CST-150 Activity 5 Guide

Contents

Part 1

Writing Classes

Overview

Students are introduced to object-oriented programming and begin designing and writing classes to model real-world entities. Write a Windows Form Application that uses a button event handler to invoke methods in classes.

Execution

Execute this assignment according to the following guidelines:

1. Start a new Visual Studio Project.
    a. Start Visual Studio and select "Create a new project" as shown in Figure 1.

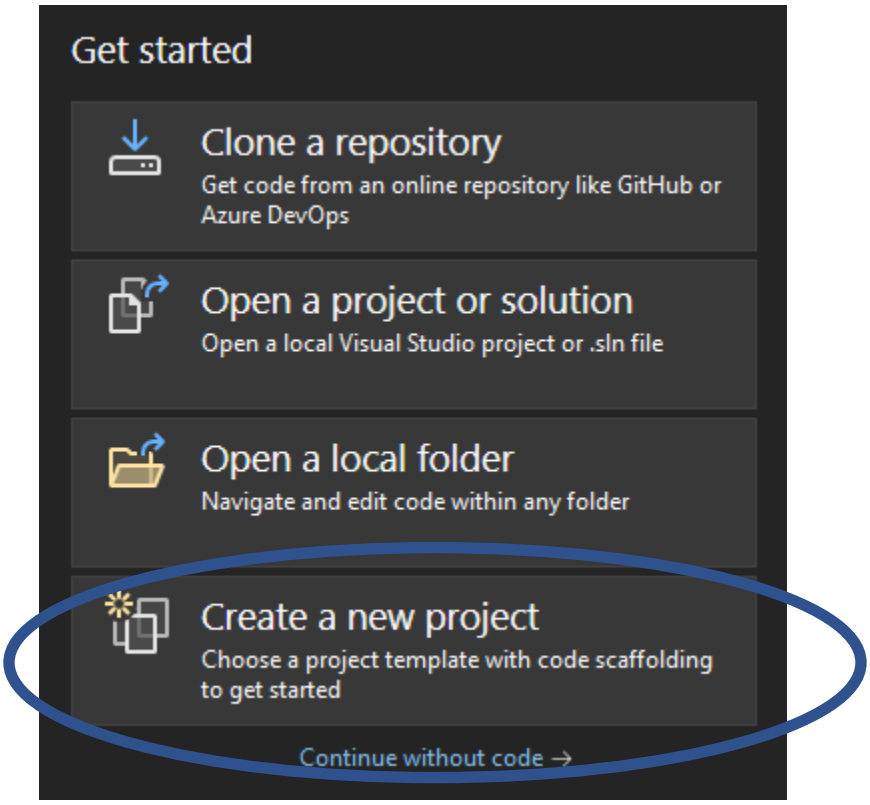

Figure 1: Select "Create a new project."

    b. Select the Project Template "WindowsFormsApp" as shown in Figure 2. Be sure to select "Windows Forms App." DO NOT select the one with (.NET Framework).

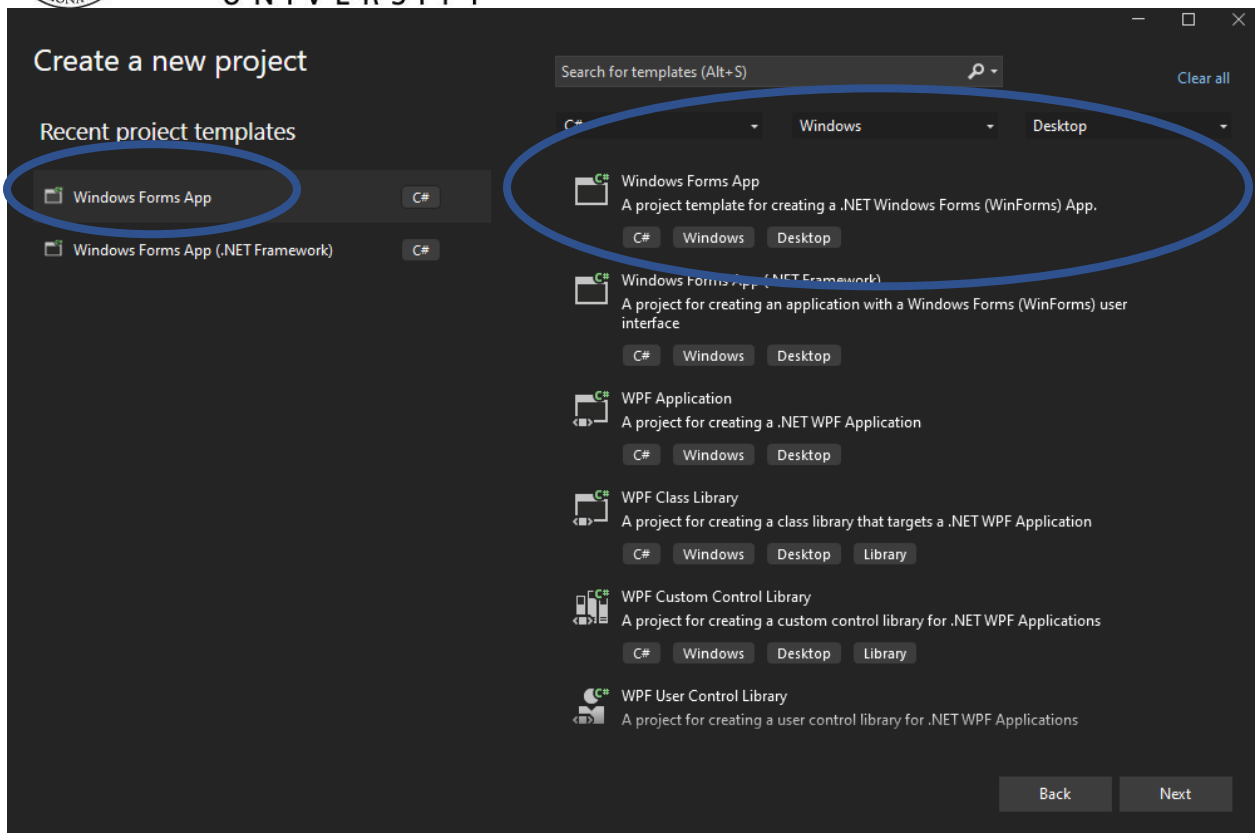

Figure 2: Select the Project Template.

- Enter the Project name: "CST-150 DogClass" as shown in Figure 3 followed by the "Next" button. Be sure the check box is not checked for "Place solution and project in the same directory."

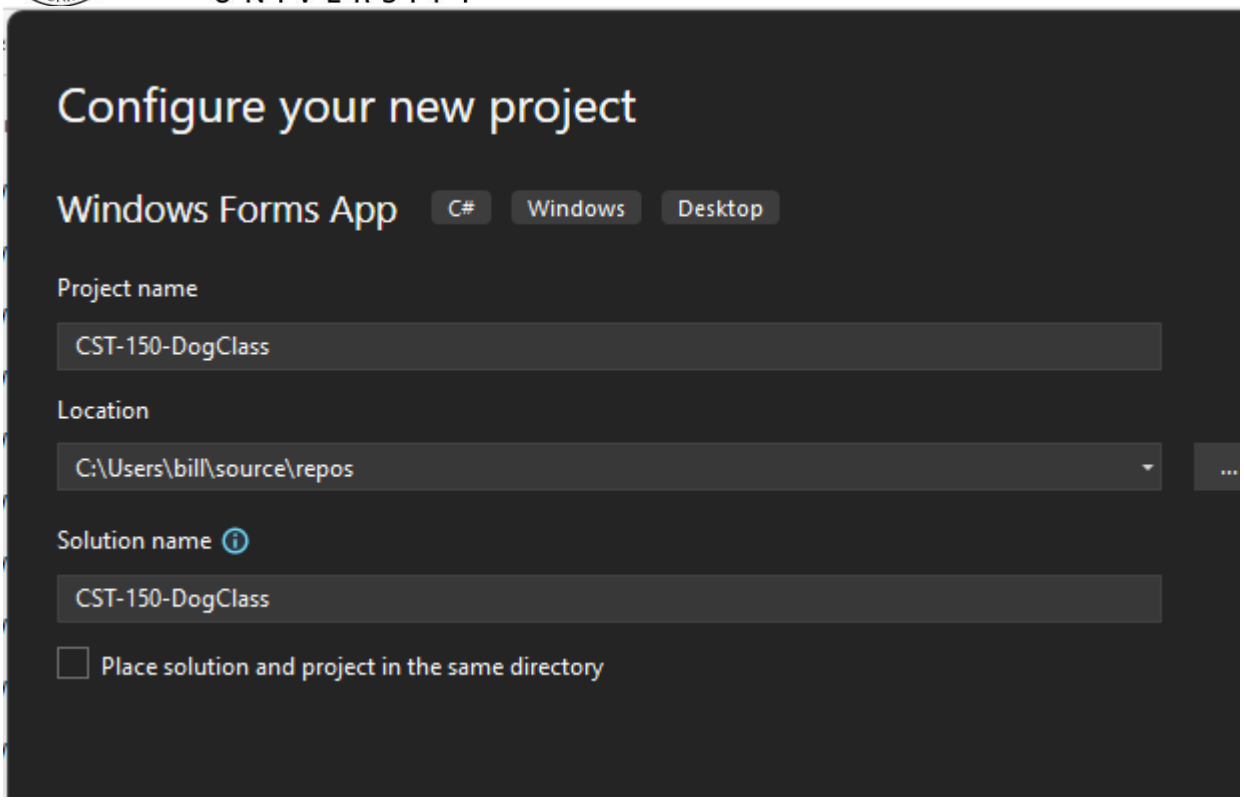

Figure 3: Enter Project Name.

d. Select the Framework as is shown in Figure 4 followed by the "Create" button. These in-class applications have all been tested using .NET 7.0. (The instructor may have the class us a different .NET version.)

