UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

Department of Computer Science

CS4125 – Group Project

Restaurant Management System

Team Orange

Ciaran Carroll - 16181492

Neale Conway - 16192206

Ronan Barry – 15179486

Contents

Narrative

There are countless challenges that a restaurant faces in its day to day operation, our plan is to create a software system that has all the features and capabilities to simplify the running of a restaurant. This system will be central to the success of managing a restaurant, instead of using the traditional approaches of taking orders using pen and paper, or writing up staff rosters on a white board, we plan to completely digitalise the running of a restaurant. This in turn will result in increased efficiency, increased accuracy and less human errors, resulting in a much more profitable business.

The system will be completely in house, there'll be a user log in, where he/she may oversee use of the. An integral part of a restaurant manager's job is to manage his/her employee's hours effectively. The admin login will simplify this process and serve as a record for his/her employee's work. The admin will also have options to edit the seating arrangements, the restaurant's menu and the restaurant's staff as he sees fit. The staff log in will be used to take orders and allocate customers seating.

Other features include a recommendation system and an awards system. The recommendation system will be based off previously placed orders, this will add a personal touch to visiting the restaurant. The award system will have 3 levels, gold, silver and bronze. Each level will receive a different deduction in price from their meal and will reward loyalty from customers.

Software Lifecycle

We researched two software development methodologies we could've chosen for our project, Agile and Waterfall. Each methodology is widely used today, Waterfall being the more traditional methodology and Agile a newer methodology which is becoming the norm for so many software companies today. We found both methodologies had their pro's and con's.

Waterfall

Initially waterfall seemed like a reasonable and straight forward methodology. With each phase having its own set of deliverables that need to be completed before moving onto the next phase. However, it became apparent that it was not ideal for numerous reasons. As we were creating an enterprise system, which is quite a substantial project for us, the rigidity and documentation needed for such a project using the waterfall method would be unsuitable. We were aware that the time needed to create the documentation would push back our proposed start date for implementation and put pressure on us to complete the project. The rigid nature of the methodology was also a turn off for us as we anticipated that we could miss out on a requirement during documentation or think of a new requirement during implementation. These issues would then become problematic to resolve.

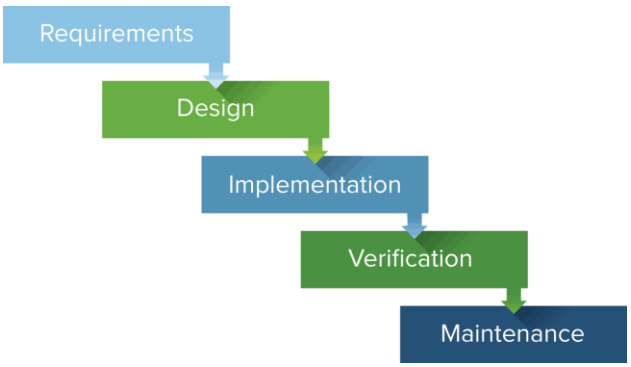


Figure 1.

Agile

Agile instantly became the obvious choice for us. The process is completely based on the incremental progress. Therefore, we know exactly what is complete and what is not. This reduces risk in the development process. An important feature of software development is the ability to adapt to various circumstances that may arise during the development process. Agile offers us the flexibility to adapt seamlessly as our needs and objectives evolve. We also have experience working in Agile environments during are placement so are already comfortable taking this approach.

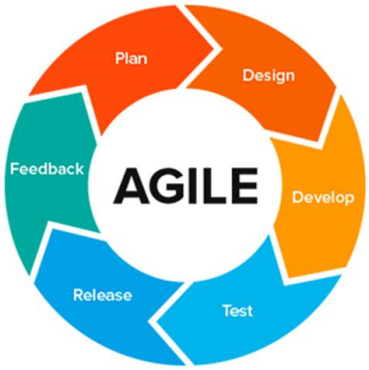


Figure 2.

Project Plan

Deliverable	Description	Responsibilty	Week
Narrative	Motivation behind project idea and project focus	Neale	4
Project Plan	Specifying Jobs and Roles	Group	4
Software Lifecycle	Discussion on model to be used	Neale	5
Requirements	Use Case Diagrams. Use Case Descriptions. Structured Use Case Descriptions. Non-Functional requirements. Tactics for handling quality attributes. GUI Prototypes	Ronan	5-6
Analysis Sketches	Identify Candidate Classes. Analysis class diagram, Sequence diagram, Entity relationship diagram	Ciaran	6
System Architecture	Architecture Diagram with Interfaces	Group	7
Code	Implementation of the System	Group	7-11
Recovered	Architectural	Group	12

Blueprints	Diagrams. Class diagrams. Interaction Diagram State Chart. Description of Patterns. Approach to Concurrency Support.		
Critique		Group	12
References		Group	12

Requirements

Functional Use Case Diagrams

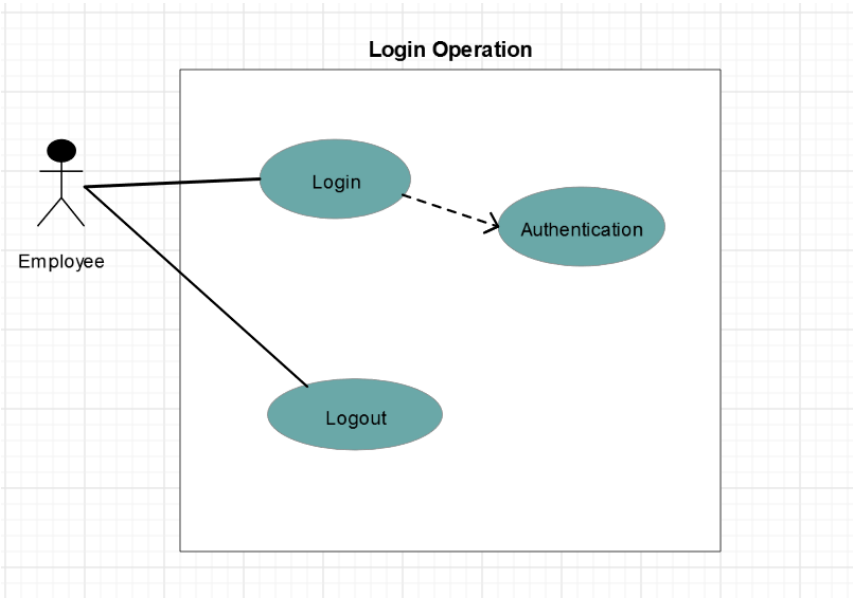


Figure 3. Login Operation

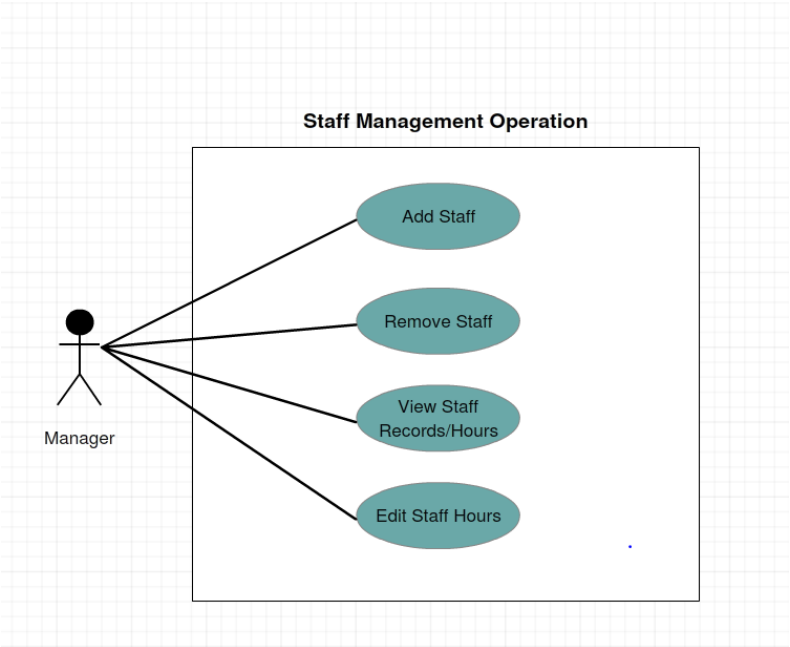


Figure 4. Staff Management Operation

Note : Advised in lab during week 8 to stray away from staff management.

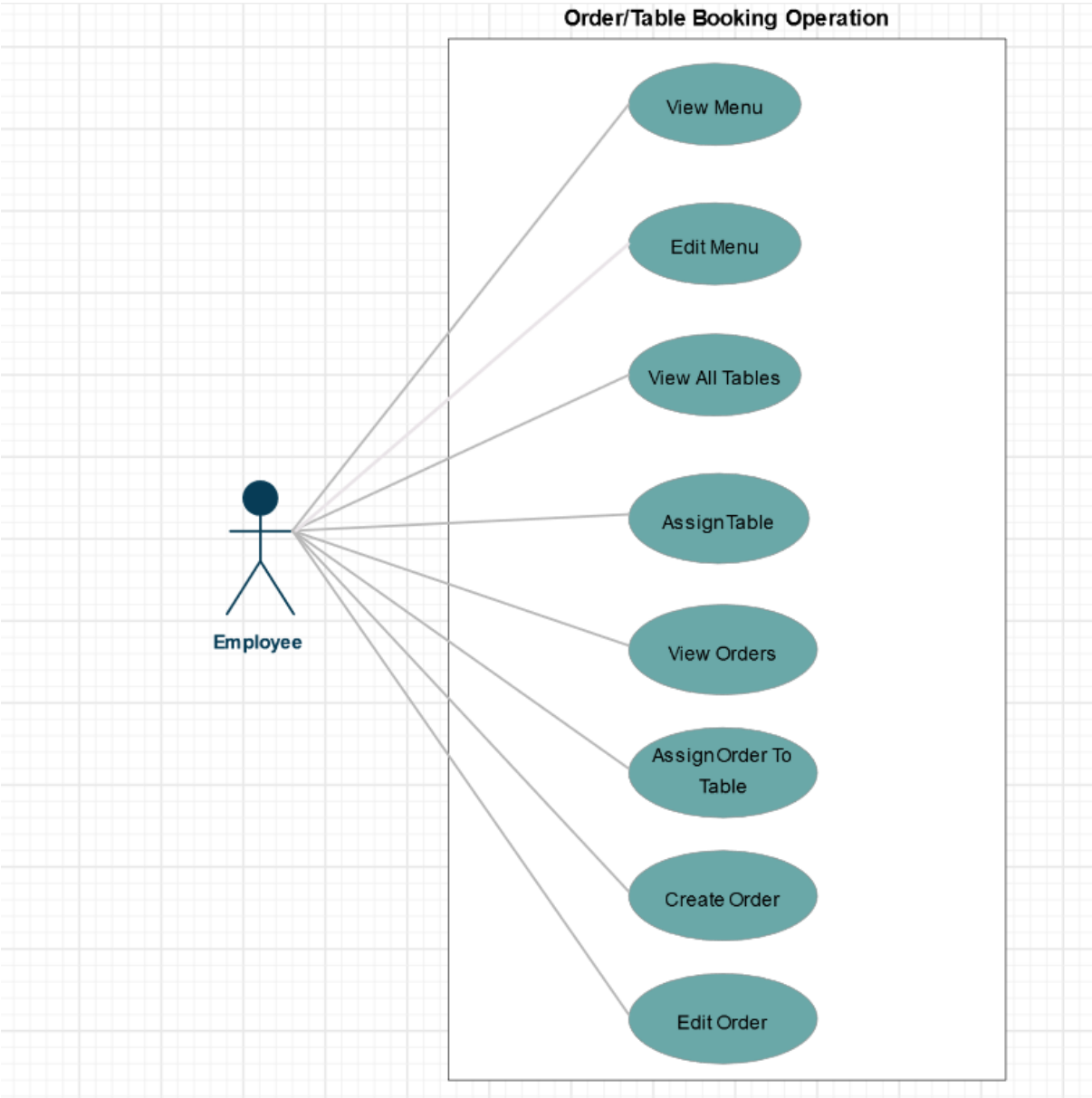


Figure 5. Order Operation

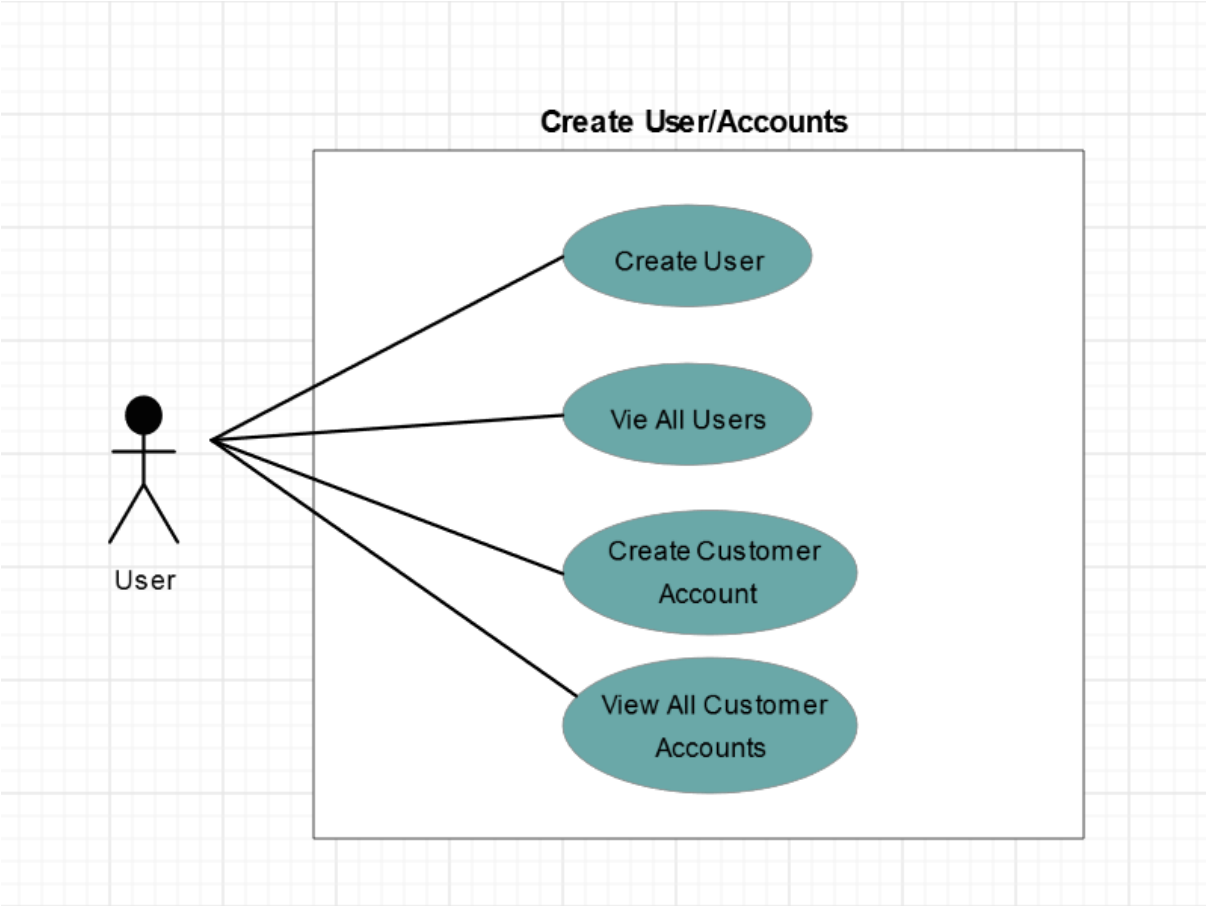


Figure 6. Create User/Account Operation

Use Case Descriptions

Login

Actor Action	**System Response**
1. User enters login credentials and presses login button.	2. Systems receives the entered details and processes the data to see if it is valid. User is made aware if valid or invalid.
	3. User will be logged in as an employee or manager.
Alternative Course	
4a. User enters incorrect details and is shown an invalid login message. 4b. User is asked to enter details again	
Non-Functional Requirement: Security	
Communication of data regarding user login should be properly encrypted.	

Logout

Actor Action	**System Response**
1. User chooses Logout	2. System display start screen
Alternative Course	
Non-Functional Requirement: Security	
System releases actor data	

View Menu

Actor Action	**System Response**
1. Manager/Employee selects view menu	2. System retrieves menu details from database
	3. System will display all menu data
Alternative Course	
4a. System fails to retrieve menu data from database 4b. System displays an error message	

Non-Functional Requirement: Usability
Data shown should be easy to understand and displayed in a pleasing manner.

Edit Menu

Actor Action	**System Response**
1. Manager selects edit menu	2. System retrieves menu details from database
	3. System will display current menu data
4. Manager selects what item to edit	5. System saves changes to database
Alternative Course	
4a. System fails to retrieve menu data from database 4b. System displays an error message 4c. System fails to save new menu data	
Non-Functional Requirement: Performance	
System should be able to complete this action within X seconds.	

View All Tables

Actor Action	**System Response**
1. Manager/Employee/User selects view all tables	2. System retrieves table details from database
	3. System will display all table data
Alternative Course	
4a. System fails to retrieve table data from database 4b. System displays an error message	
Non-Functional Requirement: Performance	
System should be able to complete this action within X seconds.	

Assign Table

Actor Action	**System Response**
1. Manager/Employee selects assign table	2. System retrieves table details from database
	3. System will display all table data

4. Manager/Employee will assign a customer to a table	5. System will store new data in the database
Alternative Course	
4a. System fails to retrieve table data from database 4b. System displays an error message 4c. Table could already be occupied 4d. Occupied message will be shown to staff member	
Non-Functional Requirement: Performance	
System should be able to complete this action within X seconds.	

Remove Table Assignment

Actor Action	**System Response**
1. Manager/Employee selects assign table	2. System retrieves table details from database
	3. System will display all table data
4. Manager/Employee will press “remove”	5. System will update database
Alternative Course	
4a. System fails to retrieve table data from database 4b. System displays an error message 4c. Table may already be empty 4d. No customer message will be shown to staff member	
Non-Functional Requirement: Performance	
System should be able to complete this action within X seconds.	

Add User

Actor Action	**System Response**
1. Manager will select “Add User”	
2. Manager will then enter all relevant data regarding staff member and shit register.	3. New staff data will be added to the database
Alternative Course	
4a. Staff member may already exist with exact same details 4b. User is asked to enter details again 4c. User notified staff member already exists	

Non-Functional Requirement: Security
Communication of data regarding user login details should be properly encrypted.

