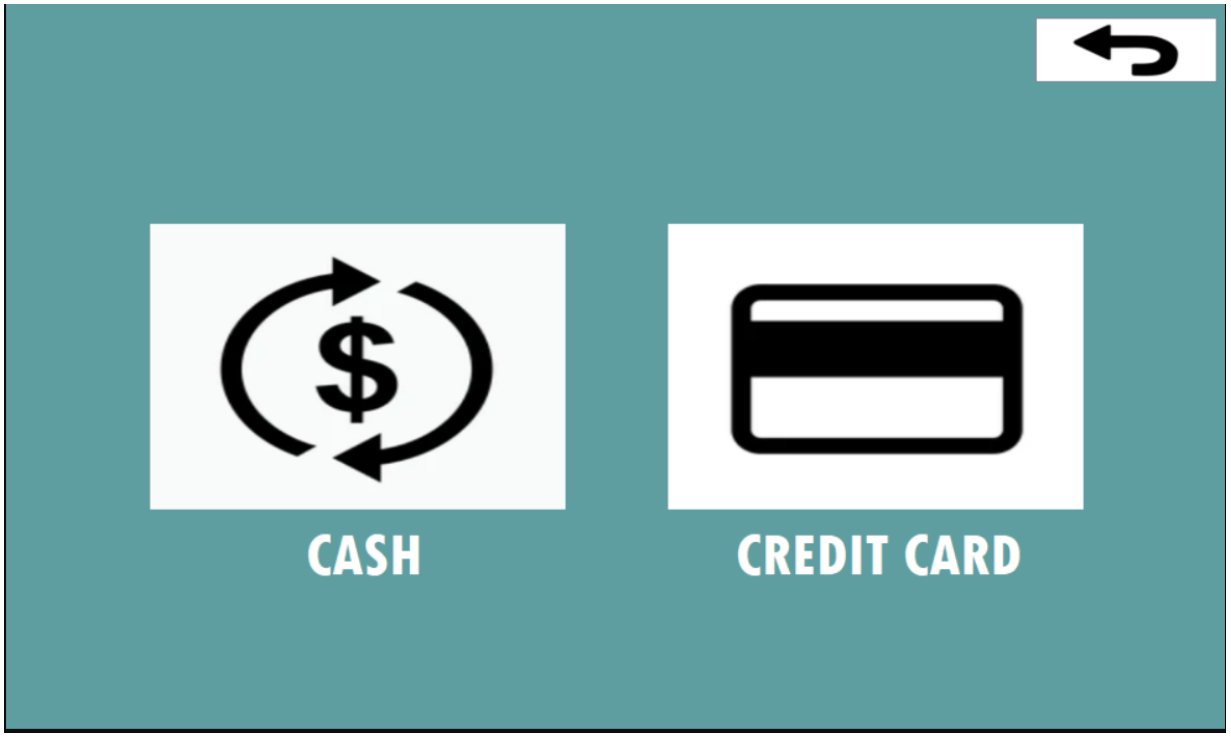
```
giveOrderButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ArrayList<Product> temp = new ArrayList<Product>();
        for (int i = 0; i < orderList.size(); i++) {
            temp.add(orderList.get(i));
        }
        GUI.cullama.addOrder(new Order(GUI.customer, temp));
        GUI.customer.addOrder(new Order(GUI.customer, temp));
        GUI.customerGui.setVisible(true);
        GUI.customer.setOrdersCost(GUI.customer.getOrdersCost());
        setVisible(false);
        bucketList.setText("");
        orderList.clear();
    }
});
```

When pressed on the "give order" button, the code above works, and the orders are added to the restaurant's order list as well.

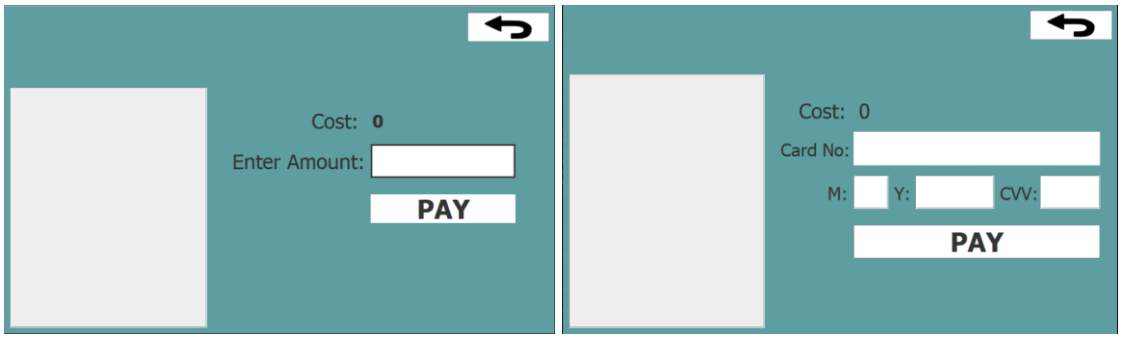
If the order is marked as ready and the payment is still not taken, the following code shows the order's price by adding each orders cost up:

```
public int getOrdersCost() {
    int cost = 0;
    for (int i = 0; i < order.size(); i++) {
        Order ordera = order.get(i);
        if(!ordera.isPaid() && ordera.isOrderReady())
            cost += ordera.getCost();
    }
    return cost;
}
```

Payment screen for customer:



Cash and credit card options:



The code for cash payment screen:

```
payButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (control.checkAmount(amountText.getText())) {
            if(Integer.parseInt(amountText.getText()) < GUI.customer.getTotalCost()) {
                underpaidText.setText("Money is not enough.");
                paidCost += Integer.parseInt(amountText.getText());
                GUI.customer.setOrdersCost(GUI.customer.getTotalCost() - paidCost);
                costInt.setText("" + GUI.customer.getTotalCost());
                GUI.cullama.setTotalMoney(GUI.cullama.getTotalMoney() + paidCost);
            }
            else {
                for(int i = 0; i < GUI.customer.getOrder().size();i++) {
                    if(GUI.customer.getOrder().get(i).isPaid() == false)
                        GUI.customer.getOrder().get(i).setPaid(true);
                }
                GUI.cullama.setTotalMoney(GUI.cullama.getTotalMoney() + Integer.parseInt(amountText.getText()));
                paidCost = 0;
                amountText.setText("");
                GUI.customer.setOrdersCost(0);
                GUI.cullama.getRestaurantTables().get(GUI.customer.getTable().getTableId() - 1).setEmpty(true);
                GUI.cullama.getRestaurantTables().get(GUI.customer.getTable().getTableId() - 1).setCustomer(new Customer("",""));
                setVisible(false);
                GUI.customerGui.setVisible(true);
                underpaidText.setText("");
            }
        }
        else {
            underpaidText.setText("Amount is invalid.");
            amountText.setText("");
        }
    }
});
```

In the pay with cash screen, if the customer's payment is less than total cost the code returns a message "money is not enough" and subtracts the given payment from order's total cost and adds the given payment to the restaurant's total money. In the else block, if the paid amount is equal or larger, the order is marked as paid, the restaurant's money is updated, and the assigned table is set as free.