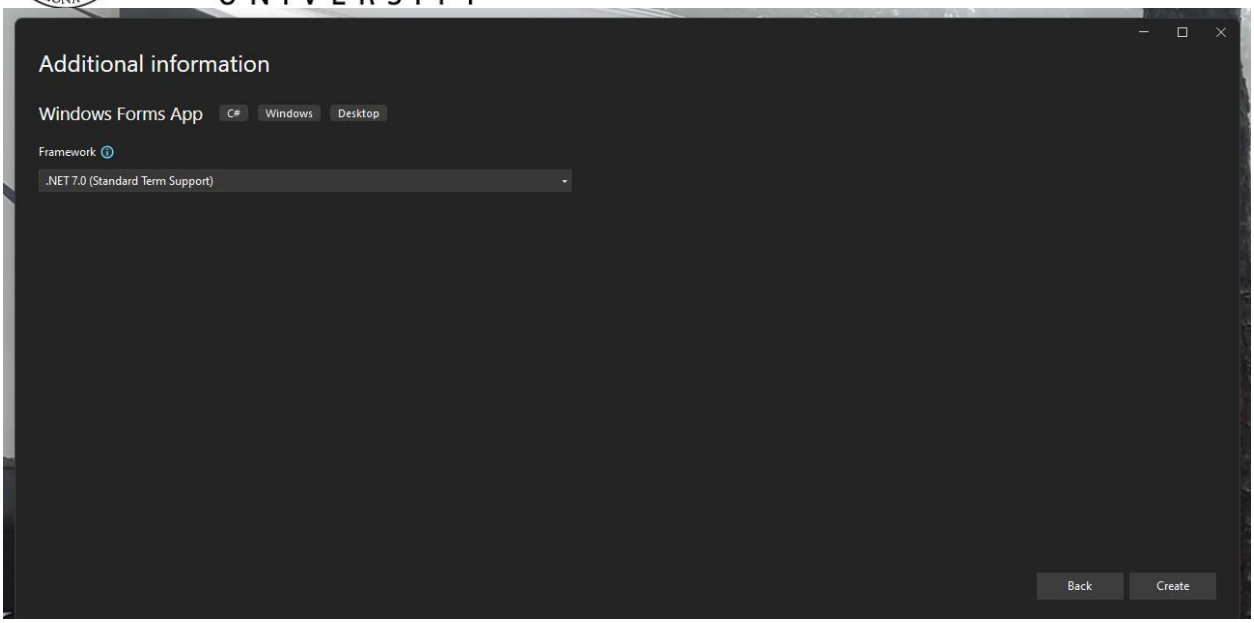

Figure 4: Select Framework.

2. Verify the application is working correctly.
    a. Follow the steps outlined in Activity 3.

3. Configure the Presentation Layer.
    a. Add a new folder "PresentationLayer" as shown in Figure 5.
    b. Remove Form1.cs as shown in Figure 5.
    c. Add a new form "FrmMain" inside the Presentation Layer as shown in Figure 5.

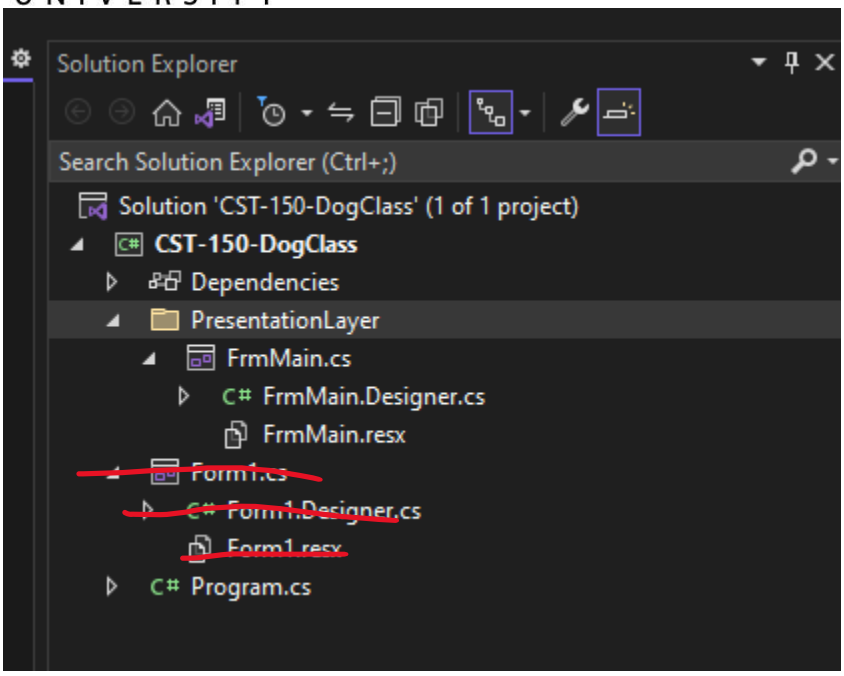

Figure 5: Add New Folder

d. Open Program.cs and update the Application.Run, so the compiler can find FrmMain in the Presentation Layeras as shown in Figure 6.

```
        /// </summary>
        [STAThread]
        0 references
        static void Main()
        {
            // To customize application configuration such as set high
            // see https://aka.ms/applicationconfiguration.
            ApplicationConfiguration.Initialize();
            Application.Run(new PresentationLayer.FrmMain());
        }
    }
}
```

Figure 6: Update Program.cs

4. Configure the Main Form in the Presentation Layer.

a. With FrmMain selected, in the properties for the form, set the font to Arial, 12pt, so this will be the default font for all controls placed on the form.

b. In FrmMain.cs, add a new button as shown in Figure 7.

c. Name: "btnAddDog"

d. Text: "Add New Dog"

e. Set: "autosize" = true

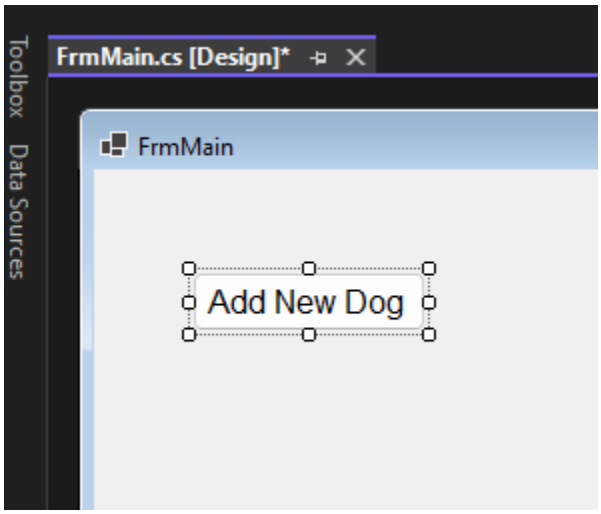

Figure 7: Add Button to Form.

f. In FrmMain, add 3 text boxes, 1 combobox, and 5 labels as shown in Figure 8. Following are the txtbox and cmbbox configurations.

g. Use the figure to position the controls and then, change the text for all labels to reflect the figure.

h. Dog Name text box: txtName

i. Neck Radius text box: txtNeck

j. Sitting name: cmbSit

k. Color name: txtColor

l. In the properties for the comboBox control, add items to the collection as is shown in Figure 9.

