**Kingston University, BSc (Hons) (top-up)**

**Coursework Coversheet**

**Draft Coursework – Subject to Moderation**

**Part 1 - To Remain with the Assignment after Marking**

|  |  |
| --- | --- |
| **Student ID: K2434194** | **Student Name: Thulana Bhajitha Vithanage** |
| **Module Code: CI6115** | **Module Name: Programming III** |
| **Assignment number: 02** | **ESoft Module Leader:** |
| **Date set:** | **Date due: 2024/12/28** |

**Guidelines for the Submission of Coursework**

1. Print this coversheet and securely attach both pages to your assignment. You can help us ensure work is marked more quickly by submitting at the specified location for your module. You are advised to keep a copy of every assignment.

2. Coursework deadlines are strictly enforced by the University.

3. You should not leave the handing in of work until the last minute. Once an assignment has been submitted it cannot be submitted again.

**Academic Misconduct**: **Plagiarism** and/or **collusion** constitute **academic misconduct** under the University's Academic Regulations. Examples of academic misconduct in coursework: making available your work to other students; presenting work produced in collaboration with other students as your own (unless an explicit assessment requirement); submitting work, taken from sources that are not properly referenced, as your own. By printing and submitting this coversheet with your coursework you are confirming that the work is your own.

|  |  |  |  |
| --- | --- | --- | --- |
| |  | | --- | | ESoft Office Use Only:  Date stamp: work received | | |  | | --- | |  | |

**Kingston University, BSc (Hons) (top-up)**

**Coursework Coversheet**

**Part 2 – Student Feedback**

|  |  |
| --- | --- |
| **Student ID:** | **Student Name:** |
| **Module Code:** | **Module Name:** |
| **Assignment number:** | **ESoft Module Leader:** |
| **Date set:** | **Date due:** |

|  |
| --- |
| Strengths (areas with well-developed answers) |

|  |
| --- |
| Weaknesses (areas with room for improvement) |

|  |
| --- |
| Additional Comments |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **ESoft Module Lecturer:** | **Provisional mark as %:** |  |
| **ESoft Module Marker:** | **Date marked:** |

[1. Introduction 3](#_Toc808450121)

[2. Class diagram 4](#_Toc69398493)

[3. Test cases 5](#_Toc494229546)

[4. Implementation 14](#_Toc1081626606)

[4.1 Pizza Customization 15](#_Toc1461568753)

[4.2 Ordering process 18](#_Toc692530682)

[4.3 User profiles and favourites 20](#_Toc1165462130)

[4.3 Real time order tracking 24](#_Toc1867556368)

[4.5 Payment and loyalty program 26](#_Toc427445383)

[4.6 Seasonal specials and promotions 28](#_Toc409057492)

[4.7 Feedback and rating 29](#_Toc1224428556)

[5. Design patterns 30](#_Toc1186465019)

[5.1 Builder pattern 31](#_Toc1713465379)

[5.2 Observer pattern 32](#_Toc1145347934)

[5.3 Strategy pattern 35](#_Toc421785118)

[5.4 Chain of responsibility pattern 37](#_Toc1032803801)

[5.5 State pattern 40](#_Toc1850472297)

[5.6 Command pattern 43](#_Toc1121497125)

[5.7 Decorator pattern 46](#_Toc1460971432)

[6. Justification of application 49](#_Toc1297764597)

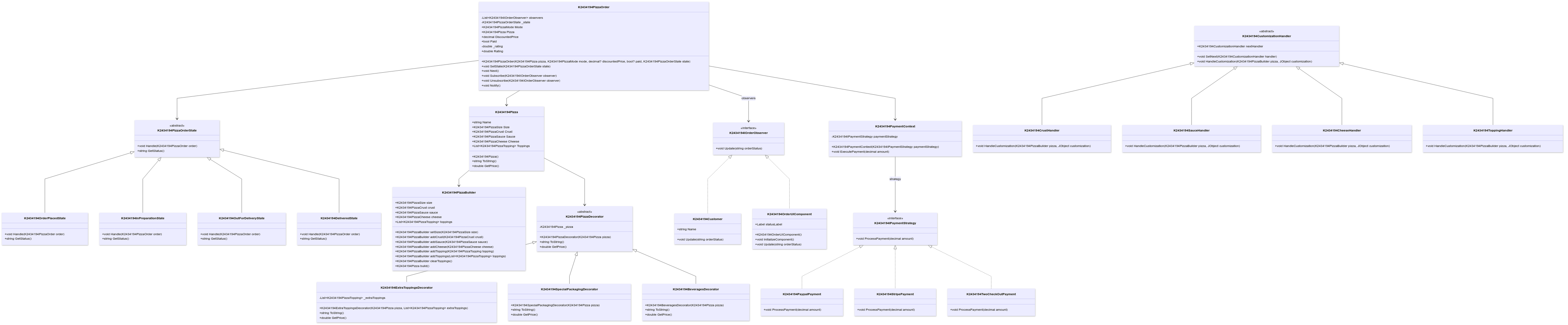
[Appendix A – link for the class diagram 50](#_Toc747682867)

[Appendix B – Link for the source code 51](#_Toc1873551697)

# Introduction

This project is developed as part of the coursework for the Programming III module, which delves into advanced object-oriented programming (OOP) concepts. The primary objective of this coursework is to design and implement a comprehensive pizza ordering system that effectively utilizes various OOP principles and design patterns.

# Class diagram

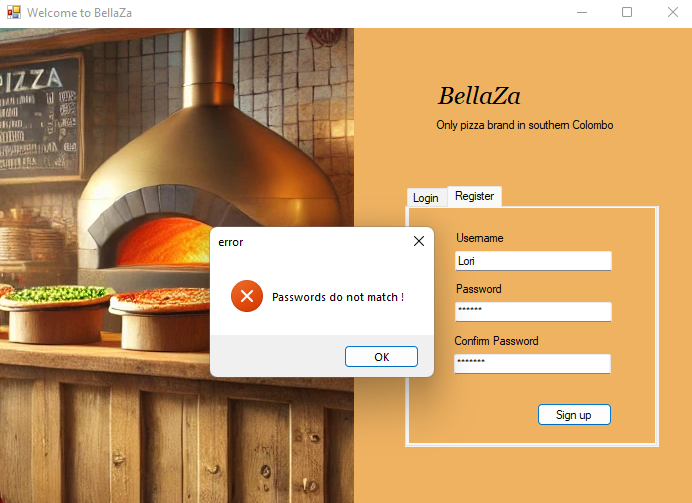


UML notation was used in drawing the class diagram. Please refer to Appendix A for the class diagram image link.