Remove Users

Actor Action	**System Response**
1. User will select “Remove user”	2. System will present a table listing all current users
3. User will select which user to remove	4. User data will be deleted, and database updated
Alternative Course	
4a. Communication with database may fail	
Non-Functional Requirement:	

View All Registered Users

Actor Action	**System Response**
1. User will select “View All Users”	2. System will present a table listing all current users
Alternative Course	
4a. Communication with database may fail	
Non-Functional Requirement: Usability	

Edit Staff Hours – Decided against this in lab on week 8 after advice from JJ.

Actor Action	**System Response**
1. Manager will select “Edit Staff Hours”	2. System will present a table listing all current staff members.
3. Manager will select the staff member which hours they wish to edit, and press “edit”	4. New data will be saved to the database

Alternative Course
4a. Communication with database may fail 4b. Hours may not be available 4c. Input may be invalid
Non-Functional Requirement: Security
Communication of data regarding user login should be properly encrypted.

Create Customer Account

Actor Action	**System Response**
1. User will select “Create Account”	
2. User will then enter all relevant data regarding customer and hit enter.	3. System will store all data in database
Alternative Course	
4a. Staff member may already exist with exact same details 4b. User is asked to enter details again 4c. User notified staff member already exists	
Non-Functional Requirement: Security	
Communication of data regarding user login should be properly encrypted.	

View All Customer Accounts

Actor Action	**System Response**
1. Manager will select “View All Accounts”	2. System will present a table listing all current user accounts.
Alternative Course	
4a. Communication with database may fail	
Non-Functional Requirement: Usability	
Data displayed should be clear and easy to understand	

Create Order

Actor Action	**System Response**
1. Manager/Employee selects Create order	

2. User enters phone number/details of customer account and assigns table	
3. Select order items	4. Items stored in database
	5. Price calculated
Alternative Course	
4a. System fails to connect to database 4b. System displays an error message 4c. Account may not exist 4d. No order message will be shown to staff member	
Non-Functional Requirement: Performance	
System should be able to complete this action within X seconds once order process is completed.	

Edit Order

Actor Action	**System Response**
1. Manager/Employee selects Edit Order	2. System retrieves all order details from database
	3. System will display all order data
4. Manager/Employee will press “edit” and change order details	5. System will update database
Alternative Course	
4a. System fails to retrieve order data from database 4b. System displays an error message 4c. Order may not exist 4d. No order message will be shown to staff member	
Non-Functional Requirement: Performance	
System should be able to complete this action within X seconds.	

Detailed Use Case Description

Use Case: Create Order(14)

Use Case	14 – Create Order	
Goal in context	Create Order for customer and assign to account and table	
Scope & level		
Preconditions	1. User must be logged in 2. Account for customer must exist or be created. 3. Table must be assigned.	
Postconditions	1. Order is created and stored in database. 2. Cost is calculated.	
Success end conditions	Order is created successfully and stored. Order completed for customer.	
Failed end connection	User is made aware input is invalid	
Primary, Secondary, Actors	User, System, payment system	
Trigger	Edit Menu is selected	
Description	Step 1	Staff presses create order
	Step 2	Sent to create order window
	Step 3	Side window of recommendations is displayed
	Step 4	Staff must enter customer account and assign a table
	Step 5	Staff then brought to order screen
	Step 6	Staff selects starter
	Step 7	Staff selects Main course
	Step 8	Staff selects Desert
	Step 9	Staff selects Beverage
	Step 10	Order is displayed on the screen
	Step 11	Order is stored in database
	Step 12	Price is calculated
Extensions		**Branching action**
	Step 4a	Staff is notified that input is not valid
Variations		**Branching action**
	Step 11a	Connection to database is failing

Use Case: Add User(8)

Use Case	8 – Add User	
Goal in context	Create a new user account for a staff member	
Scope & level		
Preconditions	Another User must be logged in	
Postconditions	New user account created	
Success end conditions	User account is created and details stored within database	
Failed end connection	User is made aware input is invalid	
Primary, Secondary, Actors	User, System	
Trigger	Add User is selected	
Description	Step 1	User presses Add User
	Step 2	Sent to create User window
	Step 3	User has fills in details, username, password, phone number, forename and surname
	Step 4	Details then checked with system
	Step 5	User account is then created and stored in database
Extensions		**Branching action**
	Step 4a	Staff is notified that input is not valid
	Step 4b	User may already exist
Variations		**Branching action**
	Step 5a	Connection to database is failing

Use Case: Edit Order(8)

Use Case	8 – Edit Order	
Goal in context	Edit order in system	
Scope & level		
Preconditions	User must be logged in Order must exist	
Postconditions	Now order data saved	
Success end conditions	New order is saved successfully	
Failed end connection	User is made aware input is invalid	
Primary, Secondary, Actors	User, System	
Trigger	Add User is selected	
Description	Step 1	User presses View Orders
	Step 2	Sent view orders window
	Step 3	User is presented with all available orders
	Step 4	Selects order to edit and presses edit
	Step 5	User is presented with current order data
	Step 6	User can now change data
	Step 7	User presses submit
	Step 8	New order data saved to database
Extensions		**Branching action**
	Step 7a	Staff is notified that input is not valid
Variations		**Branching action**
	Step 8a	Connection to database is failing

Tactics Adopted To Support Quality Attributes

Portability

We decided that java would be the best language to use for our project and would be useful in the implementation of our application. Java is a very portable language as with it use of byte code java programs can be run on any machine that has a java virtual machine on it. This means our code did not need to be modified for all operating systems or machines it will run regardless of the hardware as long as there is a java virtual machine on the system.

Reusability

Given we want our system to be easily reusable we took a very object oriented approach when designing our system and writing our code. We aimed to have a high level of reusability from the outset. We examined common functionality when creating our software and made sure to keep our system broken down into modules that can be reused in the future if needed to help with future workloads. Our system is based on a single restaurant but we also had the idea to spread it across multiple restaurants run by the same organisation, given this idea we thought having a key focus on modular components would make it easier to broaden the software to an overall network of restaurants in the future.

GUI Prototypes

Login Screen

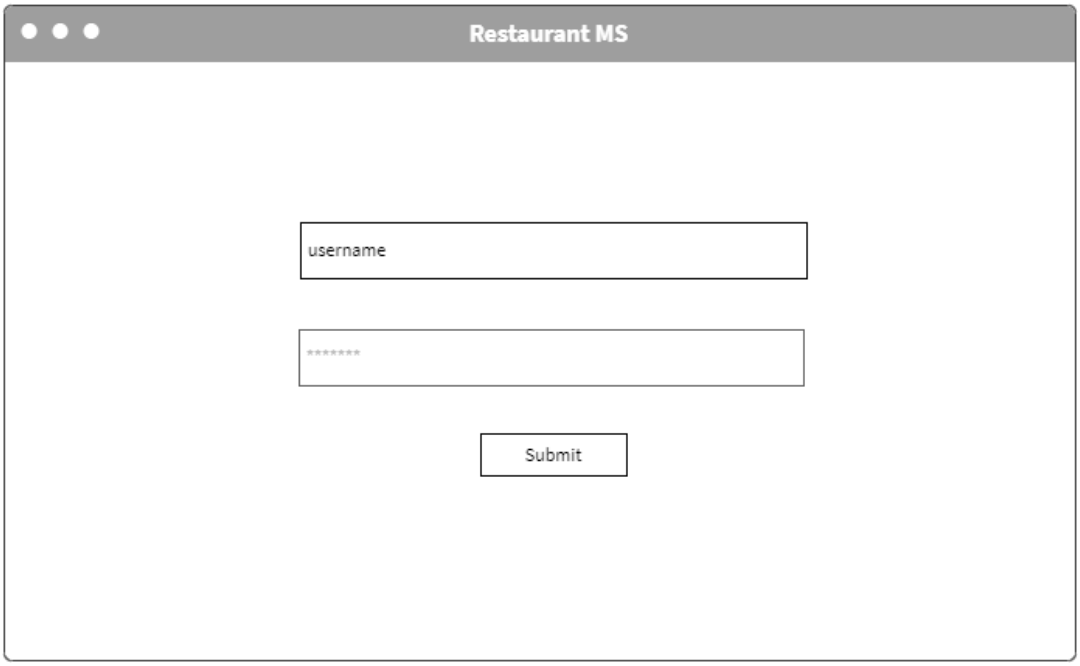


Figure 7.

Order Menu

Figure 8.

System Architecture

The Model-View-Controller (MVC) is an architectural pattern that separates an application into three main logical components: the model, the view, and the controller. Each of these components are built to handle specific development aspects of an application. We felt that this pattern was most suited to design ideas which we had and we planned to implement it using three main layers, UI, Business and Database.

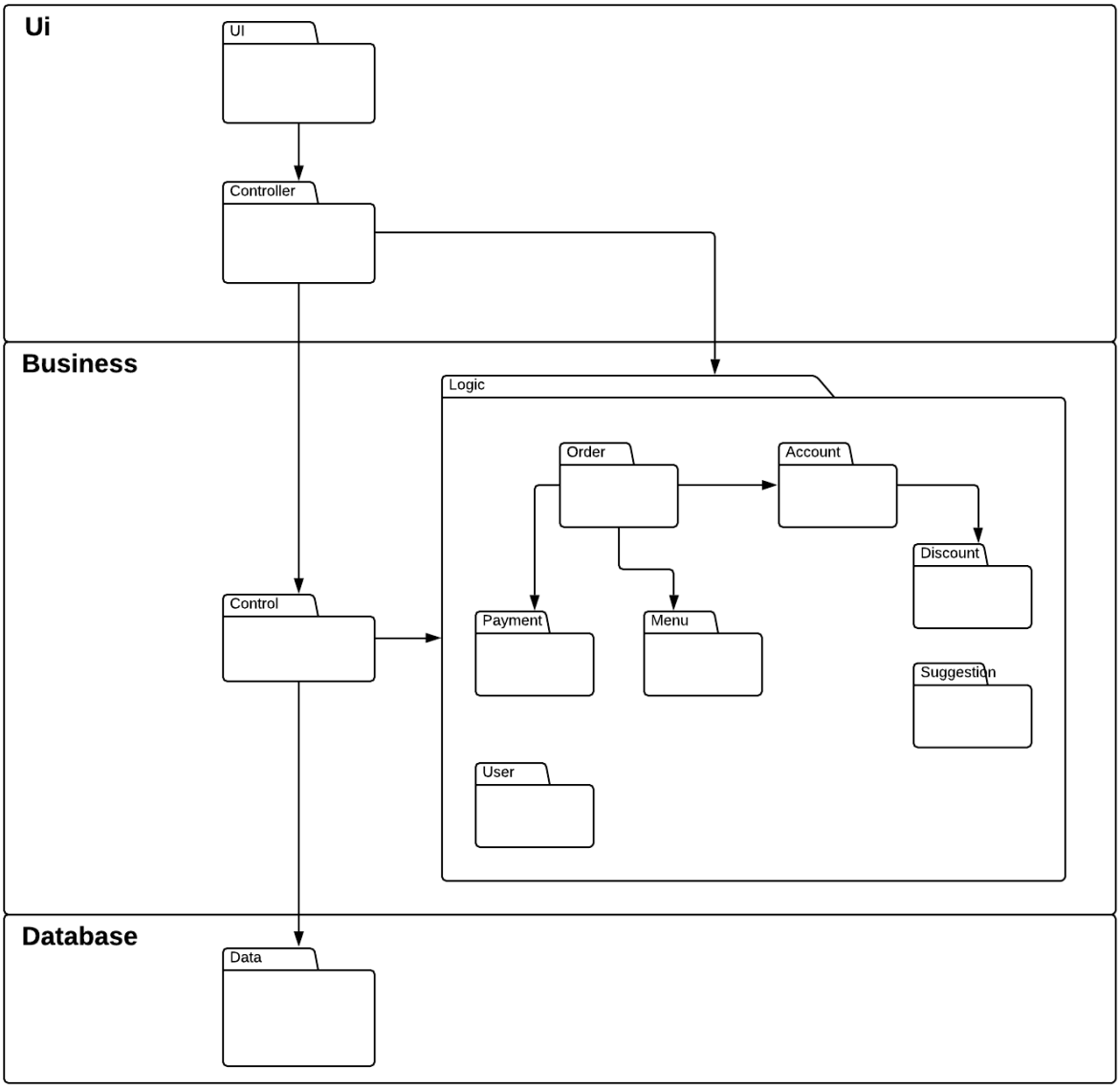


Figure 9. System Architecture

Analysis

Candidate Classes (Data Driven Design)

Noun Identification Technique

The Restaurant Management System should allow users to log in from login page using a username and password. The user should then be transferred to a main menu. The user should be able to create and edit accounts, create and edit orders and create and edit user details. There should be a menu which will be filled with different items. A discount should be applied to the final price of loyal customers. Payments should then be processed through the application when completing orders.

Classes
• Restaurant • ~~Management~~ - Irrelevant • ~~System~~ - Irrelevant • Login • ~~Username~~ - Attribute • ~~Password~~ - Attribute • User • Menu • Account • Order • ~~Detail~~ - Irrelevant • Item • Discount • Price - Attribute • ~~Customer~~ - Unnecessary • Payment • ~~Application~~ – Irrelevant