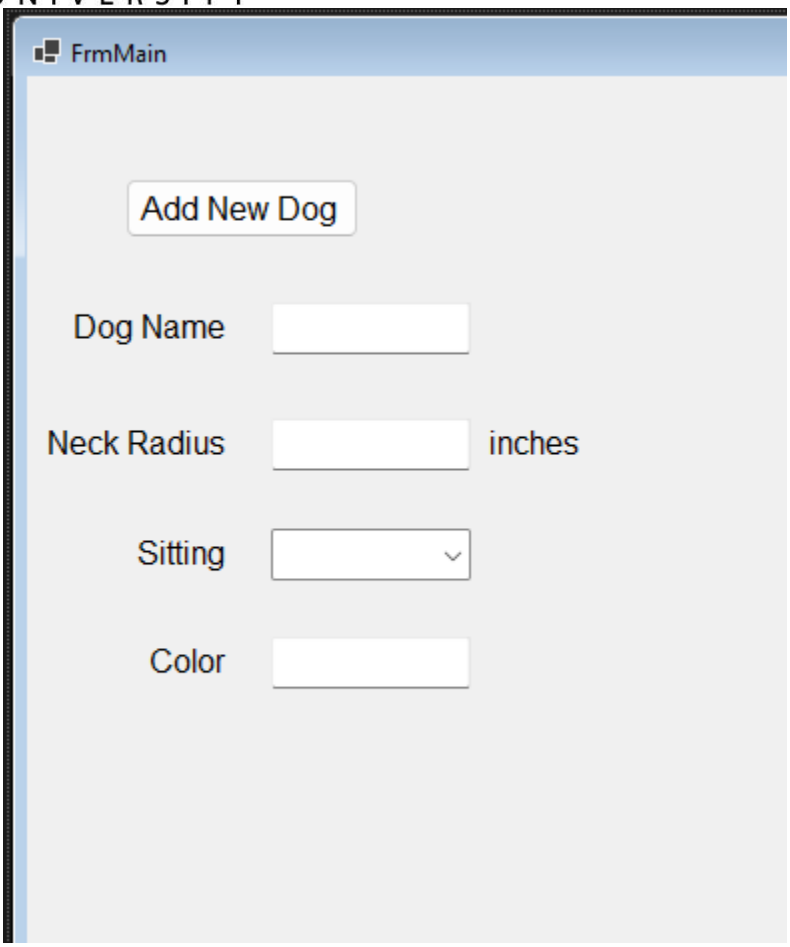

Figure 8: FrmMain Configuration

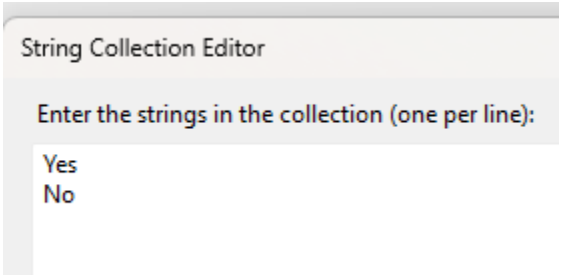

Figure 9: ComboBox Collection

m. In FrmMain, place a groupBox on the form as shown in Figure 10.
n. Name the groupBox "grbAttributes."
o. Update the groupBox Text property to "Dog Attributes."

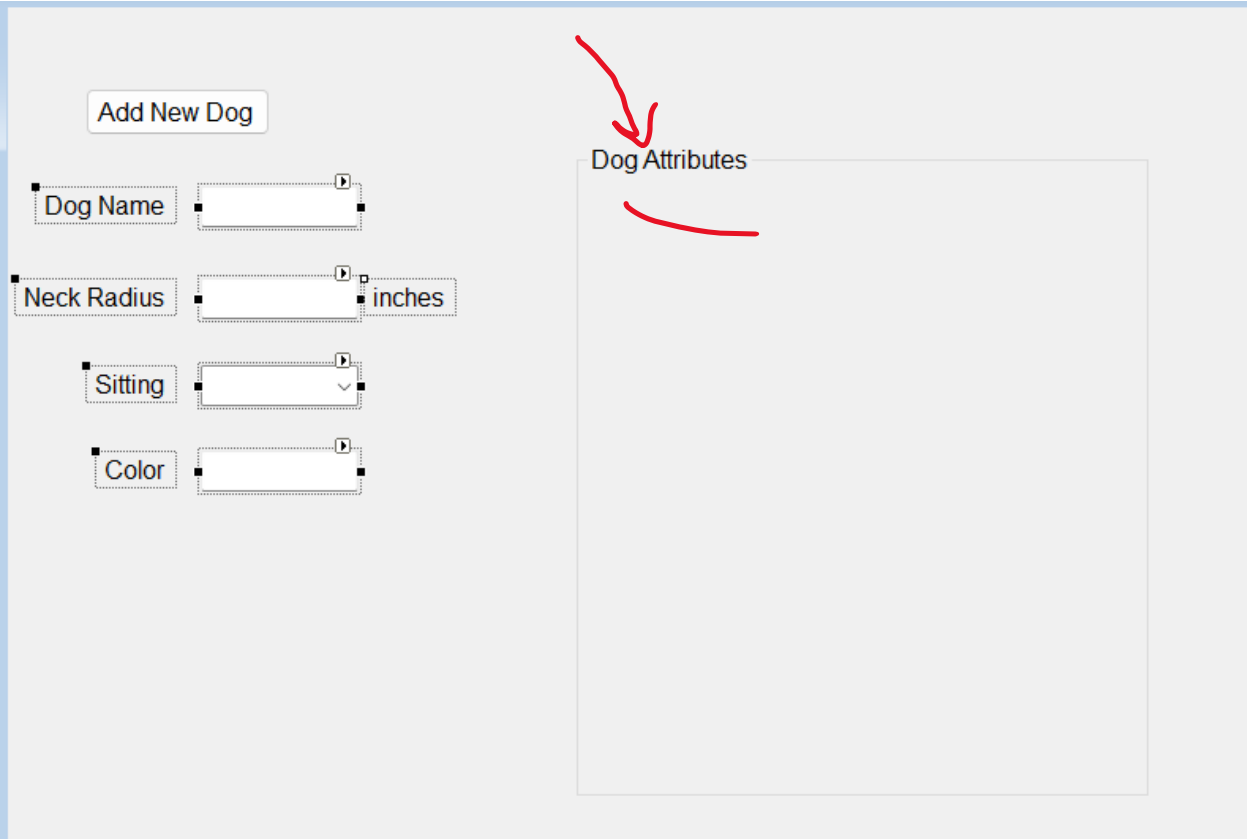

Figure 10: Add GroupBox to Form.

p. Then, select all the attributes by dragging the mouse from the top left to the bottom right of all the tools that make up the attributes.

q. Notice we did not select the Button control since this is not an attribute.

r. Now, drag all the selected tools/attributes and place them inside the groupBox that is now named Dog Attributes as shown in Figure 11.

s. Then, correctly size the groupBox so all the controls fit inside it.

t. Then, place it under the button as shown in Figure 11.

u. Notice now by just moving the groupBox, all the controls inside the groupBox move together.

v. They have been grouped together.

