CST-250 Activity 4: Building a Data-Driven Pizza Order System with N-Layer Architecture

Overview

In this lesson, you will build a product order system for pizzas. As you construct this application, you will gain hands-on experience with various form controls and learn how to effectively manage their properties. Additionally, you will create data access and business logic services for the application.

Learning Objectives

1. Install and configure a wide range of form controls.
2. Manage and read the properties of each control.
3. Create a class with properties to hold the data from each control.
4. Save all the values in the properties of a class PizzaModel object.
5. Store the PizzaModels in a list and display them on a second form.

The GCU College of Engineering and Technology Coding Guidelines and Best Practices for C#, available in Class Resources, must be followed in all CST-250 projects to ensure consistency, readability, and professionalism in student code. These guidelines promote a clear separation of concerns by organizing code into the Models, BusinessLogicLayer, and DataAccessLayer folders. The document also outlines essential standards for casing, naming conventions, code structure, commenting, and organization. Additionally, it emphasizes best practices in object-oriented design, exception handling, and the use of design patterns. Adhering to these expectations is required for successful completion of course assignments.

Part 1 – Creating the User Interface

1. Create a new Windows Forms app in Visual Studio as shown in Figure 1.

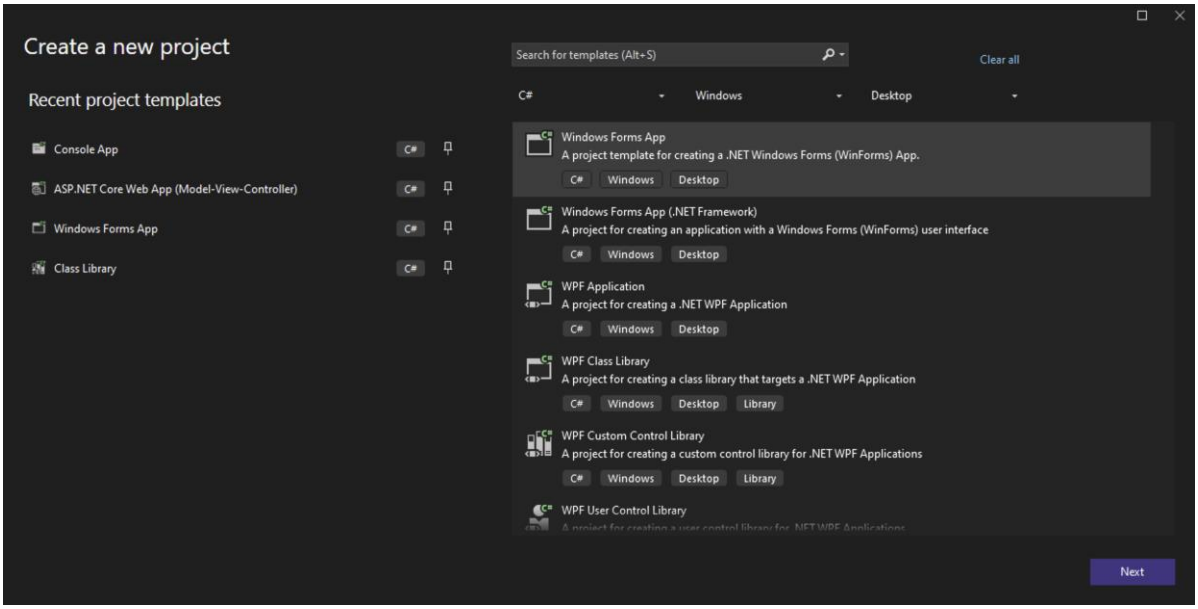


Figure 1. Creating a new Windows Forms application.

2. Name the project "PizzaMaker" as shown in Figure 2. Use the most recent long-term support .NET version.

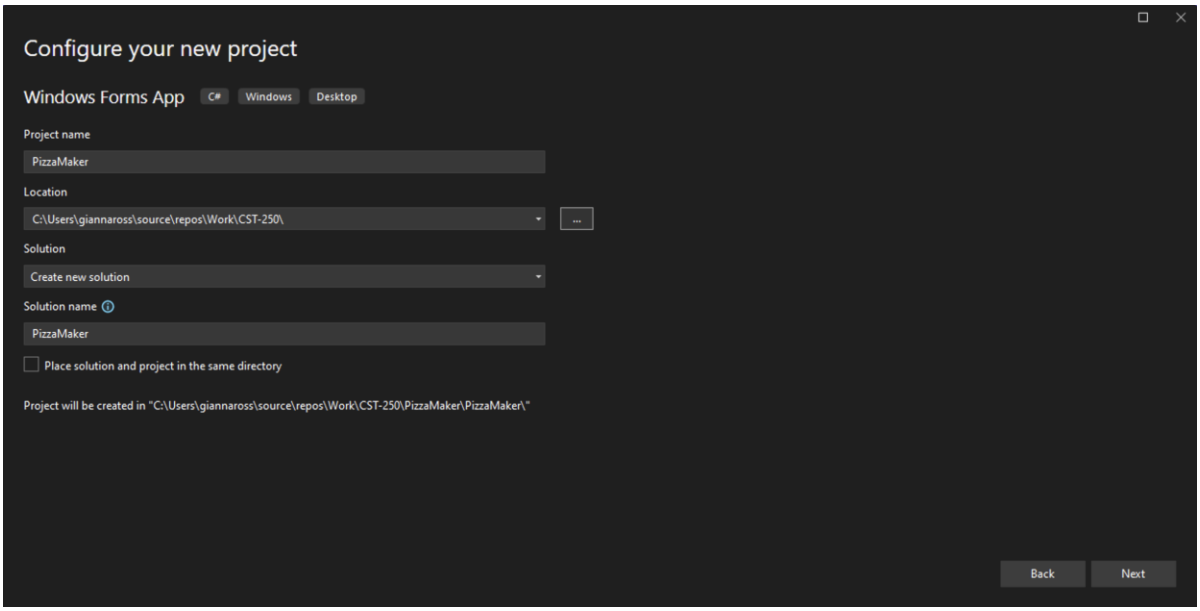


Figure 2. Naming the Windows Forms app.

- Add the project to source control.
- See Class Resources for a detailed tutorial on GitHub source control.

3. Right click on Form1 to rename as shown in Figure 3. Rename the form to "FrmPizzaMaker" as shown in Figure 4.

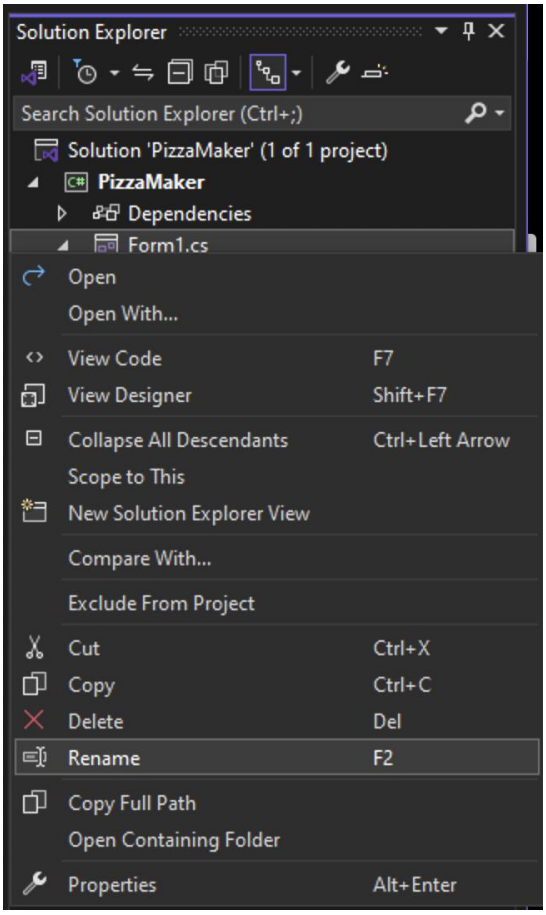


Figure 3. Renaming Form1.

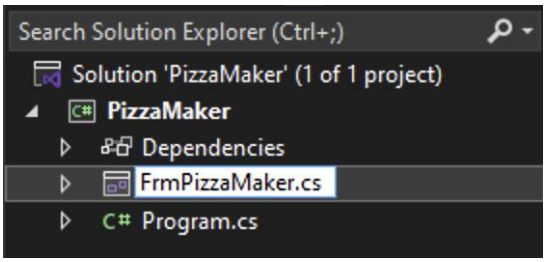


Figure 4. Updating the name of the form.

4. Next, change the Text property for the form to "Pizza Maker" as shown in Figure 5.

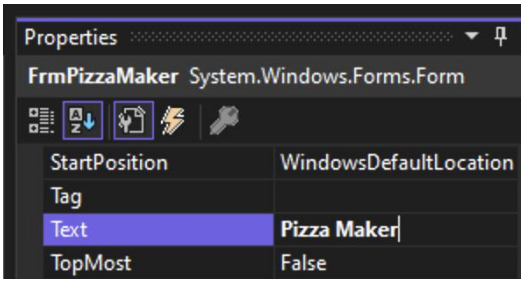


Figure 5. Updating the text for FrmPizzaMaker.

5. In the form properties, select the ellipsis in the Font property as shown in Figure 6 to change the default font for the project.

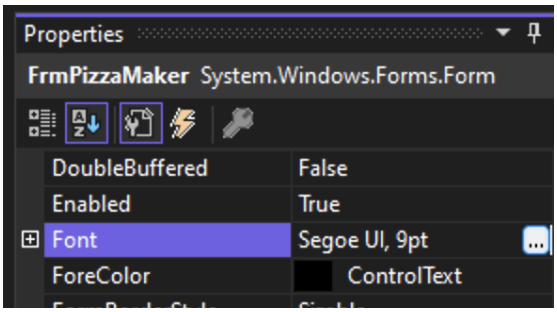


Figure 6. Accessing the Font property for a form.

6. Choose a font and size for your project as shown in Figure 7.

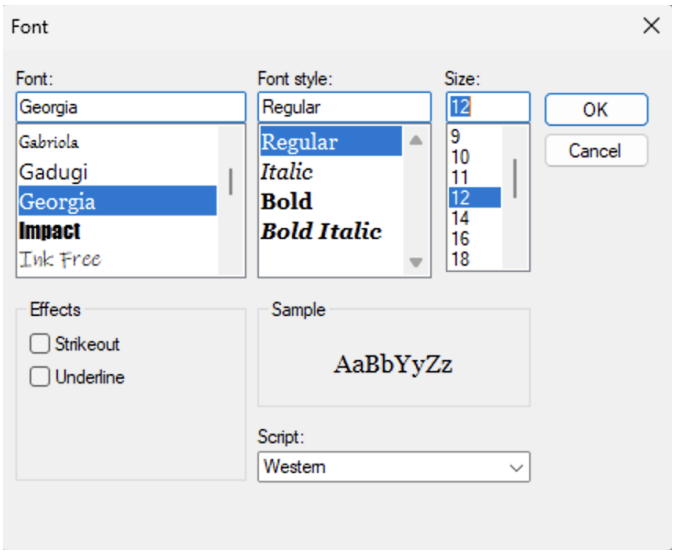


Figure 7. Changing the default font for the project.

7. Next, add a label and a text box to the top left corner of the form as shown in Figure 8. These will be used for the user to input their name.

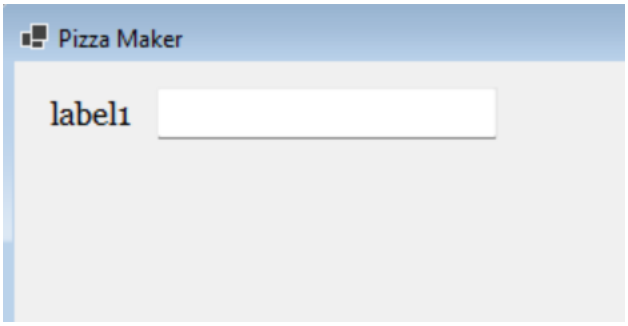


Figure 8. Setting up text entry for the user's name.

8. Change the text for the label to "Name:." Also, change the name of the text box control to "txtName" as shown in Figure 9. The form should now reflect Figure 10.

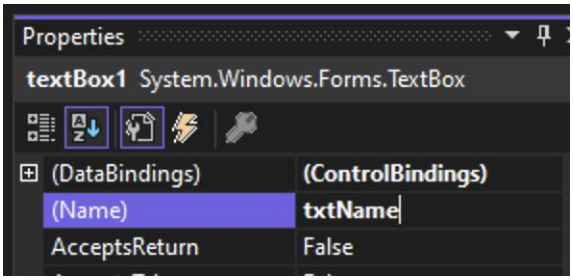


Figure 9. Changing the name of the text box.

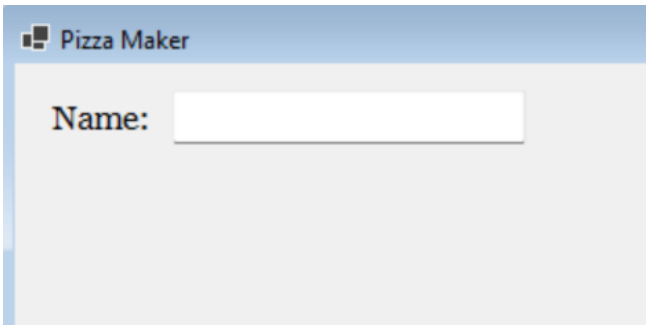


Figure 10. Updated FrmPizzaMaker with name input.

9. Add a new group box control to the form and change the text to "Ingredients" as shown in Figure 11. Use the Font property to change the text to bold as shown in Figure 12.

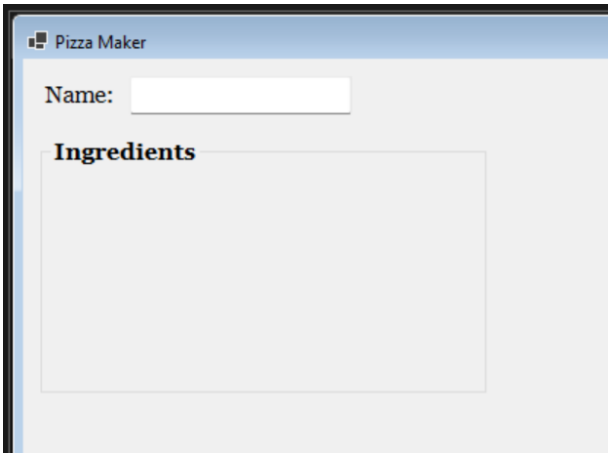


Figure 11. Adding the "Ingredients" group box.