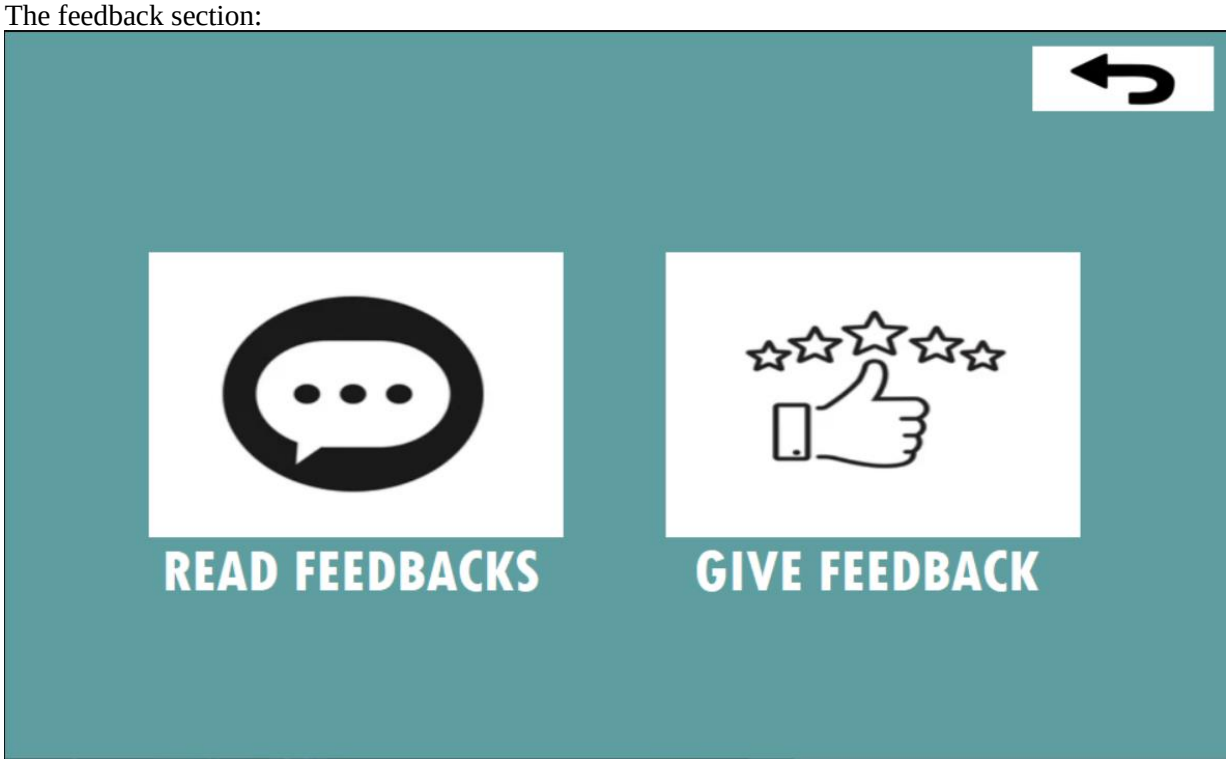
```
customerCreditCardPayButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(control.checkCreditCardNumberFormat(customerCardText.getText(), customerMText.getText()
                , customerYearText.getText(), customerCVVText.getText())) {
            setVisible(false);
            GUI.customerGui.setVisible(true);
            GUI.cullama.setTotalMoney(GUI.cullama.getTotalMoney() + GUI.customer.getTotalCost());
            for(int i = 0; i < GUI.customer.getOrder().size();i++) {
                if(GUI.customer.getOrder().get(i).isPaid() == false)
                    GUI.customer.getOrder().get(i).setPaid(true);
            }

            GUI.customer.setOrdersCost(0);
            GUI.cullama.getRestaurantTables().get(GUI.customer.getTable().getTableId() - 1).setEmpty(true);
            GUI.cullama.getRestaurantTables().get(GUI.customer.getTable().getTableId() - 1).setCustomer(new Customer("",""));
            customerCardText.setText("");
            customerMText.setText("");
            customerYearText.setText("");
            customerCVVText.setText("");
            invalidCard.setText("");
        }
        else {
            invalidCard.setText("Card is invalid.");
            customerCardText.setText("");
            customerMText.setText("");
            customerYearText.setText("");
            customerCVVText.setText("");
        }

    }
});
```

This code block is for validating the inputs given in the pay with card screen and does the same things as the previous code block.

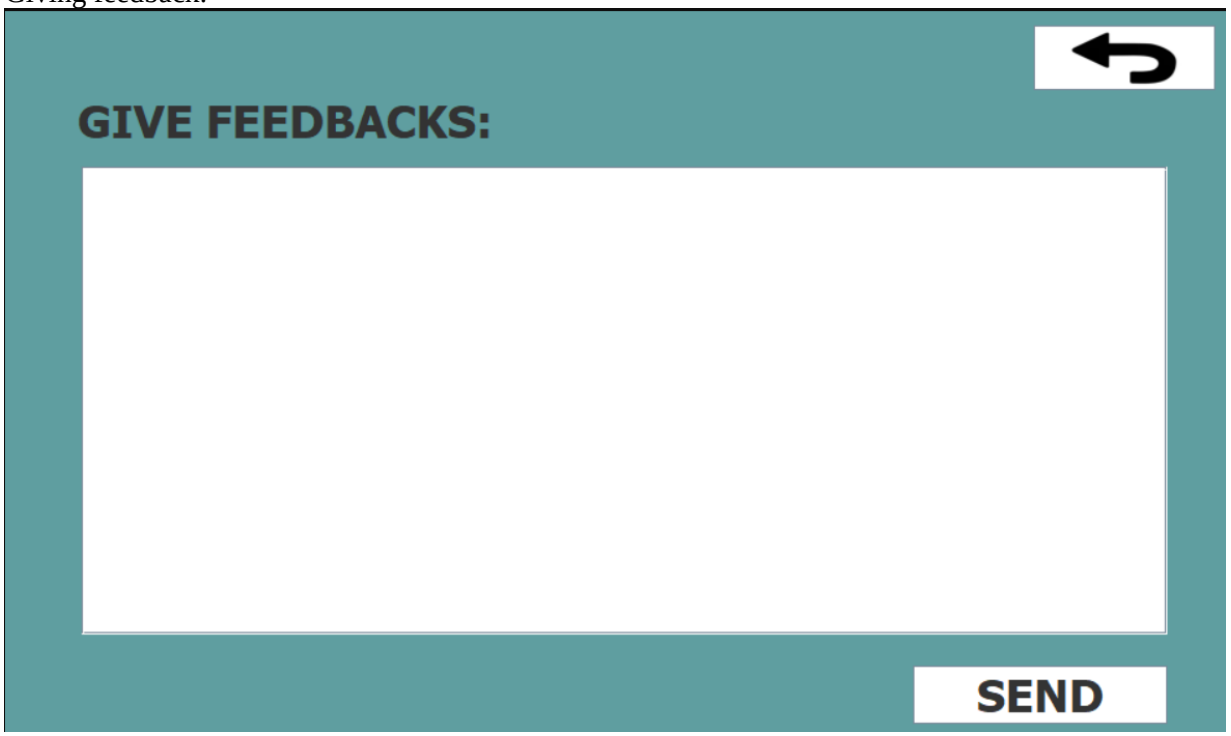
The format check for credit card:

```java
public boolean checkCreditCardNumberFormat(String cardNo, String M, String Y, String Cvv) {
    if (cardNo.length() != 16) {
        return false;
    }

    if (!cardNo.matches("\\d+")) {
        return false;
    }

    int month;
    int year;
    try {
        month = Integer.parseInt(M);
        year = Integer.parseInt(Y);
    } catch (NumberFormatException e) {
        return false;
    }

    if (month < 1 || month > 12) {
        return false;
    }

    int currentYear = Calendar.getInstance().get(Calendar.YEAR) % 100;
    if (year < currentYear || (year == currentYear && month < Calendar.getInstance().get(Calendar.MONTH) + 1)) {
        return false;
    }

    if (Cvv.length() != 3) {
        return false;
    }

    if (!Cvv.matches("\\d+")) {
        return false;
    }

    return true;
```

The feedback section:



Giving feedback:



Read feedback section:

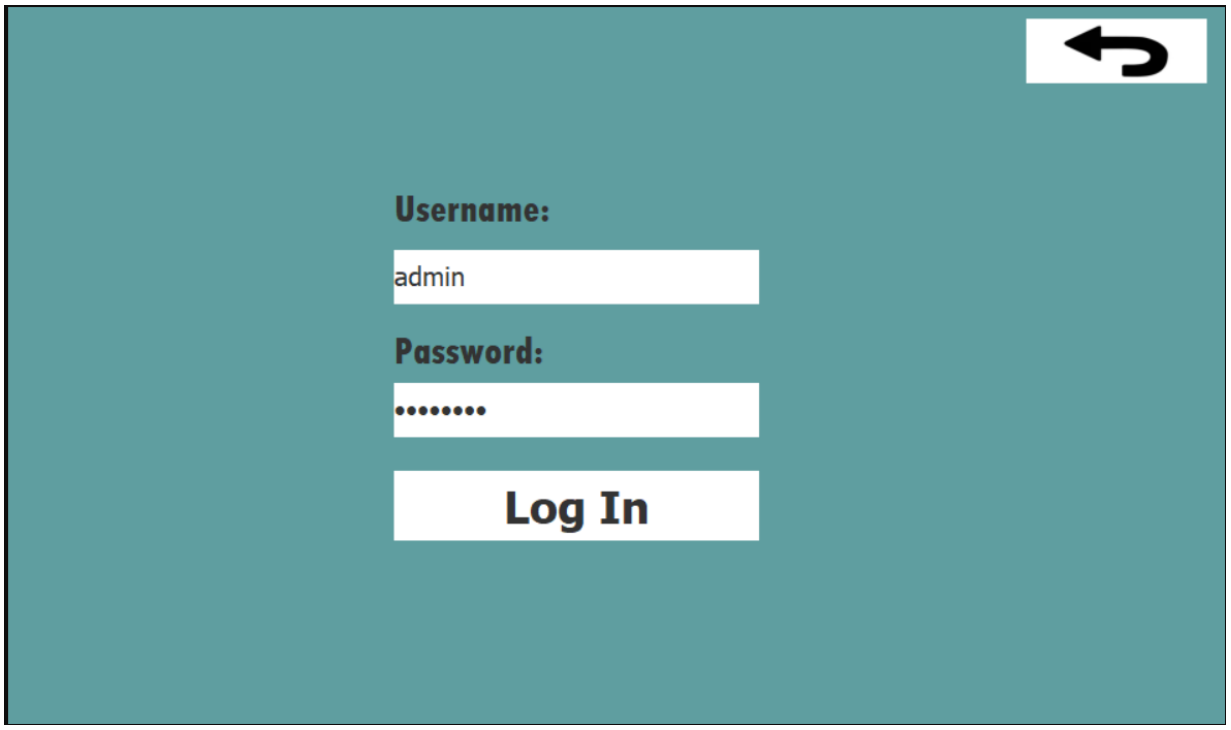
This code works when the give feedback button is pressed and calls the GUI to take a feedback:

```
customerGiveFeedbackButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        GUI.cullama.addFeedbacks(new Feedback(GUI.customer, textPane.getText()));
        setVisible(false);
        GUI.customerFeedbackGui.setVisible(true);
        textPane.setText("");
    }
});
```