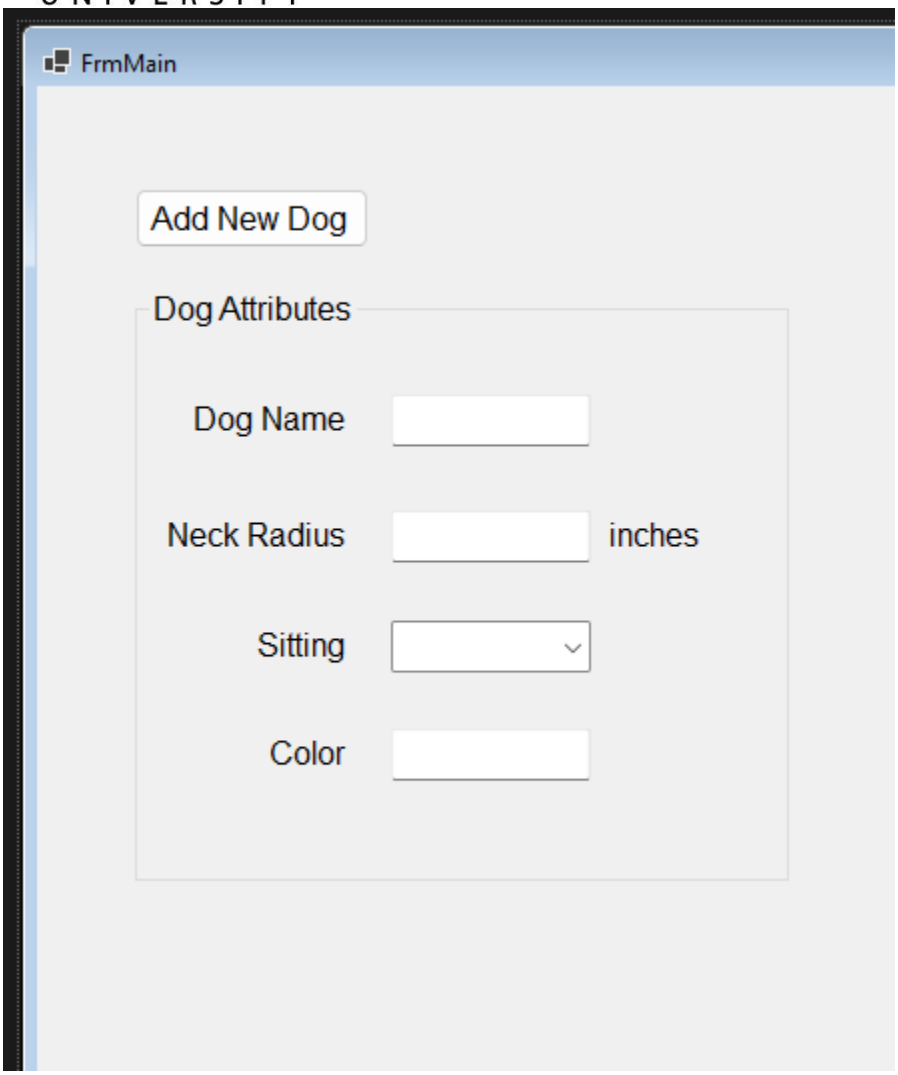

Figure 11: Place Attributes inside groupBox.

w. Now, select a DataGridView from the toolbox and drag it onto the form as shown in Figure 12 and Figure 13.

x. Be sure to size it like shown in Figure 13.

y. Name the DataGridView: "gvShowDogs."

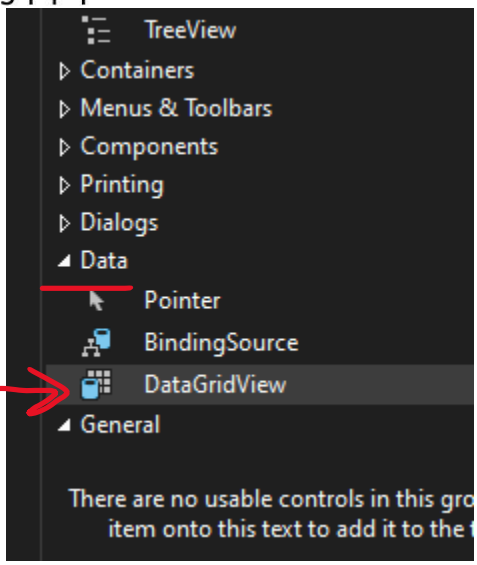

Figure 12: DataGridView Control

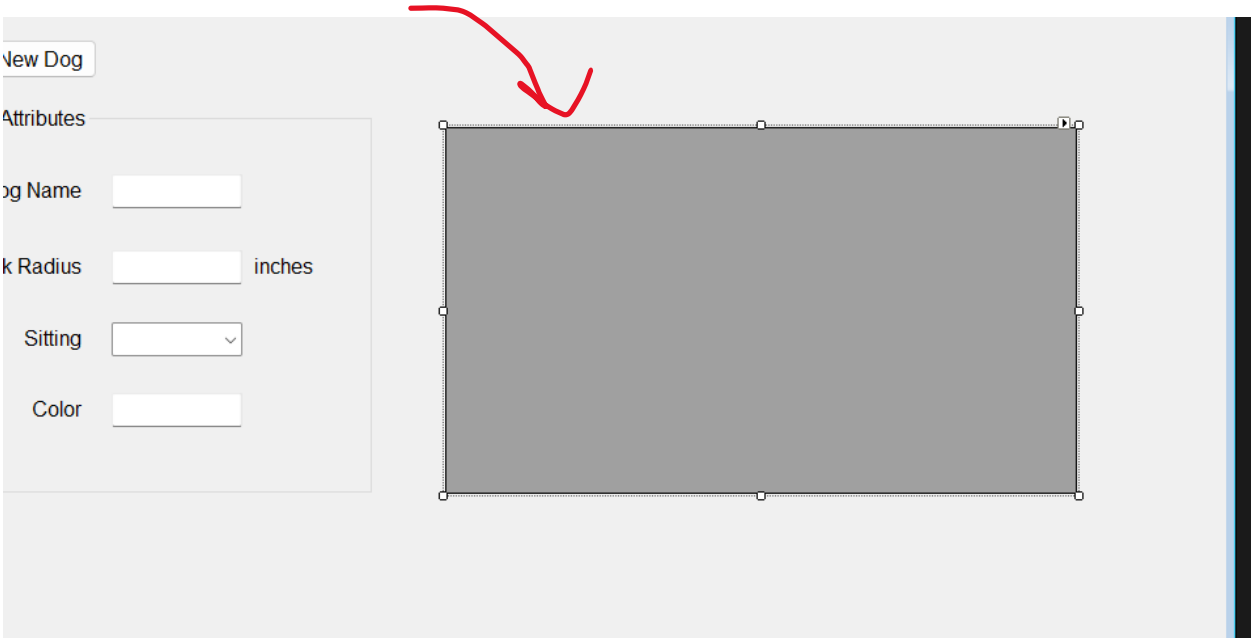

Figure 13: Drag DataGridView Control on to the Form.

5. Add code behind the form in Presentation Layer.
    a. Create a Click Event Handler for the New Dog button as shown in Figure 14.

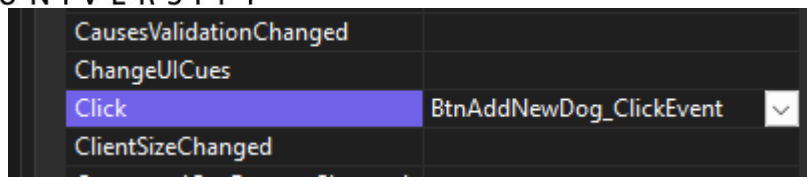

Figure 14: New Dog Click Event Handler

b. Name this click event: "BtnAddNewDog_ClickEvent."

c. Then, add method level comments as shown in Figure 15.

```
        InitializeComponent();
    }

    /// <summary>
    /// Click event to add a new dog to the datagridview tool
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    1 reference
    private void BtnAddNewDog_ClickEvent(object sender, EventArgs e)
    {

    }
}
```

Figure 15: Method Level Comments

d. Whenever a new cs page is started, we immediately add the citations at the very top as shown in Figure 16.

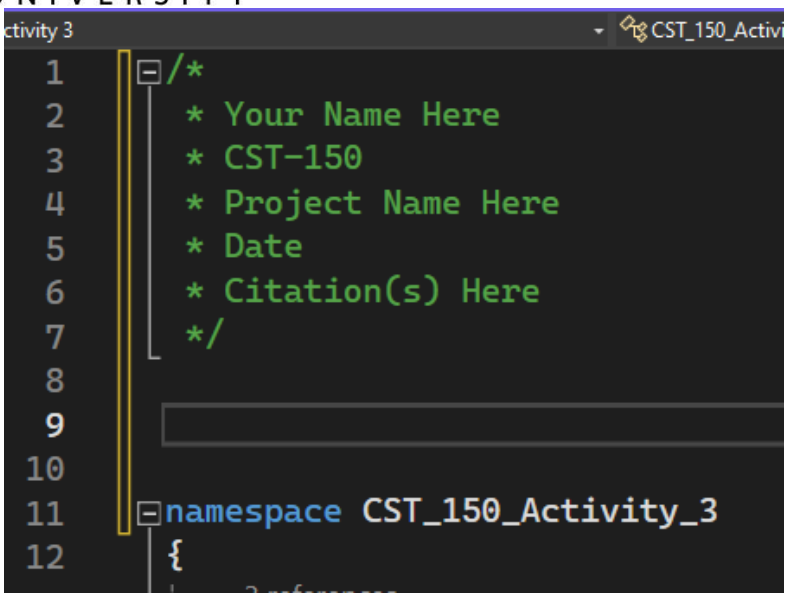

Figure 16: Citation at the top.

e. Start a new testing process. When we write a little bit of code, be sure to test it.

f. Testing – now, since we have added some new controls, let's test the groupBox first.

g. Set a break point on "var bill = txtName.Text; line. Run the code and click the New Dog button.

h. In the click event handler, read the name and verify everything is working as shown in Figure 17. Hover the mouse over the var bill (or whatever variable name you put there) and verify the value placed in the textbox is present in the variable.

i. Then test if we can read from the comboBox as shown in Figure 17. Hover the mouse over the var comboBox and verify the selected item is present in the variable name.

```
/// <summary>
/// Click event to add a new dog to the datagridview tool
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnAddNewDog_ClickEvent(object sender, EventArgs e)
{
    // testing that we can read the name inside the groupbox
1.  var bill = txtName.Text;

    // How do we read from cmbBox
2   var combobox = cmbSit.SelectedItem;

}  ≤ 1ms elapsed
}
```

Figure 17: Testing Controls

6. Addition to FrmMain.
    a. Right after color, we are going to add the weight attribute as shown in Figure 18.
    b. Name the property: "txtWeight."