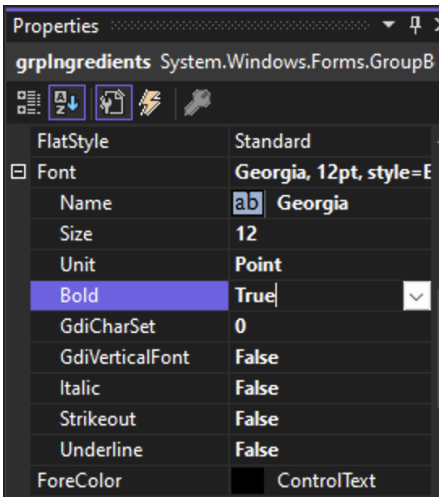


Figure 12. Changing the text to bold.

10. Update the name of the group box to grpIngredients using the naming standard for group boxes, grp, as shown in Figure 13.

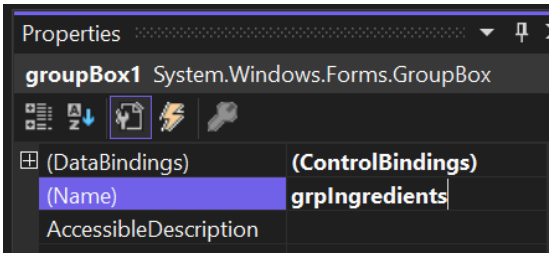


Figure 13. Renaming the "Ingredients" group box.

11. Add a checkbox control into the ingredients groups box as shown in Figure 14.

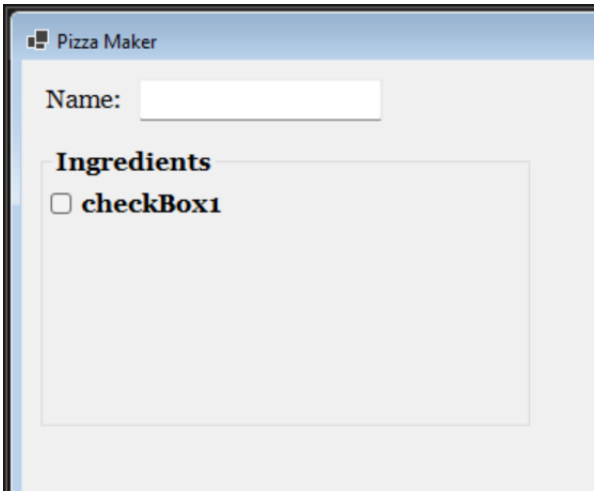


Figure 14. Adding a checkbox to the "Ingredients" group box.

12. Change the text to "Pepperoni" as shown in Figure 15. We also need to remove the bolding from the checkbox, as the control takes on the same font as the parent group box. Lastly, use the naming convention for checkboxes, chb, to name the checkbox chbPepperoni.

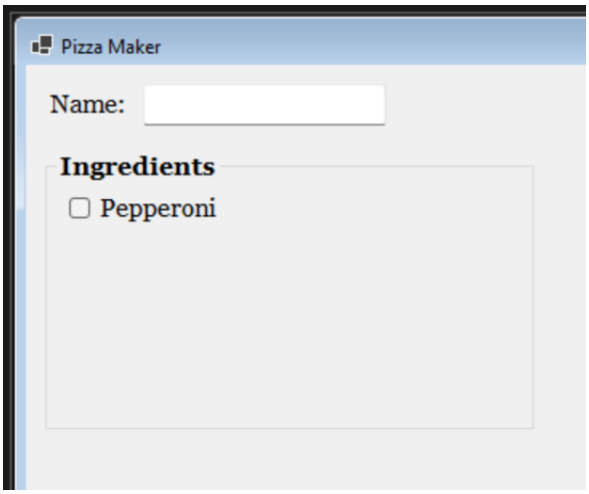


Figure 15. Formatting the pepperoni checkbox.

13. Copy the pepperoni checkbox and paste it seven times to create eight checkboxes. Change the texts for the checkboxes to "Bacon," "Olives," "Mushrooms," "Pineapple," "Sausage," "Peppers," and "Tomatoes" as shown in Figure 16. Update the names for the checkboxes based on the toppings above (chbBacon, chbOlives, etc.). Resize the "Ingredients" group box to fit the checkboxes.

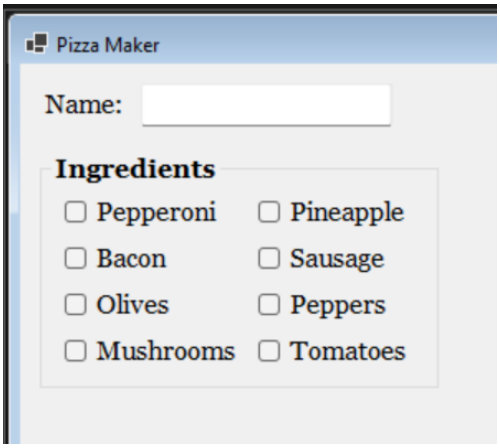


Figure 16. Finished the "Ingredients" group box.

14. Add a bolded label below the "Ingredients" group box and change the text to "Strange Add Ons" as shown in Figure 17.

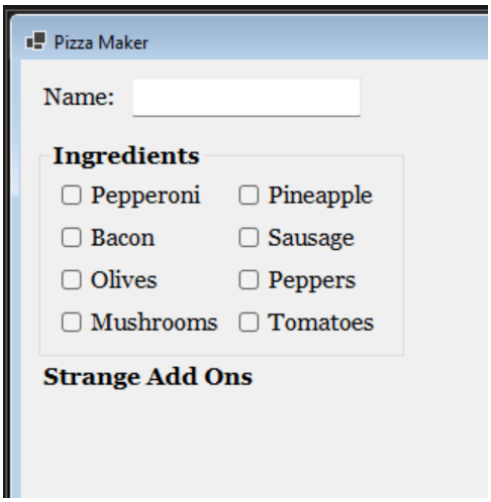


Figure 17. Adding label for strange add ons.

15. Add a list box control below the label. Rename the control to lsbStrangeAddOns using the lsb abbreviation as shown in Figure 18.

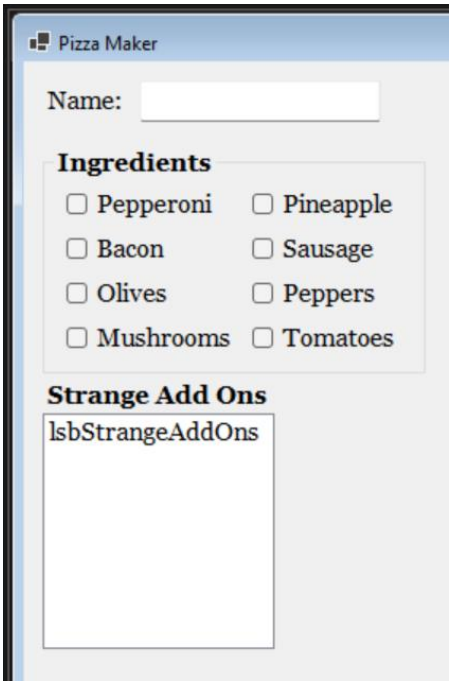


Figure 18. Adding a list box to the form.

16. Find the Items property for the list box and select the ellipsis next to "(Collection)" as shown in Figure 19. Then, fill the editor with some strange pizza toppings as shown in Figure 20. You can use the list provided here or add your own.

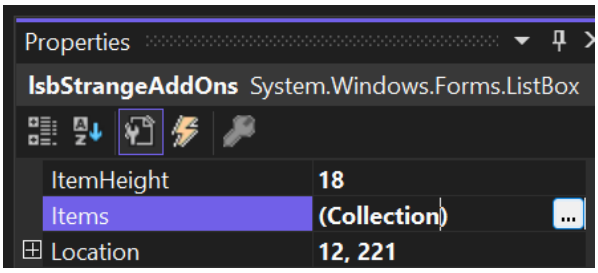


Figure 19. Editing the list box contents.

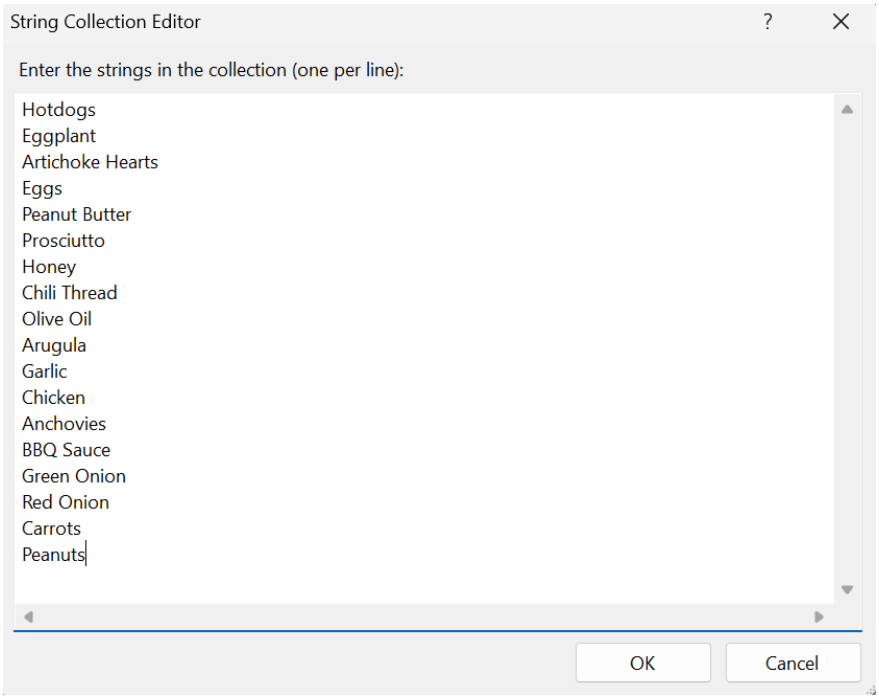


Figure 20. Strange Add On list.

17. Resize the control to fit the options. Then, find the SelectionMode property for the list box as shown in Figure 21. Change the value to "MultiSimple" so that the user can select multiple options without using "Control" or "Shift" buttons.

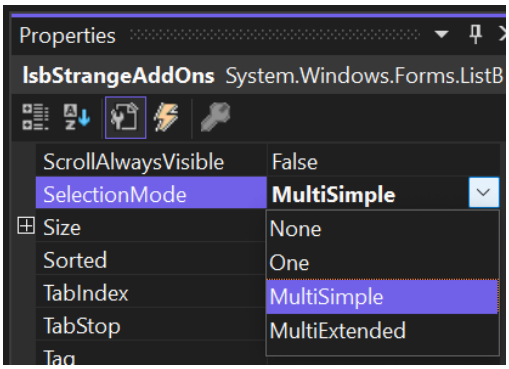


Figure 21. Updating the SelectionMode property.

18. Next, add a group box next to the "Strange Add Ons" section. Change the text to "Crust" and the font to bold as shown in Figure 22.

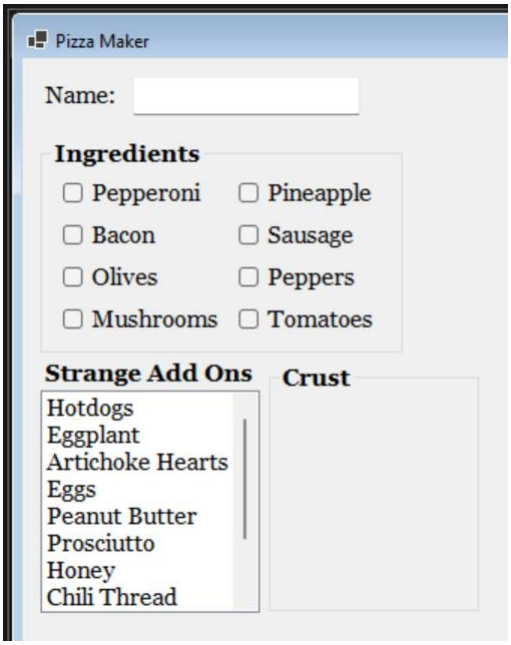


Figure 22. Creating a group box for the crust.

19. Add a new radio button to the "Crust" group box and change the text to "Thin Crust" as shown in Figure 23. The naming convention for radio buttons is rdo, so name the radio button rdoThinCrust. Make sure to remove the bolding from the button.

Figure 23. "Thin Crust" radio button.

20. Copy and paste the radio button three more times and update the name and text for each to "Deep Dish," "Stuffed Crust," and "Gluten Free" with the appropriate naming conventions for the names (rdoDeepDish, rdoStuffedCrust, etc.) as shown in Figure 24.

Figure 24. Finished the "Crust" group box.

21. Add another group box below the "Strange Add Ons" and "Crust" sections with a bolded title of "Extra Goodies" as shown in Figure 25.

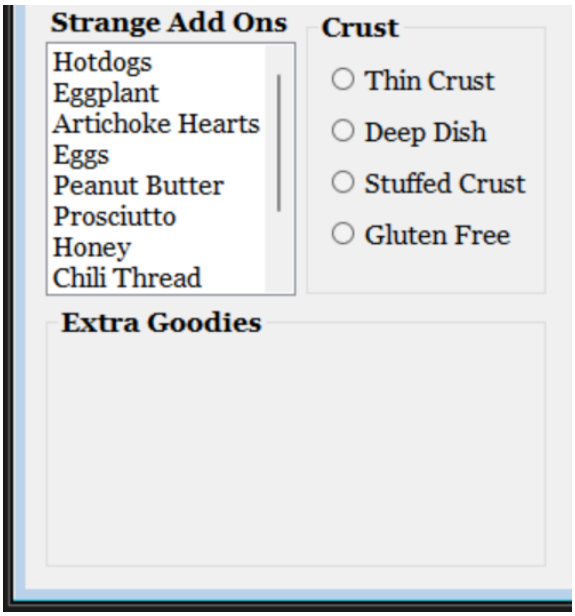


Figure 25. Adding in the "Extra Goodies" group box.

22. Add two horizonal scroll bar (HScrollBar) controls to the group box and resize them as shown in Figure 26. The naming convention for horizonal scroll bars is hsb. Name the scroll bars hsbSauce and hsbCheese. Make sure the minimum and maximum properties for both scroll bars are 0 and 100 respectively. Then, ensure the small change property is 1. This will allow the scroll bar to move in increments of 1.

Figure 26. Adding the horizonal scroll bars into the "Extra Goodies" group box.

23. Add two labels for each scroll bar as shown in Figure 27. The text for the first will be "Amount of Sauce/Cheese." The second will show a number based on the slider bar. Set the text to "00" as a placeholder and name the labels lblSauce and lblCheese respectively. Lastly, for each second label, change the TextAlign property to MiddleLeft as shown in Figure 28. Make sure none of the labels are bolded.

Figure 27. Finished the "Extra Goodies" group box.