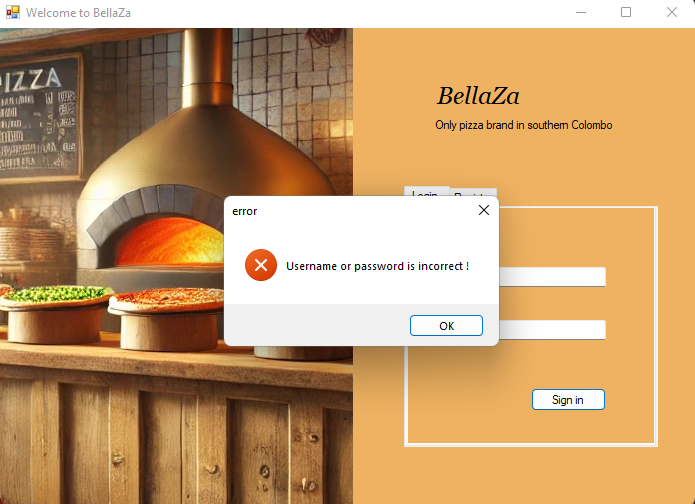
# Test cases

Unit testing, and integration testing has been carried out. Each test case is followed by necessary screenshots as evidence for the test results. Summary of the test cases is presented at the end of this chapter.

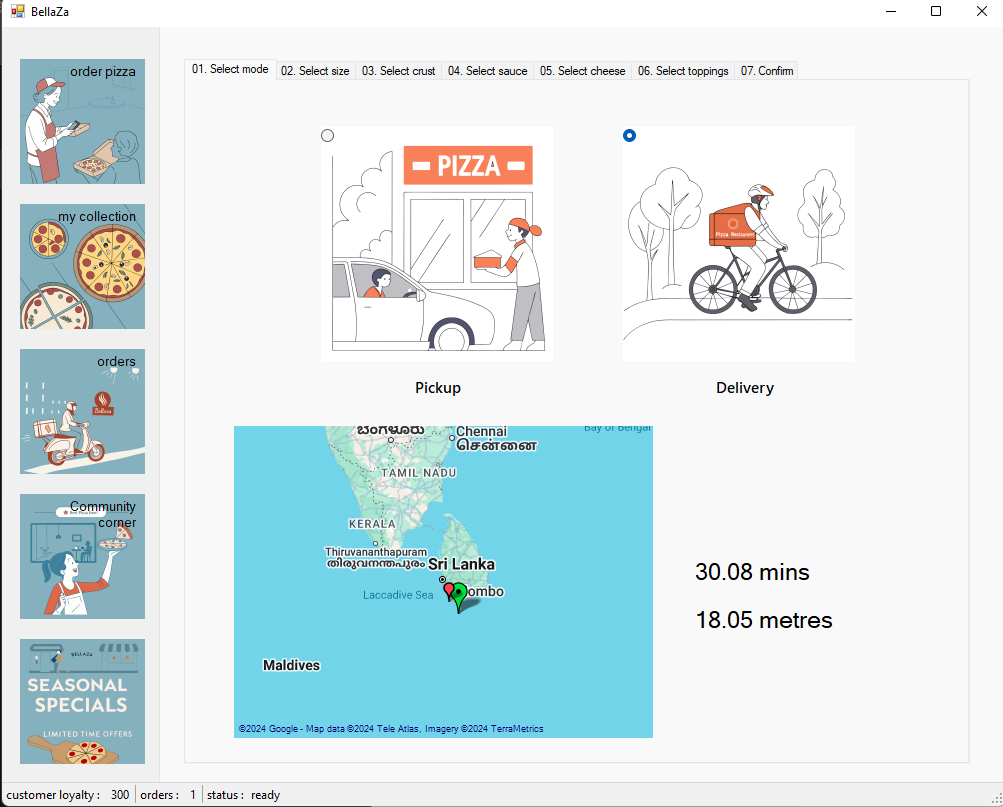
|  |  |  |
| --- | --- | --- |
| Test case number | 1 | |
| Test case Type | Unit testing | |
| Test case Name | Checking registration | |
| Test case Description | Checks registering with username and password | |
| Test data | | |
| Valid | | Invalid |
| Providing all the inputs.  Providing the same password for both password and confirm password text fields. | | Providing all the inputs.  Not providing the same password for the confirm password text field. |
| Expected Outcome | | |
| Valid | | Invalid |
| Should be directed to login tab page. | | Error message should be prompted. |



|  |  |  |
| --- | --- | --- |
| Test case number | 2 | |
| Test case Type | Unit testing | |
| Test case Name | Checking logging in | |
| Test case Description | Checks logging in with username and password | |
| Test data | | |
| Valid | | Invalid |
| Providing all the inputs.  Providing the same credentials used for the registration process. | | Providing all the inputs.  Not providing the same credentials used for the registration process. |
| Expected Outcome | | |
| Valid | | Invalid |
| Should be directed to Application home page. | | Error message should be prompted. |

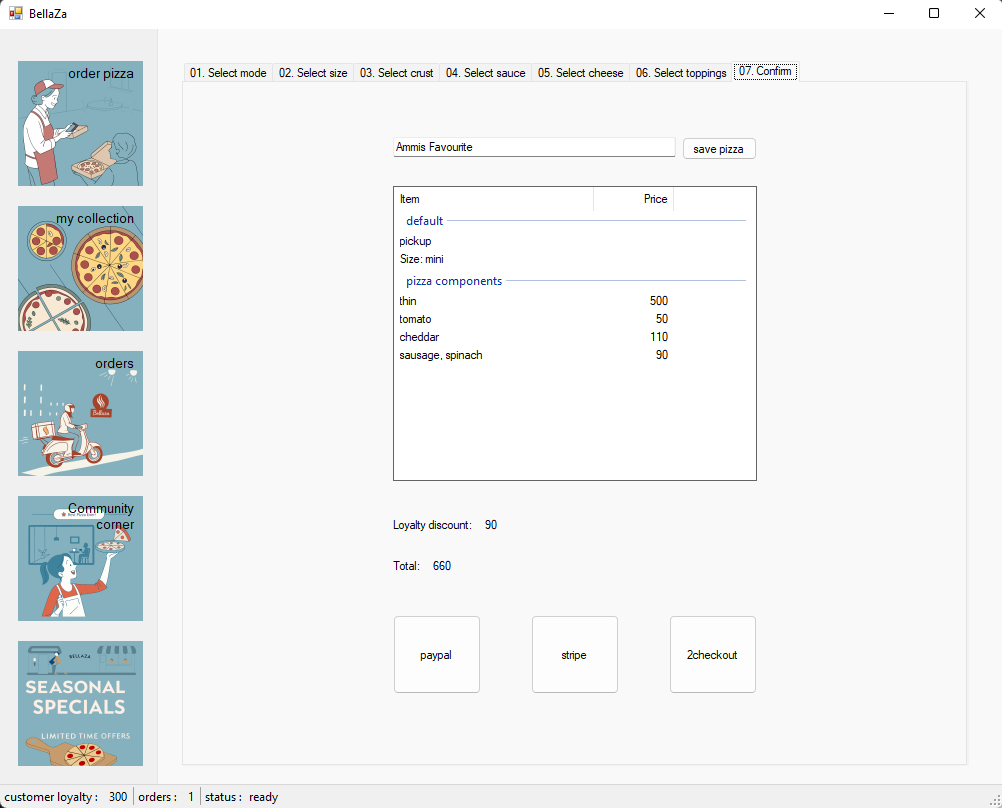


|  |  |  |
| --- | --- | --- |
| Test case number | 3 | |
| Test case Type | Unit testing | |
| Test case Name | Distance and time calculation for delivery | |
| Test case Description | Checks distance and time estimated for a delivery | |
| Test data | | |
| Valid | | Invalid |
| Selecting delivery out of delivery and pickup. Choosing a location in the map. | | Selecting delivery option.  Not choosing a location in the map. |
| Expected Outcome | | |
| Valid | | Invalid |
| Distance and time estimated should be displayed. Tab page related to the next step should be enabled. | | No distance or time estimated, should be calculated. Should not be able to proceed any further as tab pages are disabled. |