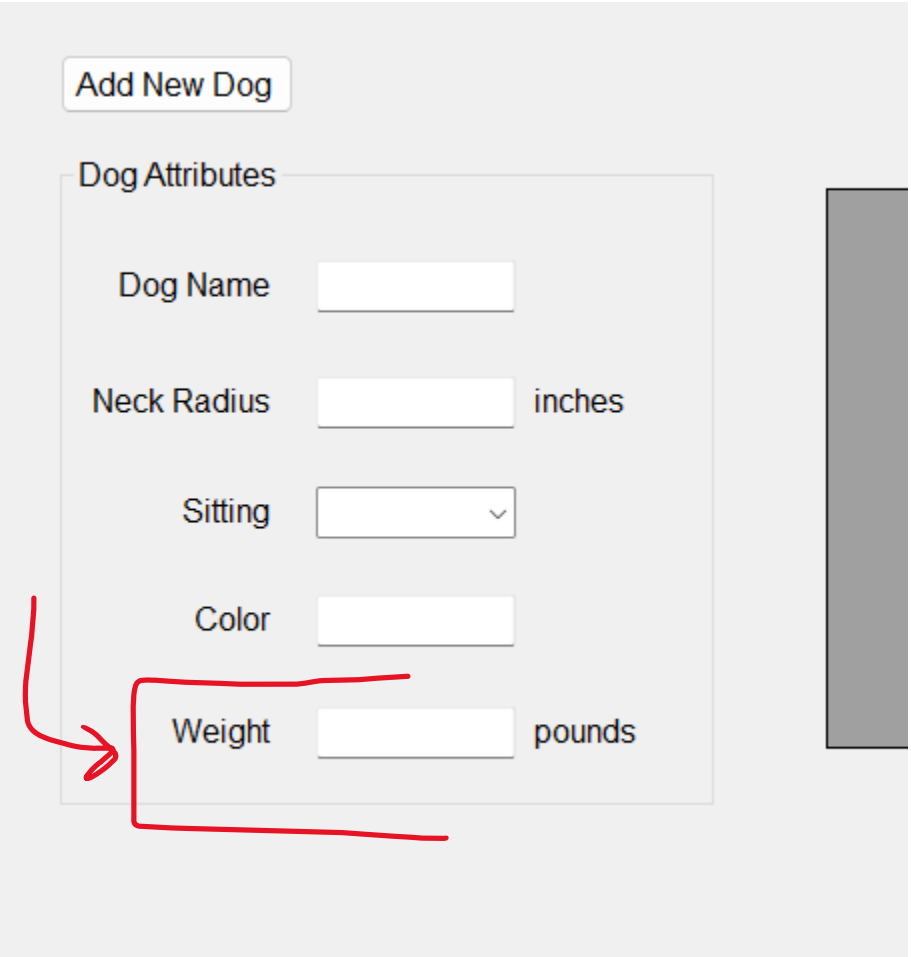

Figure 18: Add weight attribute.

7. Business Layer
    a. Create a new directory "BusinessLayer" as shown in Figure 19.
    b. Inside the Business Layer, add a new class named "Dog.cs."
    c. In Dog.cs, be sure the namespace is correct as shown in Figure 20.

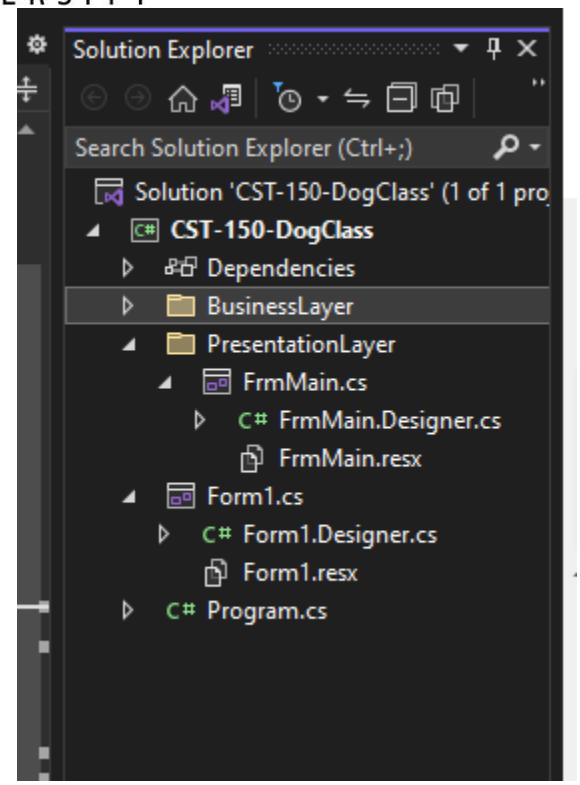

Figure 19: Create BusinessLayer.

```
using System.Threading.Tasks;

namespace CST_150_DogClass.BusinessLayer
{
    2 references
    internal class Dog
```

Figure 20: Review Namespace.

d. Inside the class "Dog.cs," the first item is to define the attributes/properties as shown in Figure 21.

e. The UML should be completed first and the properties come right from the attributes section of the UML.

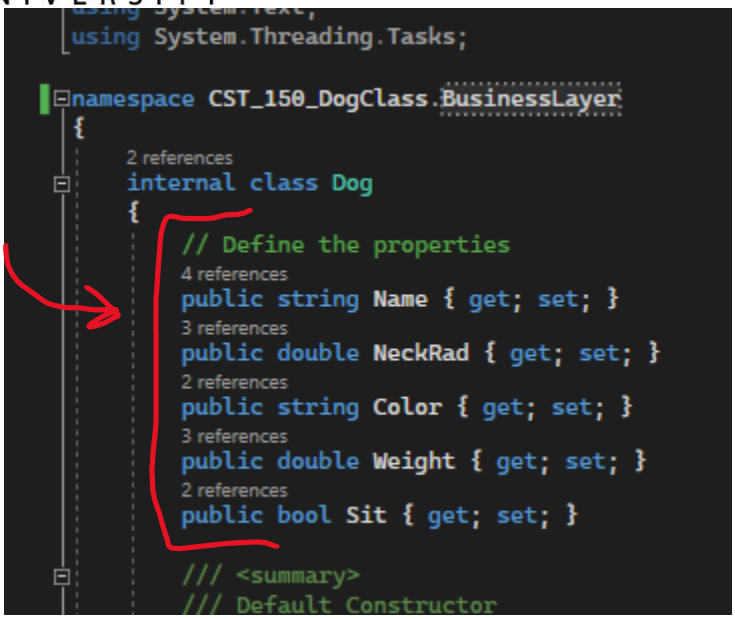

Figure 21: Define Properties.

f. The second step is to create the constructors based on the UML.

g. Right-click in the row you want to create the constructor and select "Quick Actions and Refactoring," followed by Generate Constructor.

h. Create the default constructor, so be sure none of the check boxes are selected as shown in Figure 22, then select OK.

i. Figure 23 shows the result as the Default Constructor.