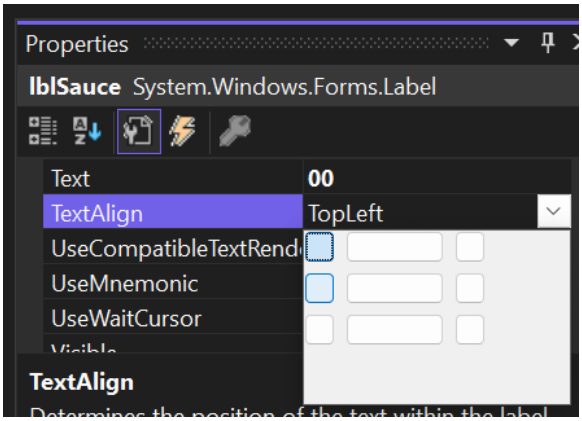


Figure 28. Updating the TextAlign property.

24. Add a new bolded label with the text of "Delivery Time" as shown in Figure 29.

Figure 29. Adding the "Delivery Time" label.

25. Add a DateTimePicker control below the new label as shown in Figure 30. Name the control dtpDeliveryTime using the naming convention for DateTimePicker controls, dtp. Resize the control so that you can see the entire date.

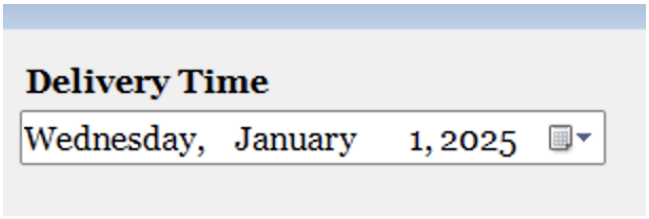


Figure 30. Adding in the date time picker control.

26. Add a bolded label below the DateTimePicker. The text should say "Pizza Box Color" as shown in Figure 31.

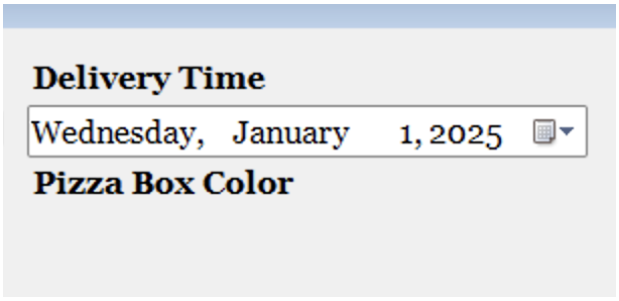


Figure 31. Adding a label for the pizza box color.

27. Add a picture box control below the label we just created as shown in Figure 32. This will allow the user to use a color picker to choose the color of their pizza box. Change the name of the picture box to picPizzaBoxColor, using the naming standard for picture boxes, pic. Use the BorderStyle property to add a border using the FixedSingle option as shown in Figure 33.

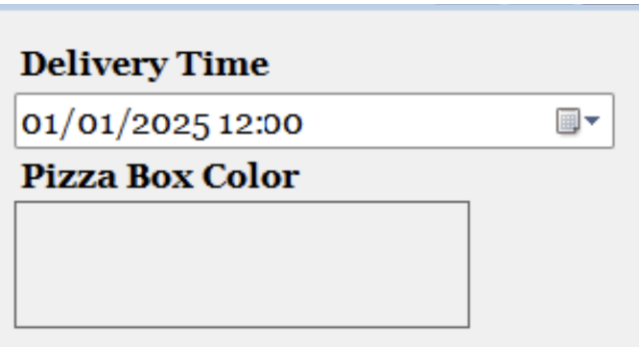


Figure 32. Adding the picture box control.

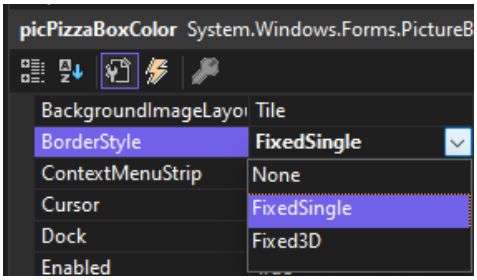


Figure 33. Adding a border to the picture box.

28. Add two more labels to the form to contain the price of the pizza as shown in Figure 34. Change the text of the first label to "Pizza Price:" and bold the text. Change the name of the other label to lblPizzaPrice and set the text to "$0" as a placeholder. This label should not be bolded. Also, change the text color to red for lblPizzaPrice.

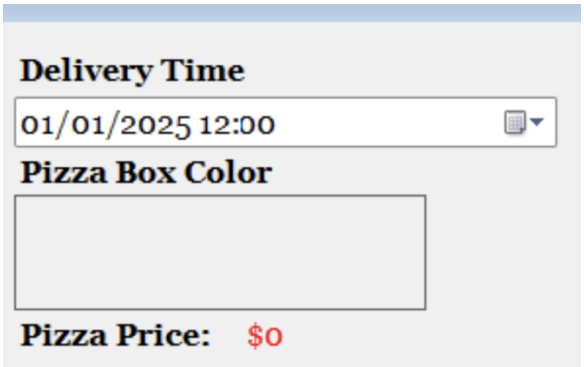


Figure 34. Adding labels for the price of the pizza.

29. Add two buttons below the pizza price as shown in Figure 35. One should say "Reset Form" and one should say "Create Pizza." The name for the buttons should be btnResetForm and btnCreatePizza respectively.

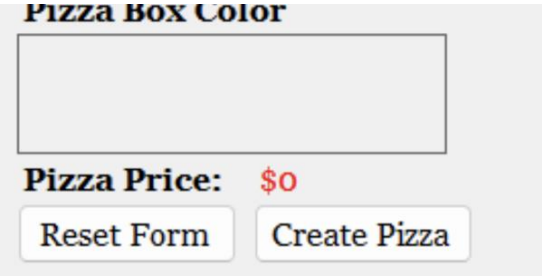


Figure 35. "Reset Form" and "Create Pizza" buttons.

30. If necessary, resize and rearrange the form controls so everything fits nicely.

- Commit all changes in the project to source control.
- Commit Message: Finish UI for the pizza maker.

- Take a screenshot of the application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Part 2 – Event Handlers

1. Create a new folder in the project named "Models." Figure 36 shows the solution explorer with the correct folder structure.

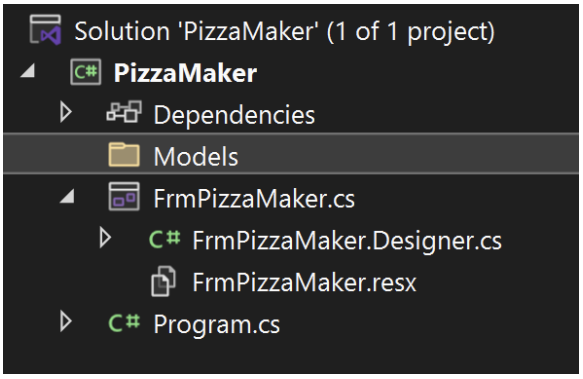


Figure 36. Folder structure for the pizza maker app.

2. In the Models folder, create a new class named PizzaModel.cs. Add a citation to the top of the class as shown in Figure 37.

```
/*
 * Gianna Ross
 * CST - 250
 * 01/01/2025
 * Pizza Maker
 * Activity 4
 */

namespace PizzaMaker.Models
{
    0 references
    internal class PizzaModel
    {

    }
}
```

Figure 37. The new PizzaModel class with a citation.

3. Add nine properties to PizzaModel as shown in Figure 38. These properties will hold the client's name (as a string), the ingredients (as a list of strings), the strange add ons (as a list of strings), the crust type (as a string), the sauce quantity (as an integer), the cheese quantity (as an integer), the delivery time (as a date time object), the pizza box color (as a color), and the price (as a decimal).

```
0 references
internal class PizzaModel
{
    // Class properties
    0 references
    public string ClientName { get; set; }
    0 references
    public List<string> Ingredients { get; set; }
    0 references
    public List<string> StrangeAddOns { get; set; }
    0 references
    public string Crust { get; set; }
    0 references
    public int SauceQty { get; set; }
    0 references
    public int CheeseQty { get; set; }
    0 references
    public DateTime DeliveryTime { get; set; }
    0 references
    public Color PizzaBoxColor { get; set; }
    0 references
    public decimal Price { get; set; }
}
```

Figure 38. Properties for PizzaModel class.

4. Create a new default constructor for the PizzaModel class as shown in Figure 39. The constructor will initialize the properties to default values. The base price for our pizza will be $15, so that will be our default.

```
/// <summary>
/// Default Constructor for Pizza Model
/// </summary>
0 references
public PizzaModel()
{
    // Declare the default properties
    ClientName = "Unknown";
    Ingredients = new List<string>();
    StrangeAddOns = new List<string>();
    Crust = "Unknown";
    SauceQty = 0;
    CheeseQty = 0;
    DeliveryTime = DateTime.Now;
    PizzaBoxColor = Color.White;
    Price = 15m;
}
```

Figure 39. Default constructor for pizza model class.

5. Next, open the code for FrmPizzaMaker and add your citation to the top. Then, add a new private class level variable of type PizzaModel named _pizza as shown in Figure 40. This variable will hold the contents of the current pizza order.

```
/*
 * Gianna Ross
 * CST - 250
 * 01/01/2025
 * Pizza Maker
 * Activity 4
 */

using PizzaMaker.Models;

namespace PizzaMaker
{
    3 references
    public partial class FrmPizzaMaker : Form
    {
        // Class level variable declarations
        private PizzaModel _pizza;
```

Figure 40. Citation and class level variable for FrmPizzaMaker.

6. Find the default constructor generated by visual studio for FrmPizzaMaker. Add a summary comment and the initialization for the _pizza variable as shown in Figure 41.

```
3 references
public partial class FrmPizzaMaker : Form
{
    // Class level variable declarations
    private PizzaModel _pizza;

    /// <summary>
    /// Default constructor for FrmPizzaMaker
    /// </summary>
    1 reference
    public FrmPizzaMaker()
    {
        InitializeComponent();
        // Initialize the current order
        _pizza = new PizzaModel();
    }
```

Figure 41. Initializing the pizza variable.

7. We will make our program so that a user cannot create their pizza without entering a name for the order. To do that, we will need to disable the "Create Pizza" button until the user enters a name. Disabling the button will be done in the constructor using the control's Enabled property as shown in Figure 42. We should also disable the "Reset Form" button.

```
/// <summary>
/// Default constructor for FrmPizzaMaker
/// </summary>
1 reference
public FrmPizzaMaker()
{
    InitializeComponent();
    // Initialize the current order
    _pizza = new PizzaModel();

    // Disable the Create Pizza button
    btnCreatePizza.Enabled = false;
    // Disable the Reset Form button
    btnResetForm.Enabled = false;
}
```

Figure 42. Hiding the button to create a pizza.

8. Next, create a method named EnablePizzaCreation that will enable both buttons as shown in Figure 43. While we could include the code in the event handler for the name text box, this will allow us to call the method in multiple locations, if necessary.

```
/// <summary>
/// Enables the reset and create buttons
/// for the order pizza form
/// </summary>
0 references
public void EnablePizzaCreation()
{
    // Enable the Create Pizza button
    btnCreatePizza.Enabled = true;
    // Enable the Reset Form button
    btnResetForm.Enabled = true;
}
```

Figure 43. Creating the EnablePizzaCreation method.

9. To see if the user has entered a name, we will attach a Leave event handler to the name text box. In the FrmPizzaMaker designer, navigate to the events section for txtName and find the Leave event handler. Create a new Leave event handler named TxtNameLeaveEH as shown in Figure 44. This event handler will trigger ever time the user leaves the txtName text box.

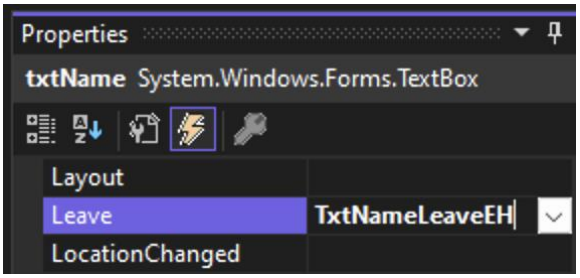


Figure 44. Attaching an event handler to txtName.

10. Add a summary comment to the new method created as shown in Figure 45.

```
/// <summary>
/// Leave event handler for txtName
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void TxtNameLeaveEH(object sender, EventArgs e)
{

}
```

Figure 45. Leave event handler method with summary comment.

11. This method will use the text box to set the client's name for the pizza variable as shown in Figure 46. Additionally, call the EnablePizzaCreation method we just created.

```
/// <summary>
/// Leave event handler for txtName
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void TxtNameLeaveEH(object sender, EventArgs e)
{
    // Set the pizzas client name to the text of txtName
    _pizza.ClientName = txtName.Text;
    // Call the Enable Pizza Creation method
    EnablePizzaCreation();
}
```

Figure 46. TxtNameLeaveEH to set client's name and show create pizza button.

12. Create a new method in FrmPizzaMaker named "UpdatePrice." This method will be called whenever we update the contents of our pizza so that we can keep our price label up to date. The method will calculate the price of our pizza as shown in Figure 47 and update the pizza price and lblPizzaPrice. Our pizza will have a base price of $15 with an additional $0.50 for each ingredient and special add-on as well as $1 extra for gluten-free crust.

```
/// <summary>
/// Update the price of the pizza
/// </summary>
0 references
public void UpdatePrice()
{
    // Declare and initialize
    decimal price = 15;

    // Add 50 cents for each ingredient
    price += (_pizza.Ingredients.Count * .50m);

    // Add 50 cents for each special add on
    price += (_pizza.StrangeAddOns.Count * .50m);

    // Add $1 if the crust if gluten free
    if (_pizza.Crust == "Gluten Free")
    {
        price += 1;
    }
    // Update the price of the pizza
    _pizza.Price = price;
    // Update lblPizzaPrice
    lblPizzaPrice.Text = $"{price:C2}";
}
```

Figure 47. Method to update the price of the pizza.

13. Call the UpdatePrice method in the FrmPizzaMaker constructor as shown in Figure 48 so that the base price is automatically shown to the user.

```
1 reference
public FrmPizzaMaker()
{
    InitializeComponent();
    // Initialize the current order
    _pizza = new PizzaModel();

    // Disable the Create Pizza button
    btnCreatePizza.Enabled = false;
    // Disable the Reset Form button
    btnResetForm.Enabled = false;
    // Update the price of the pizza
    UpdatePrice();
}
```

Figure 48. Updating the price of the pizza in the form constructor.

- Commit all changes in the project to source control.
- Commit Message: Add logic so the user cannot create a pizza without a name entered and logic to keep the price of the pizza updated.

14. Next, create a CheckedChanged event handler for chbPepperoni named ChbIngredientCheckedChangedEH as shown in Figure 49. We will use the same event handler for all eight of the ingredient checkboxes. The event handler will trigger every time the user checks or unchecks the checkbox. Remember to add a summary comment to the new method as shown in Figure 50.