Analysis Time Class diagram

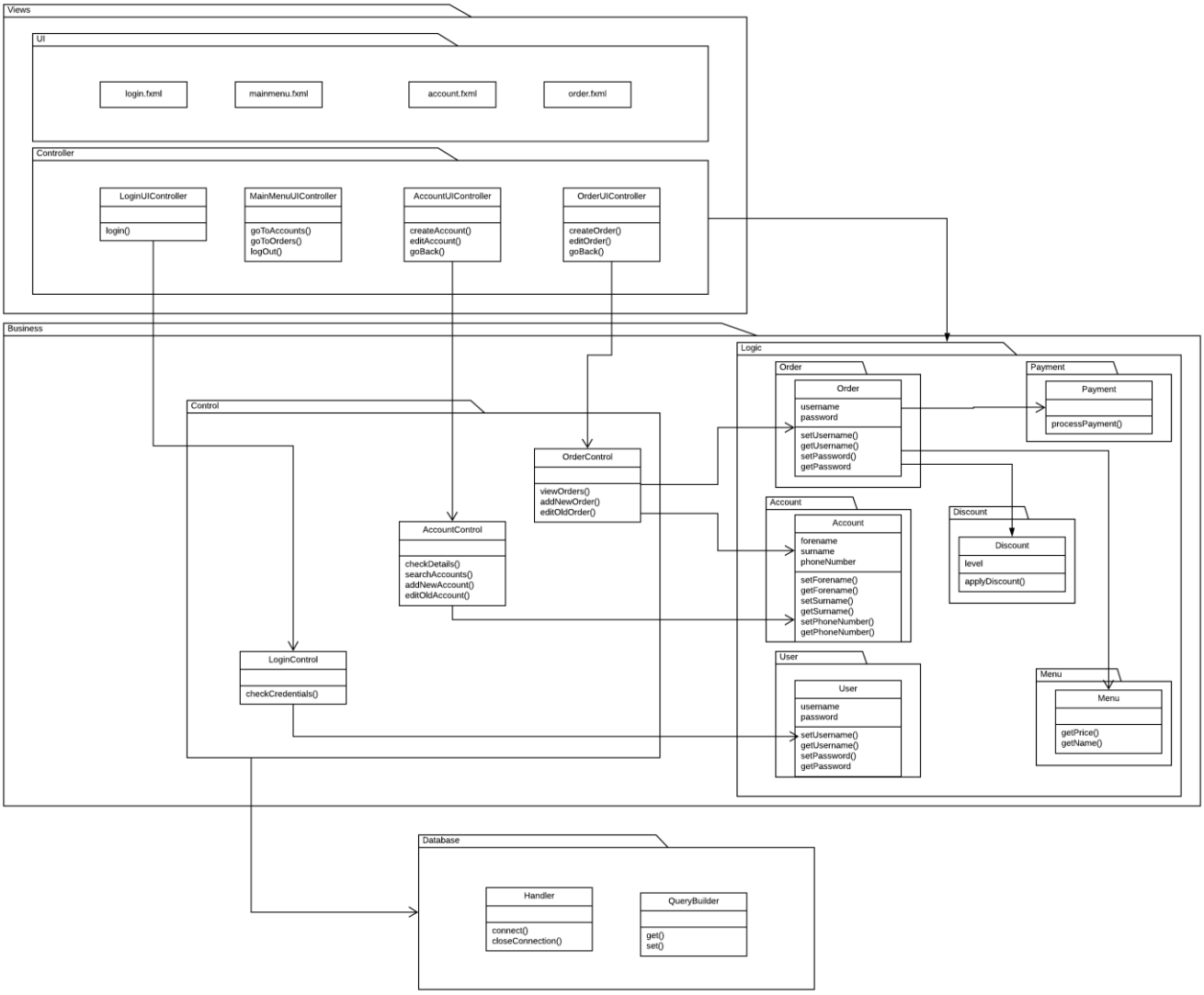


Figure 10. Analysis Time Class Diagram

Interaction Diagram – Sequence diagram

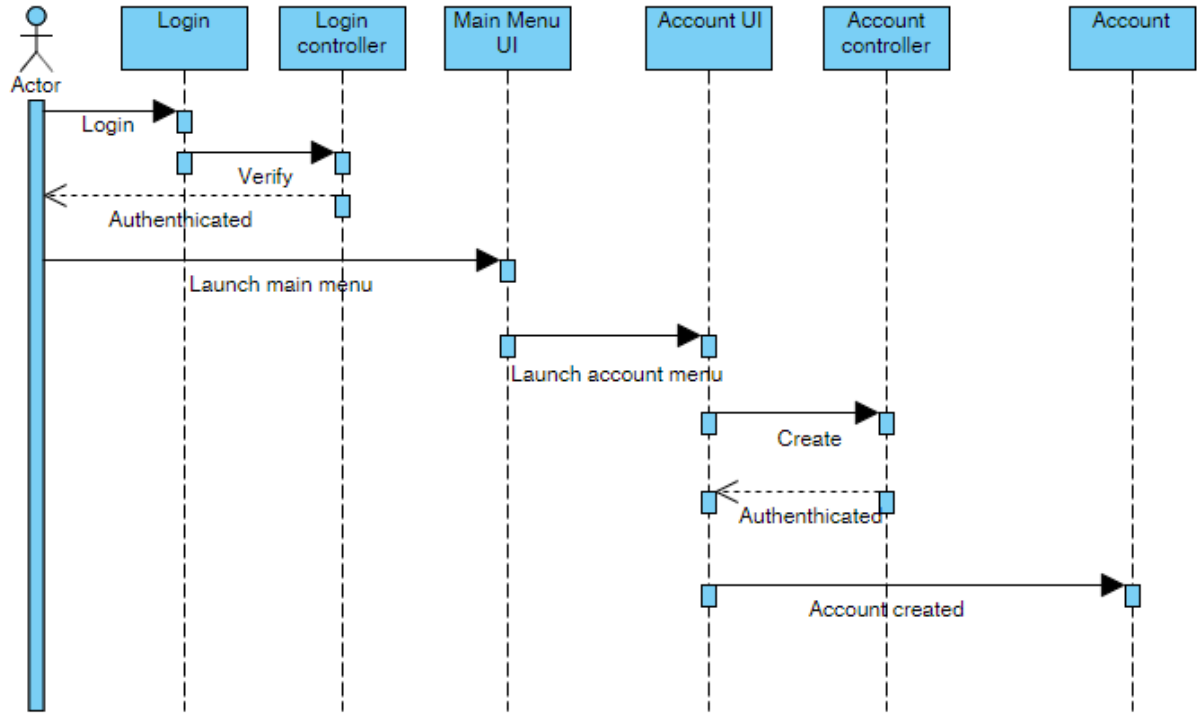


Figure 11. Interaction Diagram

Entity Relationship Diagram

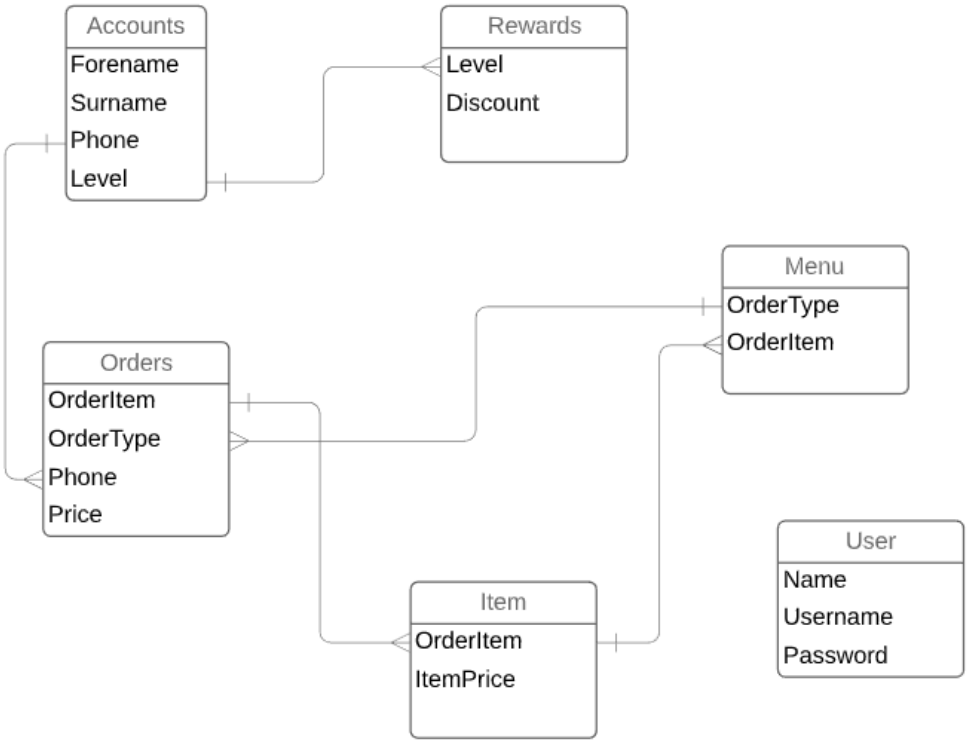


Figure 12. Entity Relationship Diagram

Code

Lines Of Code

Package	Class	Author(s)	LoC
business.model.account	Account	Neale	103
business.model.account.discount	BronzeDiscount	Neale	16
	DiscountFactory	Neale	17
	GoldDiscount	Neale	16
	IDiscount	Neale	6
	SilverDiscount	Neale	16
business.model.menu	ImenuItem	Ciaran	6
	Menu	Ciaran	58
business.model.menu.basic	Beverage	Ciaran	19
	Dessert	Ciaran	19
	Main	Ciaran	18
	Starter	Ciaran	19
business.model.menu.beverage	Beer	Ciaran	18
	Coke	Ciaran	18
	Fanta	Ciaran	18
business.model.menu.dessert	Cheesecake	Ciaran	18
	Icecream	Ciaran	18
	Pudding	Ciaran	18
business.model.menu.factory	MenuItemFactory	Ciaran	61
business.model.menu.main	Salad	Ciaran	18
	Salmon	Ciaran	18
	Steak	Ciaran	18
business.model.menu.starter	Ribs	Ciaran	18
	Soup	Ciaran	18
	Wings	Ciaran	18
business.model.order	Order	Ciaran	131
business.model.order.recommendation	Recommendation	Ciaran	34
business.model.order.state	ActiveState	Ciaran	15
	CompletedState	Ciaran	15
	OrderState	Ciaran	8
business.model.payment	PayByCard	Ciaran	11
	PayByCash	Ciaran	10
	PayContext	Ciaran	13
	PayStrategy	Ciaran	6
business.model.user	User	Ciaran	74
business.service	AccountService	Neale	144
	UserService	Ronan	124
	OrderService	Ciaran	122
data	AccountDaoSingleton	Neale	72
	IDao	Ciaran	8

	OrderDaoSingleton	Ciaran	70
	UserDaoSingleton	Ronan	70
restaurantms	RestaurantMS	Ciaran	25
ui.controller	AccountController	Neale	42
	CreateAccountController	Neale	71
	CreateOrderController	Ciaran	127
	CreateUserController	Ronan	77
	EditAccountController	Neale	67
	EditUserController	Ronan	14
	LoginController	Ciaran	48
	MainMenuController	Ciaran	36
	OrderController	Ciaran	37
	OrderDetailsController	Ciaran	81
	PaymentController	Ciaran	51
	UserController	Ronan	36
	ViewController	Ciaran	71
	ViewOrdersController	Ciaran	87
	ViewUsersController	Ronan	68
ui.view	Views	Ciaran	27
	account.fxml	Neale	22
	createuser.fxml	Ronan	44
	createaccount.fxml	Neale	38
	createorder.fxml	Ciaran	131
	editaccount.fxml	Neale	25
	edituser.fxml	Ronan	44
	login.fxml	Ciaran	30
	mainmenu.fxml	Ciaran	21
	order.fxml	Ciaran	21
	orederdetails.fxml	Ciaran	52
	payment.fxml	Ciaran	17
	user.fxml	Ronan	21
	vieworders.fxml	Ciaran	17
	viewusers.fxml	Ronan	17