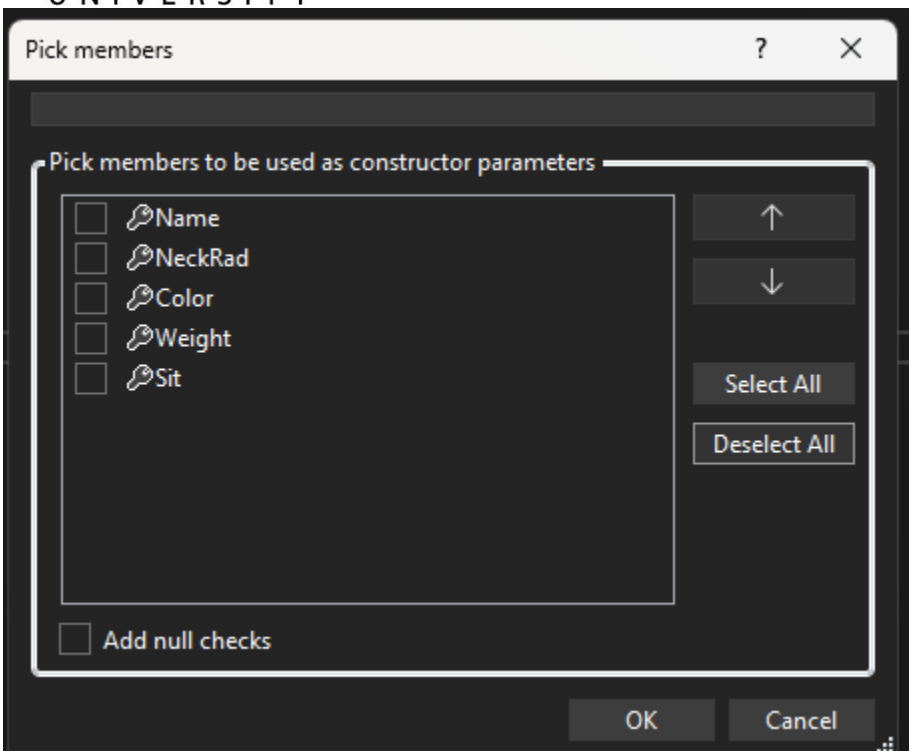

Figure 22: Create Default Constructor

```
public bool Sit { get; set; }

/// <summary>
/// Default Constructor
/// </summary>
0 references
public Dog()
{
    // Initialize the Properties
    Name = "";
    NeckRad = 0.00D;
    Color = "";
    Weight = 0.00D;
    Sit = false;
}

/// <summary>
/// Parameterized Constructor
```

Figure 23: Dog Default Constructor

j. Now, we will create the parametrized constructor.

k. Do the same, except this time, be sure all the boxes are selected as shown in Figure 24.

l. Figure 25 show the end result, Parameterized Constructor.

m. Be sure to add all the appropriate comments to all methods.

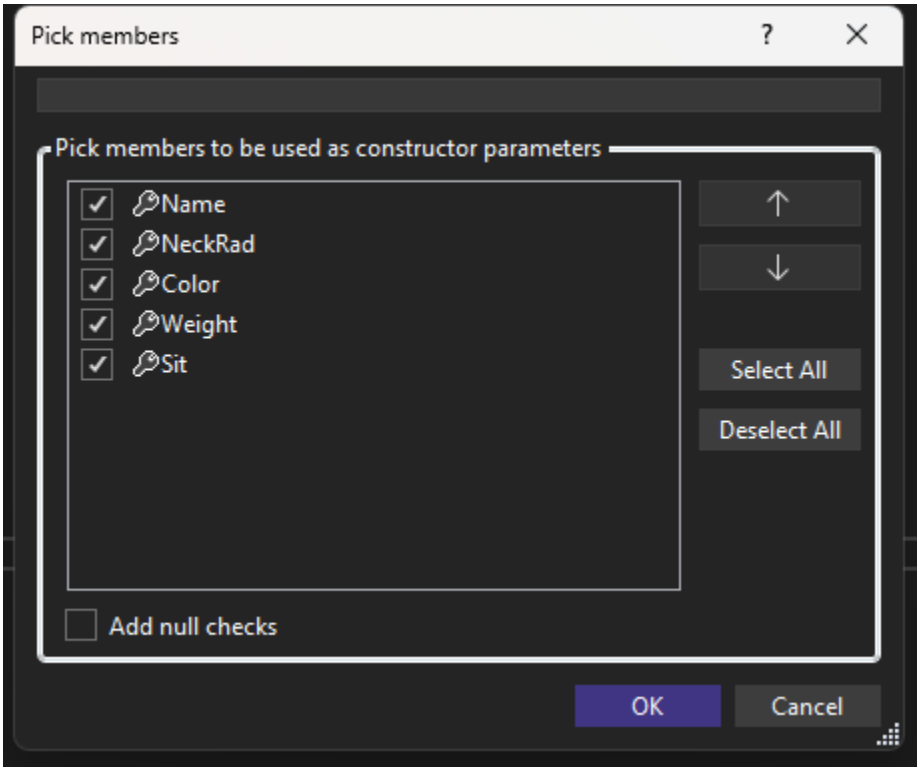

Figure 24: Be sure all boxes are checked.

```
0 references
public Dog()
{
}

/// <summary>
/// Parameterized Constructor
/// </summary>
/// <param name="name"></param>
/// <param name="neckRad"></param>
/// <param name="color"></param>
/// <param name="weight"></param>
/// <param name="sit"></param>
1 reference
public Dog(string name, double neckRad, string color, double weight, bool sit)
{
    // Main purpose of constructor is to initialize the properties
    Name = name;
    NeckRad = neckRad;
    Color = color;
    Weight = weight;
    Sit = sit;
}

/// <summary>
/// Method that takes the property NeckRad and returns the circumference
```

Figure 25: Results is a Parameterized Constructor.

n. Now, the requirements and UML are telling us we need to create the CalCircumference() method as shown in Figure 26.

```
        Sit = sit;
    }

    /// <summary>
    /// Method that takes the property NecKRad and returns
    /// the circumference in centementers.
    /// </summary>
    /// <returns></returns>
    0 references
    public double CalCircumference()
    {
        // Declare and Initialize
        const double cmConversion = 2.54D;
        double circumference = 0.00D;

        // Since NeckRad is at the class level we can use this
        // property inside this method.
        // Conversion from radius to circumference (2*Pi*r)
        circumference = 2 * Math.PI * NeckRad;
        // Then convert to centemeters from inches
        return (circumference * cmConversion);

    }
```

Figure 26: CalCircumference Method

o. Keeping all the logic in the Business Logic Layer, create the CalWeight method as shown in Figure 27.

p. The Dog.cs Class is complete.

```
        return (circumference * cmConversion);

    }

    /// <summary>
    /// Convert Weight pounds to kilograms
    /// </summary>
    /// <returns></returns>
    0 references
    public double CalWeight()
    {
        // Declare and Initialize
        const double kgConversion = 0.453592D;
        // Convert the property Weight from pounds to kilograms
        return(Weight * kgConversion);
    }
}
```

Figure 27: CalWeight Method.

8. Presentation Layer: Back to FrmMain.cs
    a. Now, back to the code behind the form in the presentation layer.
    b. The event handler will be used as our main method.
    c. We have written a lot of code, so let's test the Dog class.
    d. For now, let us go over how to instantiate our Dog class.
    e. Using the Debugger, step through the code and show how the constructor is executed by viewing the values in var name and var color as shown in Figure 28.

```
1 reference
private void BtnAddNewDog_ClickEvent(object sender, EventArgs e)
{
    // How do we read form cmbBox
    var comboBox = cmbSit.SelectedItem;

    // Test our class
    // Instantiate the Dog class and create a new Dog object called ginger
    Dog ginger = new Dog("Ginger", 12.24, "Golden Cream", 57.25, false);

    var name = ginger.Name;
    var color = ginger.Color;


}
}
```

Figure 28: Test Dog Class.

f. If there is a red line under Dog, this means the Dog class cannot be found and we need to add a using statement pointing to the Dog class location.

g. Hover over Dog and click on "Show potential fixes" as shown in Figure 29.

h. Select "using CST…" as shown in Figure 30.

i. This adds the using statement to the top of the page as shown in Figure 31.

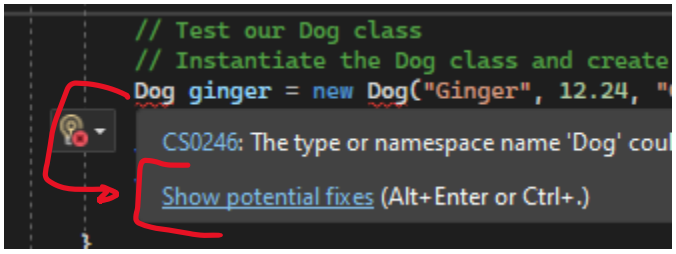

Figure 29: Hover over Dog with red line.

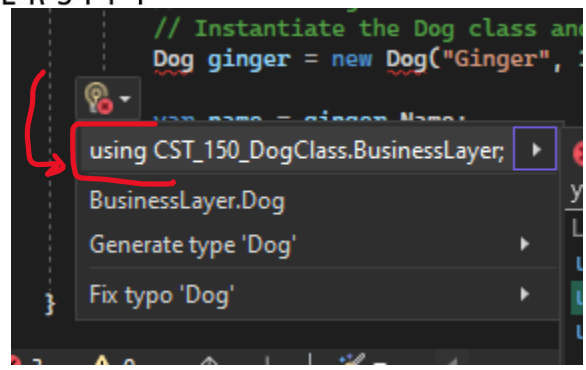

Figure 30: Select using from the pull down.