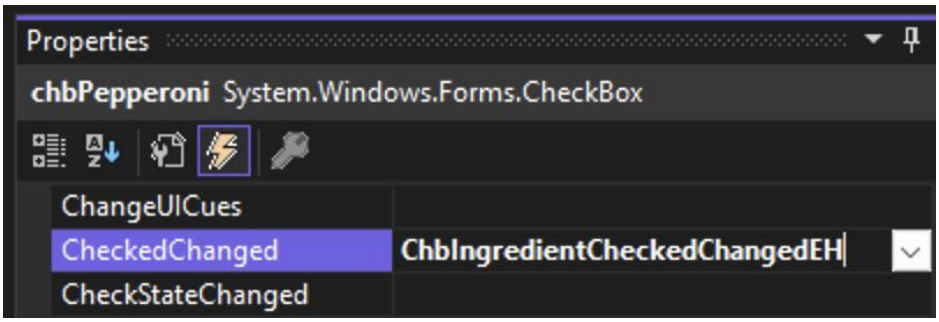


Figure 49. Creating the CheckedChanged click event handler.

```
/// <summary>
/// Checked changed event handler for ingredient check boxes
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void ChbIngredientCheckedChangedEH(object sender, EventArgs e)
{

}
```

Figure 50. Summary comment for ChbIngredientCheckedChangedEH.

15. First, use the sender object to get the CheckBox control that was changed. Then, make sure the checkbox variable is not null and add or remove the current ingredient based on the Checked property and update the pizza's price as shown in Figure 51.

```
/// <summary>
/// Checked changed event handler for ingredient check boxes
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void ChbIngredientCheckedChangedEH(object sender, EventArgs e)
{
    // Get the check box from the sender parameter
    CheckBox checkbox = sender as CheckBox;
    // Make sure the checkbox is not null
    if (checkbox != null)
    {
        // If the checkbox is checked, add the ingredient to the pizza
        if (checkbox.Checked)
        {
            // Add the current ingredient to the pizza
            _pizza.Ingredients.Add(checkbox.Text);
        }
        // If the checkbox is not checked, remove the ingredient
        else
        {
            // Remove the current ingredient from the pizza
            _pizza.Ingredients.Remove(checkbox.Text);
        }
    }
    // Update the price of the pizza
    UpdatePrice();
}
```

Figure 51. Method to add/remove ingredients from the pizza.

16. Attach the same event handler to the CheckedChanged event for the other seven checkboxes in the "Ingredients" group box as shown in Figure 52.

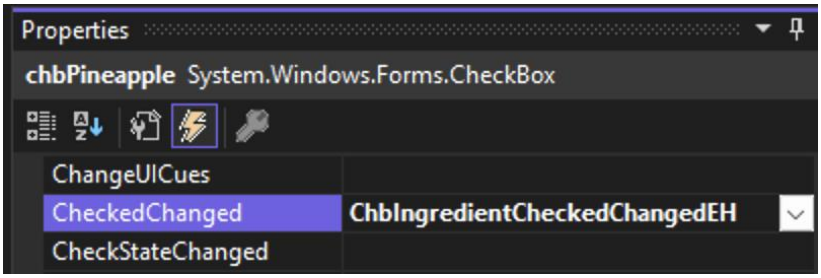


Figure 52. Adding the ChbIngredientCheckedChangedEH event handler to the other checkboxes.

17. Similarly to the "Ingredients" group box, we should add the strange add ons to the pizza as they are selected. To do this, add an event handler to lsbStrangeAddOns for the SelectedIndexChanged event named LsbStrangeAddOnsSelectedIndexChangedEH as shown in Figure 53. Add a summary comment to the new method as shown in Figure 54.

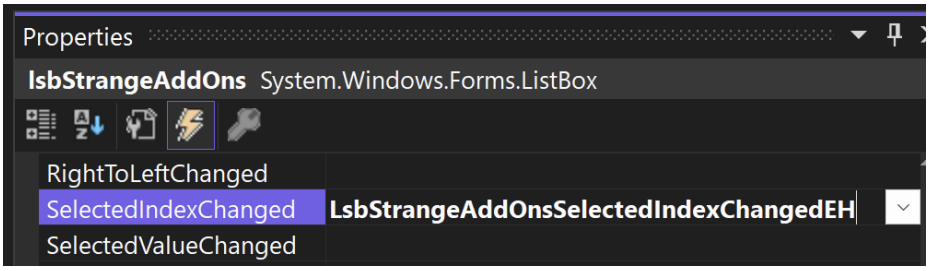


Figure 53. Adding a SelectedIndexChanged event handler.

```
/// <summary>
/// Selected Index Changed event handler for lsbStrangeAddOns
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void LsbStrangeAddOnsSelectedIndexChangedEH(object sender, EventArgs e)
{

}
```

Figure 54. LsbStrangeAddOnsSelectedIndexChangedEH method with a summary comment.

18. First, get the SelectedItems property from lsbStrangeAddOns and cast it to a string. Then, send the result to a list as shown in Figure 55. Finally, update the price of the pizza.

```
/// <summary>
/// Selected Index Changed event handler for lsbStrangeAddOns
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void LsbStrangeAddOnsSelectedIndexChangedEH(object sender, EventArgs e)
{
    // Get the list of selected items and set the StrangeAddOns property of the pizza
    _pizza.StrangeAddOns = lsbStrangeAddOns.SelectedItems.Cast<string>().ToList();
    // Update the price of the pizza
    UpdatePrice();
}
```

Figure 55. Updating the Strange Add Ons list for the pizza.

19. Next, we will work with the "Crust" group box. This set of event handlers will be done the same way as the event handlers for the "Ingredients" group box. First, create a CheckedChanged event handler for rdoThinCrust named RdoCrustCheckedChangedEH as shown in Figure 56. Make sure to add a summary comment to the new method as shown in Figure 57.

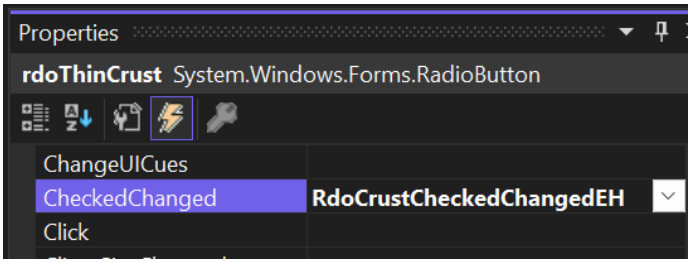


Figure 56. Adding a CheckedChanged event handler to the "Thin Crust" radio button.

```
/// <summary>
/// Checked changed event handler for crust radio buttons
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void RdoCrustCheckedChangedEH(object sender, EventArgs e)
{

}
```

Figure 57. Summary comment for RdoCrustCheckedChangedEH.

20. First, cast the sender object to a RadioButton. Then, make sure the new variable is not null and check if the radio button is checked. If it is, set the pizza's crust to the contents of the selected radio button as shown in Figure 58. Finally, update the price of the pizza.

```
/// <summary>
/// Checked changed event handler for crust radio buttons
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void RdoCrustCheckedChangedEH(object sender, EventArgs e)
{
    // Get the radio button from the sender object
    RadioButton radioButton = sender as RadioButton;
    // Make sure the radio button is not null
    if (radioButton != null && radioButton.Checked)
    {
        // Set the current crust to the pizzas crust
        _pizza.Crust = radioButton.Text;
    }
    // Update the price of the pizza
    UpdatePrice();
}
```

Figure 58. Logic to update the crust of the pizza.

21. Add RdoCrustCheckedChangedEH as the CheckedChanged event handler for the other three crust radio button as shown in Figure 59.

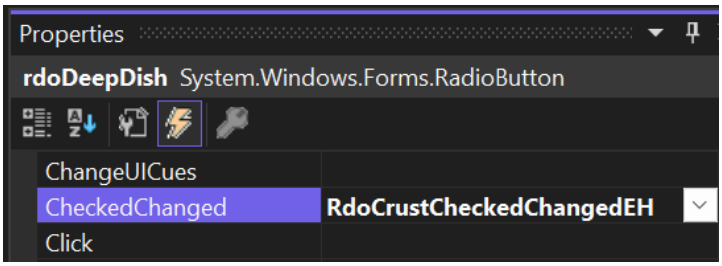


Figure 59. Adding the CheckedChanged event handler to the other crust radio buttons.

- Commit all changes in the project to source control.
- Commit Message: Add event handler for the ingredients, strange add ons, and crust for each pizza.

22. Next, add a ValueChanged event handler to hsbSauce and hsbCheese named HsbExtraGoodiesValueChangedEH as shown in Figure 60. This event will trigger whenever the value of the scroll bar is changed. We will use the same method for both hsbSauce and hsbCheese. Add a summary comment to the created method as shown in Figure 61.

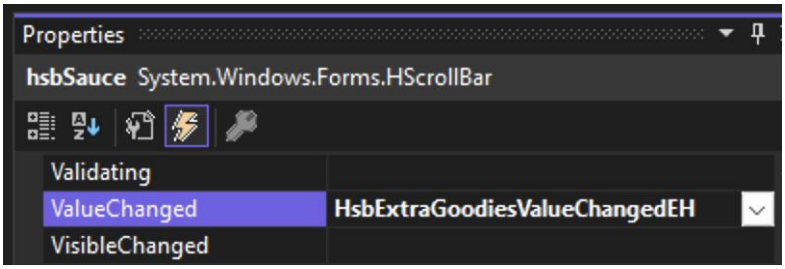


Figure 60. Adding a Scroll event handler.

```
/// <summary>
/// Value changed event handler for the horizonal scroll bars
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
2 references
private void HsbExtraGoodiesValueChangedEH(object sender, EventArgs e)
{

}
```

Figure 61. Summary comment for HsbExtraGoodiesValueChangedEH.

23. Like the checkboxes and radio buttons, start by casting the sender object to an HScrollBar variable. Then, make sure the variable is not null. We can then use the names of our scroll bars to identify which scroll bar called the method. Use that information to update the Pizza property using the scroll bars value as shown in Figure 62. Lastly, update the value of the corresponding label on the form.

```
/// <summary>
/// Value changed event handler for the horizonal scroll bars
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
2 references
private void HsbExtraGoodiesValueChangedEH(object sender, EventArgs e)
{
    // Cast the sender object to an HScrollBar
    HScrollBar scrollBar = sender as HScrollBar;
    // Make sure the scroll bar is not null
    if (scrollBar != null)
    {
        // Check if the scroll bar is hsbSauce
        if (scrollBar == hsbSauce)
        {
            // Updated the SauceQty using the scroll bars value
            _pizza.SauceQty = scrollBar.Value;
            // Update the lblSauce label
            lblSauce.Text = scrollBar.Value.ToString();
        }
        // Check if the scroll bar is hsbCheese
        else if (scrollBar == hsbCheese)
        {
            // Updated the CheeseQty using the scroll bars value
            _pizza.CheeseQty = scrollBar.Value;
            // Update the lblCheese label
            lblCheese.Text = scrollBar.Value.ToString();
        }
    }
}
```

Figure 62. Logic for horizonal scroll bars.

24. If you run your application and move either of the horizonal scroll bars to the far right, you will see that the value of the label only reaches 91 as shown in Figure 63. This is because the maximum value accounts for the scrollable range and the size of the slider itself. To fix this, we can update the FrmPizzaMaker constructor to change the maximum of both scroll bars to 100 plus the large change value of the scroll bar minus one as shown in Figure 64. This will allow us to reach the maximum value for the scroll bar.

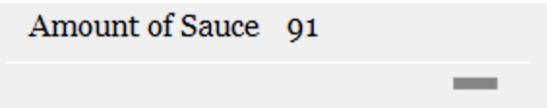


Figure 63. Actual maximum for the horizonal scroll bar.

```
/// <summary>
/// Default constructor for FrmPizzaMaker
/// </summary>
1 reference
public FrmPizzaMaker()
{
    InitializeComponent();
    // Initialize the current order
    _pizza = new PizzaModel();

    // Disable the Create Pizza button
    btnCreatePizza.Enabled = false;
    // Disable the Reset Form button
    btnResetForm.Enabled = false;
    // Update the price of the pizza
    UpdatePrice();

    // Update the maximums for hsbSauce and hsbCheese
    hsbSauce.Maximum = 100 + hsbSauce.LargeChange - 1;
    hsbCheese.Maximum = 100 + hsbCheese.LargeChange - 1;
}
```

Figure 64. Update the forms constructor for the maximums for the horizonal scroll bars.

25. Next, we will update the date time picker for the pizza's delivery time. In the form designer, find the date time picker's Format property and change the value to "Custom" as shown in Figure 65.

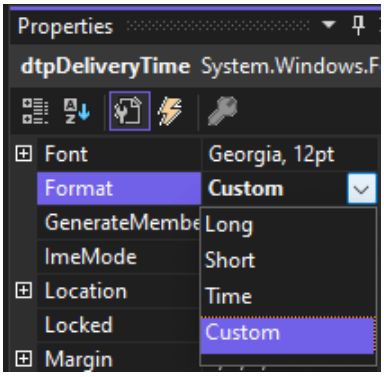


Figure 65. Updating the Format property for dtpDeliveryTime.

26. Next, find the CustomFormat property and enter "mm/dd/yyyy hh:mm" onto the field as shown in Figure 66. This will format dates to look like 01/01/2025 12:00.

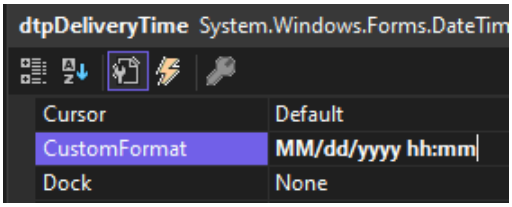


Figure 66. Adding the custom formatting to the delivery time control.

27. Next, attach a ValueChanged event handler to dtpDeliveryTime named DtpDeliveryTimeValueChangedEH as shown in Figure 67. This will trigger when the value of the date time picker is changed. Add a summary comment to the created method as shown in Figure 68.

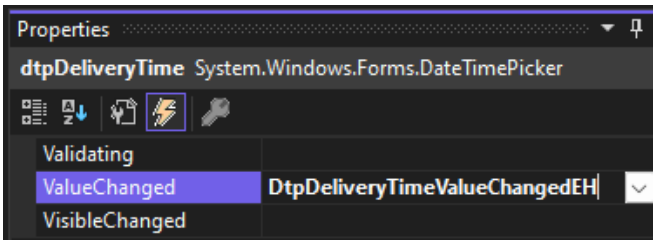


Figure 67. Adding a ValueChanged event handler to dtpDeliveryTime.

```
/// <summary>
/// Value changed event handler for dtpDeliveryTime
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void DtpDeliveryTimeValueChangedEH(object sender, EventArgs e)
{

}
```

Figure 68. Summary comment for DtpDeliveryTimeValueChangedEH.