|  |  |  |
| --- | --- | --- |
| Test case number | 4 | |
| Test case Type | Unit testing | |
| Test case Name | Integrity of pizza creation | |
| Test case Description | Checks if required selections can be left out without selecting any option. | |
| Test data | | |
| Valid | | Invalid |
| 1. Attempting to choose the pizza size after selecting either pickup or delivery 2. Attempting to choose the pizza crust after selecting the pizza size 3. Attempting to choose the pizza cheese after selecting the pizza crust 4. Attempting to choose the pizza sauce after selecting the pizza cheese 5. Attempting to choose pizza toppings after selecting the pizza sauce 6. Attempting to confirm the bill after giving required selections. | | 1. Attempting to choose the pizza size without selecting either pickup or delivery 2. Attempting to choose the pizza crust without selecting the pizza size 3. Attempting to choose the pizza cheese without selecting the pizza crust 4. Attempting to choose the pizza sauce without selecting the pizza cheese 5. Attempting to choose pizza toppings without selecting the pizza sauce 6. Attempting to confirm the bill without giving required selections. |
| Expected Outcome | | |
| Valid | | Invalid |
| 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 attempts should be successful. | | 4.7, 4.8, 4.9, 4.10, 4.11, 4.12 attempts should not be successful. |

|  |  |  |
| --- | --- | --- |
| Test case number | 5 | |
| Test case Type | Unit testing | |
| Test case Name | Confirming the bill | |
| Test case Description | Checks if the pizza is created with the given selections, and price is calculated correctly. | |
| Test data | | |
| Valid | | Invalid |
| Selecting   1. Pickup 2. Mini size 3. Thin crust 4. Tomato sauce 5. Cheddar cheese 6. spinach, black olive toppings   with the default loyalty points 300 | | (no invalid data can be provided as front-end ensures integrity of the pizza creation, refer to test case number 4) |
| Expected Outcome | | |
| Valid | | Invalid |
| According to the prices, bill should have a total of 750LKR and net total of 660LKR after the loyalty discount of 90LKR | | (no invalid data was provided) |



|  |  |  |
| --- | --- | --- |
| Test case number | 6 | |
| Test case Type | Unit testing | |
| Test case Name | Payment check | |
| Test case Description | Checks if the order is created after a payment. | |
| Test data | | |
| Valid | | Invalid |
| Paying with stripe. | | (no invalid data can be provided as the UI suggests only 3 payment methods as buttons for paying) |
| Expected Outcome | | |
| Valid | | Invalid |
| 1. Console message should be displayed detailing the payment method and amount indicating its success. 2. Number of orders should be increased in the status bar 3. New order should be displayed under orders section. | | (no invalid data was provided) |

# 4. Implementation

## 4.1 Pizza Customization

Different tab pages were designed for an intuitive pizza customization process.

Tab pages must be accessed sequentially without any restrictions to go back. Next tab page cannot be accessed without filling the required information of the current tab page.

private void sizeChanged(object sender, EventArgs e)

{

if (((RadioButton)sender).Checked)

size = (PizzaSize)((RadioButton)sender).Tag;

//Console.WriteLine(size.ToString());

tabPage2.Enabled = true;

}

private void crustChanged(object sender, EventArgs e)

{

if (((RadioButton)sender).Checked)

crust = (PizzaCrust)((RadioButton)sender).Tag;

//Console.WriteLine(crust.ToString());

tabPage3.Enabled = true;

}

private void sauceChanged(object sender, EventArgs e)

{

if (((RadioButton)sender).Checked)

sauce = (PizzaSauce)((RadioButton)sender).Tag;

//Console.WriteLine(sauce.ToString());

tabPage4.Enabled = true;

}

private void cheeseChanged(object sender, EventArgs e)

{

if (((RadioButton)sender).Checked)

cheese = (PizzaCheese)((RadioButton)sender).Tag;

//Console.WriteLine(cheese.ToString());

tabPage5.Enabled = true;

}

private void toppingsChanged(object sender, EventArgs e)

{

if(((CheckBox)sender).CheckState == CheckState.Checked)

toppings.Add((PizzaTopping)((CheckBox)sender).Tag);

if (((CheckBox)sender).CheckState == CheckState.Unchecked)

toppings.Remove((PizzaTopping)((CheckBox)sender).Tag);

tabPage6.Enabled = true;

}

private void tabPage6\_Enter(object sender, EventArgs e)

{

if(fromCollection)

{

fromCollection = false;

return;

}

Pizza newPizza = new Pizza.PizzaBuilder()

.setSize(size)

.addCrust(crust)

.addSauce(sauce)

.addCheese(cheese)

.addToppings(toppings)

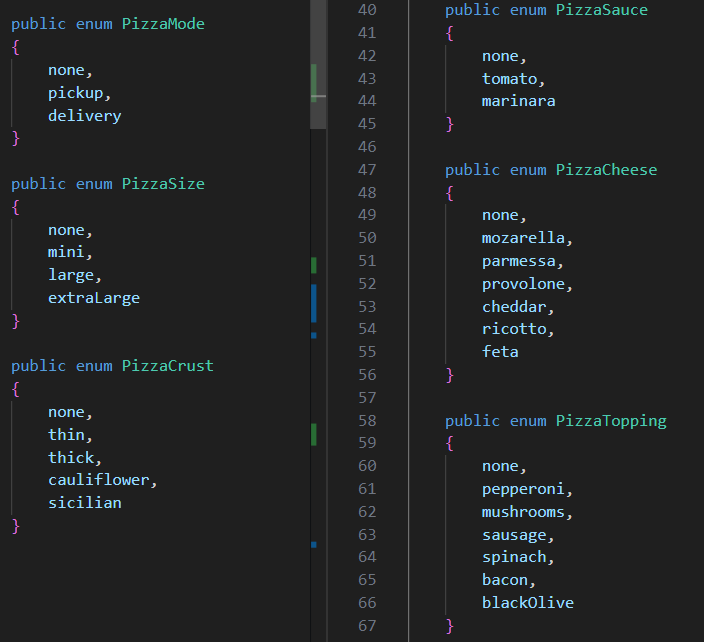
.build();

loadPizzaBill(newPizza);

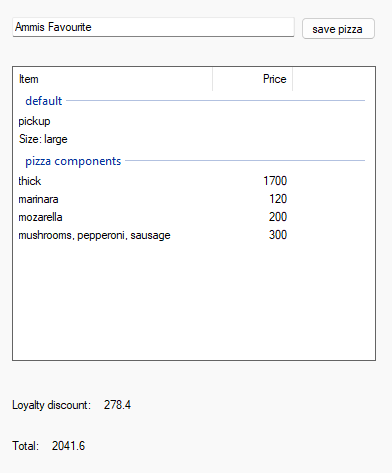
}

Selected options are saved onto private variables, which at the confirmation page, used for building the pizza. This method was used as the user may change their selection. Builder pattern was used to create the pizza in this case.