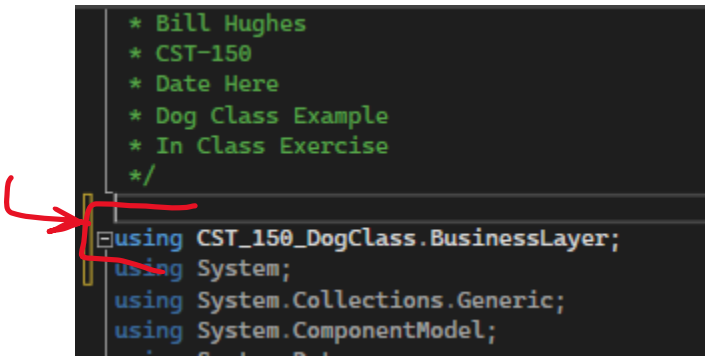

Figure 31: Using Statement.

9. Business Layer: Utility Class
    a. Following the flowchart that was created in the previous activity, the next step is to verify that the text boxes contain valid information when we add a new item as shown in Figure 32.
    b. Verification that text boxes contain valid information is business logic, so it needs to be put in the BusinessLayer.

```
/// <param name="e"></param>
1 reference
private void BtnAddNewDog_ClickEvent(object sender, EventArgs e)
{
    // Declare and Initialize


    //------------------------------------------------------
    // Verify all boxes have correct inputs
    // We will be using a Utility / Helper Class to verify inputs

    //------------------------------------------------------
```

Figure 32: Presentation Layer Event Handler

c. Add a new class named "Utility" in the BusinessLayer as shown in Figure 33.

d. This class will hold a bunch of useful methods that we can move from solution to solution and just use them as is.

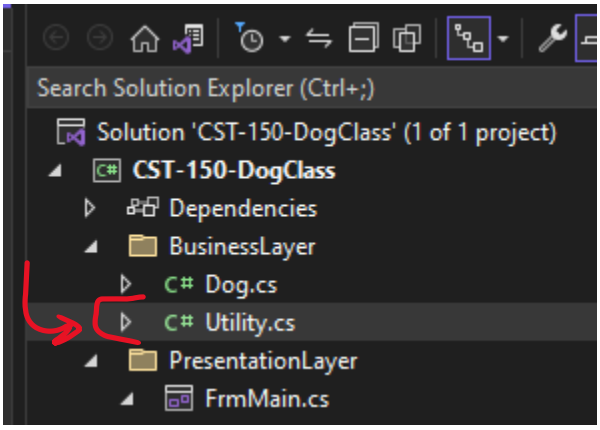

Figure 33: Add Utility Class.

e. The Utility class will have a UML, so let us create one.

f. These will typically not have attributes and only Default Constructor as shown in Figure 34.

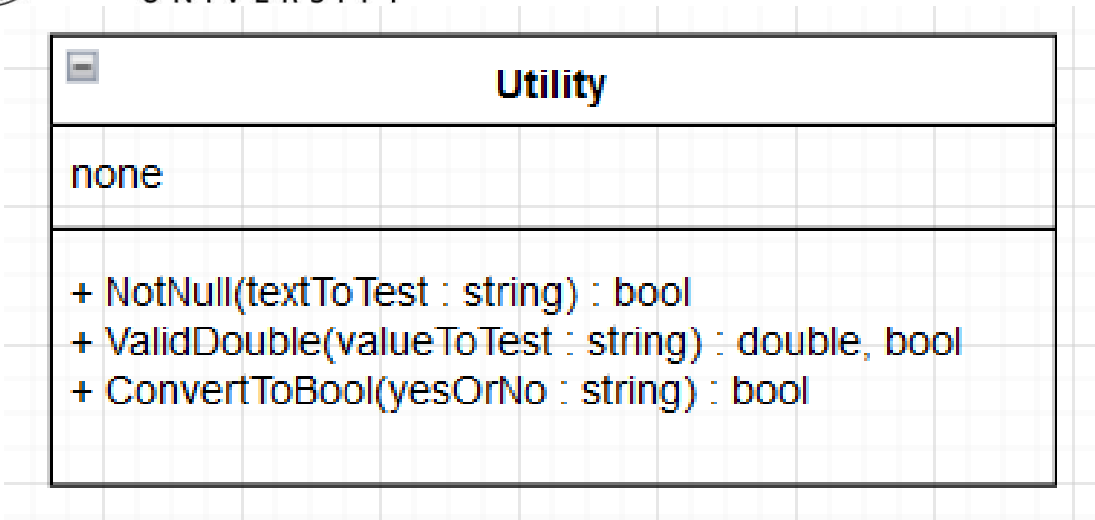

Figure 34: Utility Class UML

g. Start with the first method from the UML as shown in Figure 35.

h. Use String with cap S since we are accessing the class and method for IsNullOrWhiteSpace method.

```
0 references
internal class Utility
{

    /// <summary>
    /// Utility that returns false if the parameter string is null, empty,
    /// or just contains white spaces.
    /// </summary>
    /// <param name="textToTest"></param>
    /// <returns></returns>
    0 references
    public bool NotNull(string textToTest)
    {
        // Check if the string is empty, null, or contains only whitespaces
        if (String.IsNullOrWhiteSpace(textToTest))
        {
            return false;
        }
        return true;

    }


}
}
```

Figure 35: NotNull method.

i. Now for the second method from the UML as shown in Figure 36 with method name "ValidDouble."

j. This method introduces the Tuple. The Tuple is a data structure that can have multiple parts. We are only using 2 elements but can manage up to 8 elements. Tuple allows multiple data return types in a single data set as shown in Figure 36, return types of "(double doublValue, bool isConverted)."

k. Be sure to comment the code as is shown in the figures, which is designed to help the learning process.

```
/// <summary>
/// Tests to determine if a valid double was entered.
/// If true the string is parsed to double and true is returned.
/// If false, -1 is returned and false for bool.
/// This return type is called a Tuple.
/// </summary>
/// <param name="valueToTest"></param>
/// <returns></returns>
2 references
public (double doublValue, bool isConverted) ValidDouble(string valueToTest)
{
    // Declare and Initialize
    double convertValue = 0.00D;
    // Test to see if the string can be parsed to a double
    if(Double.TryParse(valueToTest, out convertValue))
    {
        return (convertValue, true);
    }
    // If parse fails return false and -1
    return (-1D, false);
}
```

Figure 36: Second Utility Class Method.

l. Now for the third and last method from the UML as shown in Figure 37, with method name "ConvertToBool."

m. This method converts a string "Yes" to bool "true" and anything else to bool "false."

```
        // If conversion fails return false
        return (0.00D, false);
    }

    /// <summary>
    /// Convert Yes to bool true and No to bool false
    /// </summary>
    /// <param name="YesOrNo"></param>
    /// <returns></returns>
    0 references
    public bool ConvertToBool(string YesOrNo)
    {
        if(YesOrNo == "Yes")
        {
            return true;
        }
        return false;
    }
```

Figure 37: Last Method in Utility Class.

10. Back to the Presentation Layer
    a. Following the flowchart that was created in the previous activity, the next step is to create the section for "Declare and Initialize" as shown in Figure 38 and Figure 39. Remember, we are using the Button click event handler as the main method.
    b. The main method must be kept clean.
    c. Add the variables as shown in Figure 39.

```
/// <summary>
/// Click event to add a new dog to the datagridview tool
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnAddNewDog_ClickEvent(object sender, EventArgs e)
{
    // Declare and Initialize
```

Figure 38: Declare and Initialize.

```
    // Declare and Initialize
    // Flag that tracks if all entries are valid
    bool isValidEntries = true;
    double weight = 0.00D, neckRad = 0.00D, neckCircum = 0.00D;
    // Flag to verify parsing doubles is true or false
    bool isValid = false;

    // Make sure the Error Label is not visible
    lblErrorMessage.Visible = false;

    // Instantiate our Utility class so we can use it
    Utility utility = new Utility();

    //-----------------------------------------------------
```

Figure 39: Add Variables.

d. Let's use the Utility class that we just created.

e. Instantiate the Utility class and create an object of the class named "utility" as shown in Figure 40.

f. Now, we have access to all the public methods inside the Utility class by using the object we just created named "utility.

```
    // Declare and Initialize
    // Flag that tracks if all entries are valid
    bool isValidEntries = true;
    double weight = 0.00D, neckRad = 0.00D, neckCircum = 0.00D;
    // Flag to verify parsing doubles is true or false
    bool isValid = false;

    // Make sure the Error Label is not visible
    lblErrorMessage.Visible = false;

    // Instantiate our Utility class so we can use it
    Utility utility = new Utility();

    //-----------------------------------------------------
```

Figure 40: Instantiate the Utility Class.

g. Now, test the text boxes and comboBox.

h. We call the NotNull for the two text boxes, but then we need to just test for "null" with the comboBox as shown in Figure 41.