28. Update the pizza's delivery time by getting the value from the date time picker as shown in Figure 69.

```
/// <summary>
/// Value changed event handler for dtpDeliveryTime
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void DtpDeliveryTimeValueChangedEH(object sender, EventArgs e)
{
    // Update the delivery time for the pizza
    _pizza.DeliveryTime = dtpDeliveryTime.Value;
}
```

Figure 69. Logic to update the pizza's delivery time.

- Commit all changes in the project to source control.
- Commit Message: Add event handlers for the extra goodies and delivery time.

29. For the pizza box color picker, we will attach a Click event handler to picPizzaBoxColor named PicPizzaBoxColorClickEH. Add a summary comment to the created method as shown in Figure 70.

```
/// <summary>
/// Click event handler for picPizzaBoxColor
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void PicPizzaBoxColorClickEH(object sender, EventArgs e)
{

}
```

Figure 70. PicPizzaBoxColorClickEH method with summary comment.

30. To let the user pick a color for his/her pizza box, we will need to use a ColorDialog object. This object works similarly to a form to let the user select a color. Then, if the color return was successful, the color dialog returns a DialogResult.OK. First, create a new ColorDialog object named pizzaBoxColorPicker. Then, call the ShowDialog method from pizzaBoxColorPicker and save the result to a DialogResult variable named result as shown in Figure 71. Check to make sure the result is DialogResult.OK. If so, set the pizza's PizzaBoxColor to the pizzaBoxColorPicker's Color property and update the color of the picture box using the BackColor property so that our picture box reflects the user's color choice.

```
/// <summary>
/// Click event handler for picPizzaBoxColor
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void PicPizzaBoxColorClickEH(object sender, EventArgs e)
{
    // Create a new color dialog object
    ColorDialog pizzaBoxColorPicker = new ColorDialog();
    // Call the ShowDialog method
    DialogResult result = pizzaBoxColorPicker.ShowDialog();
    // Check if the color picker returned OK
    if (result == DialogResult.OK)
    {
        // Set the pizza pizza box color
        _pizza.PizzaBoxColor = pizzaBoxColorPicker.Color;
        // Set the color of the picture box
        picPizzaBoxColor.BackColor = pizzaBoxColorPicker.Color;
    }
}
```

Figure 71. Using a ColorDialog to let the user pick a pizza box color.

31. Next, add a Click event handler to btnResetForm named BtnResetFormClickEH. Add a summary comment to the method as shown in Figure 72. Call the ResetForm method. We will create this method next.

```
/// <summary>
/// Click event handler for btnResetForm
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnResetFormClickEH(object sender, EventArgs e)
{
    // Reset the form
    ResetForm();
}
```

Figure 72. BtnResetFormClickEH with method calls.

32. Create a new private method ResetForm that returns void as shown in Figure 73. Remember to add a summary comment. Start by resetting the pizza variable by setting it to a new PizzaModel object. Then, make two method calls in the event handler, one to the ResetControls method we will create next and one to the UpdatePrice method. We will pass the entire form into the ResetControls method so that we can reset all the controls on the form.

```
/// <summary>
/// Reset the pizza maker form
/// </summary>
1 reference
private void ResetForm()
{
    // Set the pizza to a new instance
    _pizza = new PizzaModel();
    // Reset the controls of the form
    ResetControls(this);
    // Update the price of the pizza
    UpdatePrice();
}
```

Figure 73. Method to reset the form.

33. Create a new private method named ResetControls that will take a Control parentControl as a parameter. Add a summary comment and end of method comment as shown in Figure 74.

```
/// <summary>
/// Reset the controls within the parent control
/// </summary>
/// <param name="parentControl"></param>
1 reference
private void ResetControls(Control parentControl)
{

} // End of ResetControls method
```

Figure 74. Setting up the ResetControls method.

34. This method will loop through the controls within the parameter and get the type of each control. We can do this using the GetType method and save the type to a string using the Name property. Then, we will set up a switch statement to handle each type. Finally, we will check if the current control contains controls using the "HasChildren" property as shown in Figure 75. If so, we will use recursion to call the ResetControls method on the current control.

```
/// <summary>
/// Reset the controls within the parent control
/// </summary>
/// <param name="parentControl"></param>
2 references
private void ResetControls(Control parentControl)
{
    // Loop through the controls within the parent control
    foreach (Control control in parentControl.Controls)
    {
        // Get the type of the control
        Type controlType = control.GetType();
        // Save the type of the control as a string
        string type = controlType.Name.ToString();
        // Use a switch case to handle the resets
        switch (type)
        {

        }
        // Check if the control has controls (children)
        if (control.HasChildren)
        {
            // Recursively call the Reset method using the current control
            ResetControls(control);
        }
    }
} // End of ResetControls method
```

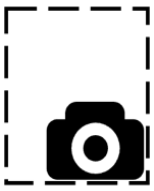
Figure 75. Setting up the reset method.

35. Next, we will finish the switch statement for the type string. We can do this by creating cases for each type of control we have on the form. For each control, cast the Control variable to the correct type and reset the control based on its type as shown in Figure 76.

```
// Use a switch case to handle the resets
switch (type)
{
    case "TextBox":
        // Cast the control to a textbox
        TextBox textbox = (TextBox)control;
        // Clear the textbox
        textbox.Clear();
        break;
    case "CheckBox":
        // Cast the control to a checkbox
        CheckBox checkbox = (CheckBox)control;
        // Make sure the checkbox is not checked
        checkbox.Checked = false;
        break;
    case "ListBox":
        // Cast the control to a list box
        ListBox listbox = (ListBox)control;
        // Clear the selected items in the list box
        listbox.ClearSelected();
        break;
    case "RadioButton":
        // Cast the control to a radio button
        RadioButton radioButton = (RadioButton)control;
        // Make sure the radio button is not checked
        radioButton.Checked = false;
        break;
    case "HScrollBar":
        // Cast the control to a horizonal scroll bar
        HScrollBar hScrollBar = (HScrollBar)control;
        // Set the scroll bars value to 0
        hScrollBar.Value = 0;
        break;
    case "DateTimePicker":
        // Cast the control to a date time picker
        DateTimePicker dateTimePicker = (DateTimePicker)control;
        // Set the date to 1/1/2025 12:00am
        dateTimePicker.Value = new DateTime(2025, 1, 1, 0, 0, 0);
        break;
    case "PictureBox":
        // Cast the control to a picture box
        PictureBox pictureBox = (PictureBox)control;
        // Change the picture box back color to the default
        pictureBox.BackColor = SystemColors.Control;
        break;
}
```

Figure 76. Using a switch case to reset each type of control.

- Commit all changes in the project to source control.
- Commit Message: Finish functionality for the UI controls on the pizza maker form.



- Take a screenshot of the application running at this point.
- Take a screenshot of the citations for PizzaModel and FrmPizzaMaker. Make sure to include the class declarations in the images.
- Take a screenshot of the default constructors for PizzaModel and FrmPizzaMaker.
- Take a screenshot of the EnablePizzaCreation, TxtNameLeaveEH, UpdatePrice, ChbIngredientCheckedChangedEH, LsbStrangeAddOnsSelectedIndexChangedEH, RdoCrustCheckedChangedEH, HsbExtraGoodiesValueChangedEH, DtpDeliveryTimeValueChangedEH, PicPizzaBoxColorClickEH, BtnResetFormClickEH, ResetForm, and ResetControls methods in FrmPizzaMaker.
- Take a screenshot of the UMLs for the PizzaModel and FrmPizzaMaker classes.
- Paste the images into a Word document.
- Put a caption below each image explaining what is being demonstrated.

Part 3 – N-Layer Architecture

1. We will begin our n-layer architecture by creating a project to hold the backend code for our application. Right click on the solution and choose Add > New Project as shown in Figure 77.

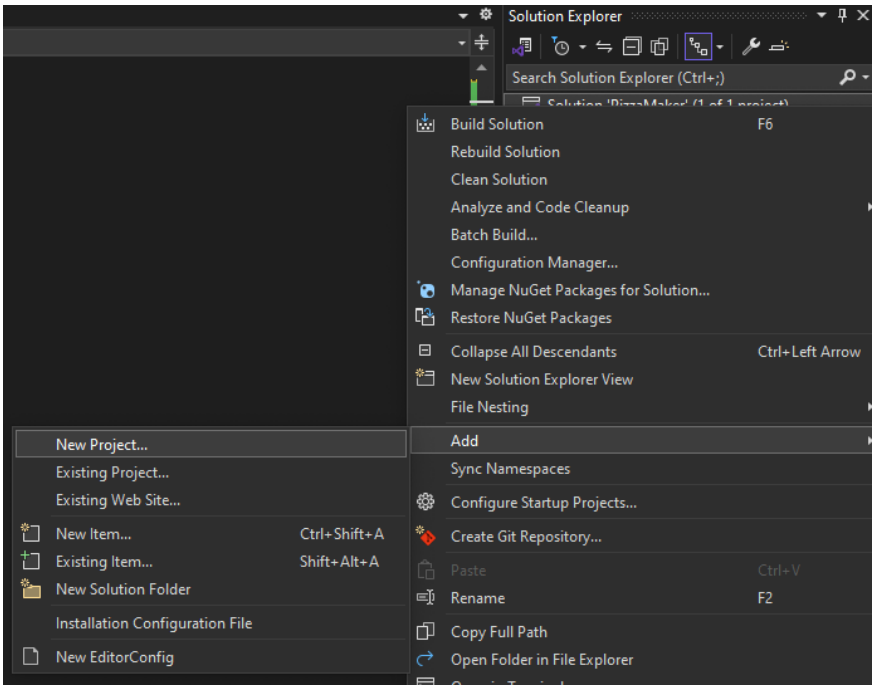


Figure 77. Adding a new project to the pizza maker.

2. Add a new Class Library project to the solution named "PizzaMakerClassLibrary" as shown in Figure 78. Use the most recent long term support version of .NET.

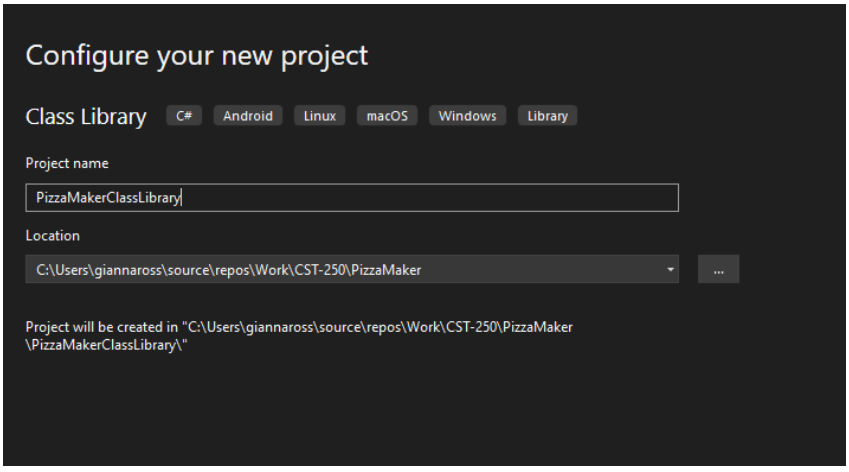


Figure 78. Naming the new class library PizzaMakerClassLibrary.

3. Delete Class1.cs. Then, create a new folder in the project named "Services." Inside the Services folder, add two folders named BusinessLogicLayer and DataAccessLayer. Figure 79 shows the solution explorer with the correct folder structure.

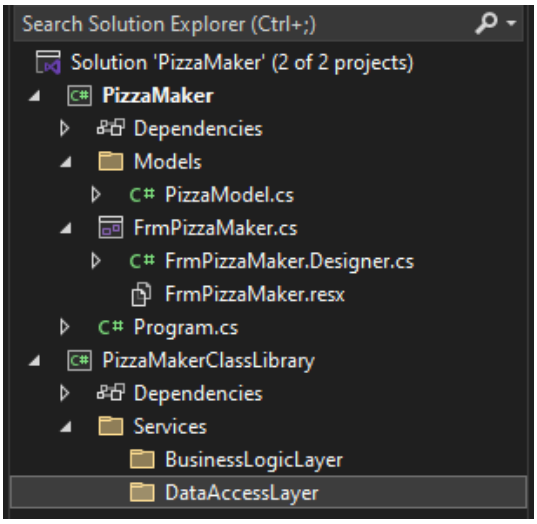


Figure 79. Folder structure for PizzaMakerClassLibrary.

4. Next, add classes for our business logic layer and data access layer. Create a class named PizzaLogic.cs in the BusinessLogicLayer folder we created. In the DataAccessLayer folder, add a class named PizzaDAO.cs. The solution explorer should reflect Figure 80 after both classes are added.

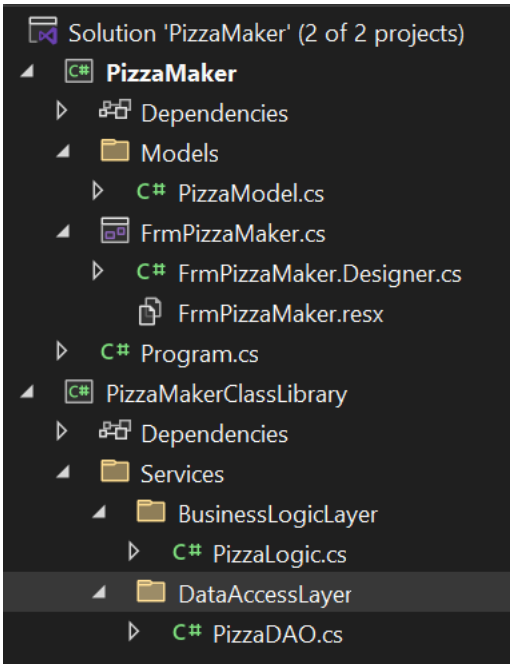


Figure 80. Updated Solution Explorer with DAO and logic classes.

5. Open PizzaDAO.cs and make the class public. This is because we will use this class in our PizzaMaker project. Add a citation to the top. Then, create a private list of type PizzaModel that will store the list of pizzas in the current order as shown in Figure 81.

```
/*
 * Gianna Ross
 * CST - 250
 * 01/01/2025
 * Pizza Maker
 * Activity 4
 */

namespace PizzaMakerClassLibrary.Services.DataAccessLayer
{
    1 reference
    public class PizzaDAO
    {
        // Class level variables
        private List<PizzaModel> _pizzaOrder;
    }
}
```

Figure 81. Adding a pizzaOrder list to our DAO.