Pizza customization includes following varieties.



User can customize the pizza and name them so that it can be stored with the given name.



## 4.2 Ordering process

Ordering process employs either builder or both the builder and chain of responsibility design patterns varying on the circumstance. After the pizza is created with the selected combinations, it is added additional details.

These additional details include, whether it should be delivered or kept at drive-thru, if need delivering, the location to which it should be delivered.

Further, the order can include extra toppings, beverages or special packaging.

private void gMapControl1\_OnMapClick(PointLatLng pointClick, MouseEventArgs e)

{

gMapControl1.Overlays.Clear();

GMapOverlay markersOverlay = new GMapOverlay("markers");

markersOverlay.Markers.Add(marker);

markersOverlay.Markers.Add(new GMarkerGoogle(pointClick, GMarkerGoogleType.red\_small));

gMapControl1.Overlays.Add(markersOverlay);

getNavigationDetails(pointClick);

tabPage7.Enabled = true;

}

private void getNavigationDetails(PointLatLng destination)

{

string result = Get("http://router.project-osrm.org/route/v1/driving/" + marker.Position.Lat +

',' + marker.Position.Lng + ';' + destination.Lat + ',' + destination.Lng + "?overview=false");

if (result == null) {

labelDuration.Text = labelDistance.Text = "something went wrong.";

}

JObject json = JObject.Parse(result);

double duration = Convert.ToDouble(json["routes"][0]["duration"]) / 60;

double distance = Convert.ToDouble(json["routes"][0]["duration"]) / 100;

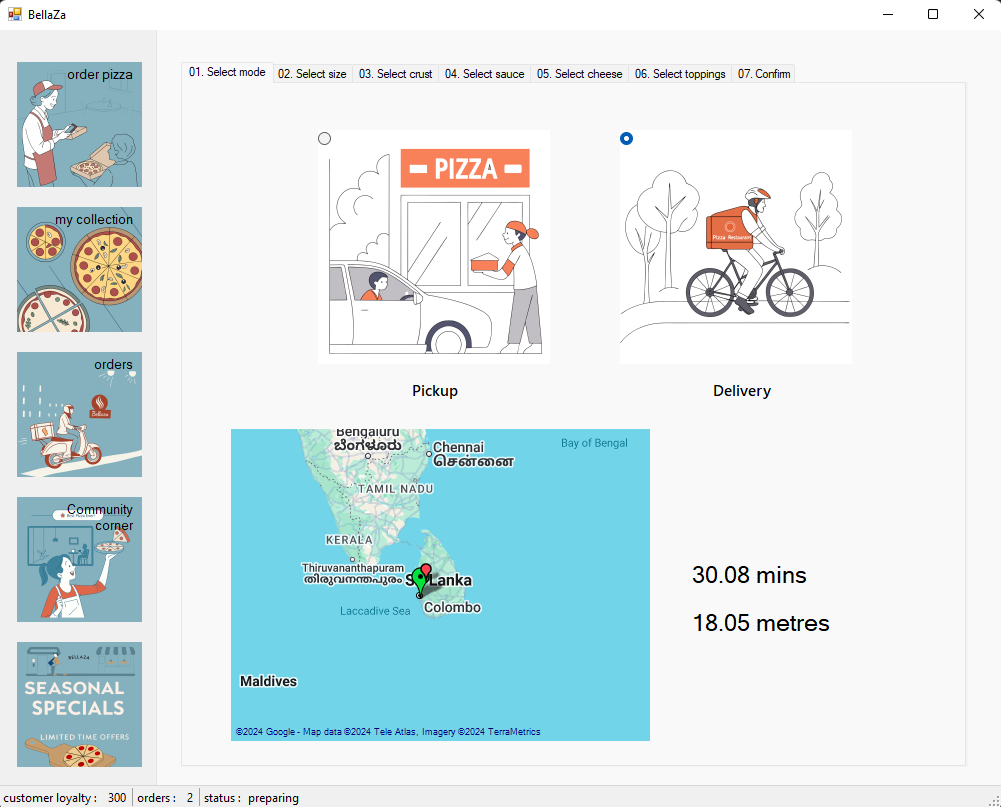
labelDuration.Text = Math.Round(duration, 2).ToString() + " mins";

labelDistance.Text = Math.Round(distance, 2).ToString() + " metres";

Console.WriteLine(result);

}

Above implementation is related to calculating the distance and time estimated for the delivery. Location of the shop is set to a pre-defined location, and the distance and ETA is calculated from there.



## 4.3 User profiles and favourites

User profiles are created to store user specific information. These include credentials, pizza collection, favourite pizzas, and orders. Moreover, loyalty points are stored as well.

public class UserProfile

{

private List<Pizza> pizzacollection;

public string Name { get; set; }

private string password;

public List<Pizza> pizzaCollection

{

get

{

return pizzacollection;

}

set

{

pizzacollection = value;

}

}

private double loyaltyPercentage;

public double LoyaltyPercentage

{

get

{

return loyaltyPercentage;

}

private set

{

if (value <= 50.0)

loyaltyPercentage = value;

}

}

public UserProfile()

{

this.Name = "default";

this.password = "default123";

pizzaCollection = new List<Pizza>();

}

public UserProfile(string name, string password)

{

this.Name = name;

this.password = password;

pizzaCollection = new List<Pizza>();

}

public bool checkPassword(string password)

{

return password == this.password;

}

public void increaseLoyalty(PizzaSize? size)

{

switch (size)

{

case PizzaSize.mini:

LoyaltyPercentage += 0.05;

break;

case PizzaSize.large:

LoyaltyPercentage += 0.06;

break;

case PizzaSize.extraLarge:

LoyaltyPercentage += 0.12;

break;

default:

LoyaltyPercentage += 0.02;

break;

}

}

}

Users can name their pizza combination and store them in their collection. Upon navigating to this collection and clicking would simply direct them to the payment page to reorder the selected pizza.

private void loadPizzaCollection()

{

flowLayoutPanel1.Controls.Clear();

for (int i = 0; i < user.pizzaCollection.Count; i++)

{

Console.WriteLine(user.pizzaCollection[i].getDescription());

Button button = new Button();

button.Text = user.pizzaCollection[i].getDescription();

button.AutoSize = true;

button.Parent = flowLayoutPanel1;

button.Tag = i;

button.Click += CollectionPizza\_Click;

}

}

private void CollectionPizza\_Click(object sender, EventArgs e)

{

Pizza pizza = user.pizzaCollection[(int)((Button)sender).Tag];

Console.WriteLine(pizza.getDescription());

fromCollection = true;

tabControl2.SelectedIndex = 0; //change the main navigation

tabPage6.Enabled = true;