Name	**Lines of Code**
Ciaran	1812
Neale	655
Rona	515

Packages	22
Classes/Files	73
Lines of Code	2982

GUI Screenshots

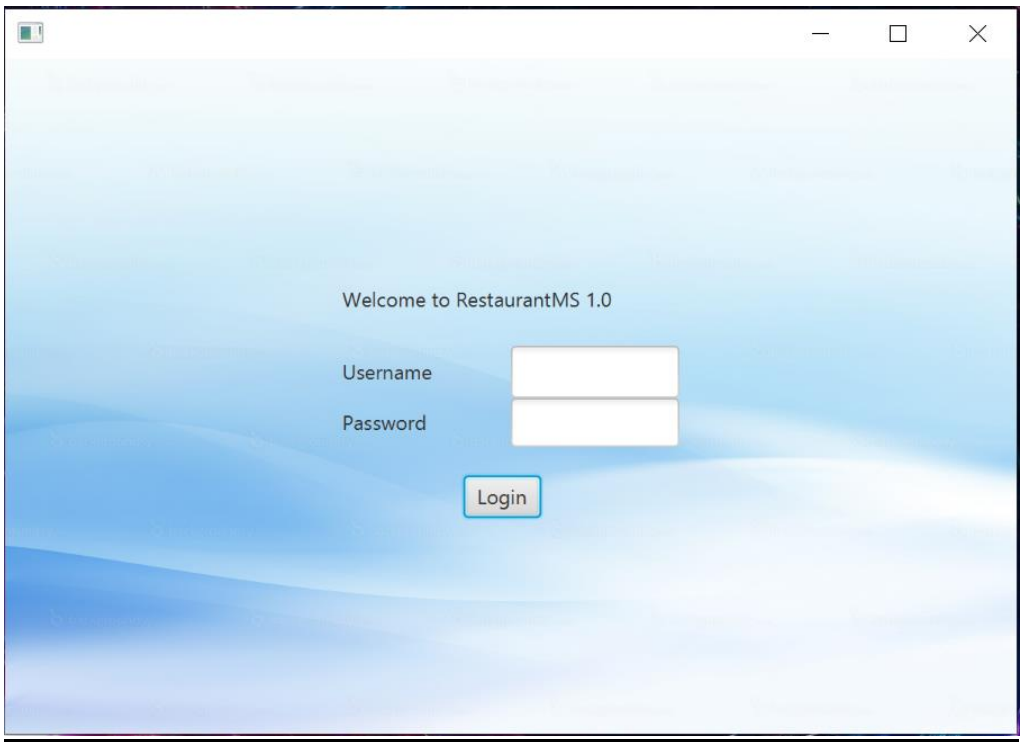


Figure 13. Start Screen

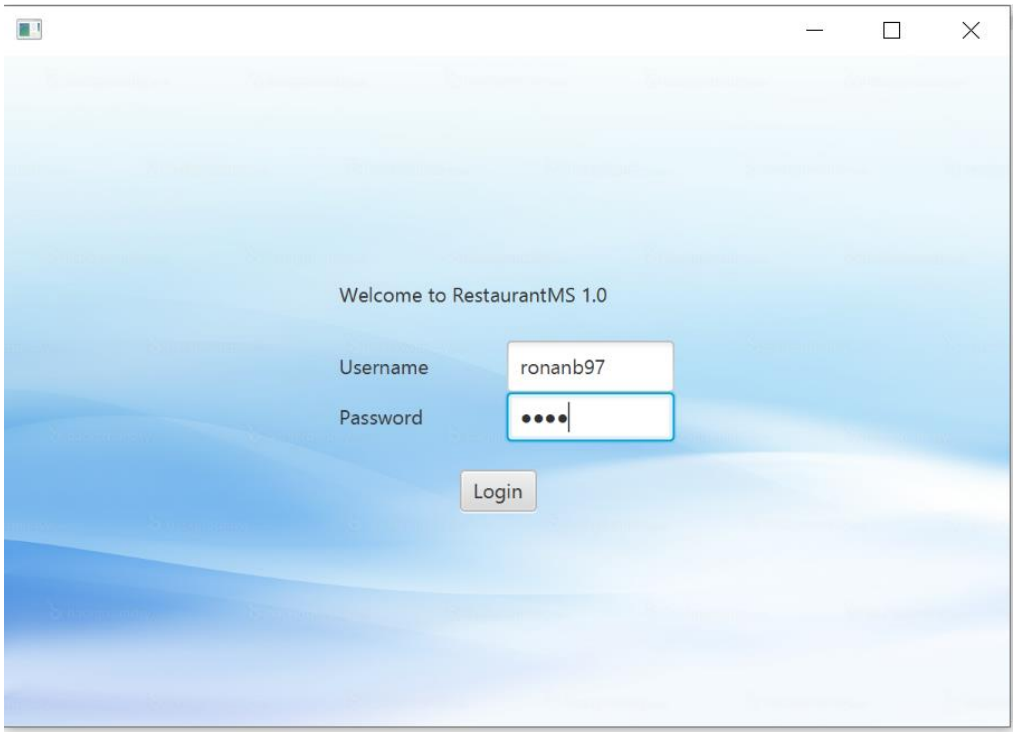


Figure 14. Login Window – With details

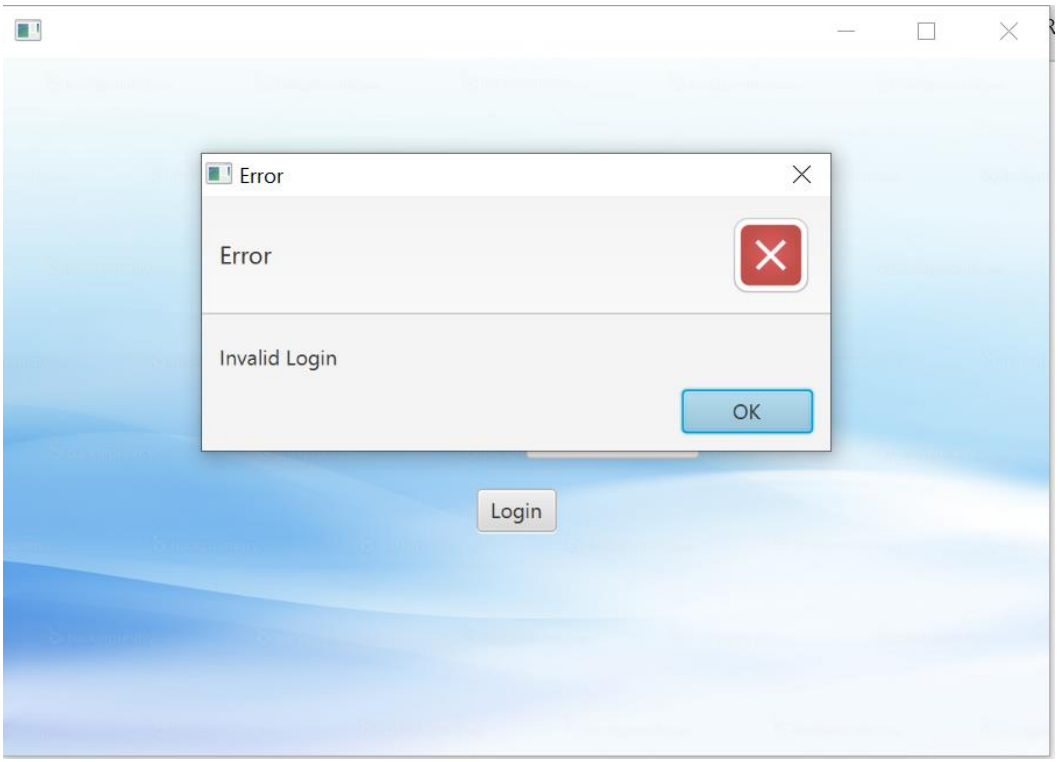


Figure 15. Login Window Fail

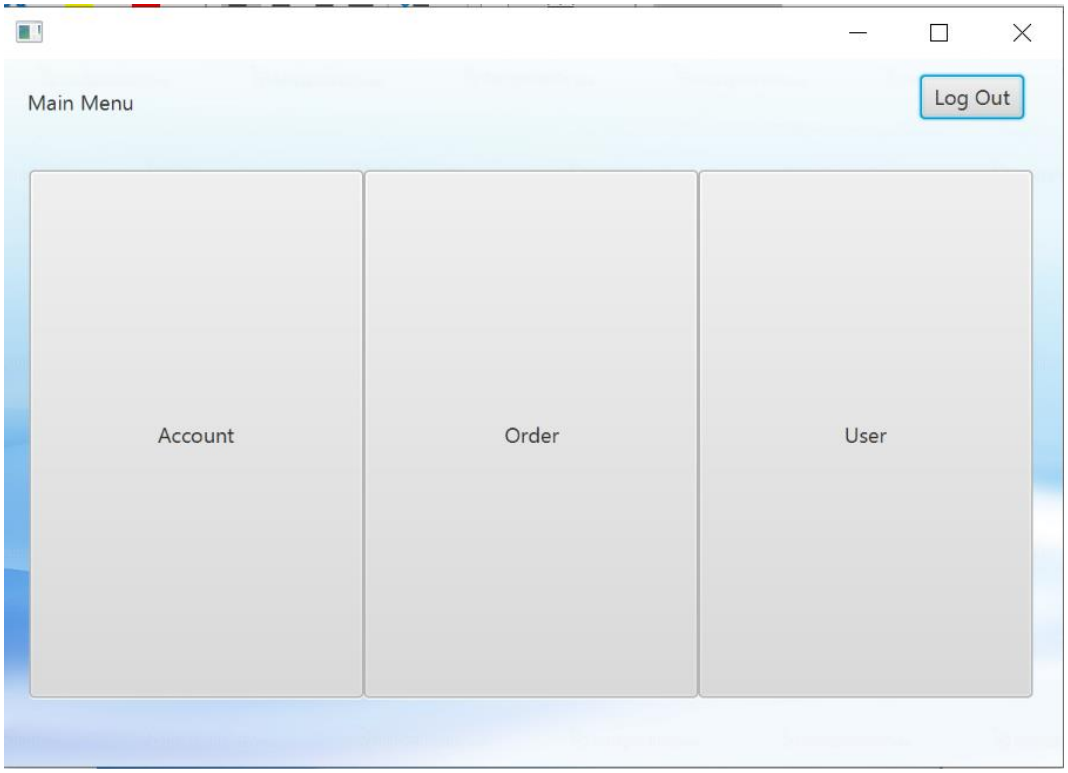


Figure 16. Main Menu Window

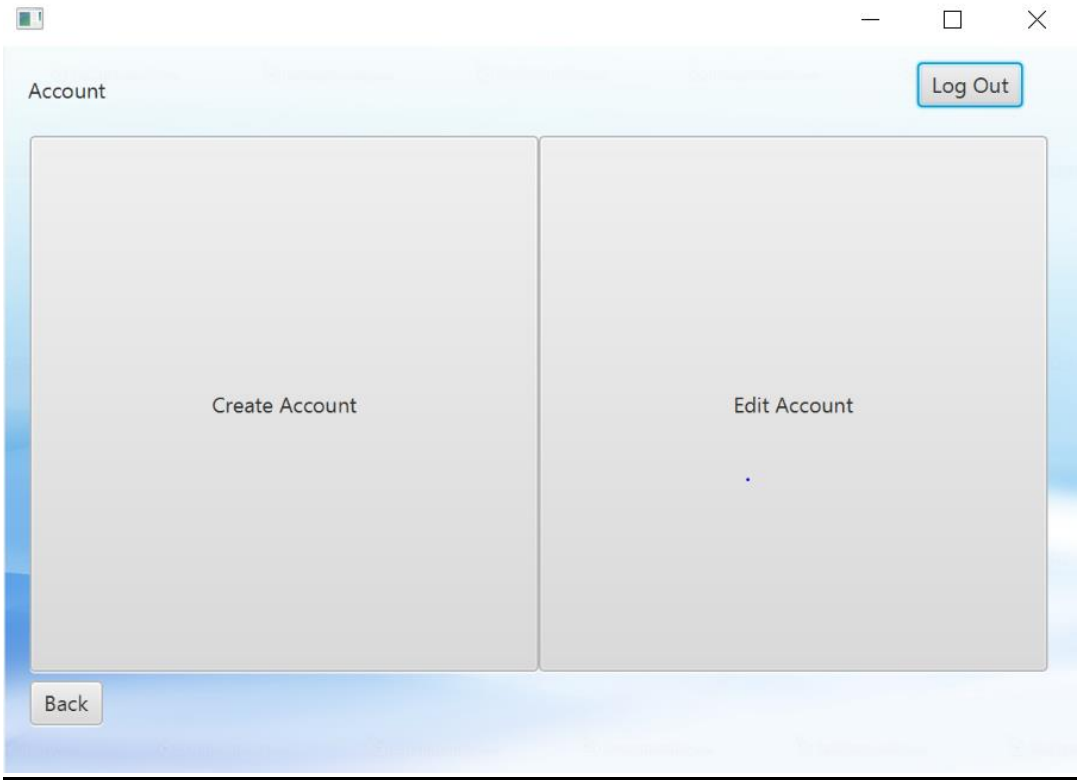


Figure 17. Account Window

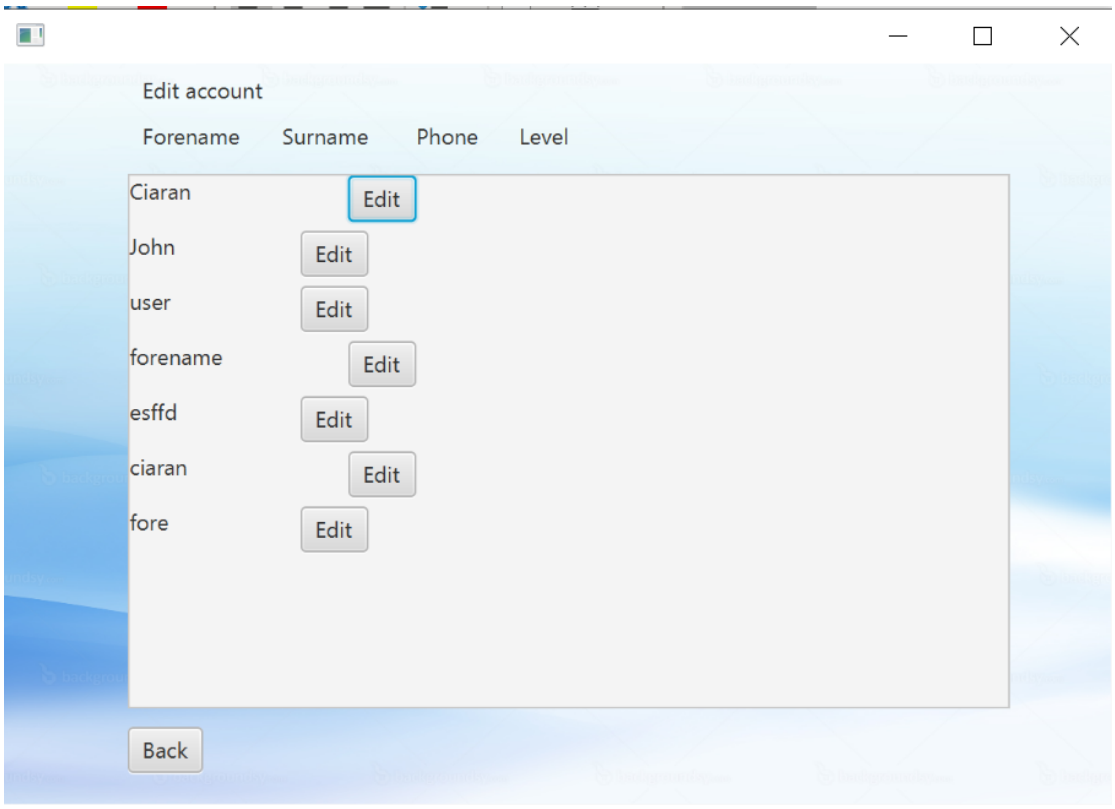


Figure 18. Edit Account Window

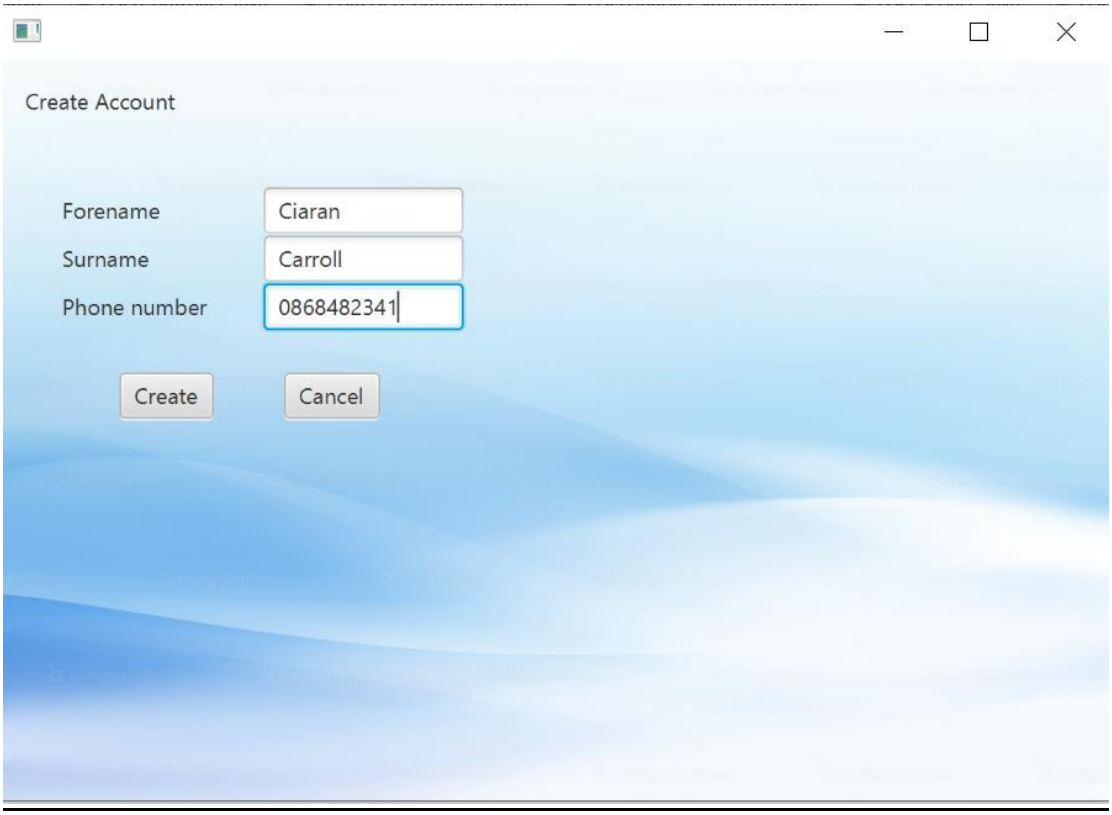


Figure 19. Create Account Window – With Details

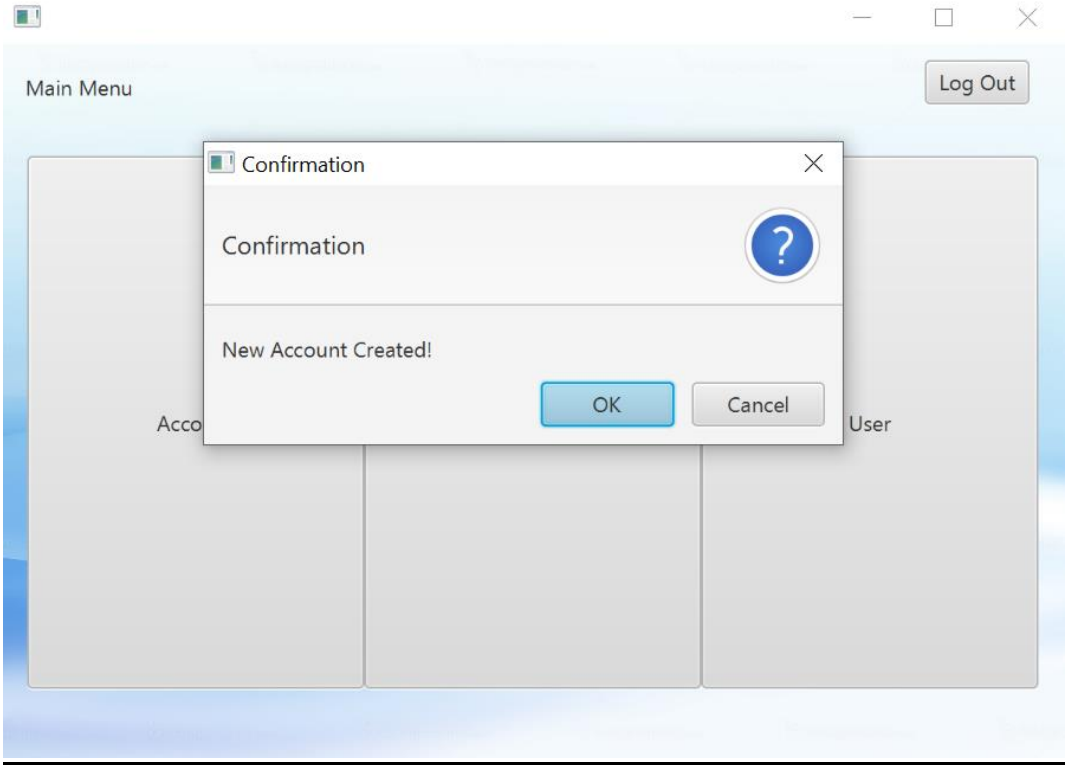


Figure 20. Successful Account Created Alert

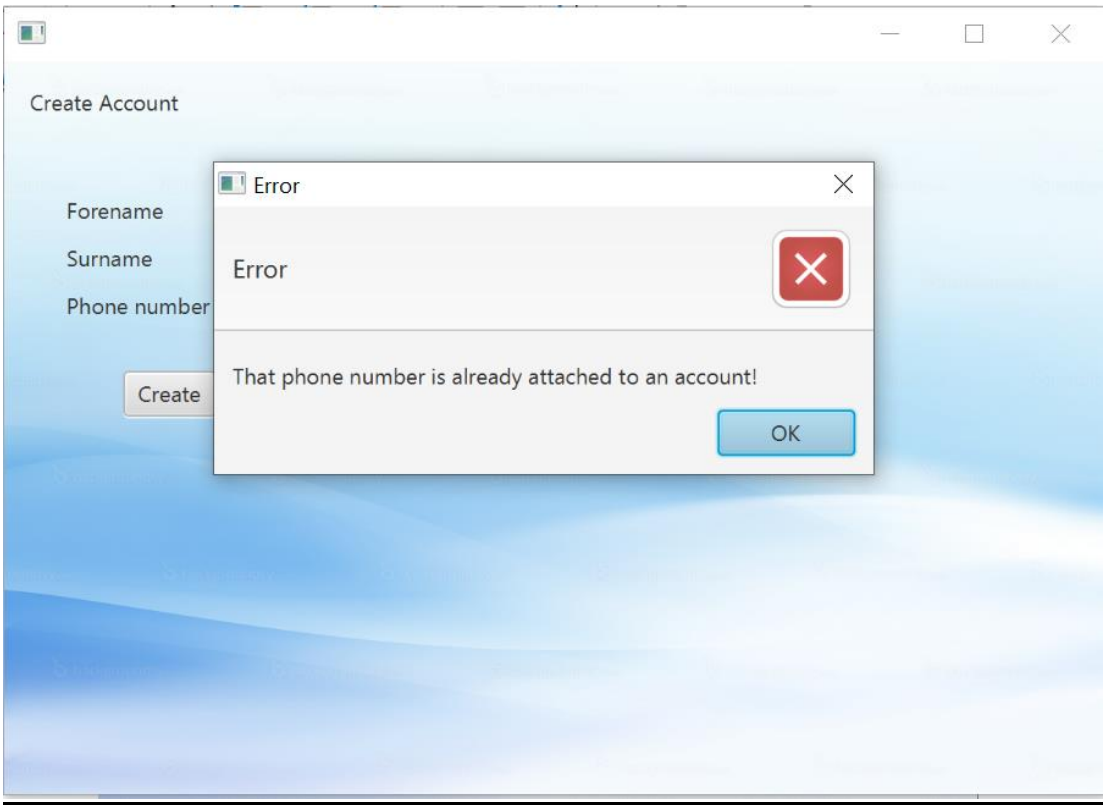


Figure 21. Account Creation Error

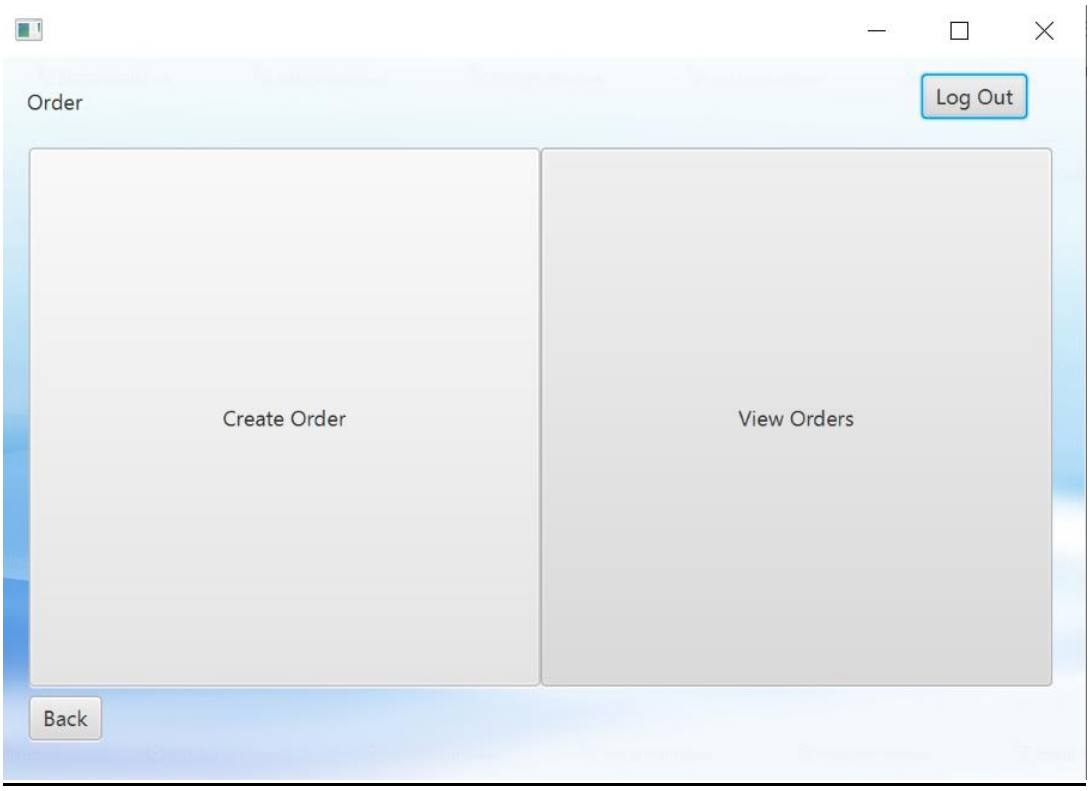


Figure 22. Order Window

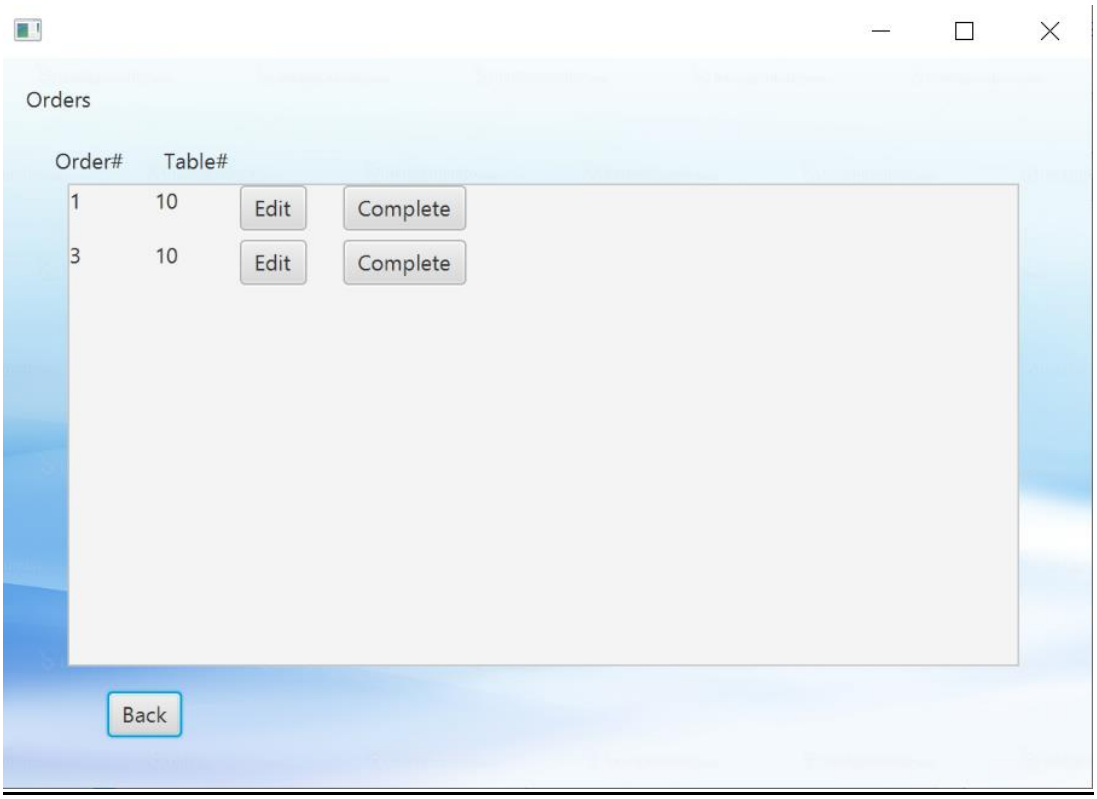


Figure 23. View Current Orders Window

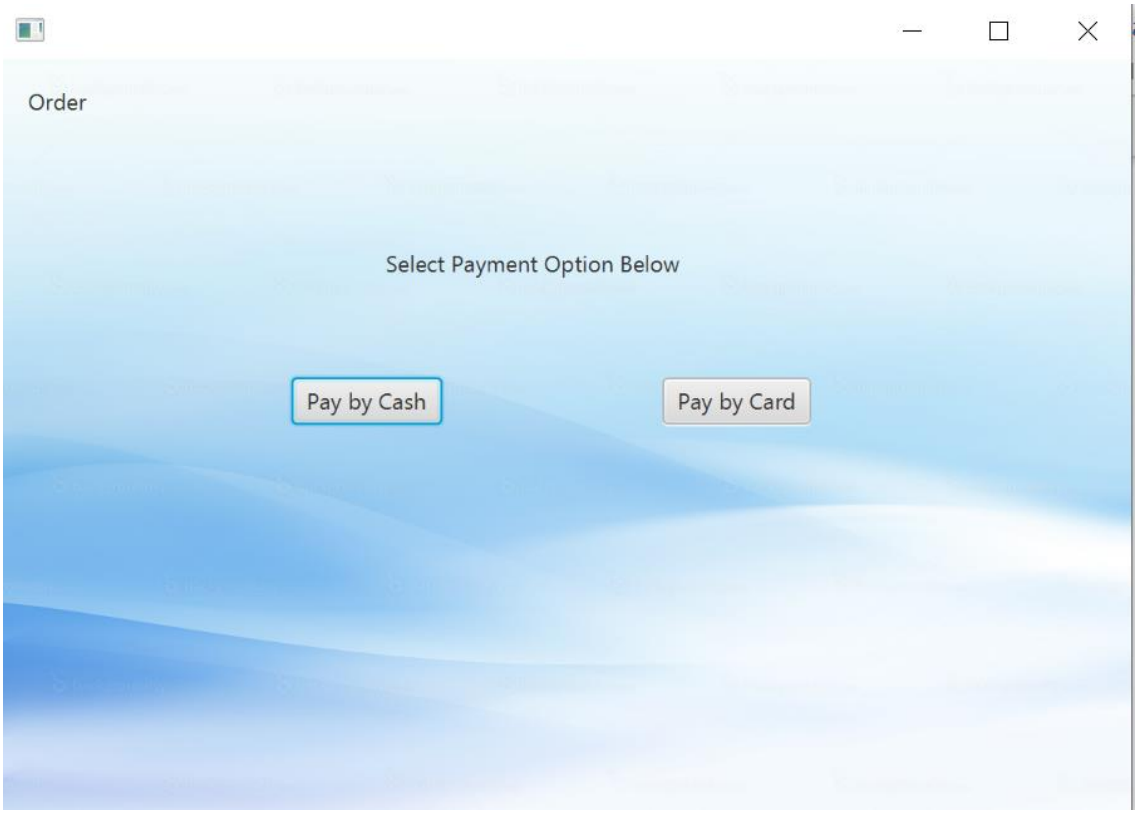


Figure 24. Payment Window

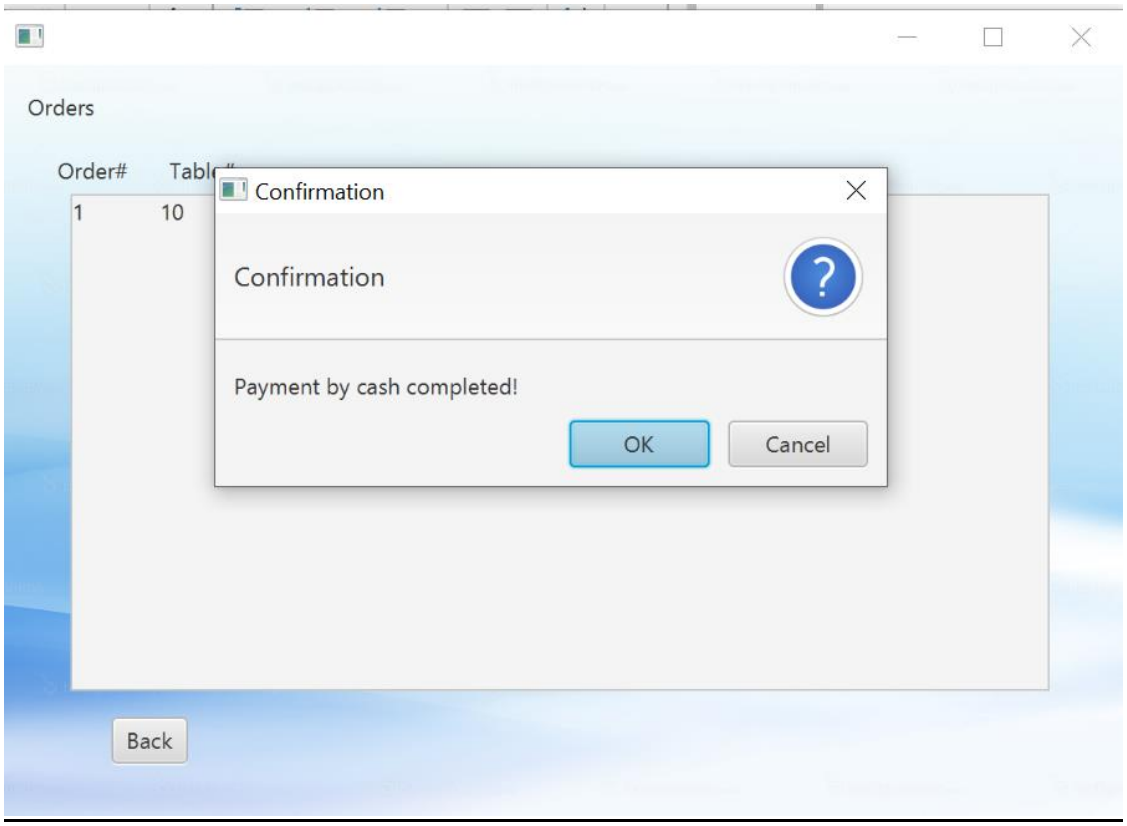


Figure 25. Successful Payment With Cash Alert

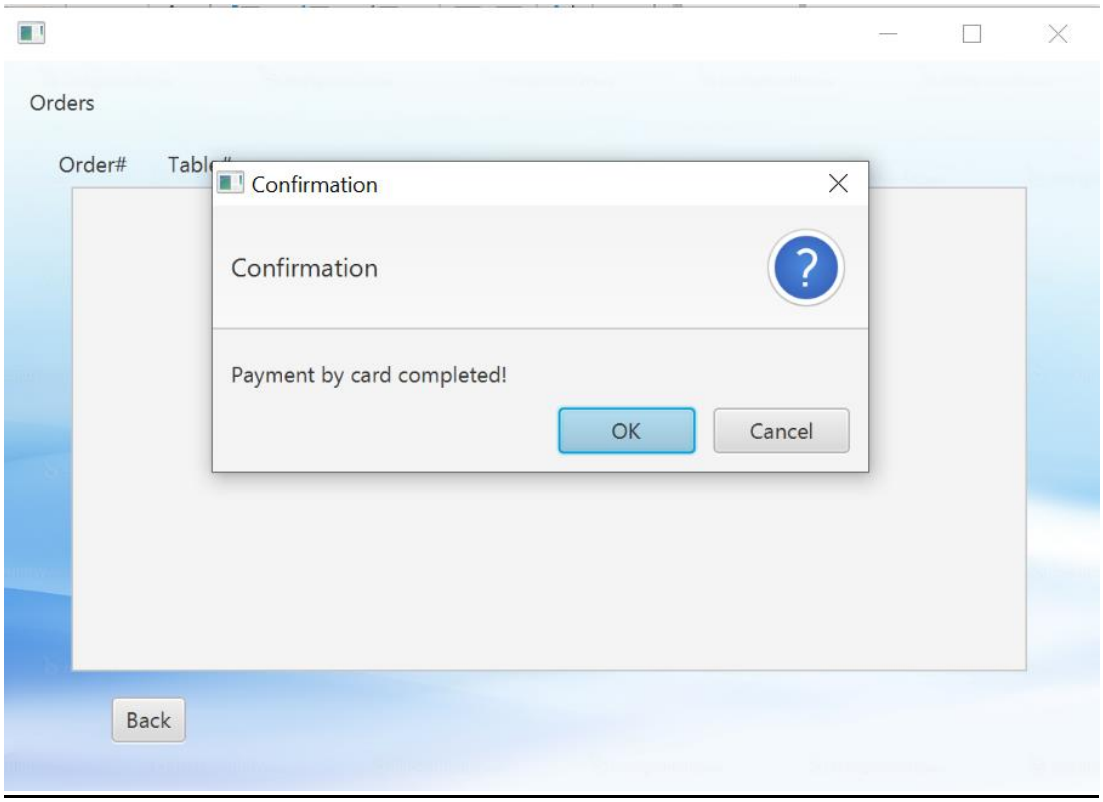


Figure 26. Successful Payment With Card Alert

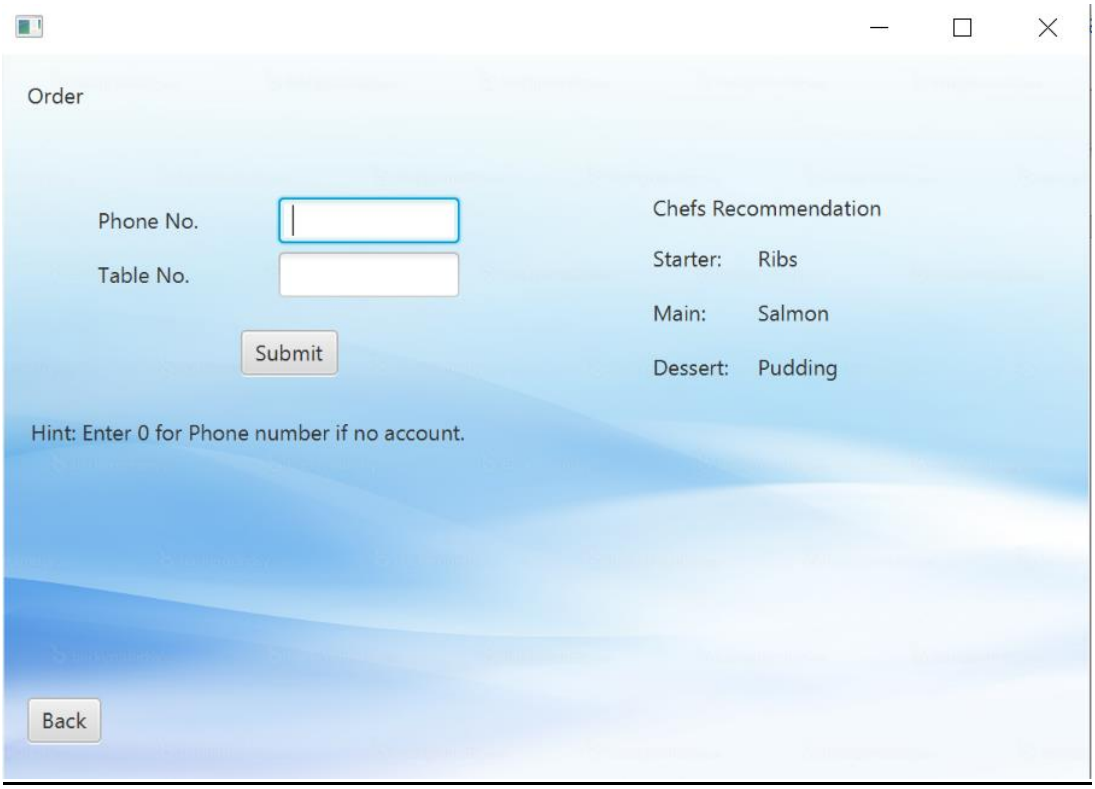


Figure 27. Assign Table Window + Recommendations

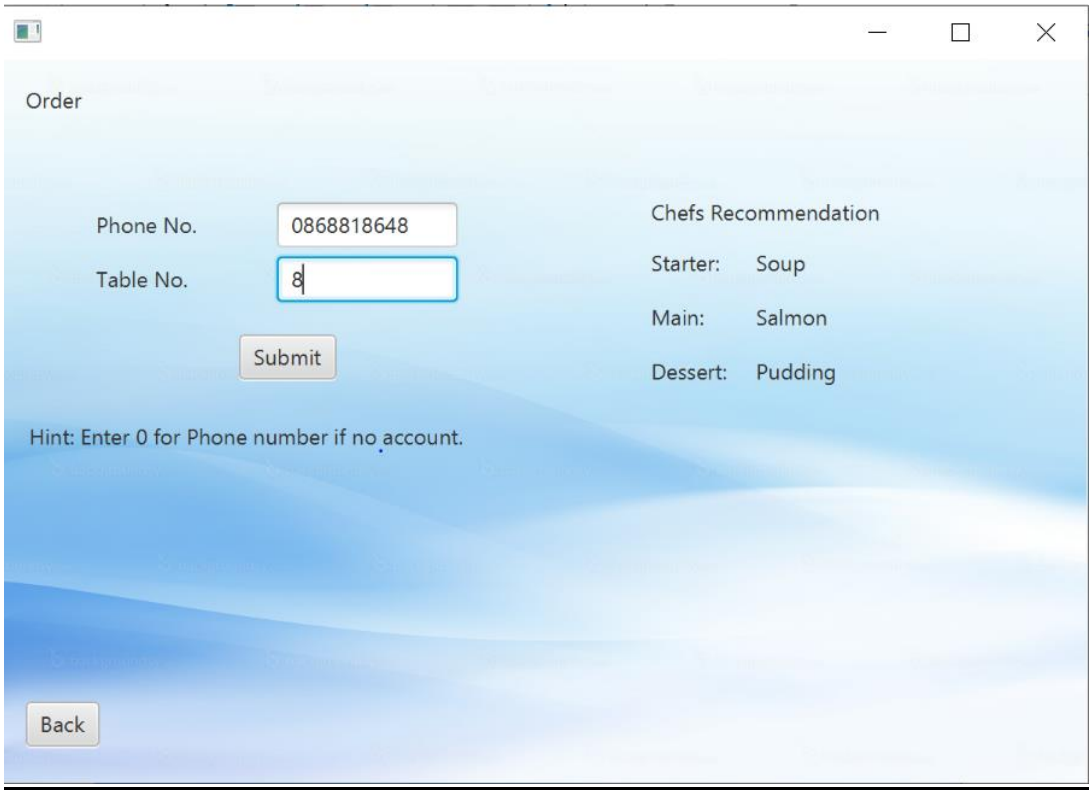


Figure 28. Assign Table Window + Recommendations – With details

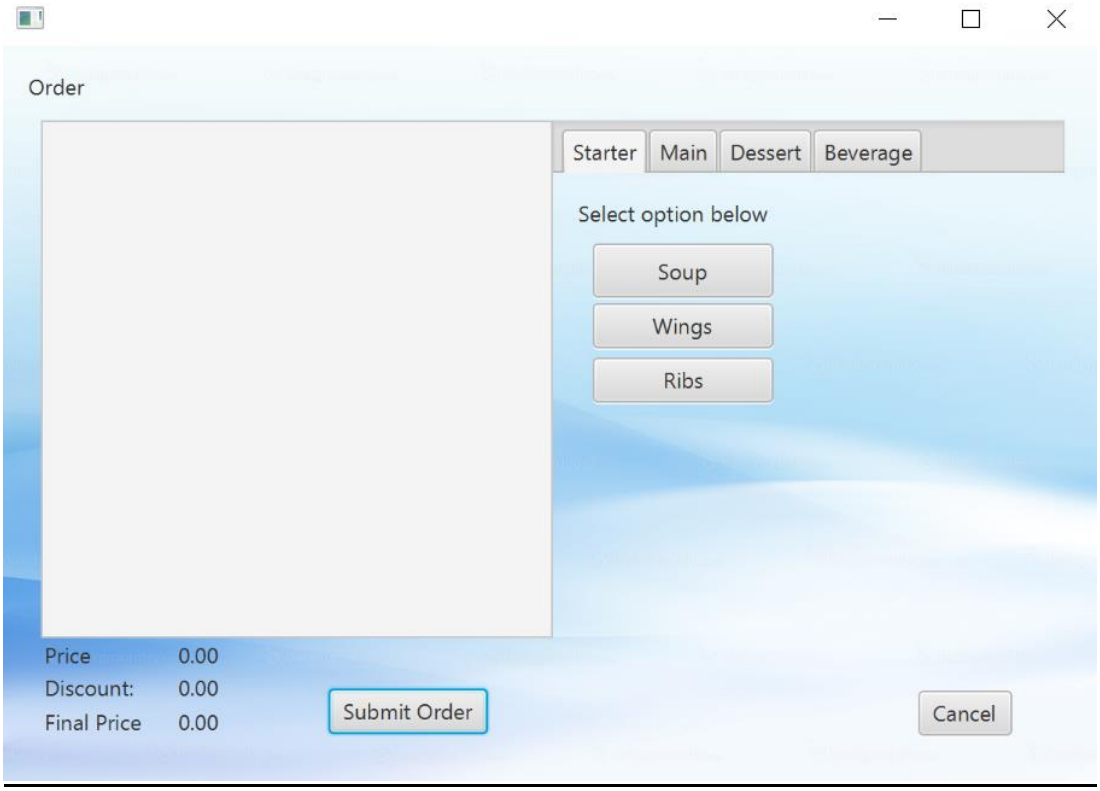


Figure 29. Menu Window/Order Creation Window

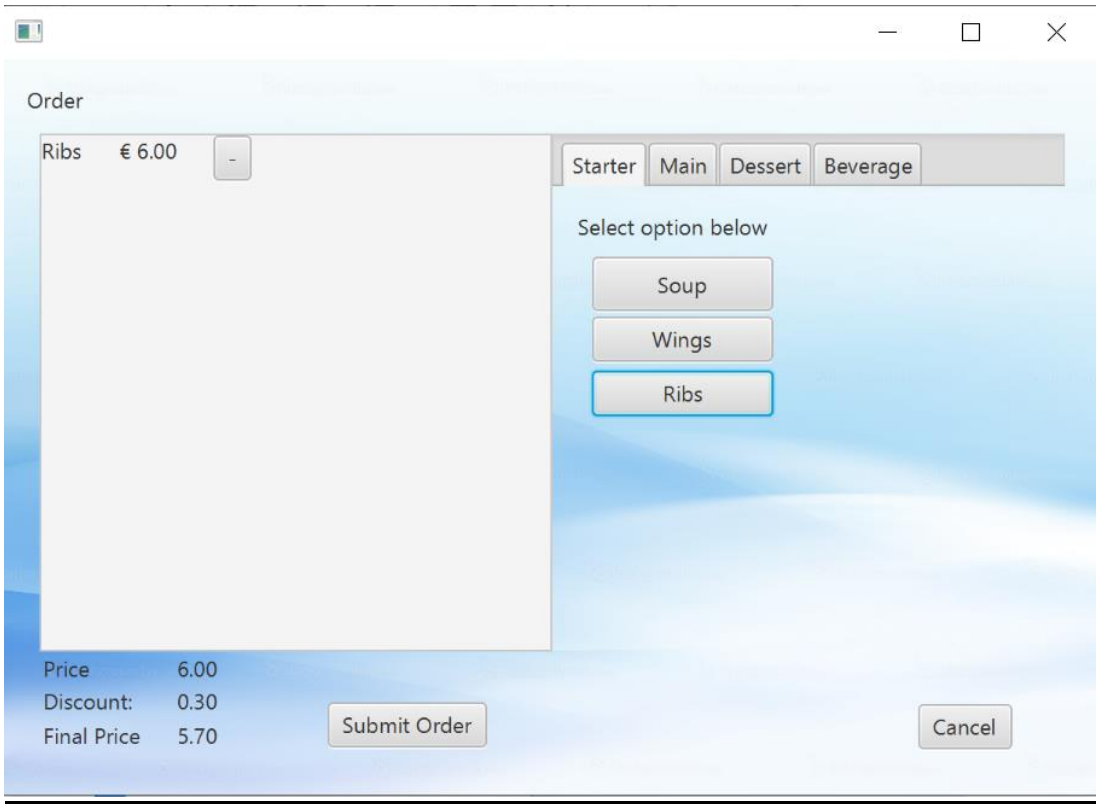


Figure 30. Starter Selection

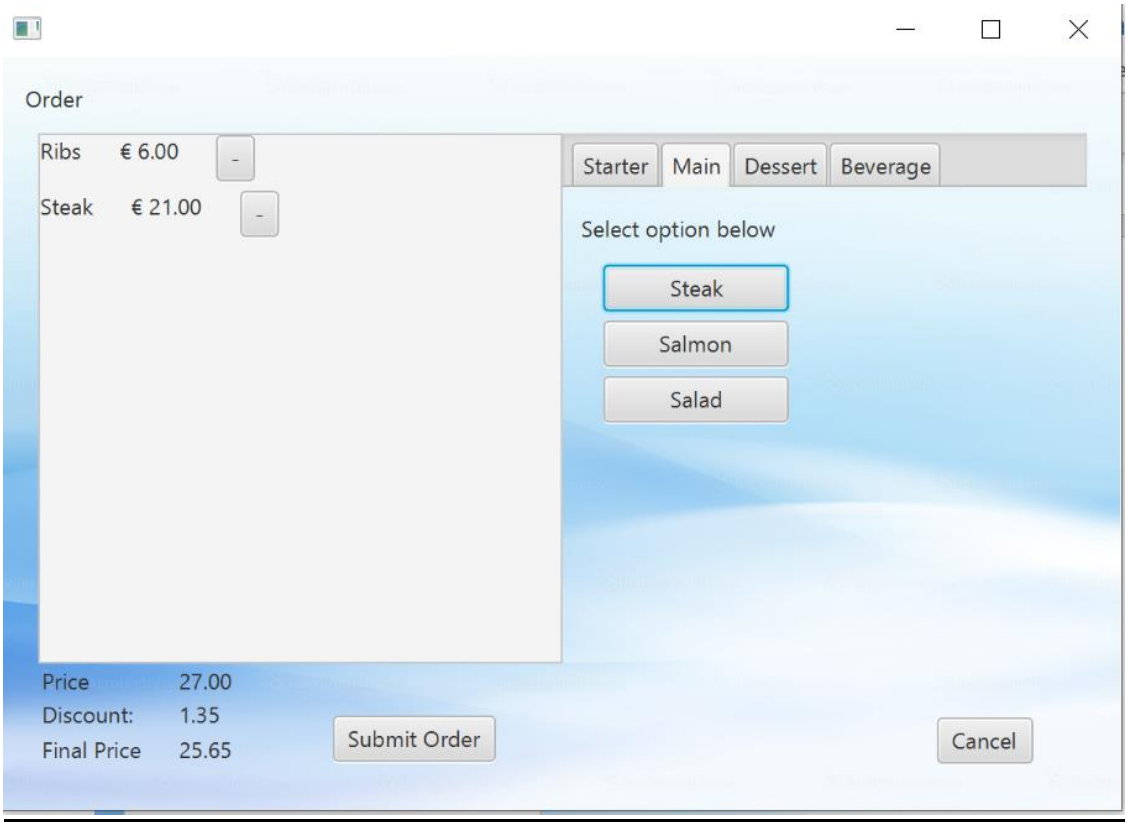


Figure 31. Main Selection

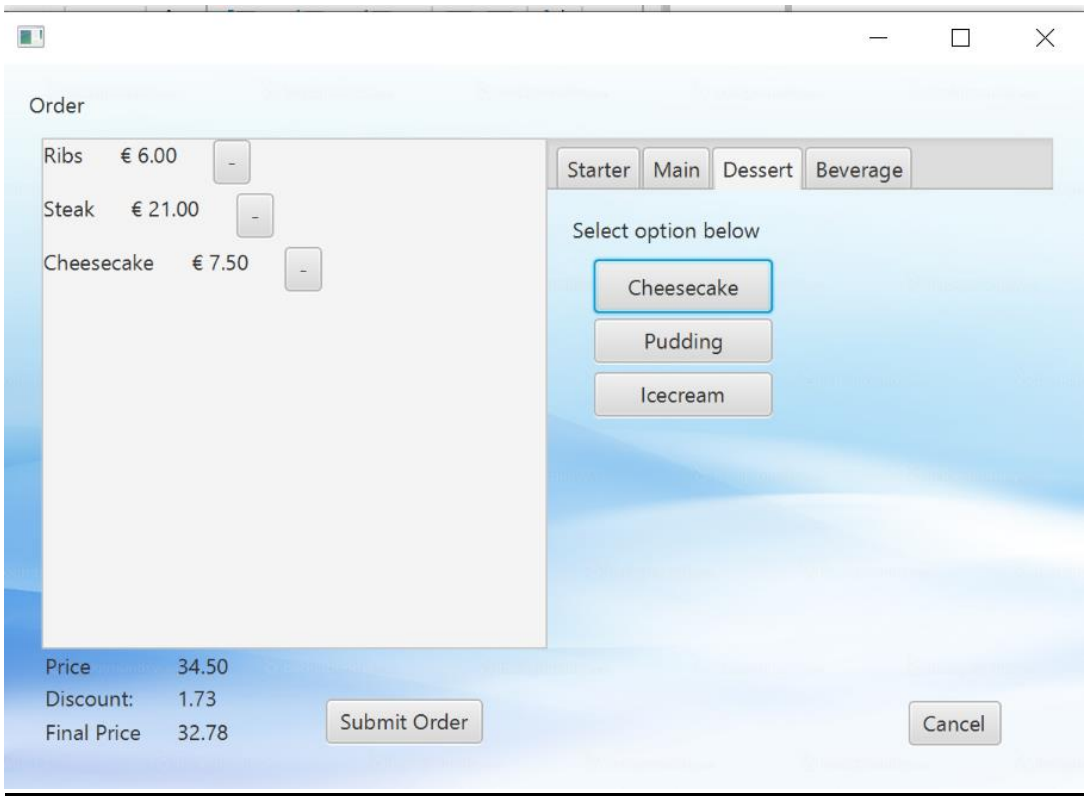


Figure 32. Desert Selection

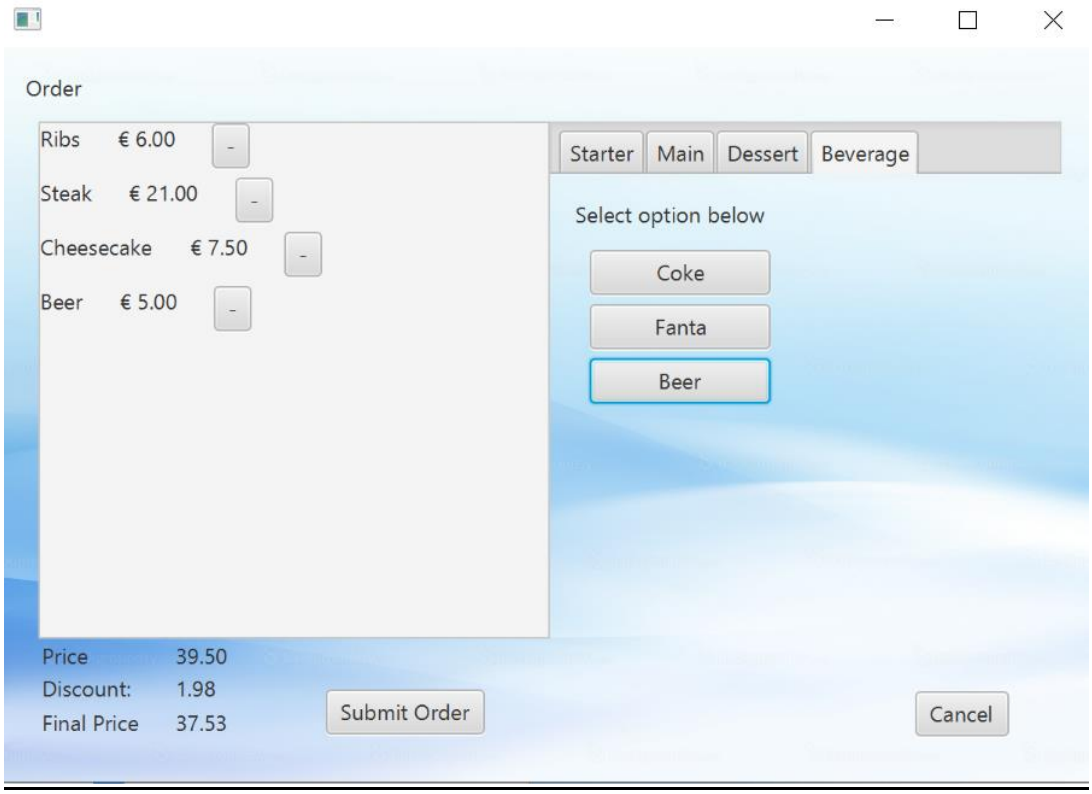


Figure 33. Beverage Selection

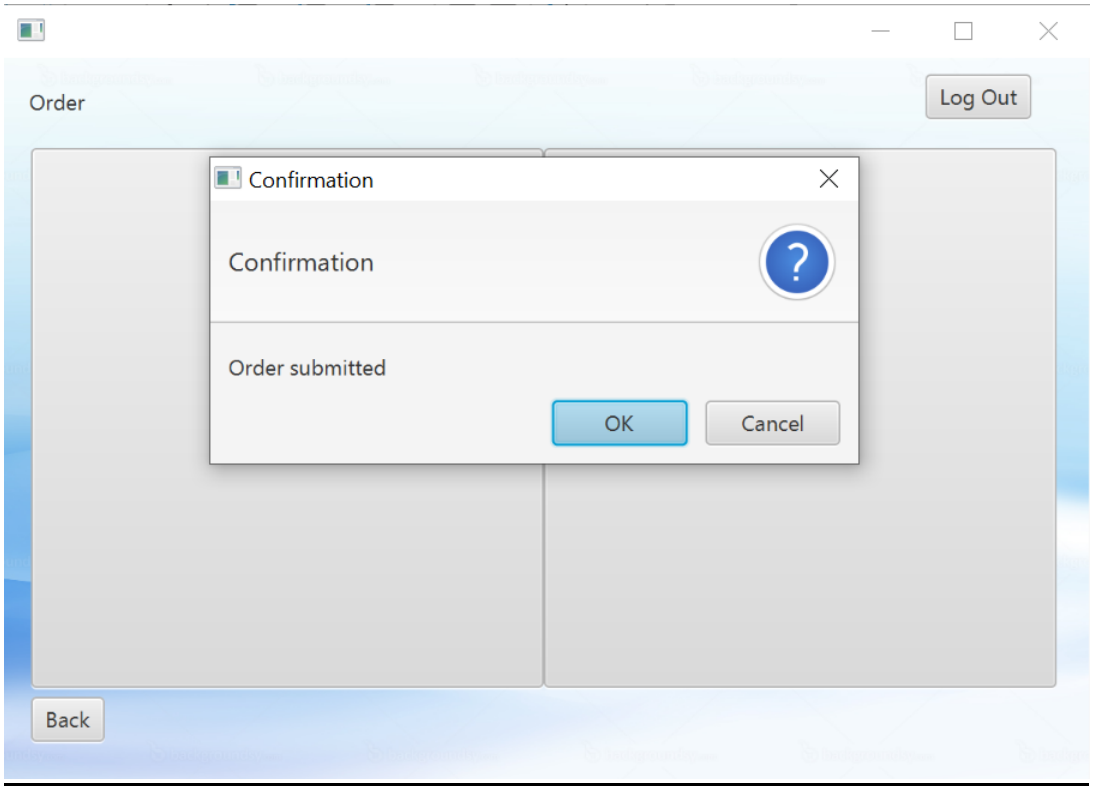