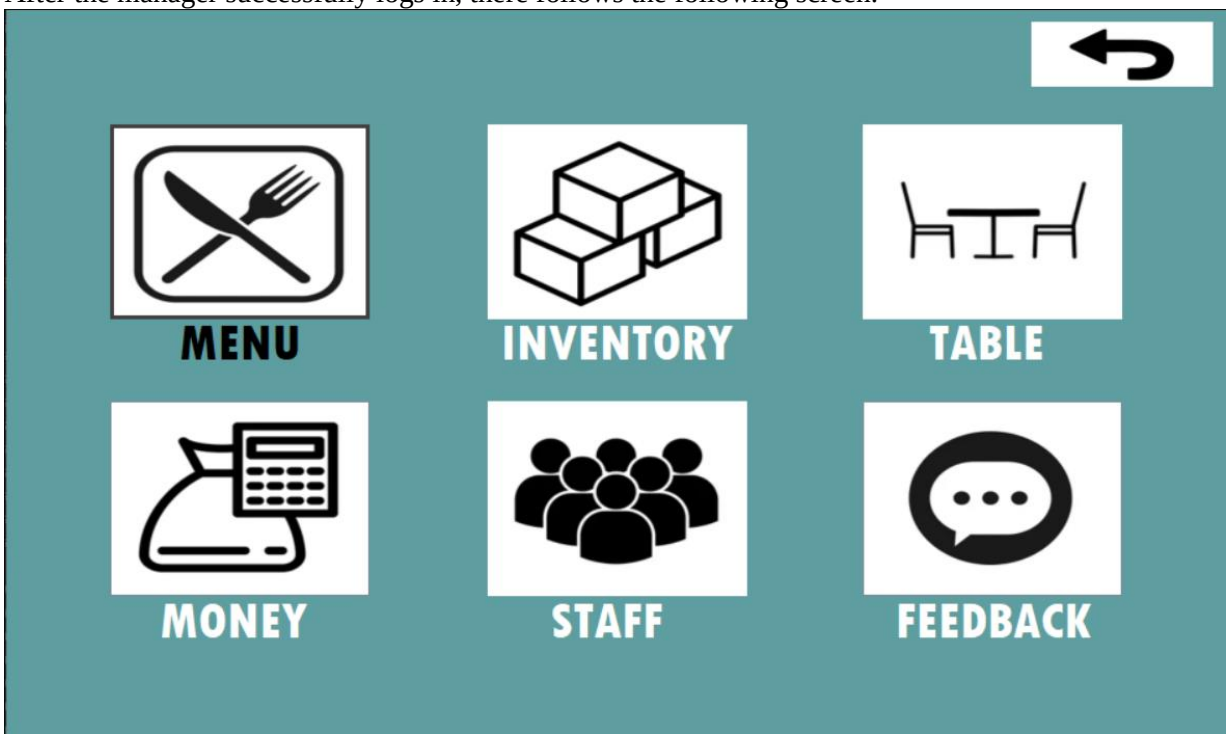
This code works when read feedback button is pressed and displays the given feedbacks. If there is no feedback given, it prints “There is no comment yet”

```
customerReadFeedbackButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        GUI.customerReadFeedbackGui.setVisible(true);
        if(GUI.cullama.getFeedbacks().size() != 0)
            GUI.customerReadFeedbackGui.customerReadFeedbackText.setText(GUI.cullama.getFeedbackString());
        else
            GUI.customerReadFeedbackGui.customerReadFeedbackText.setText("There is no comment yet.");
        setVisible(false);
    }
});
```

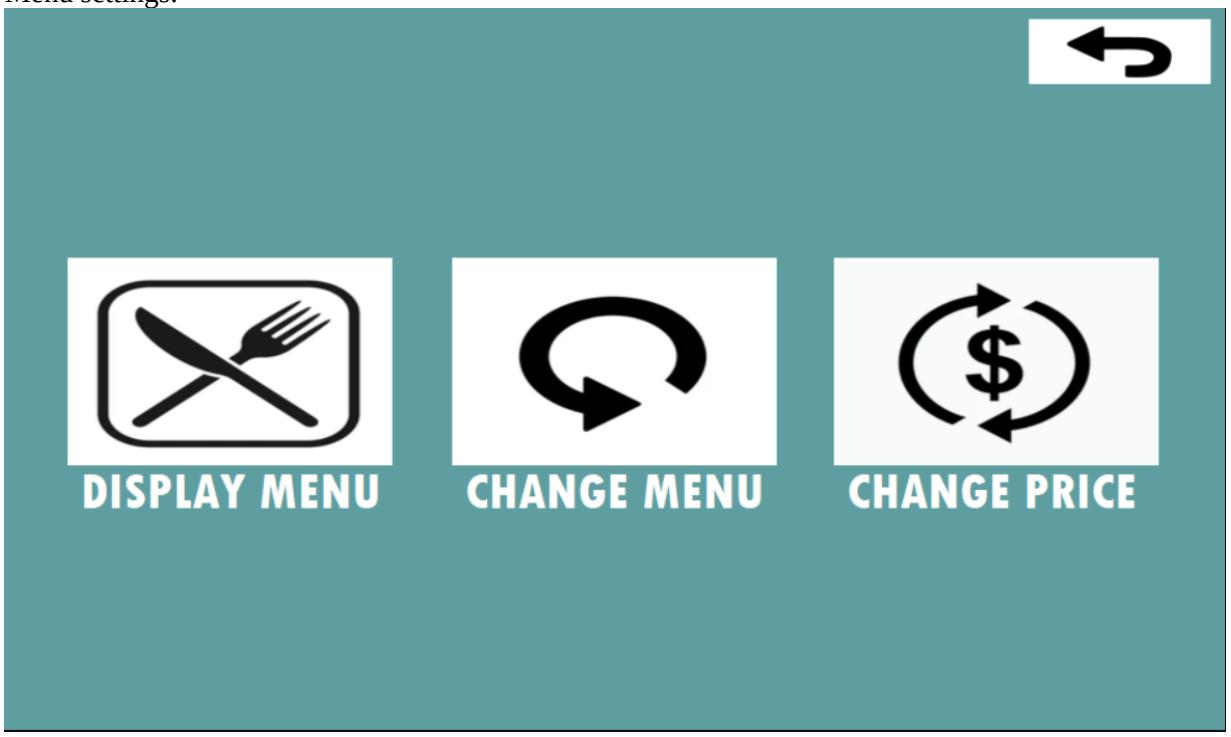
Manager Log In:



After the manager successfully logs in, there follows the following screen:



Menu settings:



Manager can display the current menu of the restaurant:

And also change the items on it:

This code block adds the items to the restaurant's menu if the checkbox on the left of the item is checked.

```
JCheckBox food11 = new JCheckBox("11- " + GUI.cullama.getRestaurantMenu().getProducts().get(10).getElementName());
if (GUI.cullama.getRestaurantMenu().returnExist(GUI.cullama.getRestaurantMenu().getProducts().get(10))) {
    food11.setSelected(GUI.cullama.getRestaurantMenu().returnExist(GUI.cullama.getRestaurantMenu().getProducts().get(10)));

}
food11.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseEntered(MouseEvent e) {
        food11.setBorder(BorderFactory.createLineBorder(Color.DARK_GRAY, 5));
        food11.setForeground(Color.WHITE);
    }
    @Override
    public void mouseExited(MouseEvent e) {
        food11.setBorder(null);
        food11.setForeground(Color.BLACK);
    }
    @Override
    public void mouseClicked(MouseEvent e) {
        if(food11.isSelected()) {
            GUI.cullama.getRestaurantMenu().addToMenu(GUI.cullama.getRestaurantMenu().getProducts().get(10));
            checkedFood++;
            checkedFoodNumber.setText("Checked: " + checkedFood);

        }
        else {
            GUI.cullama.getRestaurantMenu().deleteFromMenu(GUI.cullama.getRestaurantMenu().getProducts().get(10));
            checkedFood--;
            checkedFoodNumber.setText("Checked: " + checkedFood);
        }
    }
});
```

The following code block updates the menu according to the checked items on the update menu:

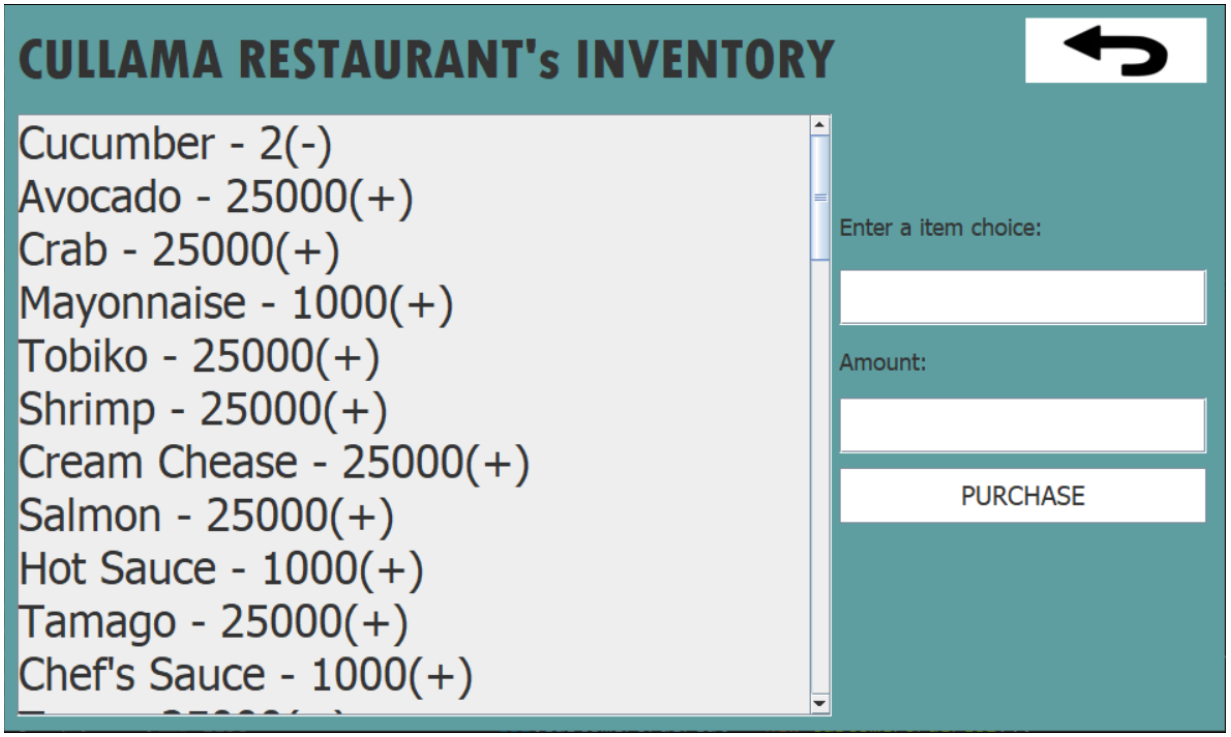
```
managerChangeMenuButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(checkedFood != 15) {
            checkedFalse.setText("Menu must include exactly 15 product.");
        }
        else {
            FileOperations operators = new FileOperations();
            operators.writeRestaurantStaff(GUI.cullama);
            GUI.managerDisplayMenu.managerShowMenuText.setText(GUI.cullama.getRestaurantMenu().getMenuString());
            checkedFalse.setText("");
            GUI.managerMenu.setVisible(true);
            GUI.customerOrderGui = new CustomerOrderGUI();
            setVisible(false);
        }
    }
});
```

Updating prices:

The following code block gets the updated prices and assigns them to the products after checking if they are integer or not in the priceControl function.

```
managerChangePriceButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String[] array = {food1Price.getText(),food2Price.getText(),food3Price.getText(),food4Price.getText(),food5Price.getText(),
                food6Price.getText(),food7Price.getText(),food8Price.getText(),food9Price.getText(),food10Price.getText(),
                food11Price.getText(),food12Price.getText(),food13Price.getText(),food14Price.getText(),food15Price.getText(),
                food16Price.getText(),food17Price.getText(),food18Price.getText(),food19Price.getText(),food20Price.getText(),
                food21Price.getText(),food22Price.getText(),food23Price.getText(),food24Price.getText(),food25Price.getText(),
                food26Price.getText(),food27Price.getText(),food28Price.getText(),food29Price.getText(),food30Price.getText()};
        if(control.priceControl(array)) {
            GUI.cullama.getRestaurantMenu().getProducts().get(0).setPrice(Integer.parseInt(food1Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(1).setPrice(Integer.parseInt(food2Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(2).setPrice(Integer.parseInt(food3Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(3).setPrice(Integer.parseInt(food4Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(4).setPrice(Integer.parseInt(food5Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(5).setPrice(Integer.parseInt(food6Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(6).setPrice(Integer.parseInt(food7Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(7).setPrice(Integer.parseInt(food8Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(8).setPrice(Integer.parseInt(food9Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(9).setPrice(Integer.parseInt(food10Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(10).setPrice(Integer.parseInt(food11Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(11).setPrice(Integer.parseInt(food12Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(12).setPrice(Integer.parseInt(food13Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(13).setPrice(Integer.parseInt(food14Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(14).setPrice(Integer.parseInt(food15Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(15).setPrice(Integer.parseInt(food16Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(16).setPrice(Integer.parseInt(food17Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(17).setPrice(Integer.parseInt(food18Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(18).setPrice(Integer.parseInt(food19Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(19).setPrice(Integer.parseInt(food20Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(20).setPrice(Integer.parseInt(food21Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(21).setPrice(Integer.parseInt(food22Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(22).setPrice(Integer.parseInt(food23Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(23).setPrice(Integer.parseInt(food24Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(24).setPrice(Integer.parseInt(food25Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(25).setPrice(Integer.parseInt(food26Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(26).setPrice(Integer.parseInt(food27Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(27).setPrice(Integer.parseInt(food28Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(28).setPrice(Integer.parseInt(food29Price.getText()));
            GUI.cullama.getRestaurantMenu().getProducts().get(29).setPrice(Integer.parseInt(food30Price.getText()));
        }
        else {
            invalidInput.setText("Inputs are in invalid format.");
        }
```

Inventory Section:



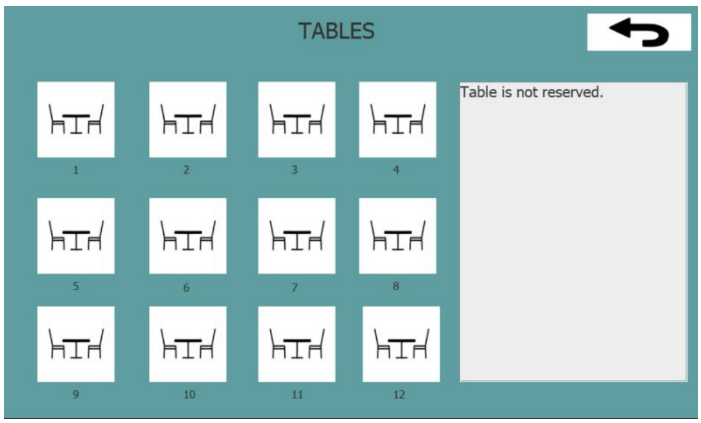
The following code block performs purchasing action according to the given item choice and amount. If the inputs are incorrect prints a warning:

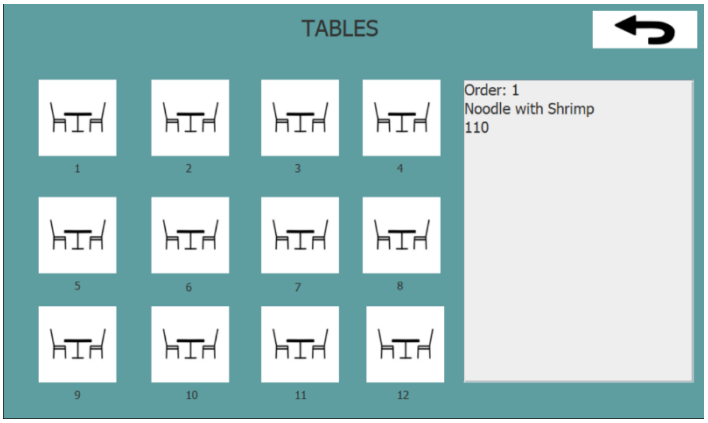
```
purchaseButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(control.checkAmount(amount.getText()) && control.checkItem(item.getText())) {
            boolean isFound = false;
            for (int i = 0; i < GUI.cullama.getRestaurantInventories().size(); i++) {
                if(GUI.cullama.getRestaurantInventories().get(i).getIngredient().getIngredientName().equals(item.getText())) {
                    GUI.cullama.getRestaurantInventories().get(i).setAmount(GUI.cullama.getRestaurantInventories().get(i).getAmount() + Integer.parseInt(amount.getText()));
                    inventory.setText(GUI.cullama.inventoryString());
                    item.setText("");
                    amount.setText("");
                    error.setText("   Item has been purchased.");
                    isFound= true;
                    break;
                }
            }
            if(!isFound) {
                error.setText("          Item not found.");
                item.setText("");
                amount.setText("");
            }
        }
        else {
            item.setText("");
            amount.setText("");
            error.setText("             Invalid input.");
        }
    }
});
```