6. Set up a constrictor for the PizzaDAO to initialize the _pizzaOrder variable as shown in Figure 82.

```
// Class level variables
private List<PizzaModel> _pizzaOrder;

/// <summary>
/// Default constructor for the pizza DAO
/// </summary>
0 references
public PizzaDAO()
{
    // Initialize the _pizzaOrder list
    _pizzaOrder = new List<PizzaModel>();
}
```

Figure 82. Initializing the pizza orders list.

- Commit all changes in the project to source control.
- Commit Message: Add a class library and pizza DAO and pizza logic classes.

7. We are getting errors for the PizzaModel object because there is no reference to the PizzaMaker project, where the model is stored. To fix this, right click on PizzaMakerClassLibrary and select Add > Project Reference… Check the box for the PizzaMaker project as shown in Figure 83 and select OK.

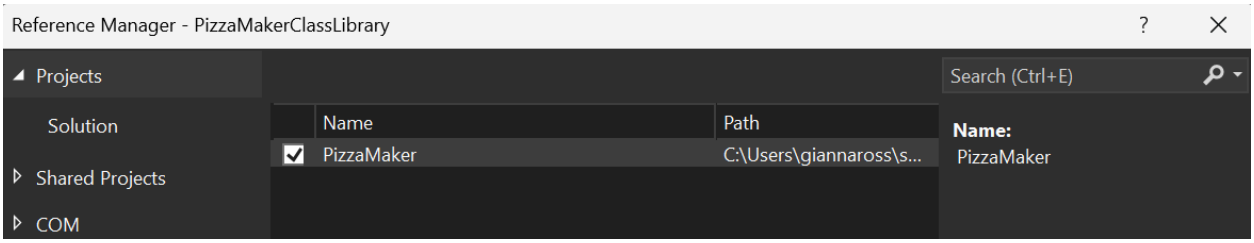


Figure 83. Adding the PizzaMaker project reference.

8. As you can see in Figure 84, this project reference creates an error because a Class Library cannot reference a Windows Forms project. Additionally, the Windows Forms project will need to reference the Class Library to use the business and data access layers. This will create a circular reference, which is not allowed. To fix this, we can move the Models folder from the PizzaMaker project to the PizzaMakerClassLibrary project. The updates solution explorer should reflect Figure 85. You may need to delete the Models folder from the PizzaMaker project.

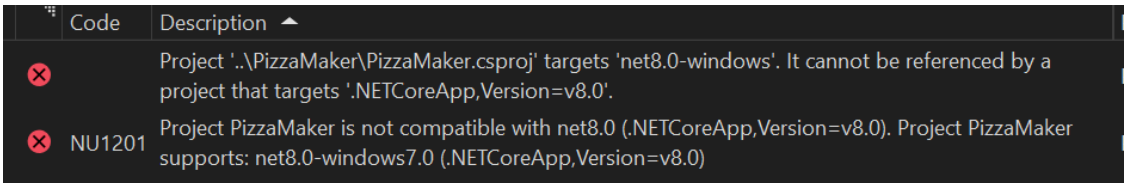


Figure 84. Errors created by Class Library referencing a Windows Forms application.

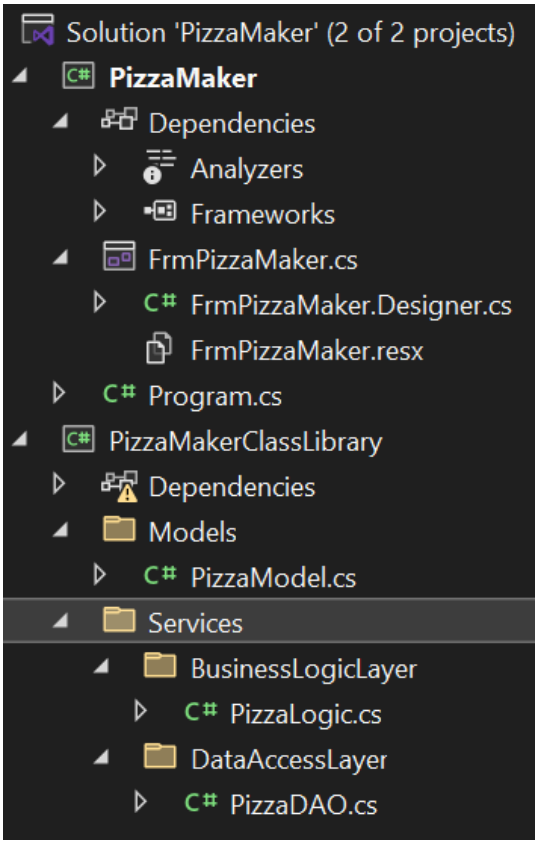


Figure 85. Updated solution explorer after moving Models folder WoWor.

9. Next, remove the dependency to PizzaMaker by expanding the Dependencies tab until you reach the PizzaMaker project. Right click on the project and select "Remove" as shown in Figure 86.

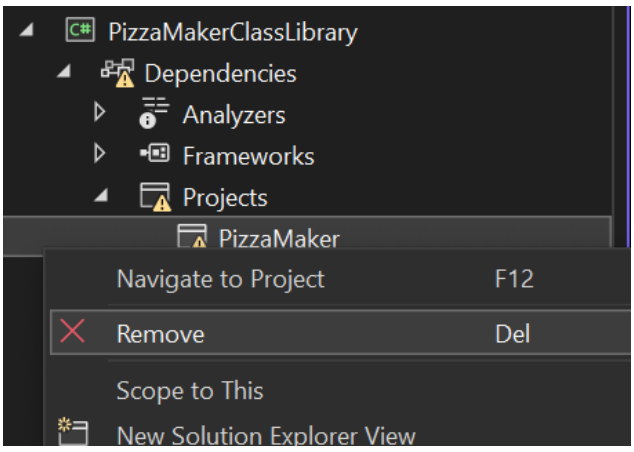


Figure 86. Removing the project reference to PizzaMaker.

10. In the PizzaModel file, update the class to be public instead of internal. You may also have to add a using statement for System.Drawing for the color type for the pizza box color as shown in Figure 87. Lastly, update the namespace to be PizzaMakerClassLibrary.Models.

```
using System.Drawing;

namespace PizzaMakerClassLibrary.Models
{
    1 reference
    public class PizzaModel
    {
        // Class properties
        1 reference
        public string ClientName { get; set; }
        1 reference
```

Figure 87. Using statement for System.Drawing.

11. In PizzaDAO, add a using statement for PizzaMakerClassLibrary.Models as shown in Figure 88. This should resolve all outstanding errors.

```
using PizzaMakerClassLibrary.Models;

namespace PizzaMakerClassLibrary.Services.DataAccessLayer
{
    2 references
    public class PizzaDAO
    {
        // Class level variables
        private List<PizzaModel> _pizzaOrder;

        /// <summary>
        /// Default constructor for the pizza DAO
        /// </summary>
        0 references
        public PizzaDAO()
        {
            // Initialize the _pizzaOrder list
            _pizzaOrder = new List<PizzaModel>();
        }
```

Figure 88. Adding a using statement for PizzaMakerClassLibrary.Models.

12. Add a public method to the DAO named AddPizzaToOrder that will return the number of pizzas in the current order as an int as shown in Figure 89. The method will accept a PizzaModel object as the new pizza. Remember to add a summary comment to the method.

```
    _pizzaOrder = new List<PizzaModel>();
}

/// <summary>
/// Add a pizza to the current order
/// </summary>
/// <param name="newPizza"></param>
/// <returns></returns>
0 references
public int AddPizzaToOrder(PizzaModel newPizza)
{

}
```

Figure 89. Adding the AddPizzaToOrder method in the DAO.

13. The method will add the new pizza to the pizza order variable we created earlier. Then, return the number of pizzas in the list as shown in Figure 90.

```
/// <summary>
/// Add a pizza to the current order
/// </summary>
/// <param name="newPizza"></param>
/// <returns></returns>
0 references
public int AddPizzaToOrder(PizzaModel newPizza)
{
    // Add the new pizza to the pizzaOrder list
    _pizzaOrder.Add(newPizza);
    // Return the number of pizzas in pizzaOrder
    return _pizzaOrder.Count;
}
```

Figure 90. Logic to add a new pizza in the DAO.

- Commit all changes in the project to source control.
- Commit Message: Add logic to the DAO to add a new pizza.

14. Open PizzaLogic.cs and add a citation to the top. Additionally, change the access specifier for the class to public as shown in Figure 91.

```
/*
 * Gianna Ross
 * CST - 250
 * 01/01/2025
 * Pizza Maker
 * Activity 4
 */

namespace PizzaMakerClassLibrary.Services.BusinessLogicLayer
{
    0 references
    public class PizzaLogic
    {

    }
}
```

Figure 91. Setting up the pizza logic class.

15. Declare a private class level PizzaDAO variable in the PizzaLogic class named _pizzaDAO as shown in Figure 92. This will be used to access the DAO methods we create. To initialize the variable, add a default constructor to the PizzaLogic class.

```
1 reference
public class PizzaLogic
{
    // Declare class level variables
    private PizzaDAO _pizzaDAO;

    /// <summary>
    /// Default constructor for PizzaLogic
    /// </summary>
    0 references
    public PizzaLogic()
    {
        // Initialize the pizza DAO object
        _pizzaDAO = new PizzaDAO();
    }
}
```

Figure 92. Adding a DAO variable to PizzaLogic.

16. Add a new method to PizzaLogic named AddPizzaToOrder. Like the DAO method, it will accept a PizzaModel as the parameter. However, this method will return two values, a boolean representing if the pizza was valid or not and the integer showing how many pizzas are in the order as shown in Figure 93. Remember to add a summary comment to the method.

```
/// <summary>
/// Add a new pizza to the current order
/// </summary>
/// <param name="newPizza"></param>
/// <returns></returns>
0 references
public (bool isValidPizza, int pizzasInOrder) AddPizzaToOrder(PizzaModel newPizza)
{

}
```

Figure 93. Setting up the AddPizzaToOrder method in the business logic layer.

17. Currently, this method will always return true for isValidPizza. Later, you will add in data integrity checking to make sure each pizza that is added to the pizza order is valid. Declare and initialize a variable to store the number of pizzas in the order. Then, call the AddPizzaToOrder method from the DAO and save the result in the variable we just created. Finally, return the variable and true for isValidPizza as shown in Figure 94.

```
/// <summary>
/// Add a new pizza to the current order
/// </summary>
/// <param name="newPizza"></param>
/// <returns></returns>
0 references
public (bool isValidPizza, int pizzasInOrder) AddPizzaToOrder(PizzaModel newPizza)
{
    // Declare and initialize
    int pizzas = -1;

    // Call the DAO AddPizzaToOrder
    pizzas = _pizzaDAO.AddPizzaToOrder(newPizza);
    // Return the pizzas variable
    return (true, pizzas);
}
```

Figure 94. Logic to add a pizza to the order from the business logic layer.

- Commit all changes in the project to source control.
- Commit Message: Set up the PizzaLogic class and add logic to add a pizza to the order.

18. Back in the PizzaMaker project, add a project reference to the PizzaMakerClassLibrary project as shown in Figure 95.

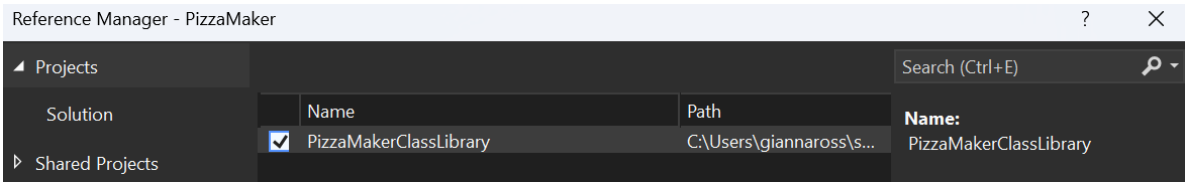


Figure 95. Adding a project reference to PizzaMaker.

19. Next, create a Click event handler for the "Create Pizza" button named BtnCreatePizzaClickEH. Add a summary comment to the event as shown in Figure 96.

```
/// <summary>
/// Click event handler for btnCreatePizza
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnCreatePizzaClickEH(object sender, EventArgs e)
{

}
```

Figure 96. BtnCreatePizzaClickEH with a summary comment.

20. Add a new private PizzaLogic class level variable named _pizzaLogic to FrmPizzaMaker as shown in Figure 97. Additionally, make sure the using statement for Models uses PizzaMakerClassLibrary. Initialize the _pizzaLogic variable in the form constructor.

```
3 references
public partial class FrmPizzaMaker : Form
{
    // Class level variable declarations
    private PizzaModel _pizza;
    private PizzaLogic _pizzaLogic;

    /// <summary>
    /// Default constructor for FrmPizzaMaker
    /// </summary>
    1 reference
    public FrmPizzaMaker()
    {
        InitializeComponent();
        // Initialize the current order
        _pizza = new PizzaModel();
        // Initialize the business logic layer
        _pizzaLogic = new PizzaLogic();

        // Disable the Create Pizza button
        btnCreatePizza.Enabled = false;
        // Disable the Reset Form button
        btnResetForm.Enabled = false;
        // Update the price of the pizza
        UpdatePrice();

        // Update the maximums for hsbSauce and hsbCheese
        hsbSauce.Maximum = 100 + hsbSauce.LargeChange - 1;
        hsbCheese.Maximum = 100 + hsbCheese.LargeChange - 1;
    }
```

Figure 97. _pizzaLogic variable for FrmPizzaMaker.

21. Declare two variables in BtnCreatePizzaClickEH to contain the results of the AddPizzaToOrder call from the business logic layer. Then, call the method and store the results in the variables we just created as shown in Figure 98. Lastly, if the pizza was valid, reset the form by calling the ResetForm method we created earlier.

```
/// <summary>
/// Click event handler for btnCreatePizza
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnCreatePizzaClickEH(object sender, EventArgs e)
{
    // Declare and initialize
    bool isValidPizza = false;
    int pizzasInOrder = -1;

    // Use the pizzaLogic to call AddPizzaToOrder
    (isValidPizza, pizzasInOrder) = _pizzaLogic.AddPizzaToOrder(_pizza);

    // Check if the pizza was valid
    if (isValidPizza)
    {
        // Reset the form
        ResetForm();
    }
}
```

Figure 98. Logic for BtnCreatePizzaClickEH to add the pizza to the current order.

- Commit all changes in the project to source control.
- Commit Message: Finish the logic to add a new pizza to the order.

22. We will use the isValidPizza variable to enable a button. This button, when clicked, will take us to another form that will show the user their full pizza order. First, add another button named btnSeeFullOrder to FrmPizzaMaker as shown in Figure 99. Change the text of the button to "See Full Order."