Figure 34. Successful Order Created and Submitted

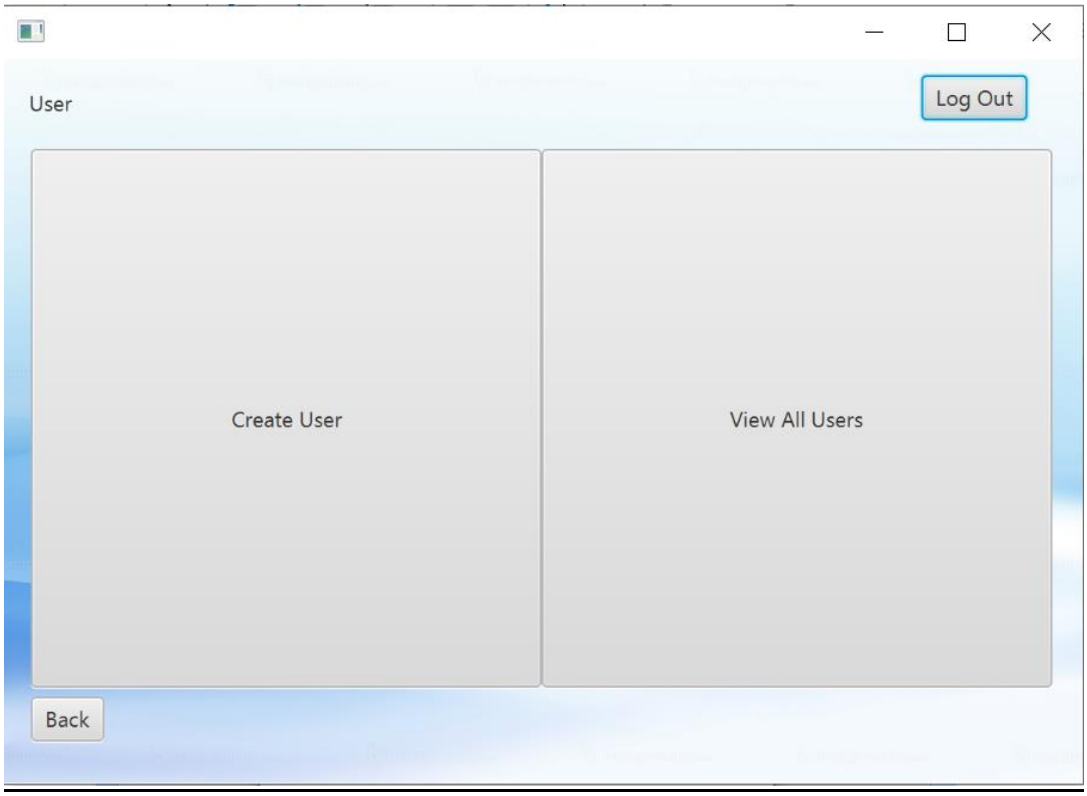


Figure 35. User Window

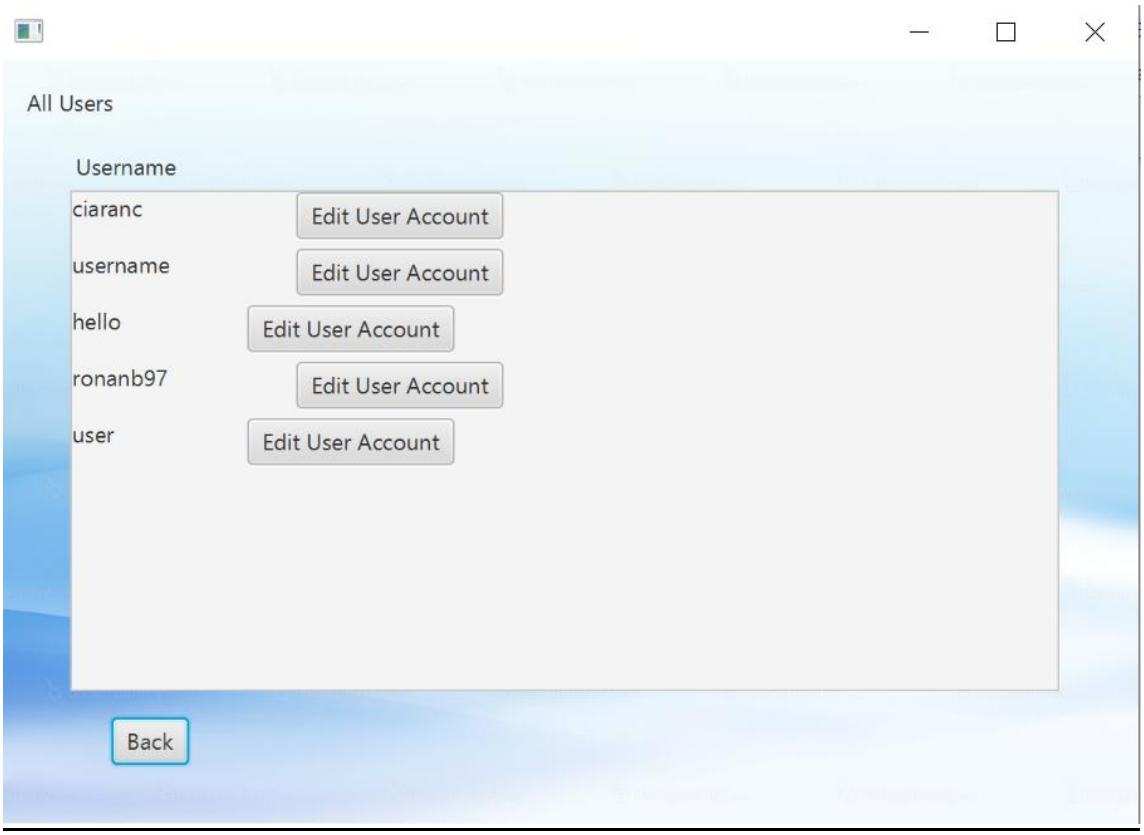


Figure 36. View All Users Window

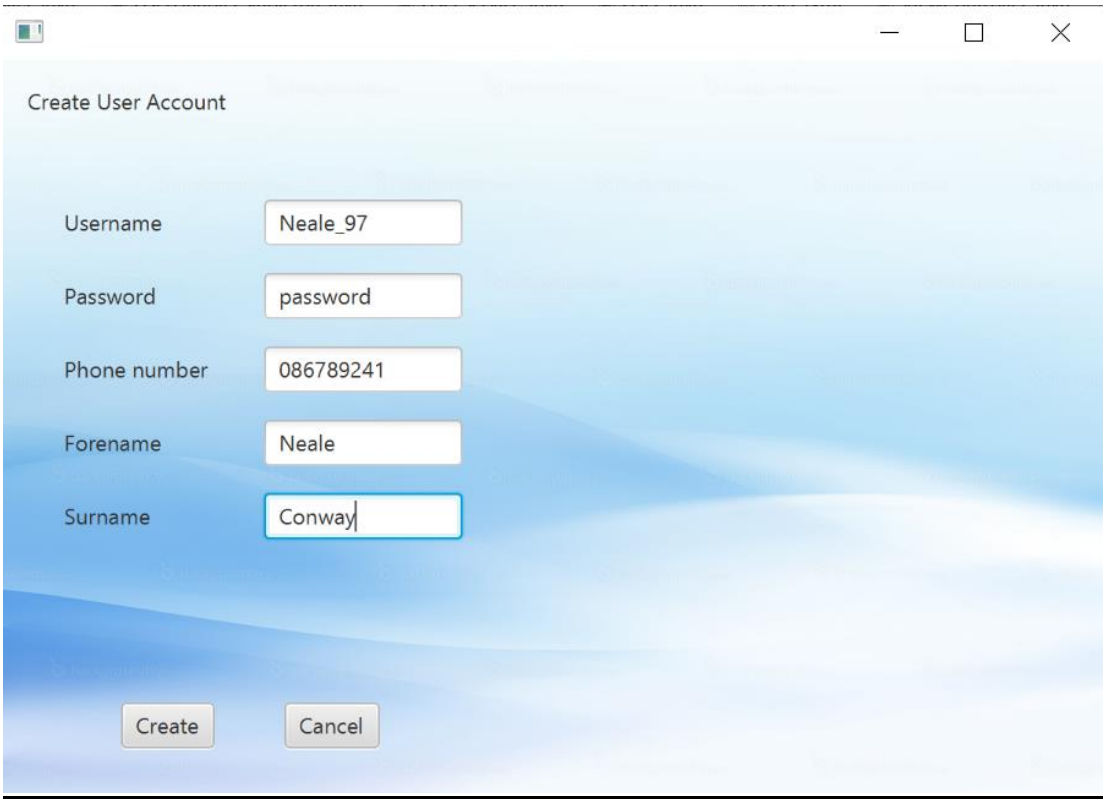


Figure 37. Create User Window – With details

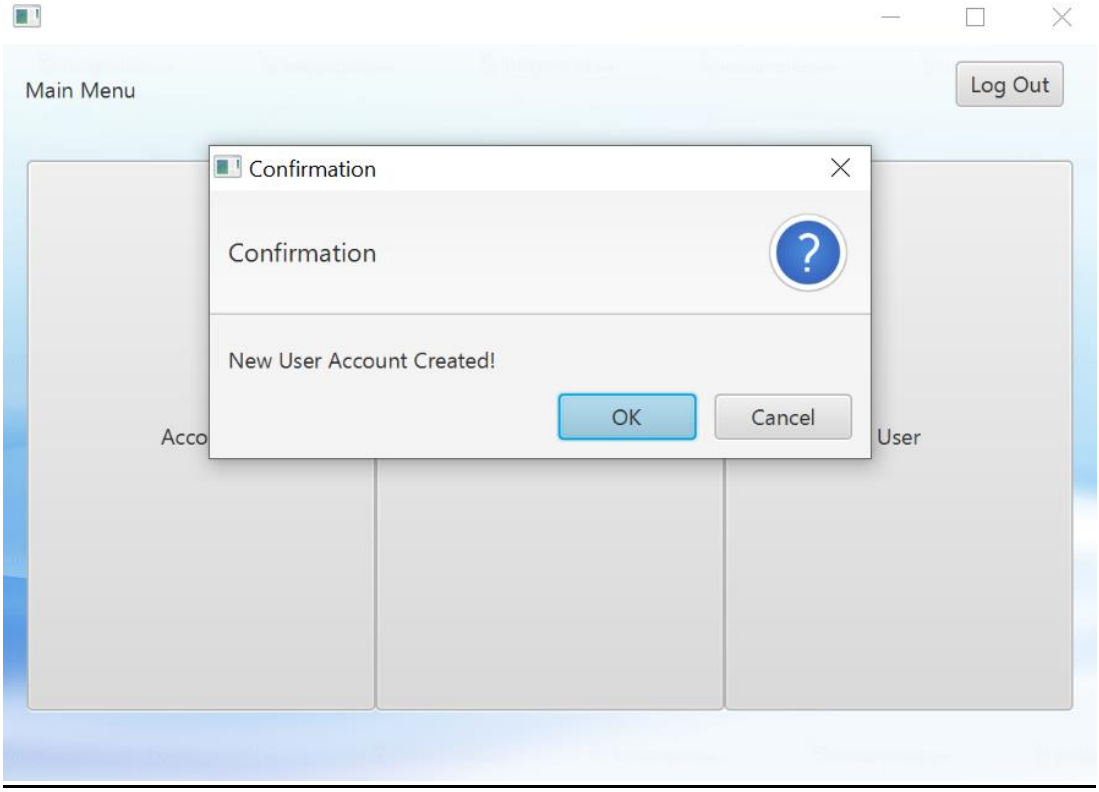


Figure 38. Successful User Created Alert

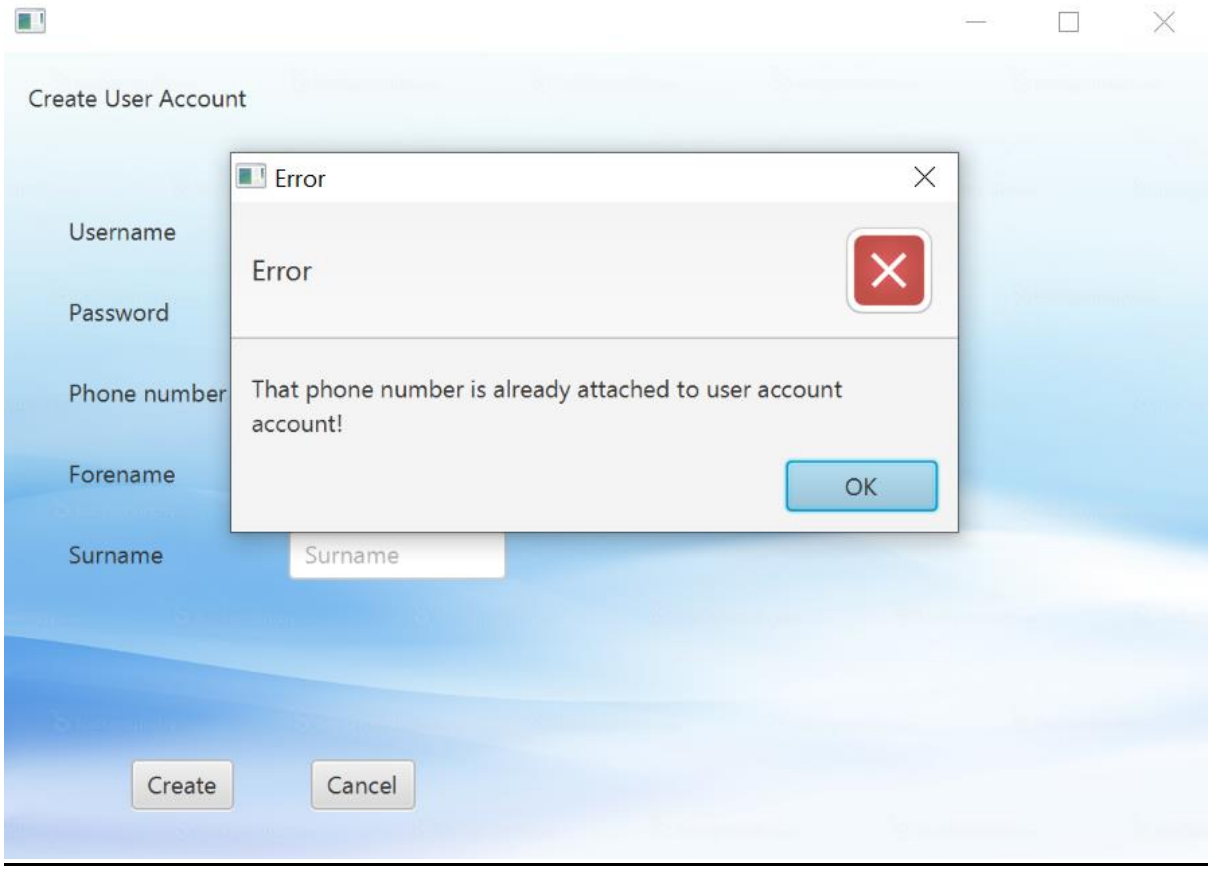


Figure 39. Unsuccessful Creation of User Account

Design Patterns

Singleton Pattern

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class. We used this design pattern for our dao classes because it made sense that only a single connection to the data was created. Example of this can be seen below.

```
public class AccountDaoSingleton implements IDao{

    private static AccountDaoSingleton connector;


    public static AccountDaoSingleton getInstance()
    {
        if (connector==null)
        {
            connector = new  AccountDaoSingleton();
        }
        return connector;
    }

    private static File getConnection()
    {
        File con = new File("Accounts.txt");
        return con;

    }

    @Override
    public ArrayList<String> getData()
    {
        File con = connector.getConnection();

        ArrayList<String> strings = new ArrayList<>();
        String lineFromFile;
        try {
            Scanner in = new Scanner(con);
            while(in.hasNext())
            {
                lineFromFile = in.nextLine();
                strings.add(lineFromFile);
            }
            in.close();
        } catch (FileNotFoundException ex) {
            Logger.getLogger(OrderDaoSingleton.class.getName(
        }
        return strings;
    }
```

Figure 40.

Factory Method Pattern