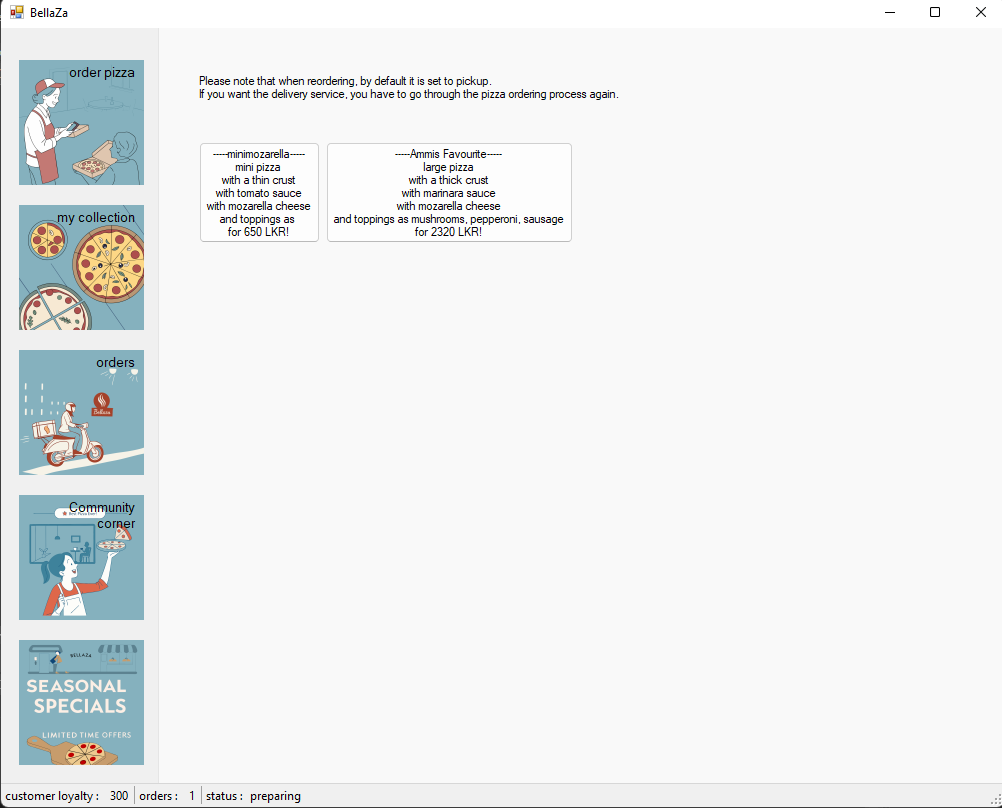
tabControl1.SelectedIndex = tabControl1.TabCount - 1;

textPizzaName.Text = pizza.Name;

mode = PizzaMode.pickup;

loadPizzaBill(pizza);

}



## 4.3 Real time order tracking

A status bar was implemented to keep track of the total number of orders for the client, and the current state of the most recent order.

Further, order section in the application facilitates both past and ongoing orders with their relevant order state, with current orders at the top of the list.

State pattern and observer pattern have been used in implementing this feature.

private void AddNewOrder()

{

statusBarOrderCount.Text = (int.Parse(statusBarOrderCount.Text) + 1).ToString();

var component = new OrderUIComponent(orderPlaceHolder);

ordersPanel.Controls.Add(component);

component.Show();

component.Dock = DockStyle.Top;

realTimeOrderProcessor.Enabled = true;

}

Although, it was not possible to stimulate a real-world delivery tracking, a timer was used to mimic the process programmatically. After each interval, state was set to its next possible state which was reflected in the UI.

private void realTimeOrderProcessor\_Tick(object sender, EventArgs e)

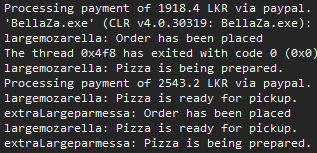
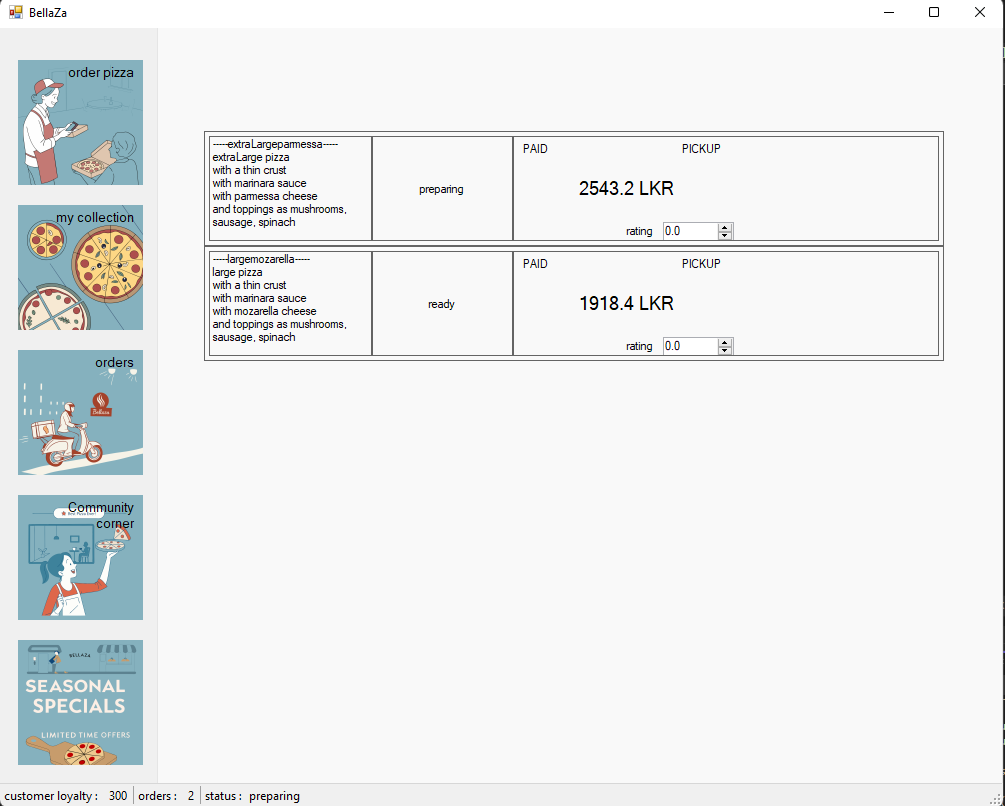
{

foreach (OrderUIComponent component in ordersPanel.Controls)

component.Order.Next();

statusCurrentState.Text = ((OrderUIComponent)ordersPanel.Controls[ordersPanel.Controls.Count - 1]).Status;

}



## 4.5 Payment and loyalty program

Strategy pattern was implemented to facilitate or stimulate the payment process with different payment methods such as PayPal, stripe, and twocheckout.

private void PayButton\_Click(object sender, EventArgs e)

{

var paymentStrategy = (IPaymentStrategy)((Button)sender).Tag;

var paymentContext = new PaymentContext(paymentStrategy);

paymentContext.ExecutePayment(netTotal);

AddNewOrder();

if (!tabPage5.Enabled)