The following code block checks if the amount of an item is below the minimum acceptable percentage and prints a “-“ next to that item or “+” otherwise:

```
public String inventoryString() {
    String inventory = "<html>";
    String isEnough = "";
    for (int i = 0; i < restaurantInventories.size(); i++) {
        if(restaurantInventories.get(i).percentage() <= restaurantInventories.get(i).getMinPercentage()) {
            isEnough = "(-)";
        }
        else {
            isEnough = "(+)";
        }
        inventory += restaurantInventories.get(i).getIngredient().getIngredientName() + " - ";
        inventory += restaurantInventories.get(i).getAmount() + isEnough + "<br>";
        isEnough = "";
    }
    return inventory;
}
```

Manager's table GUI:





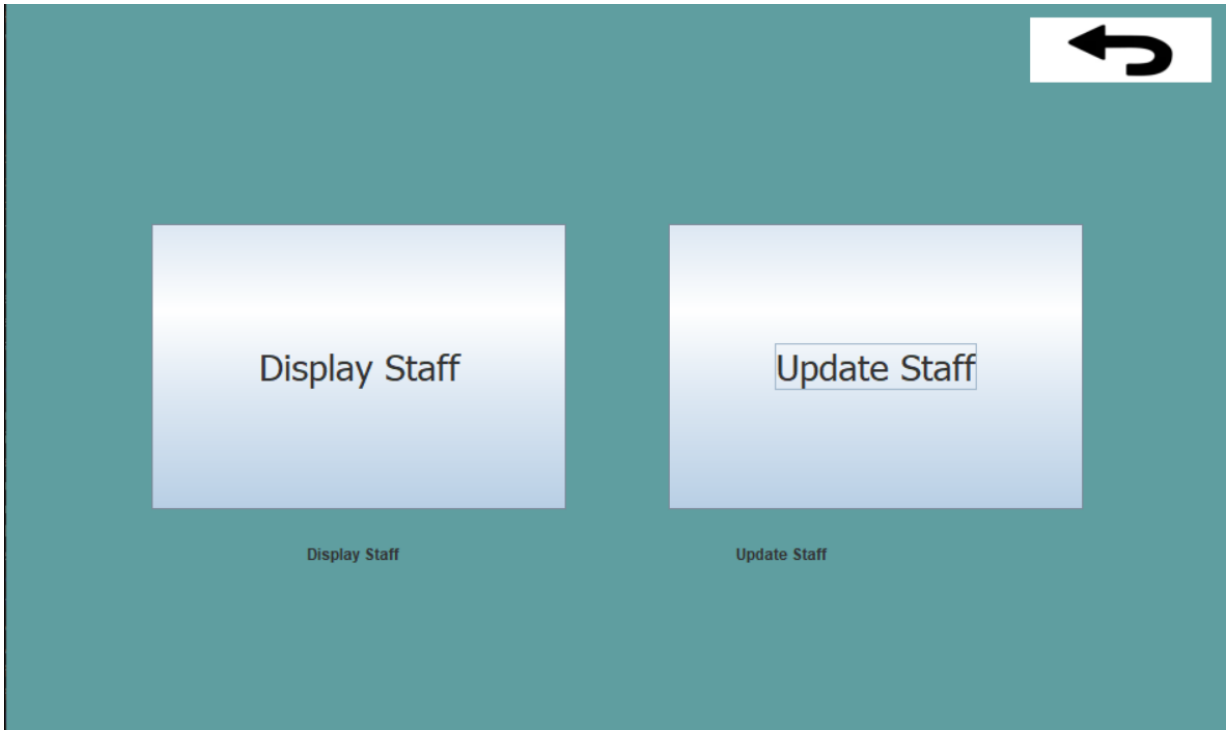
The following code block displays the customers order on that table if that table is reserved or prints "table is not reserved" otherwise.

```
tables2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        if(GUI.cullama.getRestaurantTables().get(1).isEmpty()) {
            lblNewLabel_2.setText("Table is not reserved.");
        }
        else {
            lblNewLabel_2.setText(GUI.cullama.getCustomerOrders(GUI.cullama.getRestaurantTables().get(1).getCustomer()));

        }

    }
});
```

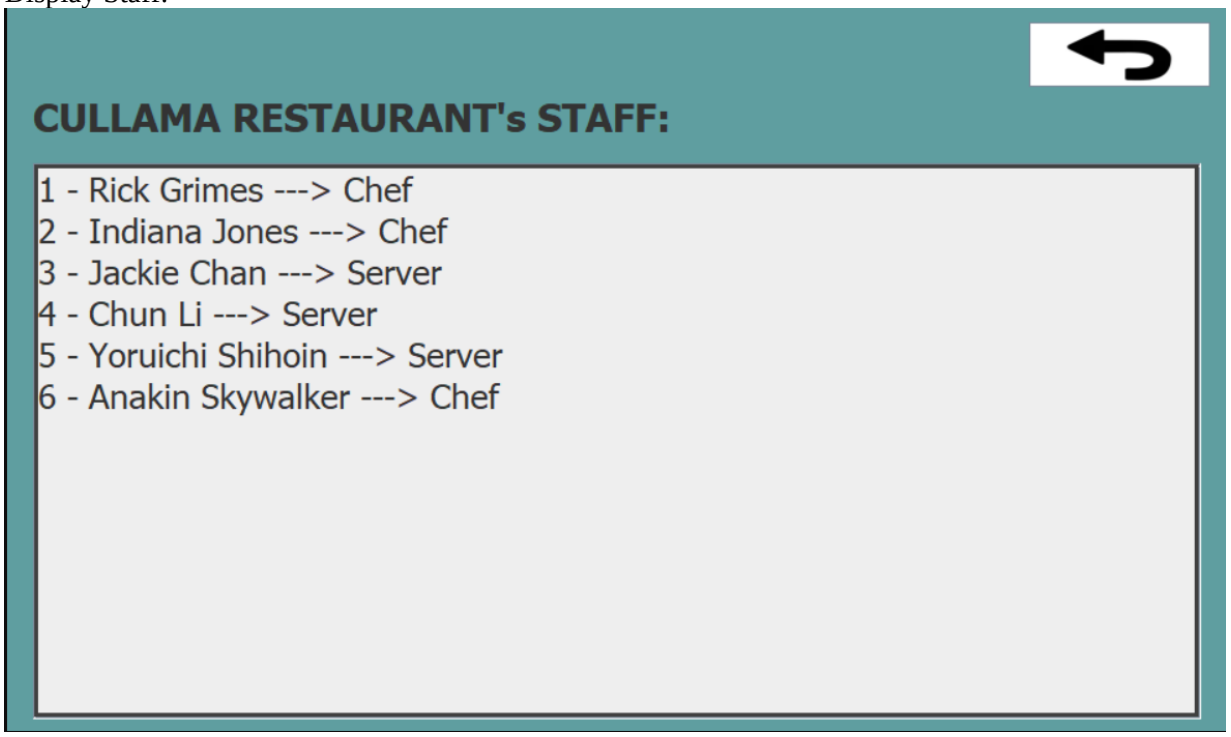
```
public String getCustomerOrders(Customer customer) {
    String orders = "";
    for (int i = 0; i < customers.size(); i++) {
        if(customers.get(i).getName().equals(customer.getName()) && customers.get(i).getSurname().equals(customer.getSurname())) {
            orders = customers.get(i).showOrdersCustomer();
            break;
        }

    }
    return orders;
}
```

Money Screen : Shows the restaurant's total money

Manager’s Staff GUI:



Display Staff:

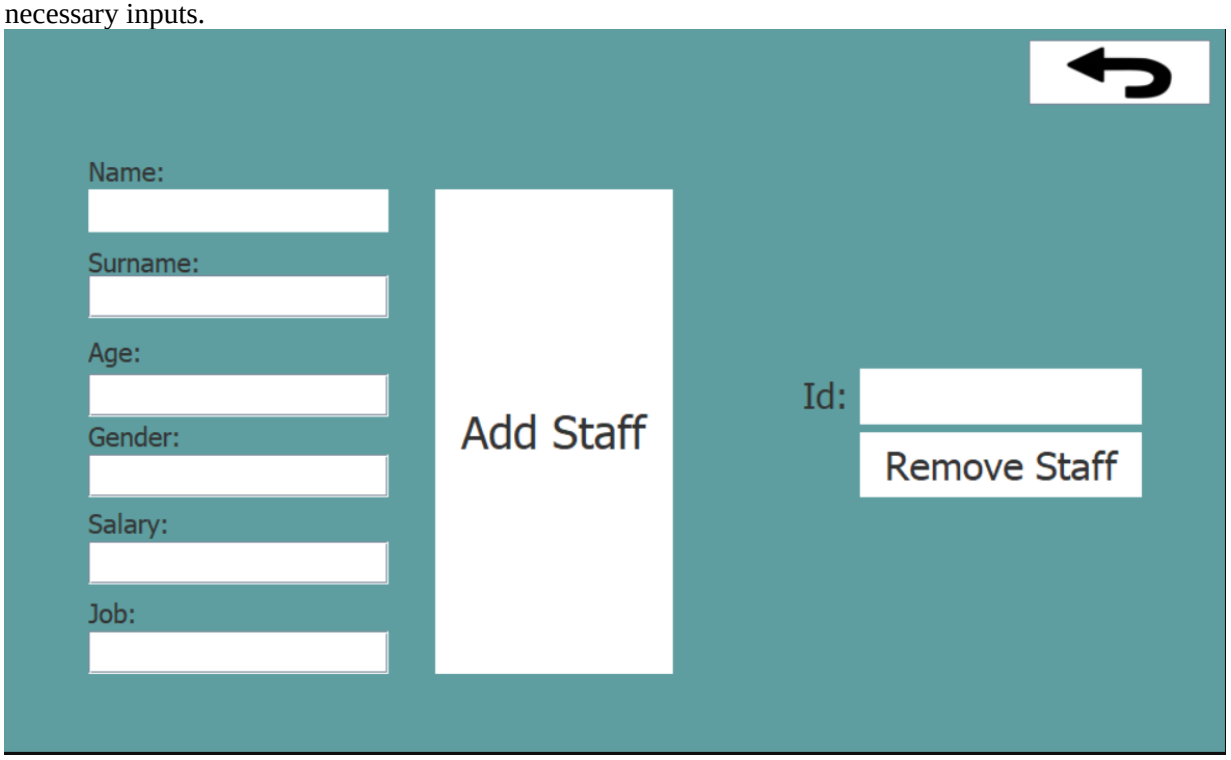


This code block displays the staffs:

```
public String staffString() {
    String staff = "<html>";
    for (int i = 0; i < restaurantStaff.size(); i++) {
        staff += restaurantStaff.get(i).getId() + " - " + restaurantStaff.get(i).getName() + " " + restaurantStaff.get(i).getSurname() + "  --->  " + restaurantStaff.get(i).getJob()
    }
    staff += "</html>";
    return staff;
}
```

Update Staff: Here, the manager can create a new staff or remove an already existing one by providing

necessary inputs.



The following code block deletes a staff according to the staff's id. The code iterates in staffs and when it finds the staff that has a matching id with the given input, it deletes the staff.

```
btnRemoveStaff.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        if(control.checkAmount(idInput.getText())) {
            boolean flag = false;
            int count = 0;
            for(int i = 0; i < GUI.cullama.getRestaurantStaff().size(); i++) {
                if(Integer.parseInt(idInput.getText()) == GUI.cullama.getRestaurantStaff().get(i).getId()) {
                    flag = true;
                    break;
                }
                count++;
            }
            if(flag)
            {
                GUI.cullama.removeRestaurantStaff(GUI.cullama.getRestaurantStaff().get(count));
                idInput.setText("");
                invalidInput.setText("Staff Removed");
            }
            else
            {
                idInput.setText("");
                invalidInput.setText("There is no such ID registered");
            }

        }
        else {
            idInput.setText("");
            invalidInput.setText("Invalid ID");
        }
    }
});
```

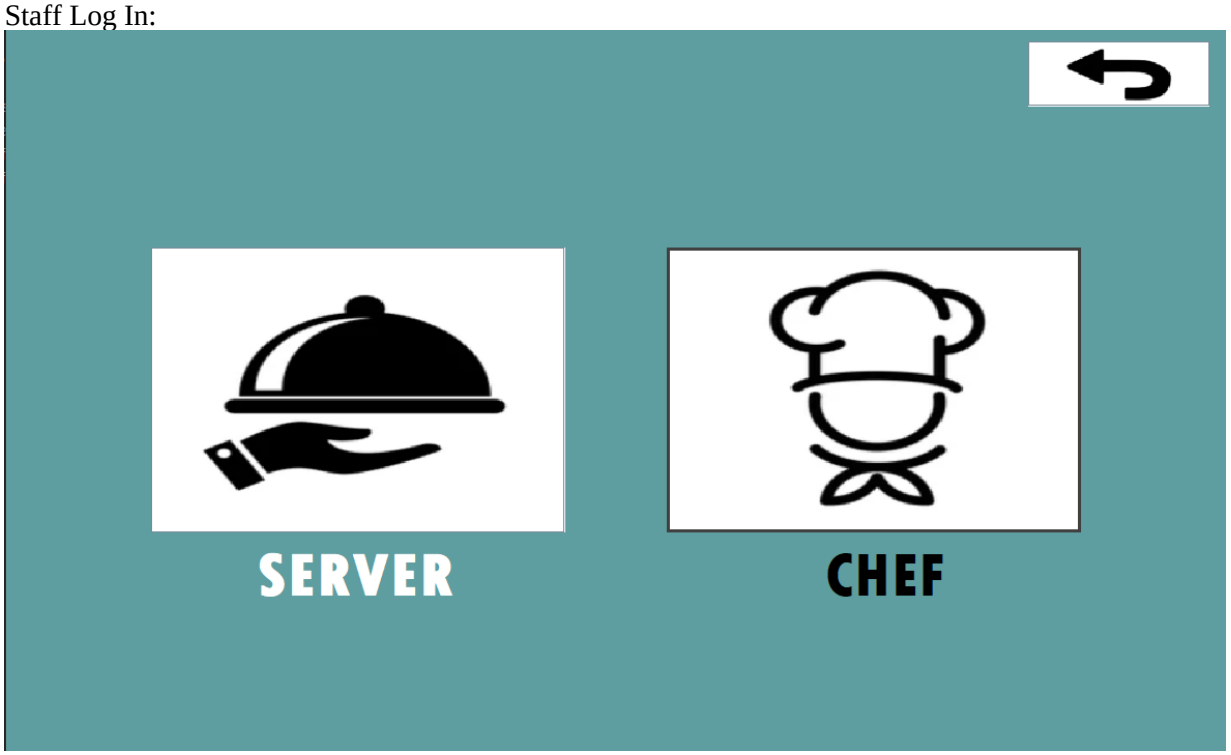
The following code controls inputs and adds a new staff according to the given inputs. This section will

be explained in details in the further parts of the report.(*)

```
btnNewButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        if(control.staffControl(nameInput.getText(),surnameInput.getText(),ageInput.getText(),genderInput.getText(),salaryInput.getText(),jobInput.getText())) {
            SimpleStaffFactory stafffactory = new SimpleStaffFactory();
            Staff staff = staffFactory.createStaff((GUI.cullama.getRestaurantStaff().get(GUI.cullama.getRestaurantStaff().size()-1).getId()+1), nameInput.getText(), surnam

            GUI.cullama.addRestaurantStaff(staff);
            invalidInput.setText("Staff Confirmed");
            nameInput.setText("");
            surnameInput.setText("");
            ageInput.setText("");
            genderInput.setText("");
            salaryInput.setText("");
            jobInput.setText("");
        }
        else {
            invalidInput.setText("Invalid Input");
            nameInput.setText("");
            surnameInput.setText("");
            ageInput.setText("");
            genderInput.setText("");
            salaryInput.setText("");
            jobInput.setText("");

        }

    }
});
```

Manager Feedbeck is as same as customer's feedback GUI: Manager can only display the feedback not write one.

Staff Log In:

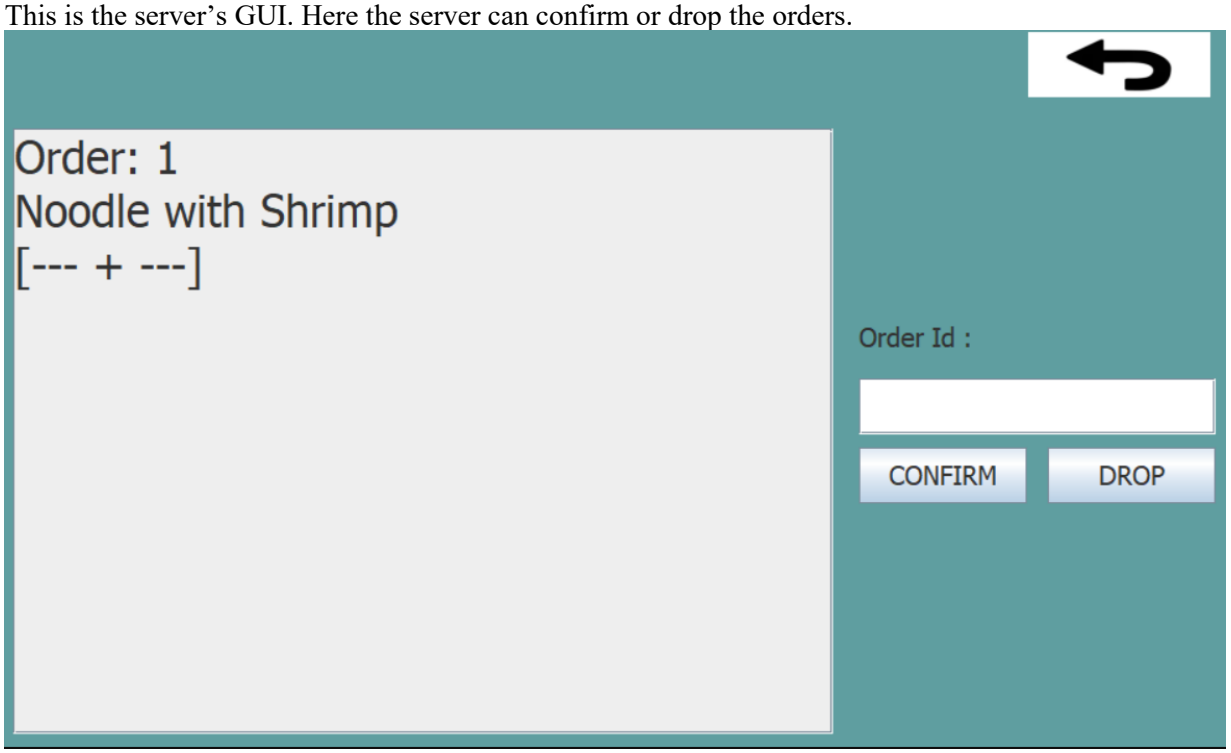


Server:

Each staff has a unique username and password assigned to them and they log in to the system using those. This code checks if the log in inputs are correct:

```
JButton serverLoginButton = new JButton("Log In");
serverLoginButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        boolean flag = false;
        for (int i = 0; i < GUI.cullama.getRestaurantStaff().size(); i++) {
            if(GUI.cullama.getRestaurantStaff().get(i).getName().equals(serverUsernameText.getText()) &&
                    GUI.cullama.getRestaurantStaff().get(i).getPassword().equals(serverPasswordText.getText()) &&
                    GUI.cullama.getRestaurantStaff().get(i).getJob().equals("Server")) {
                GUI.staffServerGui.setVisible(true);
                setVisible(false);
                flag = true;
                invalid.setText("");
                serverUsernameText.setText("");
                serverPasswordText.setText("");
                GUI.staffServerGui.orders.setText(GUI.cullama.serverNotConfirmedOrders());
                break;
            }
        }
        if(!flag) {
            invalid.setText("                    No such user found.");
            serverUsernameText.setText("");
            serverPasswordText.setText("");
        }
    }
});
```

This is the server's GUI. Here the server can confirm or drop the orders.



The following code block displays the orders that are placed by customers which are not confirmed and not dropped on the server's GUI. If the order's contents requirements are met by inventory items' amounts, the code prints a "+" under the order and "-" otherwise.

```
public String serverNotConfirmedOrders() {
    String orders = "<html>";
    for (int i = 0; i < restaurantOrders.size(); i++) {
        Order ordera = restaurantOrders.get(i);
        if(!ordera.isConfirmed() && !ordera.isDropped()) {
            orders += "Order: " + (i + 1) + "<br>";
            for (int j = 0; j < ordera.getOrderProductList().size(); j++) {
                orders += ordera.getOrderProductList().get(j).getElementName() + "<br>";
            }
        }
        if(ordera.isEnough(restaurantInventories) && !ordera.isConfirmed() && !ordera.isDropped()) {
            orders += ("[---      +      ---]<br>");
            orders += " <br>";
        }
        else if(!ordera.isEnough(restaurantInventories) && !ordera.isConfirmed() && !ordera.isDropped()) {
            orders += ("[---      -      ---]<br>");
            orders += "<br>";
        }
    }
    orders += "</html>";
    return orders;
}
```

If the server's choice is confirm, this code checks if the given order id is valid and if it is valid it checkes if the inventory requirements for that order is enough and if it is enough, the code sets the order as confirmed and decreases the used amounts of the ingredients from inventory.

```
JButton btnNewButton = new JButton("CONFIRM");
btnNewButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(!(orderId.getText().equals("") || GUI.cullama.getRestaurantOrders().size() < Integer.parseInt(orderId.getText()))) {
            if(GUI.cullama.getRestaurantOrders().get(Integer.parseInt(orderId.getText()) - 1).isEnough(GUI.cullama.getRestaurantInventories())) {
                GUI.cullama.getRestaurantOrders().get(Integer.parseInt(orderId.getText()) - 1).setConfirmed(true);
                GUI.cullama.getRestaurantOrders().get(Integer.parseInt(orderId.getText()) - 1).
                getCustomer().setOrderConfirmed(GUI.cullama.getRestaurantOrders().get(Integer.parseInt(orderId.getText()) - 1));
                GUI.cullama.decreaseInventory(GUI.cullama.getRestaurantOrders().get(Integer.parseInt(orderId.getText()) - 1));
                orders.setText(GUI.cullama.serverNotConfirmedOrders());
            }
            orderId.setText("");
            error.setText("");
        }
        else {
            error.setText("There is no such order.");
            orderId.setText("");
        }
    }
});
```

This following function is used for setting selected orders of a customer as confirmed in the above code.

```
public void setOrderConfirmed(Order order) {
    boolean flag = false;
    int index = -1;
    for (int i = 0; i < this.order.size(); i++) {
        if(order.getOrderProductList().size() == this.order.get(i).getOrderProductList().size() && !(this.order.get(i).isConfirmed())) {
            for (int j = 0; j < order.getOrderProductList().size(); j++) {
                if(order.getOrderProductList().get(j) == this.order.get(i).getOrderProductList().get(j))
                    flag = true;
                else {
                    flag = false;
                    break;
                }
            }
        }
        if(flag) {
            index = i;
            break;
        }
    }
    if(flag && index != -1) {
        this.order.get(index).setConfirmed(true);
    }
}
```

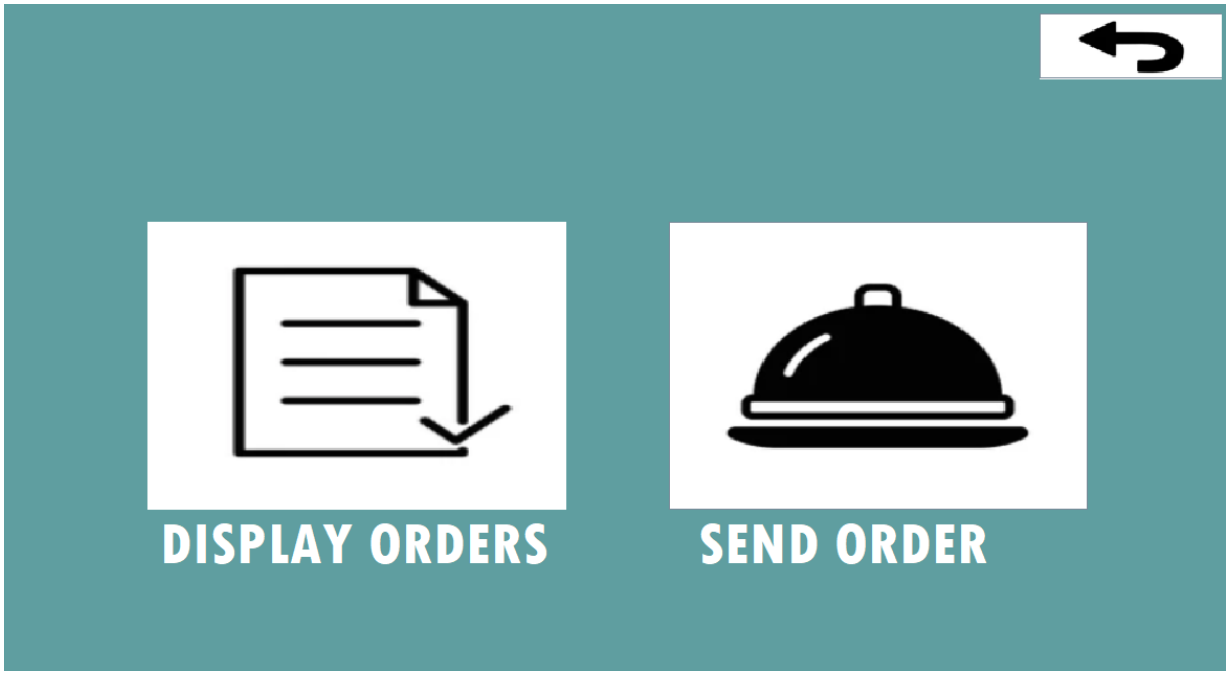
If the server's choice is drop, this code checks if the inventory requirements is not enough and if so sets the order as dropped.

```
JButton btnDrop = new JButton("DROP");
btnDrop.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(!(orderId.getText().equals("") || GUI.cullama.getRestaurantOrders().size() < Integer.parseInt(orderId.getText()))) {
            if(!(GUI.cullama.getRestaurantOrders().get(Integer.parseInt(orderId.getText()) - 1).isEnough(GUI.cullama.getRestaurantInventories()))) {
                GUI.cullama.getRestaurantOrders().get(Integer.parseInt(orderId.getText()) - 1).setDropped(true);
                GUI.cullama.getRestaurantOrders().get(Integer.parseInt(orderId.getText()) - 1).
                getCustomer().setOrderDropped(GUI.cullama.getRestaurantOrders().get(Integer.parseInt(orderId.getText()) - 1));
                orders.setText(GUI.cullama.serverNotConfirmedOrders());
            }
            orderId.setText("");
            error.setText("");
        }
        else {
            error.setText("There is no such order.");
            orderId.setText("");
        }
    }
});
```