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. We used this pattern to create our MenuItem and Discount objects when they were accessed from our database. The factories were able to turn the strings stored into objects. The Account factory can be seen below.

```
public class DiscountFactory {
    public IDiscount getDiscount(String name) {
        switch(name)
        {
            case "Bronze":
                return new BronzeDiscount();
            case "Silver":
                return new SilverDiscount();
            case "Gold":
                return new GoldDiscount();
            default:
                return null;
        }
    }
}
```

Figure 41.

Strategy Pattern

In Strategy pattern, we create objects which represent various strategies and a context object whose behaviour varies as per its strategy object. The strategy object changes the executing algorithm of the context object. We implemented this strategy as part of our Payment package. This allowed us to decide between PayByCard and PayByCash at runtime. Snippets of this seen below.

```
public interface PayStrategy {

    public void receivePayment(double price);
}


public class PayByCash implements PayStrategy{

    @Override
    public void receivePayment(double price)
    {
        //Add new Cash payment to database
    }
}
public class PayContext {
   private PayStrategy strategy;

  public void setPaymentStrategy(PayStrategy strategy) {
    this.strategy = strategy;
  }

  public void pay(double price) {
    strategy.receivePayment(price);
  }
}
```

Figure 42.

State Pattern

State design pattern is used when an Object changes its behaviour based on its internal state. If we have to change the behaviour of an object based on its state, we can have a state variable in the Object and use if-else condition block to perform different actions based on the state. State pattern is used to provide a systematic and lose-coupled way to