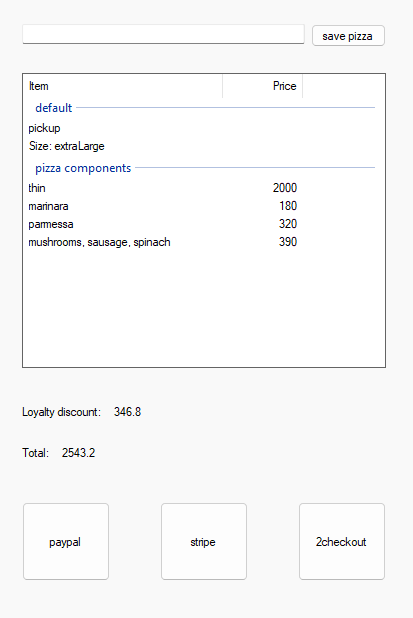
{

tabPage6.Enabled = false;

tabControl1.SelectedIndex = 0;

}

}



A loyalty program is introduced to each user. With each new pizza order, the loyalty percentage is increased. This increment is based on the pizza size, higher the size is higher the percentage increment.

Then, this increased percentage is used to provide discount for the next order. However, the increment takes place with a maximum percentage reach as a restriction.

user.increaseLoyalty(orderPlaceHolder.Pizza.size);

public void increaseLoyalty(PizzaSize? size)

{

switch (size)

{

case PizzaSize.mini:

LoyaltyPercentage += 0.05;

break;

case PizzaSize.large:

LoyaltyPercentage += 0.06;

break;

case PizzaSize.extraLarge:

LoyaltyPercentage += 0.12;

break;

default:

LoyaltyPercentage += 0.02;

break;

}

}

public double LoyaltyPercentage

{

get

{

return loyaltyPercentage;

}

private set

{

if (value <= 50.0)

loyaltyPercentage = value;

}

}

## 4.6 Seasonal specials and promotions

Decorator class has been used to introduce seasonal specials and promotions. Although, it can be used to define many such promotions, for the demonstration, one such class has been created.

public class SeasonalSpecialDecorator : PizzaDecorator

{

public SeasonalSpecialDecorator(Pizza pizza) : base(pizza)

{

}

public override string ToString()

{

var baseDescription = base.ToString();

var packagingDescription = "Seasonal discount included";

return baseDescription + "\n" + packagingDescription;

}

public override decimal getPrice()

{

var basePrice = base.getPrice();

var seasonalDiscount = (decimal)0.15;

return basePrice \* ((decimal)1.0 - seasonalDiscount);

}

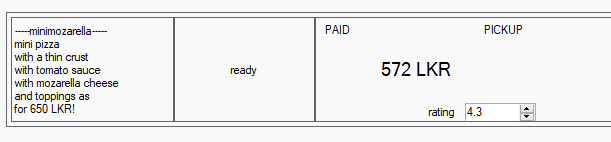
}

Furthermore, a JSON price configuration can be used to determine the offer or promotion for each pizza item for each size.

## 4.7 Feedback and rating

Command pattern was used in getting feedback and ratings.

OrderUIComponent presents a rating feature which takes the input and transform into a command which can be executed. Upon execution user’s pizza collection is updated with the relevant rating for that pizza order.



# 5. Design patterns

## 5.1 Builder pattern

Builder pattern was used for constructing the pizza step by step, sequentially. Although, it necessarily, must be not in order, builder methods were called according to the step-by-step pizza creation in the user interface of the application.

public class PizzaBuilder

{

public PizzaSize size;

public PizzaCrust crust;

public PizzaSauce sauce;

public PizzaCheese cheese;

public List<PizzaTopping> toppings = new List<PizzaTopping>();

public PizzaBuilder setSize(PizzaSize size)

{

this.size = size;

return this;

}

public PizzaBuilder addCrust(PizzaCrust crust)

{

this.crust = crust;

return this;

}

public PizzaBuilder addSauce(PizzaSauce sauce)

{

this.sauce = sauce;

return this;

}

public PizzaBuilder addCheese(PizzaCheese cheese)

{

this.cheese = cheese;

return this;

}

public PizzaBuilder addTopping(PizzaTopping topping)

{

if (toppings.Count < PizzaMaxToppings.getMaxToppingsCount(this.size))

this.toppings.Add(topping);

return this;

}

public PizzaBuilder addToppings(List<PizzaTopping> toppings)

{

foreach (PizzaTopping topping in toppings)

addTopping(topping);

return this;

}

public PizzaBuilder clearToppings()

{

this.toppings.Clear();

return this;

}

public Pizza build()

{

return new Pizza(this);

}

}

Maximum number of toppings applied to the pizza is decided based on the pizza size.

## 5.2 Observer pattern

Observer pattern was used to notify client application when a change in status of the pizza order occurs.

public class PizzaOrder

{

private List<IOrderObserver> observers = new List<IOrderObserver>();

private PizzaOrderState \_state;

public PizzaMode Mode { get; }

public Pizza Pizza { get; }

public decimal DiscountedPrice { get; }

public bool Paid { get; }

private double \_rating;

public double Rating

{

get { return \_rating; }

set

{

if (value <= 5.0)

{

\_rating = value;

}

}

}

public PizzaOrder(Pizza pizza, PizzaMode mode, decimal? discountedPrice = null, bool? paid = null, PizzaOrderState state = null)

{

this.\_state = state ?? new OrderPlacedState();

this.Mode = mode;

this.Pizza = pizza;

this.Paid = paid ?? true;

this.DiscountedPrice = discountedPrice ?? pizza.getPrice();

}

public void SetState(PizzaOrderState state)

{

this.\_state = state;

Notify();

}

public void Next()

{

\_state.Handle(this);

}

public void Subscribe(IOrderObserver observer)

{

observers.Add(observer);

}

public void Unsubscribe(IOrderObserver observer)

{

observers.Remove(observer);

}

private void Notify()

{

foreach (var observer in observers)

{

observer.Update(\_state.ToString());

}

}

}

Observer interface is as follow.

public interface IOrderObserver

{

void Update(string orderStatus);

}

A custom created UI component acts as an observer to the order, where upon status change it updates its user interface to reflect the status change.

public partial class OrderUIComponent : UserControl, IOrderObserver

{

...

public void Update(string status)

{

labelState.Text = status;

if(status.ToLower() != "order placed") buttonCancel.Visible = false;

}

}

}

## 5.3 Strategy pattern

Strategy pattern was used to implement different payment methods to a certain payment context. This ensured, different payment gateway processing for the same amount.

public interface IPaymentStrategy

{

void ProcessPayment(decimal amount);

}

public class PaypalPayment : IPaymentStrategy

{

public void ProcessPayment(decimal amount)

{

Console.WriteLine($"Processing payment of {amount} LKR via paypal.");

}

}

public class StripePayment : IPaymentStrategy

{

public void ProcessPayment(decimal amount)

{

Console.WriteLine($"Processing payment of {amount} LKR via stripe.");

}

}

public class TwoCheckOutPayment : IPaymentStrategy

{

public void ProcessPayment(decimal amount)

{

Console.WriteLine($"Processing payment of {amount} LKR via 2CheckOut.");

}

}

// Payment context class

public class PaymentContext

{

private readonly IPaymentStrategy paymentStrategy;

public PaymentContext(IPaymentStrategy strategy)

{

paymentStrategy = strategy;

}

public void ExecutePayment(decimal amount)

{

paymentStrategy.ProcessPayment(amount);

}

}

Three payment strategies were employed; stripe, PayPal, and 2checkout, although they don't really implement their relevant payment gateway.