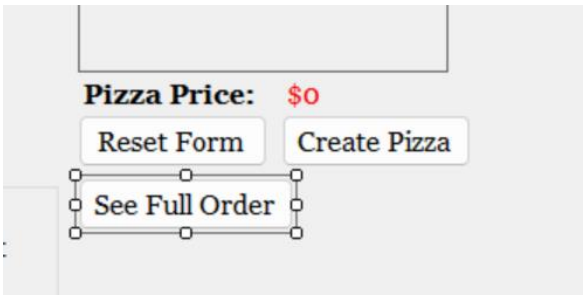


Figure 99. Adding a See Full Order button to FrmPizzaMaker.

23. In the form constructor, disable the button as shown in Figure 100.

```
/// <summary>
/// Default constructor for FrmPizzaMaker
/// </summary>
1 reference
public FrmPizzaMaker()
{
    InitializeComponent();
    // Initialize the current order
    _pizza = new PizzaModel();
    // Initialize the business logic layer
    _pizzaLogic = new PizzaLogic();

    // Disable the Create Pizza button
    btnCreatePizza.Enabled = false;
    // Disable the Reset Form button
    btnResetForm.Enabled = false;
    // Disable the See Full Order button
    btnSeeFullOrder.Enabled = false;
    // Update the price of the pizza
    UpdatePrice();

    // Update the maximums for hsbSauce and hsbCheese
    hsbSauce.Maximum = 100 + hsbSauce.LargeChange - 1;
    hsbCheese.Maximum = 100 + hsbCheese.LargeChange - 1;
}
```

Figure 100. Disabling btnSeeFullOrder in the FrmPizzaMaker constructor.

24. In BtnCreatePizzaClickEH, check if the pizza returned valid. If it did, enable btnSeeFullOrder and reset the form as shown in Figure 101. If the pizza was not valid, display a message box to the user.

```
/// <summary>
/// Click event handler for btnCreatePizza
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnCreatePizzaClickEH(object sender, EventArgs e)
{
    // Declare and initialize
    bool isValidPizza = false;
    int pizzasInOrder = -1;

    // Use the pizzaLogic to call AddPizzaToOrder
    (isValidPizza, pizzasInOrder) = _pizzaLogic.AddPizzaToOrder(_pizza);

    // Check if the pizza was valid
    if (isValidPizza)
    {
        // Enable the See Full Order button
        btnSeeFullOrder.Enabled = true;
        // Reset the form
        ResetForm();
    }
    else
    {
        // Show a failure message to the user
        MessageBox.Show("Your pizza order is not complete.");
    }
}
```

Figure 101. Enable btnSeeFullOrder if the pizza is valid.

25. Add a second form to the PizzaMaker project. Right click the project and choose Add > Form (Windows Forms)… as shown in Figure 102.

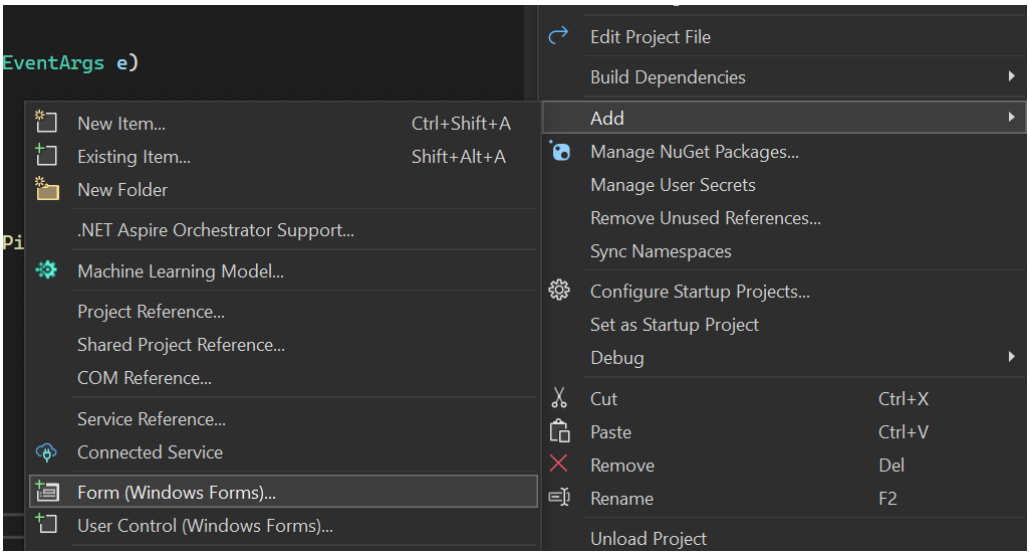


Figure 102. Adding a new form to the PizzaMaker project.

26. Name the new form FrmOrderDetails as shown in Figure 103.

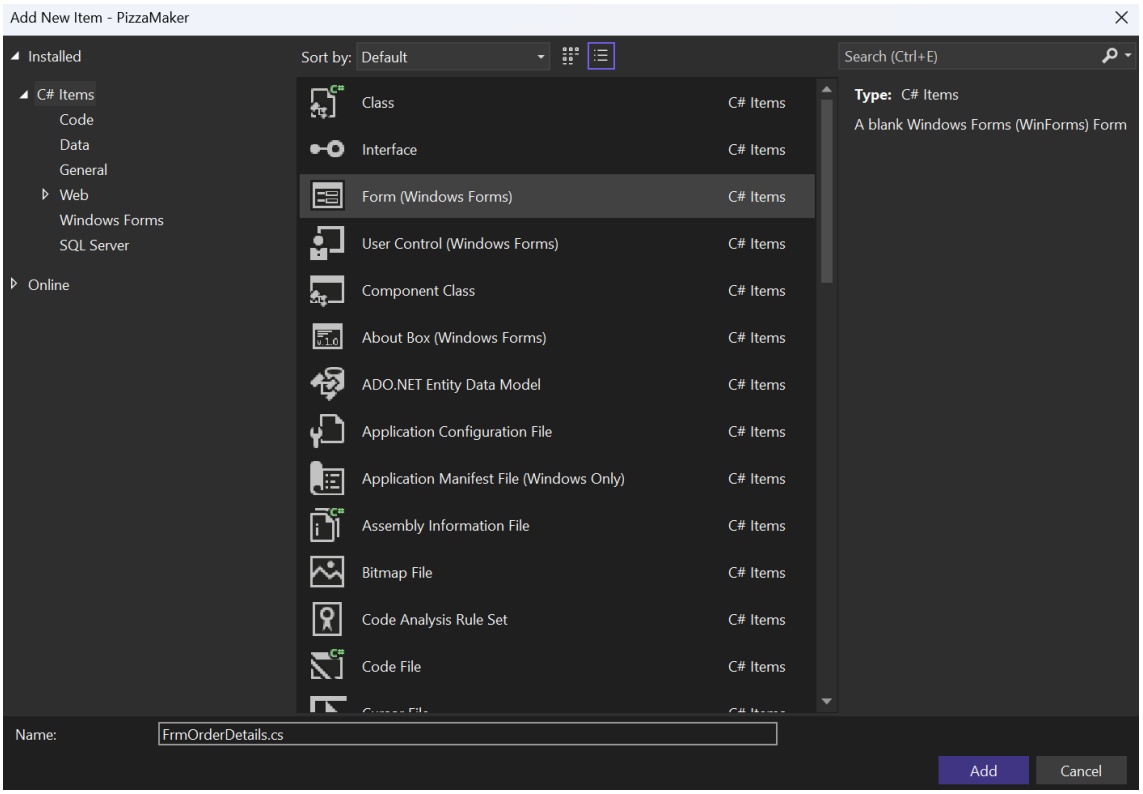


Figure 103. Naming the new form.

27. Change the text on the form to say "Pizza Order Details" as shown in Figure 104.

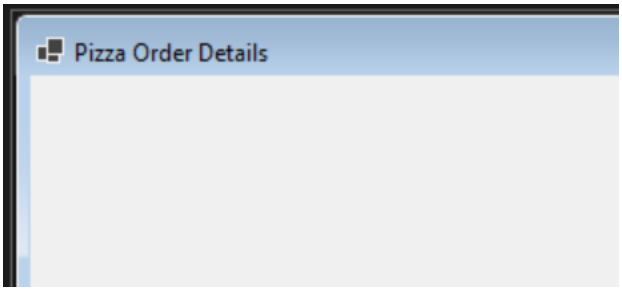


Figure 104. Updating the text for FrmOrderDetails.

28. Add a new label to the form named lblOrderDetails as shown in Figure 105. Change the default text for the label to "Order Details."

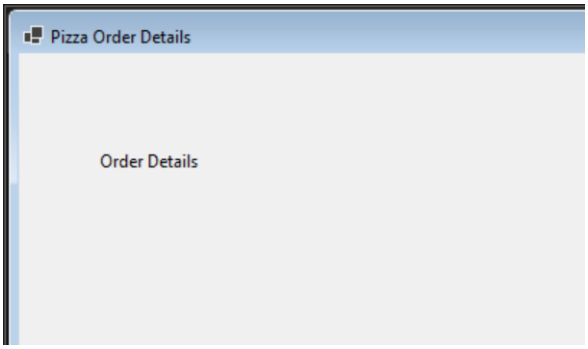


Figure 105. FrmOrderDetails with lblOrderDetails.

- Commit all changes in the project to source control.
- Commit Message: Add and complete the UI for a second form to show the current pizza order.

29. Open the code behind FrmOrderDetails and add a citation to the top of the class. Create two private class level variables: a List<PizzaModel> _pizzaOrders and a PizzaLogic _pizzaLogic as shown in Figure 106.

```
/*
 * Gianna Ross
 * CST - 250
 * 01/01/2025
 * Pizza Maker
 * Activity 4
 */

using PizzaMakerClassLibrary.Models;
using PizzaMakerClassLibrary.Services.BusinessLogicLayer;

namespace PizzaMaker
{
    4 references
    public partial class FrmOrderDetails : Form
    {
        // Declare class level variables
        private List<PizzaModel> _pizzaOrder;
        private PizzaLogic _pizzaLogic;

        1 reference
        public FrmOrderDetails()
        {
            InitializeComponent();
        }
    }
}
```

Figure 106. Adding a property to FrmOrderDetails.

30. Next, add a parameterized constructor for FrmPizzaConfirm to receive a list of pizzas. So that the form is setup correctly, we need to call the InitializeComponent method as shown in Figure 107. We should also take a PizzaLogic variable as a parameter, as this will allow our pizza order list to be persistent.

```
/// <summary>
/// Parameterized constructor for FrmOrderDetails
/// </summary>
/// <param name="pizzaOrder"></param>
1 reference
public FrmOrderDetails(List<PizzaModel> pizzaOrderList, PizzaLogic pizzaBusinessLogic)
{
    // Initialize the form
    InitializeComponent();
    // Initialize the class level variables
    _pizzaOrder = pizzaOrderList;
    _pizzaLogic = pizzaBusinessLogic;
}
```

Figure 107. Parameterized constructor for FrmOrderDetails.

31. Back in FrmPizzaMaker, create a Click event handler for btnSeeFullOrder named BtnSeeFullOrderClickEH. Add a summary comment as shown in Figure 108.

```
/// <summary>
/// Click event handler for btnSeeFullOrder
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnSeeFullOrderClickEH(object sender, EventArgs e)
{

}
```

Figure 108. BtnSeeFullOrderClickEH with a summary comment.

32. This method will need to pass the list of pizzas in the current order to FrmOrderDetails. To do that, we will need to create DAO and business logic methods to get the list of pizzas. In PizzaDAO, create a new public method named GetPizzaOrder that returns a list of pizza models as shown in Figure 109. Remember to add a summary comment.

```
    return pizzaOrder.Count;
}

/// <summary>
/// Get the list of pizzas in the current order
/// </summary>
/// <returns></returns>
0 references
public List<PizzaModel> GetPizzaOrder()
{

}
```

Figure 109. DAO method to get the full list of pizza orders.

33. In GetPizzaOrder, return the pizzaOrder list as shown in Figure 110.

```
/// <summary>
/// Get the list of pizzas in the current order
/// </summary>
/// <returns></returns>
0 references
public List<PizzaModel> GetPizzaOrder()
{
    // Return the pizzaOrder list
    return _pizzaOrder;
}
```

Figure 110. Finishing the DAO GetPizzaOrder method.

34. In PizzaLogic.cs, create a mirror method with a summary comment that calls and returns the DAO method as shown in Figure 111.

```
/// <summary>
/// Get the list of pizzas in the current order
/// </summary>
/// <returns></returns>
0 references
public List<PizzaModel> GetPizzaOrder()
{
    // Get and return GetPizzaOrder from the DAO
    return _pizzaDAO.GetPizzaOrder();
}
```

Figure 111. GetPizzaOrder method in PizzaLogic.

35. Back in FrmPizzaMaker, declare and initialize a variable to hold the pizza list in BtnSeeFullOrderClickEH. Call the pizzaLogic method to get the current list of pizzas. Then, create a new FrmOrderDetails object using the pizza list as shown in Figure 112.

```
/// <summary>
/// Click event handler for btnSeeFullOrder
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnSeeFullOrderClickEH(object sender, EventArgs e)
{
    // Declare and initialize
    List<PizzaModel> pizzaList;
    // Get the pizza list from pizzaLogic
    pizzaList = _pizzaLogic.GetPizzaOrder();
    // Create a new form with the pizza list
    FrmOrderDetails frmOrderDetails = new FrmOrderDetails(pizzaList, _pizzaLogic);
}
```

Figure 112. Finishing BtnSeeFullOrderClickEH method.

36. Create a new public method in FrmOrderDetails named DisplayPizzas with a summary comment as shown in Figure 113.

```
/// <summary>
/// Display the pizzas on the form
/// </summary>
0 references
public void DisplayPizzas()
{

}
```

Figure 113. Starting the DisplayPizzas method.

37. DisplayPizzas will loop through the pizza order list and add each one to the order details label. First, clear lblOrderDetails. Then, set up a foreach loop to get each PizzaModel in PizzaOrder. For each PizzaModel, format the data and add it to the label as shown in Figure 114. By using string.Join, we can list the ingredients and strange add ons out, separated by commas.

```
/// <summary>
/// Display the pizzas on the form
/// </summary>
1 reference
public void DisplayPizzas()
{
    // Clear the label
    lblOrderDetails.Text = "";

    // Loop through the pizza order list
    foreach (PizzaModel pizza in _pizzaOrder)
    {
        lblOrderDetails.Text +=
            $"Name: {pizza.ClientName}\n" +
            $"Ingredients: {string.Join(", ", pizza.Ingredients)}\n" +
            $"Strange Add Ons: {string.Join(", ", pizza.StrangeAddOns)}\n" +
            $"Crust: {pizza.Crust}\n" +
            $"Sauce: {pizza.SauceQty}\n" +
            $"Cheese: {pizza.CheeseQty}\n" +
            $"Delivery Time: {pizza.DeliveryTime}\n" +
            $"Pizza Box Color: {pizza.PizzaBoxColor}\n" +
            $"Price: {pizza.Price}\n\n";
    }
}
```

Figure 114. Adding each pizza to lblOrderDetails.