achieve this through Context and State implementations. This was implemented in our Order class where orders could have an Active or Completed state. The state classes had a method within which would change the order variable to the opposite case when called. Snippets seen below.

```
public void changeState()
{
    state.change(this);
}
public class ActiveState implements OrderState{

    @Override
    public void change(Order order){
        order.setState(new CompletedState());
    }
    @Override
    public int getStatus(){
        return 0;
    }
}
```

Figure 43.

MVC Pattern

The Model View Controller (MVC) design pattern specifies that an application consist of a data model, presentation and controller. The pattern requires that each of these be separated into different objects. The Model contains only the pure application data, it contains no logic describing how to present the data to a user. The View presents the model's data to the user. The view knows how to access the model's data, but it does not know what this data means or what the user can do to manipulate it. The Controller exists between the view and the model. It listens to events triggered by the view and executes the appropriate reaction to these events. In most cases, the reaction is to call a method on the model. For our system the view was the FXML files which presented the data to the user. The Controller was that controllers designated to all these FXML files and these controllers interacted with the Model and the Service classes.

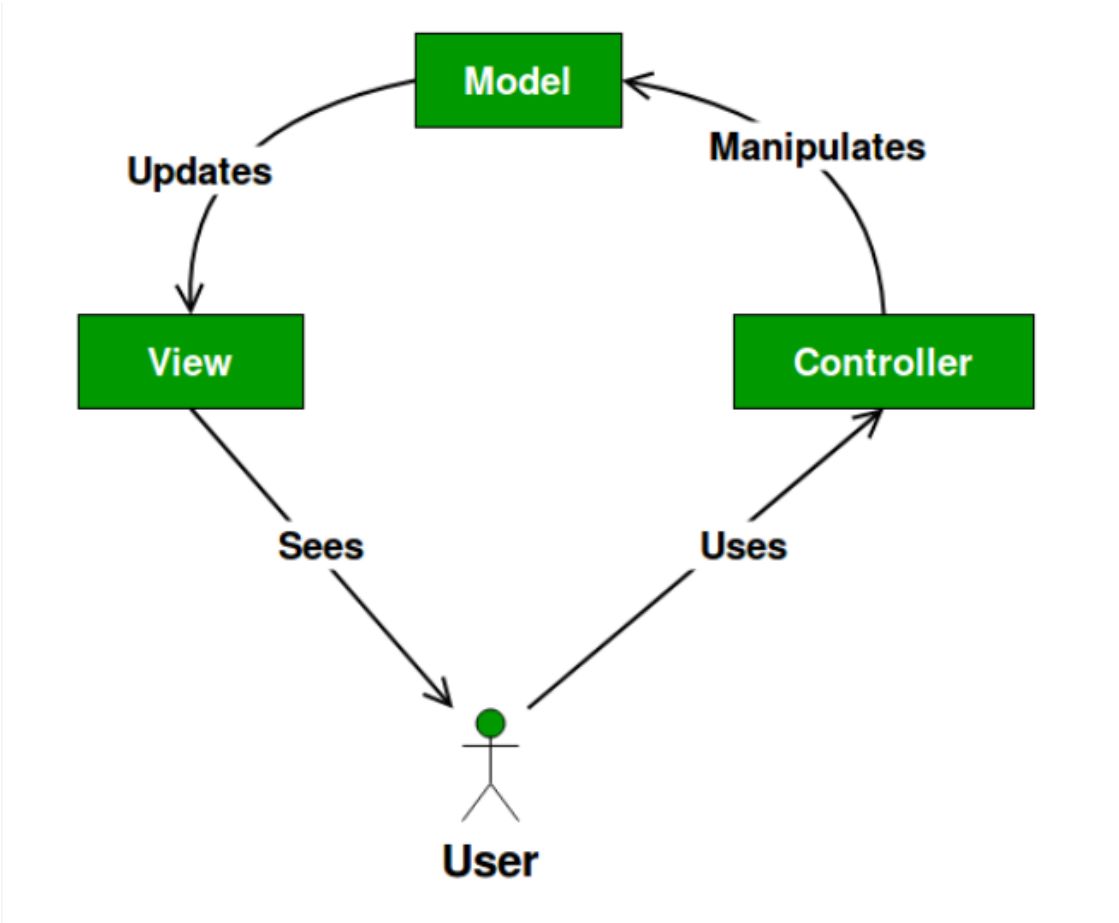


Figure 44.

GitHub

We used Github to control the versions of our system. GitHub is a website and cloud-based service that helps developers store and manage their code, as well as track and control changes to their code. Contribution tab for repository seen below.