This following function is used for setting selected orders of a customer as confirmed in the above code.

```
public void setOrderDropped(Order order) {
    boolean flag = false;
    int index = -1;
    for (int i = 0; i < this.order.size(); i++) {
        if(order.getOrderProductList().size() == this.order.get(i).getOrderProductList().size() && !(this.order.get(i).isDropped())) {
            for (int j = 0; j < order.getOrderProductList().size(); j++) {
                if(order.getOrderProductList().get(j) == this.order.get(i).getOrderProductList().get(j))
                    flag = true;
                else {
                    flag = false;
                    break;
                }
            }
        }
        if(flag) {
            index = i;
            break;
        }
    }
    if(flag && index != -1) {
        this.order.get(index).setDropped(true);
    }
}
```

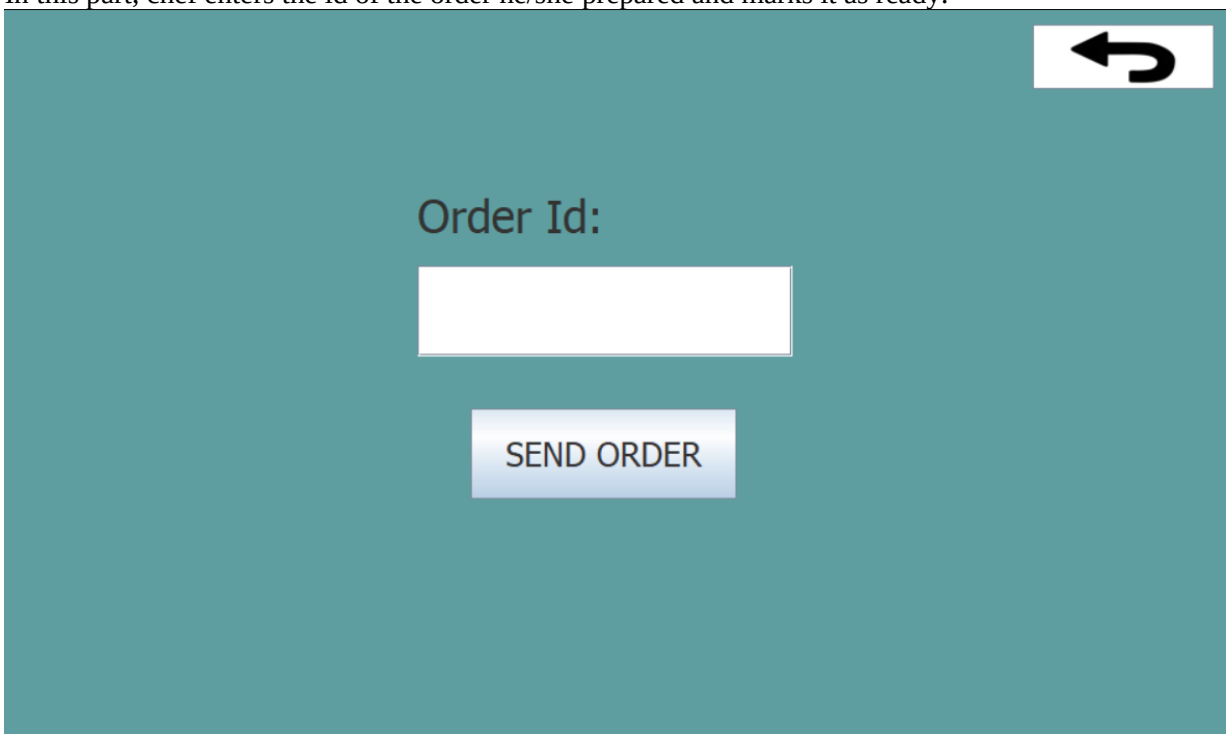
Chef: The chef can display orders or mark them as prepared in send order section.



If the order is confirmed and not marked as ready yet, this function prints these orders on the chefs display order section.

```java
public String showChefOrders() {
    String orders = "<html>";
    for (int i = 0; i < restaurantOrders.size(); i++) {
        Order ordera = restaurantOrders.get(i);
        if(ordera.isConfirmed() && !ordera.isOrderReady()) {
            orders += "Order: " + (i + 1) + "<br>";
            for (int j = 0; j < ordera.getOrderProductList().size(); j++) {
                orders += ordera.getOrderProductList().get(j).getElementName() + "<br>";
            }
        }
    }
    orders += "</html>";
    return orders;
}
```

In this part, chef enters the id of the order he/she prepared and marks it as ready.



This function is used for setting the order ready if the order is not dropped.

```java
btnNewButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {

        if(!(textField.getText().equals("") || GUI.cullama.getRestaurantOrders().size() < Integer.parseInt(textField.getText())))
        {

            if(!GUI.cullama.getRestaurantOrders().get(Integer.parseInt(textField.getText()) - 1).isDropped())
            {
                GUI.cullama.getRestaurantOrders().get(Integer.parseInt(textField.getText()) - 1).setOrderReady(true);
                GUI.cullama.getRestaurantOrders().get(Integer.parseInt(textField.getText()) - 1).
                getCustomer().setOrderReady(GUI.cullama.getRestaurantOrders().get(Integer.parseInt(textField.getText()) - 1));
            }

        }
        textField.setText("");

    }
});
```

FACTORY DESIGN PATTERN

In our project we decided to use simple factory design pattern because we thought that most optimal design pattern would be simple factory design pattern. In the project we used simple factory design pattern for creating objects of Chef and Server classes. The reason behind why we decided to use simple factory design pattern is to make code more portable and easy to edit. Staff interface was created from the common functions and methods of Chef and Server classes. Chef and Server classes implement Staff interface. Before using simple factory design pattern, in order to create objects of Chef and Server classes we had to look into our main code where the objects are initialized and in order to check which class' object we must create we had to use if else conditional statements. But after using simple factory design pattern we moved these if else statements to StaffFactory class in order to edit or access these statements easily. createStaff function of the StaffFactory class decides which class' object will be added according to the job of the staff that will be added. If we want to add a new staff with a new type of job later with a simple factory design pattern, all we need to do is adding another if statement for the new job to the StaffFactory class. Instead of searching for the code where we added staff and editing that in the main code, with this change in StaffFactory class, our code is more portable. For example, in the future if restaurant manager decides to add address delivery order there should be a courier. With simple factory design pattern courier can be added easily with less effort.

```
public class SimpleStaffFactory {

    public Staff createStaff(int id, String name, String surname, int age,String gender, String job, int salary, String password)
    {
        if(job.equals("Server"))
        {
            return new Server(id,name,surname,age,gender,job,salary,password);
        }
        else
        {
            return new Chef(id,name,surname,age,gender,job,salary,password);
        }
    }
}
```

```
SimpleStaffFactory staffFactory = new SimpleStaffFactory();
Staff staff = staffFactory.createStaff(Integer.parseInt(rowArr[0]),rowArr[1],rowArr[2],Integer.parseInt(rowArr[3]),rowArr[4],rowArr[5],Integer.parseInt(rowArr[6]),rowArr[7]);
restaurantStaff.add(staff);
```

```
SimpleStaffFactory staffFactory = new SimpleStaffFactory();
Staff staff = staffFactory.createStaff((GUI.cullama.getRestaurantStaff().get(GUI.cullama.getRestaurantStaff().size()-1).getId()+1), nameInput.getText(), surnameInput.getText(),
GUI.cullama.addRestaurantStaff(staff);
```

CHAPTER FIVE

CONCLUSION AND FUTURE WORKS

First of all, the “Restaurant Information System” is fully completed with some changes in it. The changes are:

1. Removed the courier and cashier staff.
2. Removed address delivery order.
3. Removed order factory and added staff factory instead.

Summary of the project:

As planned at the beginning, we have three interfaces for customers, managers and staff. Each having its own functions and extent in the code.

In the staffs’ interface, there are two parts. Server and chef. Server checks the given orders by customer and confirms the order so that the order will be send to the chef. After the order is prepared by the chef, the order is marked as ready, and the payment is taken after that.

When a customer logs in, the available options are placing an order, making a payment, and writing and reading feedback. Customer chooses the foods and beverages from the restaurant’s menu. The order is sent to the server’s screen and here the server confirms or drops the order. If the order is confirmed A table is assigned to the customer and the order is sent to the chef to be prepared. After preparing the order, the chef marks it as ready, and the payment is taken from the customer.

As for the manager’s interface, the manager has full authority over the system. The manager can change and update everything and can view every information about the restaurant which are inventory, staff, menu, tables, feedback and money. The manager can add new staff and remove the existing ones, change the items on the restaurant’s menu, view feedback and delete the inappropriate ones. Manager’s log in id and password are held in a text file to check if the entered id and password is correct.

Possible future works:

1- A system for booking a table for a time in the future may be added. With that upgrade customers will be able to place their orders on a chosen table for a given time in the future.
2- Address delivery could be added by bringing back the courier.