38. In BtnSeeFullOrderClickEH, call the DisplayPizzas method on the created form. Then, use the ShowDialog method to display the form as shown in Figure 115. By using ShowDialog instead of Show, the user is unable to click on the parent form, FrmPizzaMaker, while FrmOrderDetails is open.

```
/// <summary>
/// Click event handler for btnSeeFullOrder
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnSeeFullOrderClickEH(object sender, EventArgs e)
{
    // Declare and initialize
    List<PizzaModel> pizzaList;
    // Get the pizza list from pizzaLogic
    pizzaList = _pizzaLogic.GetPizzaOrder();
    // Create a new form with the pizza list
    FrmOrderDetails frmOrderDetails = new FrmOrderDetails(pizzaList, _pizzaLogic);
    // Update the label with the pizza order
    frmOrderDetails.DisplayPizzas();
    // Show the form
    frmOrderDetails.ShowDialog();
}
```

Figure 115. Finishing BtnSeeFullOrderClickEH.

- Commit all changes in the project to source control.
- Commit Message: Finish functionality to send pizza order to a second form.

39. Next, we will add logic to save a pizza order to a text file. In PizzaDAO, create a new public method named WriteOrderToFile that will return a boolean value as shown in Figure 116. Make sure to add a summary comment.

```
    return pizzaOrder;
}

/// <summary>
/// Write the pizza order to a text file
/// </summary>
/// <returns></returns>
0 references
public bool WriteOrderToFile()
{

}
```

Figure 116. Starting the WriteOrderToFile method.

40. First, declare and initialize a file path variable by using the Path.Combine method, which will combine the given strings to create a new file path. We can get our current directory using AppDomain.CurrentDomain.BaseDirectory and combine it with an "App_Data" folder so that our data is separate from the other files. We should also declare a pizzaString variable to hold each formatted pizza before it gets written to the file. Check if the "App_Data" directory exists. If it does not, create it as shown in Figure 117.

```
/// <summary>
/// Write the pizza order to a text file
/// </summary>
/// <returns></returns>
0 references
public bool WriteOrderToFile()
{
    // Declare and initialize
    string filePath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "App_Data");
    string pizzaString = "";

    // Check if the directory exists
    if (!Directory.Exists(filePath))
    {
        // Create the directory
        Directory.CreateDirectory(filePath);
    }
}
```

Figure 117. WriteOrderToFile method with directory check.

41. Set up a try-catch statement to surround the writing for the file. If the write statement fails, return false. Create a using statement for a stream writer object using the file path variable we created earlier combined with "PizzaOrder.txt" as our text file. Loop through the pizzaOrder list and format and write each pizza to the text file as shown in Figure 118. Finally, return true.

```
    Directory.CreateDirectory(filePath);
}

// Set up a try-catch for the file writer
try
{
    // Create a using statement for StreamWriter
    using (StreamWriter streamWriter = new StreamWriter(Path.Combine(filePath, "PizzaOrder.txt")))
    {
        // Loop through the pizza order list
        foreach (PizzaModel pizza in _pizzaOrder)
        {
            pizzaString =
                $"Name: {pizza.ClientName}\n" +
                $"Ingredients: {string.Join(", ", pizza.Ingredients)}\n" +
                $"Strange Add Ons: {string.Join(", ", pizza.StrangeAddOns)}\n" +
                $"Crust: {pizza.Crust}\n" +
                $"Sauce: {pizza.SauceQty}\n" +
                $"Cheese: {pizza.CheeseQty}\n" +
                $"Delivery Time: {pizza.DeliveryTime}\n" +
                $"Pizza Box Color: {pizza.PizzaBoxColor}\n" +
                $"Price: {pizza.Price}\n\n";
            streamWriter.WriteLine(pizzaString);
        }
    }
    // Return true
    return true;
}
catch
{
    // Return false
    return false;
}
```

Figure 118. Finishing the WriteOrderToFile method.

42. In PizzaLogic.cs, create a method to mirror the DAO method we just created as shown in Figure 119. The method will call WriteOrderToFile from the DAO.

```
/// <summary>
/// Write the pizza order to a text file
/// </summary>
/// <returns></returns>
0 references
public bool WriteOrderToFile()
{
    // Get and return WriteOrderToFile from the DAO
    return _pizzaDAO.WriteOrderToFile();
}
```

Figure 119. WriteOrderToFile method in pizzaLogic.

43. Back in the design for FrmOrderDetails, add a new button named btnSaveOrder and change the text to "Save Order" as shown in Figure 120.

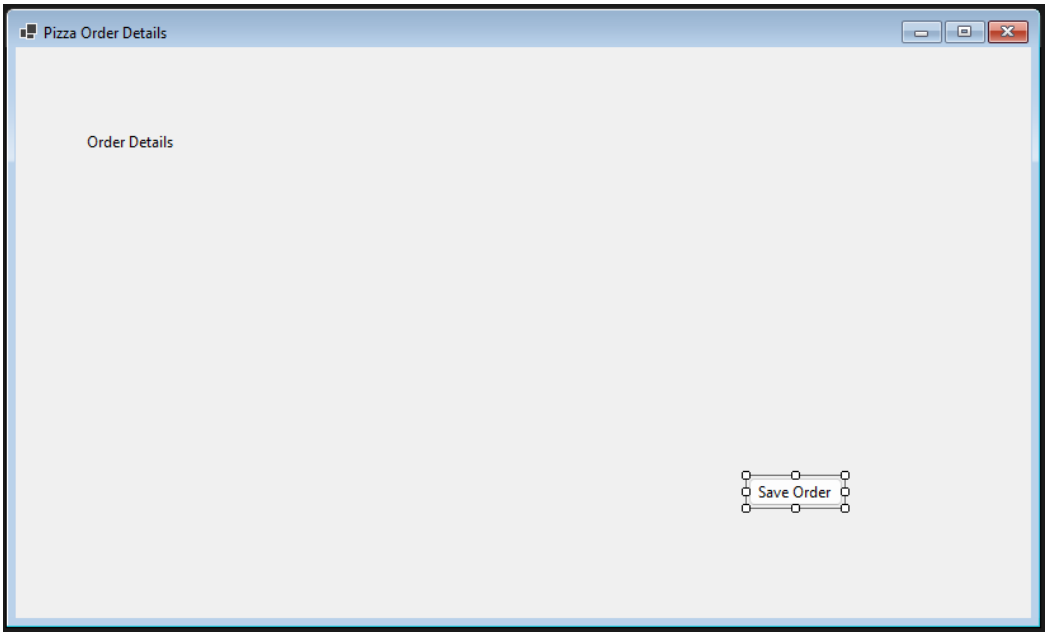


Figure 120. Adding the Save Order button.

44. Add a Click event handler to the button named BtnSaveOrderClickEH with a summary comment as shown in Figure 121.

```
/// <summary>
/// Click event handler for btnSaveOrder
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnSaveOrderClickEH(object sender, EventArgs e)
{

}
```

Figure 121. BtnSaveOrderClickEH with a summary comment.

45. First, declare and initialize a variable to store the return from the WriteOrderToFile method. Then, call the method from pizzaLogic and save the result in isSaveSuccess. Based on that result, show the user a message using MessageBox.Show as shown in Figure 122.

```
/// <summary>
/// Click event handler for btnSaveOrder
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnSaveOrderClickEH(object sender, EventArgs e)
{
    // Declare and initialize
    bool isSaveSuccess = false;

    // Write the order to the file
    isSaveSuccess = _pizzaLogic.WriteOrderToFile();

    // Check if the save was successful
    if (isSaveSuccess)
    {
        // Show a success message to the user
        MessageBox.Show("The pizza order was saved.");
    }
    else
    {
        // Show a failure message to the user
        MessageBox.Show("An error occurred while trying to save your order. Please try again later.");
    }
}
```

Figure 122. Saving the pizza order to a text file.

- Commit all changes in the project to source control.
- Commit Message: Add logic to send the pizzas to a text file.

- Take a screenshot of the FrmOrderDetails form running at this point with two orders.
- Take a screenshot of the success message box.
- Take a screenshot of PizzaOrder.txt with the saved orders.
- Take a screenshot of the default constructor for FrmOrderDetails.
- Take a screenshot of the DisplayPizzas and BtnSaveOrderClickEH methods in FrmOrderDetails.
- Take a screenshot of the BtnCreatePizzaClickEH, and BtnSeeFullOrderClickEH methods in FrmPizzaMaker.
- Take a screenshot of the AddPizzaToOrder, GetPizzaOrder, and WriteOrderToFile methods in the PizzaLogic and PizzaDAO classes.
- Take a screenshot of the updated UMLs for the FrmOrderDetails, FrmPizzaMaker, PizzaModel, PizzaLogic, and PizzaDAO classes.
- Paste the images into a Word document.
- Put a caption below each image explaining what is being demonstrated.

Pizza Maker Challenges

Add the following features to enhance your program.

1. **Pizza Validation**

   Update the AddPizzaToOrder method in PizzaLogic to make sure pizzas passed in are valid. A valid pizza has a name, a crust selection, at least one ingredient, and values greater than 0 for cheese and sauce.

2. **Back Button**

   Add a "Back" button to FrmOrderDetails so that the user can access FrmPizzaMaker to add more pizzas to his/her order.

- Commit the challenges to source control with a commit message.

- Take a screenshot of the application running after the new feature have been added.
- Take a screenshot of each method added or updated to implement the feature.
- Take a screenshot of the updated UMLs for any classes with added methods.
- Paste the images into a Word document.
- Put a caption below each image explaining what is being demonstrated.

What You Learned in this Activity

Throughout Activity 4, you have acquired skills and knowledge that can be applied not only to building pizza ordering systems but also to any form-based application development. Here are some of the key takeaways from this lesson.

1. **Understanding of Form Controls**: You have learned how to install and configure a wide range of form controls, understanding their properties and functions. This includes controls like text boxes, checkboxes, radio buttons, scroll bars, and more.
2. **Data Handling and Class Creation**: By creating a class like `PizzaModel`, you have practiced mirroring UI elements with backend data structures, enabling you to manage complex data effortlessly.
3. **Dynamic UI Management**: You have developed skills in dynamically managing UI elements based on user interactions, such as enabling or disabling controls or updating their values.
4. **Data Collection and Display**: You have implemented techniques to collect data from various form controls, store them in a structured format like an list, and then display this collection on another form.
5. **Application of Learned Concepts to Various Domains**: You have seen how the techniques learned can be adapted to create a wide array of applications, from event planners to custom gift baskets, demonstrating the flexibility of your new skills.
6. **Integration of User Feedback**: By coding the "Create Pizza" button to generate pizza orders based on user selections and then displaying these on a second form, you have learned how to integrate user feedback into application flow.
7. **Practical Application of Theoretical Knowledge**: Finally, you have put theoretical knowledge into practice by designing a fully functional application that consolidates multiple UI controls and backend data handling in a coherent and user-friendly interface.

These skills form a solid foundation for further development in C# and .NET, equipping you with the tools needed to create advanced and user-centric applications. Whether for academic, personal, or commercial use, the knowledge gained here will serve as a stepping stone to more complex projects and applications.

Check for Understanding

These quiz questions will help you review important concepts from the lesson. These are not graded. You do not have to submit your score.

1. What does the `TextBox` control in the Pizza Maker app typically store?

a. Boolean values

b. Integer values

c. String values

d. None of the above

2. Which control is best suited for selecting a date in the Pizza Maker app?

a. Checkbox

b. Radio Button

c. Date Time Picker

d. List Box

3. How are multiple selection options presented in the Pizza Maker app?

a. Checkbox

b. Text Edit

c. Group Box

d. H-Scroll Bar

4. Which control would be suitable for adjusting quantities like sauce and cheese in the Pizza Maker app?

a. Radio Button

b. Scroll Bar

c. Date Time Picker

d. Text Edit

5. What does the "Create Pizza" button do in the Pizza Maker app?
    a. It closes the application.
    b. It resets the form.
    c. It sorts the pizza order.
    d. It saves the current instance of PizzaModel.

Answer Key

1. What does the "TextBox" control in the Pizza Maker app typically store?

   c. String values

2. Which control is best suited for selecting a date in the Pizza Maker app?

   c. Date Time Picker

3. How are multiple selection options presented in the Pizza Maker app?

   a. Checkbox

4. Which control would be suitable for adjusting quantities like sauce and cheese in the Pizza Maker app?

   b. Scroll Bar

5. What does the "Create Pizza" button do in the Pizza Maker app?

   d. It saves the current instance of PizzaModel.

Deliverables

You must submit two separate files:

- One PDF containing Parts 1–3.
- One ZIP file containing the complete codebase for Part 4.

Part 1: Activity Submission

1. Cover Page
    - Use APA formatting for the cover page.
2. GitHub Repository Link
    - At the top of the second page, provide a link to your private GitHub repository.
    - Be sure to add your instructor and grading assistant (if applicable) as collaborators.
3. Updated Flowchart
    - Include the latest flowchart reflecting your current application design.
4. Updated UML Diagram(s)
    - Provide UML diagrams that reflect the current implementation.
5. Base Application Screenshots
    - Include screenshots as required in the activity instructions.
    - Each screenshot must have a clear caption directly below the image explaining what is being demonstrated.
    - All code must be properly commented.
6. Section Headers
    - Use clear headers as outlined in these deliverables to identify each part of the assignment within the PDF.
7. Citations
    - Include a citation at the top of each page containing code, flowcharts, or UML diagrams.
8. File Naming
    - Name the PDF according to the activity title and include your full name.
9. Coding Standards Compliance

- Ensure GCU College of Engineering and Technology Coding Guidelines and Best Practices, found in Class Resources, have been followed to ensure consistency, readability, and professionalism in student code.

Part 2: Activity Challenges

1. Challenge Screenshots
    a. Include screenshots that demonstrate completion of the Challenge requirements as described in the Activity guide.
2. Minimum Features
    a. Implement and document all features from the Challenges section.

Part 3: Planning for Topic 5 Activity 5

1. Flowchart for Topic 5 Activity 5
    a. Use the requirements as outlined in Topic 5 Activity 5 to create a high-level flowchart. Showcase the logic and data flow that can be used as a template for the software design.
2. UML Diagram(s) for Topic 5 Activity 5
    a. Provide UML diagram(s) based on the same requirements.

Part 4: Code Submission (Separate ZIP File)

1. Compressed Project Folder
    a. Compress the full folder containing all source code and necessary files into a .zip file.
2. Executable Submission
    a. Ensure the application runs correctly. The ZIP file will be unzipped and tested during grading, so include all dependencies.
3. File Name
    a. Name the ZIP file according to the activity and include your full name.

Submission Reminder

Do not submit a single ZIP file that contains both the PDF and the code. Each file must be uploaded separately.