## 5.4 Chain of responsibility pattern

This was used when trying to order seasonal promotions, and community-best pizzas. These mentioned pizza details are shipped as a Json file. Therefore, it was required to consider the cases when a particular item such as cheese doesn't have a specification for the pizza creation. To address such cases, this pattern was employed as relevant handler to the config will only respond.

public abstract class CustomizationHandler

{

protected CustomizationHandler nextHandler;

public CustomizationHandler SetNext(CustomizationHandler handler)

{

nextHandler = handler;

return handler;

}

public abstract void HandleCustomization(PizzaBuilder pizza, JObject customization);

}

Handlers for crust, sauce, cheese, and toppings were created based on the customizationHandler.

public class CrustHandler : CustomizationHandler

{

public override void HandleCustomization(PizzaBuilder pizza, JObject customization)

{

if (customization.ContainsKey("crust"))

{

pizza.addCrust(customization["crust"].ToObject<PizzaCrust>());

Console.WriteLine($"Set crust to: {customization}");

}

else if (nextHandler != null)

{

nextHandler.HandleCustomization(pizza, customization);

}

}

}

public class SauceHandler : CustomizationHandler

{

public override void HandleCustomization(PizzaBuilder pizza, JObject customization)

{

if (customization.ContainsKey("sauce"))

{

pizza.addSauce(customization["sauce"].ToObject<PizzaSauce>());

Console.WriteLine($"Set crust to: {customization}");

}

else if (nextHandler != null)

{

nextHandler.HandleCustomization(pizza, customization);

}

}

}

public class CheeseHandler : CustomizationHandler

{

public override void HandleCustomization(PizzaBuilder pizza, JObject customization)

{

if (customization.ContainsKey("cheese"))

{

pizza.addCheese(customization["cheese"].ToObject<PizzaCheese>());

Console.WriteLine($"Set crust to: {customization}");

}

else if (nextHandler != null)

{

nextHandler.HandleCustomization(pizza, customization);

}

}

}

public class ToppingHandler : CustomizationHandler

{

public override void HandleCustomization(PizzaBuilder pizza, JObject customization)

{

if (customization.ContainsKey("Toppings"))

{

foreach(JProperty topping in customization["Toppings"])

pizza.addTopping(topping.ToObject<PizzaTopping>());

Console.WriteLine($"Added topping: {customization}");

}

else if (nextHandler != null)

{

nextHandler.HandleCustomization(pizza, customization);

}

}

}

}

Additionally, the base class CustomizationHandler was created in a way, upon calling the nextHandler() it returns the handler set as the next handler, so that when implementing, functions can be chained for a single handler.

## 5.5 State pattern

State pattern was used to indicate and update different states regarding the pizza order. Each state represented a different stage in the order process, such as "Order Placed", "In Preparation", "Out for Delivery", and "Delivered". The state transitions are handled by the state objects themselves, which encapsulate the behaviour associated with each state.

State context is the PizzaOrder class which also acts as the notifying agent for its subscribers.

public abstract class PizzaOrderState

{

public abstract void Handle(PizzaOrder context);

}

Concrete order states are implemented as follow.

public class OrderPlacedState : PizzaOrderState

{

public override void Handle(PizzaOrder context)

{

Console.WriteLine(context.Pizza.Name + ": Order has been placed");

context.SetState(new InPreparationState());

}

public override string ToString()

{

return "order placed";

}

}

public class InPreparationState : PizzaOrderState

{

public override void Handle(PizzaOrder context)

{

Console.WriteLine(context.Pizza.Name + ": Pizza is being prepared.");

if(context.Mode == PizzaMode.pickup)

context.SetState(new ReadyForPickupState());

else

context.SetState(new OutForDeliveryState());

}

public override string ToString()

{

return "preparing";

}

}

public class OutForDeliveryState : PizzaOrderState

{

public override void Handle(PizzaOrder context)

{

Console.WriteLine(context.Pizza.Name + ": Pizza is out for delivery.");

context.SetState(new DeliveredState());

}

public override string ToString()

{

return "delivering";

}

}

public class ReadyForPickupState : PizzaOrderState

{

public override void Handle(PizzaOrder context)

{

Console.WriteLine(context.Pizza.Name + ": Pizza is ready for pickup.");

}

public override string ToString()

{

return "ready";

}

}

public class DeliveredState : PizzaOrderState

{

public override void Handle(PizzaOrder context)

{

Console.WriteLine(context.Pizza.Name + ": Pizza is delivered.");

}

public override string ToString()

{

return "delivered";

}

}

public class OrderCancelledState : PizzaOrderState

{

public override void Handle(PizzaOrder context)

{

Console.WriteLine(context.Pizza.Name + ": Pizza order is cancelled.");

}

public override string ToString()

{

return "cancelled";

}

}

## 5.6 Command pattern

Command pattern was used to give ratings as the feedback for a particular order, cancel the order.

public interface ICommand

{

void Execute();

}

public class PizzaOrderCommand : ICommand

{

private PizzaOrder \_order;

public PizzaOrderCommand(PizzaOrder order)

{

\_order = order;

}

public void Execute()

{

\_order.Next();

}

}

public class CancelOrderCommand : ICommand

{

private readonly OrderService \_orderService;

private readonly PizzaOrder \_order;

public CancelOrderCommand(OrderService orderService, PizzaOrder order)

{

\_orderService = orderService;

\_order = order;

}

public void Execute()

{

\_orderService.CancelOrder(\_order);

}

}

public class RateOrderCommand : ICommand

{

private readonly OrderService \_orderService;

private readonly PizzaOrder \_order;

private readonly double \_rating;

public RateOrderCommand(OrderService orderService, PizzaOrder order, double rating)

{

\_orderService = orderService;

\_order = order;

\_rating = rating;

}

public void Execute()

{

\_orderService.RateOrder(\_order, \_rating);

}

}

Order service is used to facilitate the process.

public class OrderService

{

public void CancelOrder(PizzaOrder order)

{

order.SetState(new OrderCancelledState());

}

public void RateOrder(PizzaOrder order, double rating)

{

order.Rating = rating;

}

}

CommandInvoker class was used to keep track the of the commands, and execute them one by one.

public class CommandInvoker

{

private readonly List<ICommand> \_commands = new List<ICommand>();

public void AddCommand(ICommand command)

{

\_commands.Add(command);

}

public void ExecuteCommands()

{

foreach (var command in \_commands)

{

command.Execute();

}

\_commands.Clear();

}

}

However, undo() method was not implemented as cancelOrder is basically the undo() method of placing an order. Further, a new rating would replace the previous rating. Therefore, undo() method implementation was not necessary for both the cases.