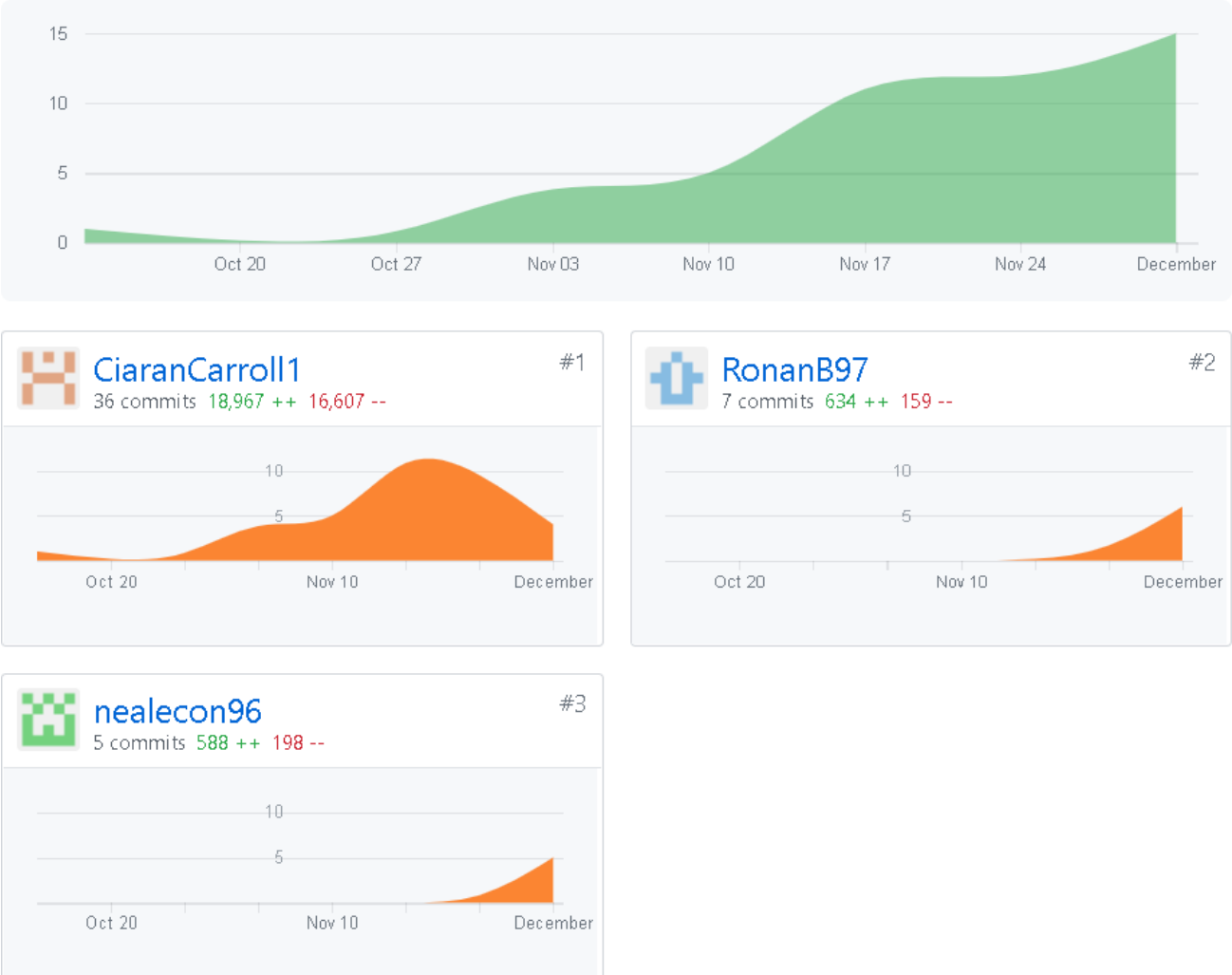


Figure 45.

Note: Line additions for Ciaran Carroll unusually high due to commiting entire NetBeans project by mistake.

Testing – Junit

Junit is a Regression Testing Framework used by developers to implement unit testing in Java, and accelerate programming speed and increase the quality of code. Developing unit tests made it much easier to source bugs as the the tests could tell where the issues lay within a class. Examples of Junit tests shown below.

```
21      public GoldDiscountTest() {
22      }
23
24      /**
25       * Test of getName method, of class GoldDiscount.
26       */
27      @Test
28      public void testGetName() {
29          System.out.println("getName");
30          GoldDiscount instance = new GoldDiscount();
31          String expResult = "Gold";
32          String result = instance.getName();
33          assertEquals(expResult, result);
34      }
35
36      /**
37       * Test of getDiscount method, of class GoldDiscount.
38       */
39      @Test
40      public void testGetDiscount() {
41          System.out.println("getDiscount");
42          GoldDiscount instance = new GoldDiscount();
43          double expResult = 0.0;
44          double result = instance.getDiscount();
45          assertEquals(expResult, result, 0.80);
46      }
47
48  }
49
```

```
Test Results ×
JavaFXApplication6 ×
Tests passed: 100.00 %
Both tests passed. (0.198 s)
business.logic.account.discount.GoldDiscountTest passed
    testGetName passed (0.007 s)
    testGetDiscount passed (0.0 s)

getName
getDiscount
```

Figure 46.

```
77          */
78         @Test
79         public void testGetMealCount() {
80             System.out.println("getMealCount");
81             Account instance = test;
82             int expResult = 0;
83             int result = instance.getMealCount();
84             assertEquals(expResult, result);
85         }
86
87         /**
88          * Test of toString method, of class Account.
89          */
90         @Test
91         public void testToString() {
92             System.out.println("toString");
93             Account instance = test;
94             String expResult = "Forename,Surname,123,0,Bronze";
95             String result = instance.toString();
96             assertEquals(expResult, result);
97         }
98
99     }
00
```

st Results ×

vaFXApplication6 ×

Tests passed: 100.00 %

All 6 tests passed. (0.251 s)

- business.logic.account.AccountTest passed
  - testGetSurname passed (0.003 s)
  - testToString passed (0.001 s)
  - testGetMealCount passed (0.0 s)
  - testGetPhoneNumber passed (0.0 s)
  - testGetDiscount passed (0.0 s)
  - testGetForename passed (0.0 s)

getSurname
toString
getMealCount
getPhoneNumber
getDiscount
getForename

Figure 47.

Recovered Blueprints

State Chart – Order

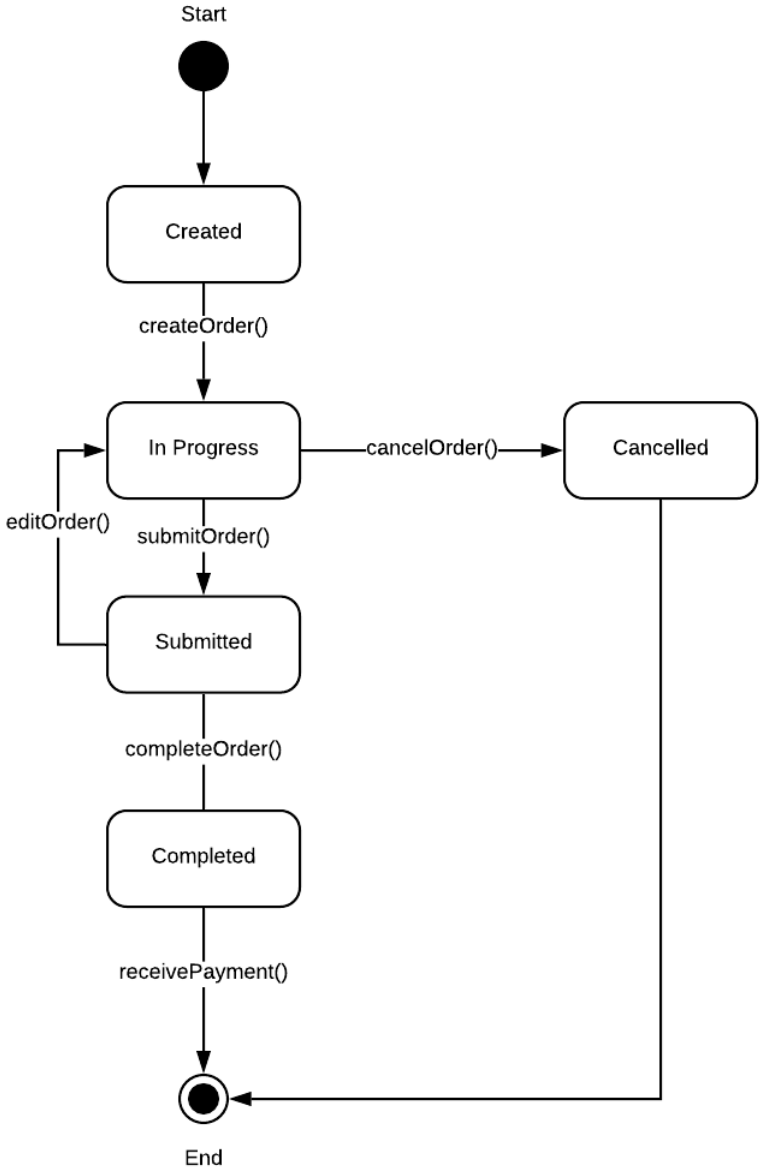


Figure 48.

Architectural Diagram

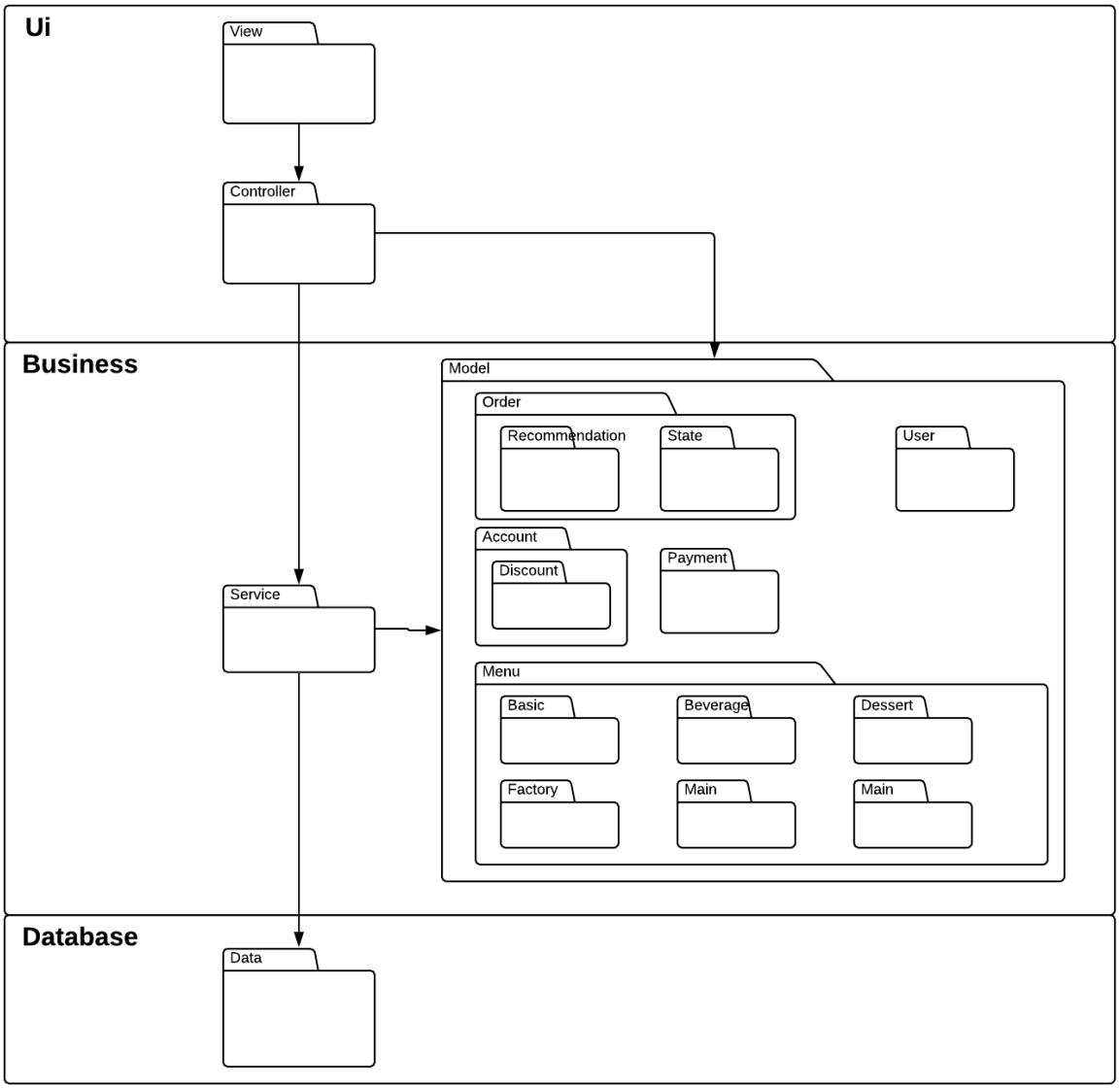
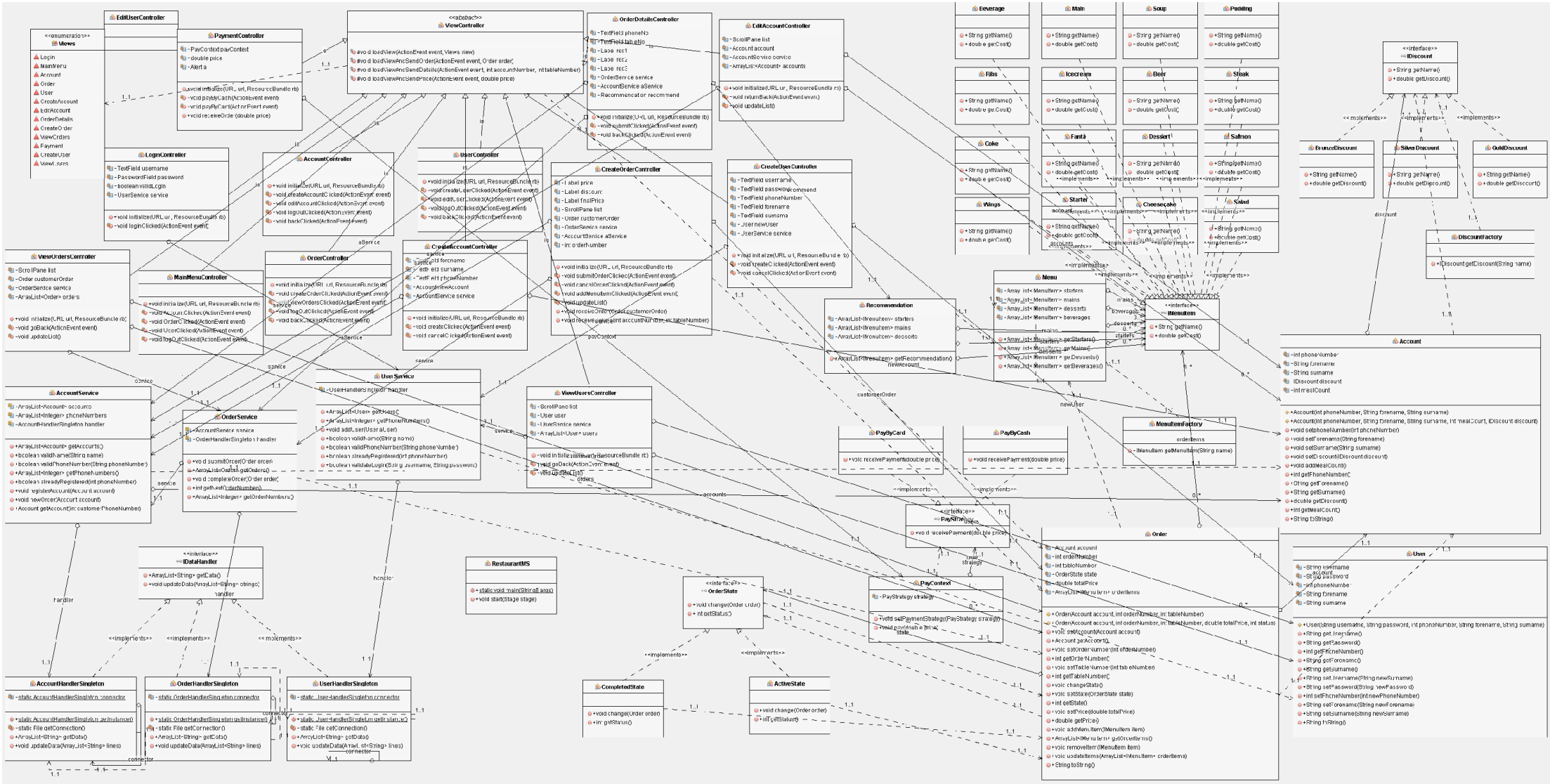


Figure 49.

Design Time Class Diagram

Figure 50.


EditUserController
PaymentController
ViewController
OrderDetailsController
EditAccountController
LoginController
AccountController
UserController
CreateOrderController
CreateUserController
ViewOrdersController
MainMenuController
OrderController
CreateAccountController
Recommendation
Menu
IMenuItem
Account
AccountService
OrderService
UserService
ViewUsersController
PayByCard
PayByCash
MenuItemFactory
Order
User
PayStrategy
PayContext
IDataHandler
RestaurantMS
OrderState
CompletedState
ActiveState
AccountHandlerSingleton
OrderHandlerSingleton
UserHandlerSingleton
IDiscount
DiscountFactory
Beverage
Main
Soup
Pudding
Ribs
Icecream
Beer
Steak
Coke
Fanta
Dessert
Salmon
Wings
Starter
Cheesecake
Salad

Critique

Analysis vs Design diagrams

Our diagrams from before implementation had changed a lot by the time we were done with the project. This is mainly due to difference in the quantity of classes and packages which we expected compared to what we actually implemented. We did not account for how much we would need to break down each use case in order to have a well-functioning system. It was a good learning exercise to see how much a project would change over time and how you have to adapt to those changes.

Implementation

We are mainly happy with what we managed to implement but unfortunately due to time constraints we were not able to implement everything that we discussed in the planning phase. We also had a fairly major change of plan mid-way through the project which set us back as regards completing all use cases.

If we were to get to do the project again, we would hope to create a real database and web service to access this data so we would be able to run application off different systems. This would be a more appropriate design as it would allow us to connect the visual systems of the Kitchen with the interactive systems of floor staff.

Design Patterns

We were happy with our implementations of design patterns, but we feel there was other patterns which would have been appropriate for the system that we missed out on. Namely, the observer and decorator patterns. The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. This could have been used to update all the UIs when a change occurred. The decorator pattern is a design pattern that allows behaviour to be added to an individual object, dynamically, without affecting the behaviour of other objects from the same class. This would have been useful as it would have provided greater flexibility than static inheritance.

References

Lucidchart :
https://www.lucidchart.com/

Lecture Notes:
CS4125

Tutorials Point (Design patterns):
https://www.tutorialspoint.com/index.htm

Geekforgeeks(Design patterns):
https://www.geeksforgeeks.org/