## 5.7 Decorator pattern

Decorator pattern was used to add extra items, or modify the pizza without affecting its overall structure, and billing details. These extra items based on the user preference, includes extra toppings, special packaging, and beverages.

Specially, in the case of a mini pizza order, since the number of toppings are limited to 2 options, this implementation allows the user to add more toppings as extra items.

public abstract class PizzaDecorator : Pizza

{

protected Pizza \_pizza;

public PizzaDecorator(Pizza pizza)

{

\_pizza = pizza;

}

public override string ToString()

{

return \_pizza.ToString();

}

public override decimal getPrice()

{

return \_pizza.getPrice();

}

}

Based on the PizzaDecorator, extraToppings, specialPakcaging, and Beverage decorators are created.

public class ExtraToppingsDecorator : PizzaDecorator

{

private List<PizzaTopping> \_extraToppings;

public ExtraToppingsDecorator(Pizza pizza, List<PizzaTopping> extraToppings) : base(pizza)

{

\_extraToppings = extraToppings;

}

public override string ToString()

{

var baseDescription = base.ToString();

var extraToppingsDescription = "Extra toppings: " + string.Join(", ", \_extraToppings);

return baseDescription + "\n" + extraToppingsDescription;

}

public override decimal getPrice()

{

var basePrice = base.getPrice();

var extraToppingsPrice = \_extraToppings.Count \* 150; // taking each topping amount 150Rs

return basePrice + extraToppingsPrice;

}

}

public class SpecialPackagingDecorator : PizzaDecorator

{

public SpecialPackagingDecorator(Pizza pizza) : base(pizza)

{

}

public override string ToString()

{

var baseDescription = base.ToString();

var packagingDescription = "Special packaging included.";

return baseDescription + "\n" + packagingDescription;

}

public override decimal getPrice()

{

var basePrice = base.getPrice();

var packagingPrice = (decimal) 200.0;

return basePrice + packagingPrice;

}

}

public class BeveragesDecorator : PizzaDecorator

{

public BeveragesDecorator(Pizza pizza) : base(pizza)

{

}

public override string ToString()

{

var baseDescription = base.ToString();

var packagingDescription = "Beverages included.";

return baseDescription + "\n" + packagingDescription;

}

public override decimal getPrice()

{

var basePrice = base.getPrice();

var beveragePrice = (decimal) 400.0;

return basePrice + beveragePrice;

}

}

public class SeasonalSpecialDecorator : PizzaDecorator

{

public SeasonalSpecialDecorator(Pizza pizza) : base(pizza)

{

}

public override string ToString()

{

var baseDescription = base.ToString();

var packagingDescription = "Seasonal discount included";

return baseDescription + "\n" + packagingDescription;

}

public override decimal getPrice()

{

var basePrice = base.getPrice();

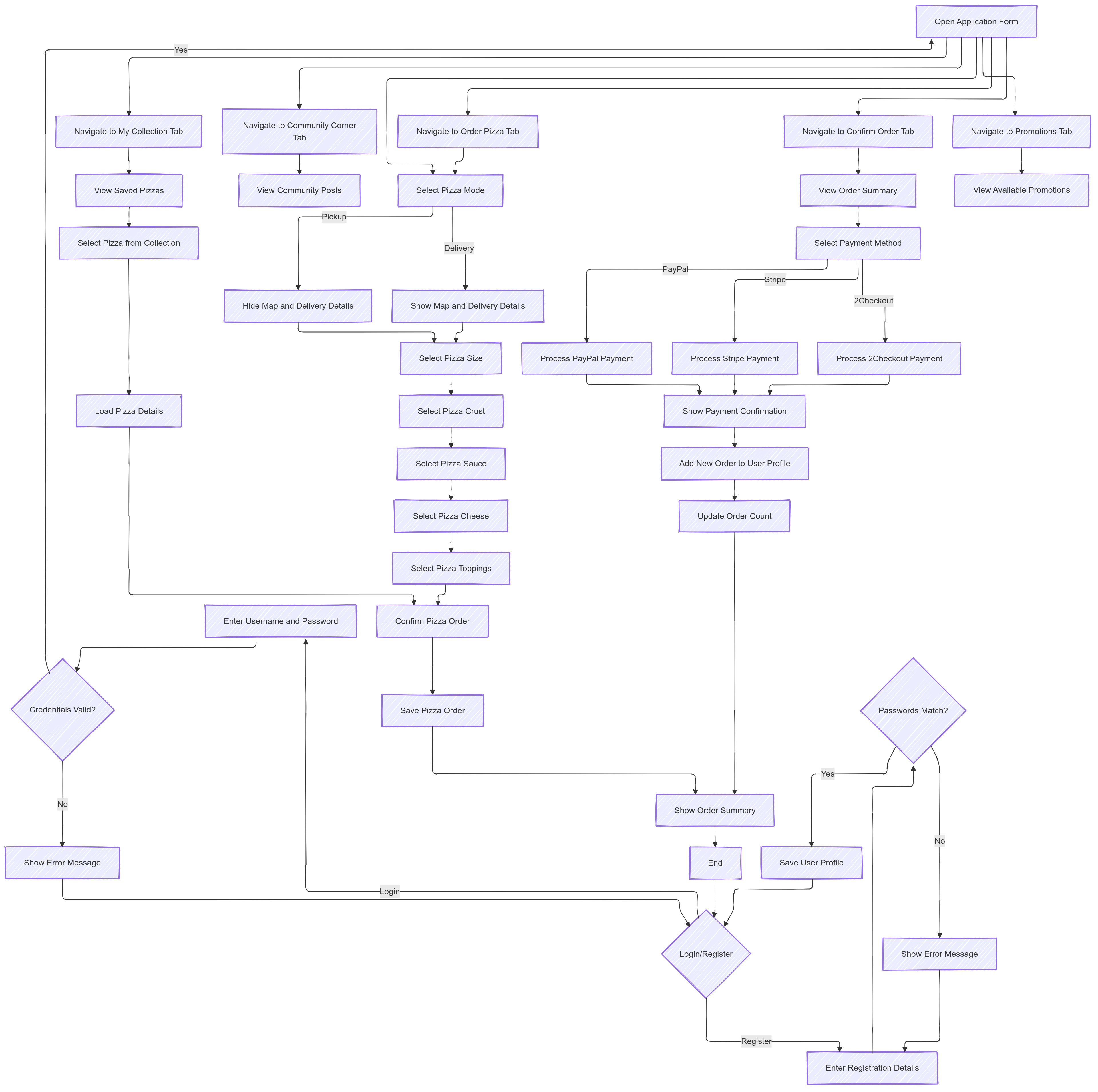
var seasonalDiscount = (decimal)0.15;

return basePrice \* ((decimal)1.0 - seasonalDiscount);

}

}

# 6. Justification of application



# Appendix A – link for the class diagram

[BellaZa class diagram](https://github.com/Thulana2006/BellaZa/blob/master/Resources/Diagrams/class%20diagram.png)

# Appendix B – Link for the source code

[BellaZa github repository](https://github.com/Thulana2006/BellaZa)