i. Use a flag, so we know if any of the entries are not valid as is shown by the "isValidEntries" in the figure.

```
// Instantiate our Utility class so we can use it
Utility utility = new Utility();

//-----------------------------------------------------
// Verify all boxes have correct inputs
// We will be using a Utility / Helper Class to verify inputs
if (!utility.NotNull(txtName.Text) || !utility.NotNull(txtColor.Text) || (cmbSit.SelectedItem == null))
{
    isValidEntries = false;
}
```

Figure 41: Validate Data Entry.

j. Now, test double input for neck size.

k. This is where we call the ValidDouble method from Utility class as shown in Figure 42.

```
    isValidEntries = false;
}
// Now test the text box for valid double neck size
(neckRad, isValid) = utility.ValidDouble(txtNeck.Text);
if(!isValid)
{
    isValidEntries = false;
}
// Now test the weight
(weight, isValid) = utility.ValidDouble(txtWeight.Text);
```

Figure 42: Validate Double Data Type.

l. Now test double input for weight.

m. Again, this is where we call the ValidDouble method from Utility class as shown in Figure 43.

n. Now, if isValidEntries remained true then we know we are good to go. But if it is false, we need to display an error message.

```
(neckCircum, isValid) = utility.ValidDouble(txtNeck.Text);
if (!isValid)
{
    isValidEntries = false;
}
// Now test the text box for valid double neck size
// If valid the weight will contain the double value
(weight, isValid) = utility.ValidDouble(txtWeight.Text);
if (!isValid)
{
    isValidEntries = false;
}

//------------------------------------------------
```

Figure 43: Validate Weight Input.

11. Presentation Layer: Open FrmMain Design View

a. If any data entry is not valid, we need to display a message to the user.
b. Add a label to the form as shown in Figure 44.
c. Name: lblErrorMessage.
d. Make the ForeColor property "Red."
e. Make the visibility property "False."
f. Update the Text property to "Please fix the incorrect data entry…Then try again…"

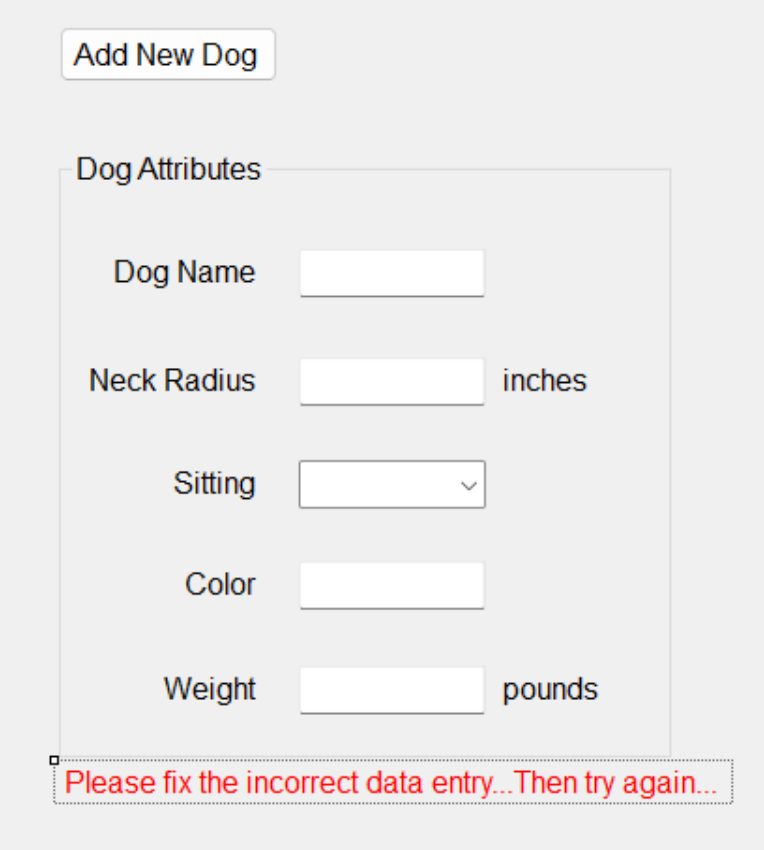

Figure 44: Add Error Message.

12. Back to the code behind the FrmMain in the Presentation Layer.
    a. To manage our label, be sure it is not visible when the program starts as shown in Figure 45.
    b. Also, be sure the label is not visible anytime we click the button as shown in Figure 46.

```
public partial class FrmMain : Form
{
    1 reference
    public FrmMain()
    {
        InitializeComponent();
        // Error Label not visible
        lblErrorMessage.Visible = false;
    }
```

Figure 45: Verify the Label is not Visible.

```
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
1 reference
private void BtnAddNewDog_ClickEvent(object sender, EventArgs e)
{
    // Declare and Initialize
    // Flag that tracks if all entries are valid
    bool isValidEntries = true;
    double weight = 0.00D, neckRad = 0.00D, neckCircum = 0.00D;
    // Flag to verify parsing doubles is true or false
    bool isValid = false;

    // Make sure the Error Label is not visible
    lblErrorMessage.Visible = false;

    // Instantiate our Utility class so we can use it
    Utility utility = new Utility();

    //-------------------------------------------------------
```

Figure 46: On Event Handler hide the button.

c. If isValidEntries is true, we can populate the datagridview.

d. Start by adding the if statement to begin the entire process as shown in Figure 47.

e. Inside this if statement is where we create a new Dog. The first step is to instantiate the Dog class and create a new dogObject as is shown in Figure 47.

f. Then, the second line of code adds the new Dog to the data grid view using the dogObject that was just created.

g. Notice how the methods are called to CalCircumference() and CalWeight() right inside the Add statement. These methods are all inside the Dog class and are being

invoked using the dogObject that was just created in the previous line. No arguments need to be sent to the methods since the methods are using the Properties of the class that were defined when the class was instantiated using the Parameterized constructor which is designed to initialize the Class Properties.

```
// datagridview with the entry
// else display error message.
if(isValidEntries)
{
    // If we are here we know we have valid entries
    // so lets populate the datagridview
    Dog dogObject = new Dog(txtName.Text, neckRad, txtColor.Text, weight, utility.ConvertToBool(cmbSit.Text));
    gvShowDogs.Rows.Add(dogObject.Name, dogObject.CalCircumference(), dogObject.Sit, dogObject.CalWeight(), dogObject.Color);

}
else
{
    // If we are here there is a problem with an entry
```

Figure 47: isValidEntries is True.

h. Then, we can add in the "else" that will show the error message as shown in Figure 48.

i. Since we already put the text in the label, all we need to do is show it.

```
//-------------------------------------------------------
// if isValidEntries is still true we can populate the
// datagridview with the entry
// else display error message.
if(isValidEntries)
{
    // If we are here we know we have valid entries
    // so lets populate the datagridview

}
else
{
    // If we are here there is a problem with an entry
    // Show the error message.
    lblErrorMessage.Visible = true;
}
```

Figure 48: Add the else.

j. Before we can add the rows to the datagridview, we need to add the columns to configure the entire grid.

k. Let's do this when the form is loaded. Create a load event handler for the form.

l. Call the load event handler "FrmMainLoadEventHandler" as shown in Figure 49.

```
        var name = ginger.Name;
        var color = ginger.Color;

    }


    /// <summary>
    /// When the form loads execute this event handler
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    1 reference
    private void FrmMainLoadEventHandler(object sender, EventArgs e)
    {

    }
}
```

Figure 49: Form Load Event Handler.

m. In the "FrmMainLoadEventHandler," add the columns for the grid view as shown in Figure 50 part A. This section of code also inserts the header text that is shown above each column.

n. Also, format the double columns so there are only 2 decimals as shown in Figure 50 part B.

```
    }


    /// <summary>
    /// When the form loads execute this event handler
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    1 reference
    private void FrmMainLoadEventHandler(object sender, EventArgs e)
    {
        // Set the number of rows to 5 and configure each row
        gvShowDogs.ColumnCount = 5;
        gvShowDogs.Columns[0].Name = "Name";
        gvShowDogs.Columns[1].Name = "Neck Circum";
        gvShowDogs.Columns[2].Name = "Sitting";
        gvShowDogs.Columns[3].Name = "Weight";
        gvShowDogs.Columns[4].Name = "Color";

        // Format number in neck and weight for 2 decimals
        gvShowDogs.Columns[1].DefaultCellStyle.Format = "#.00";
        gvShowDogs.Columns[3].DefaultCellStyle.Format = "#.00";
    }
```

Figure 50: Format Data Grid View Columns.

o. Run the application and enter values in the text boxes and pull down.

p. Verify the data grid view is correctly populated as shown in Figure 51.

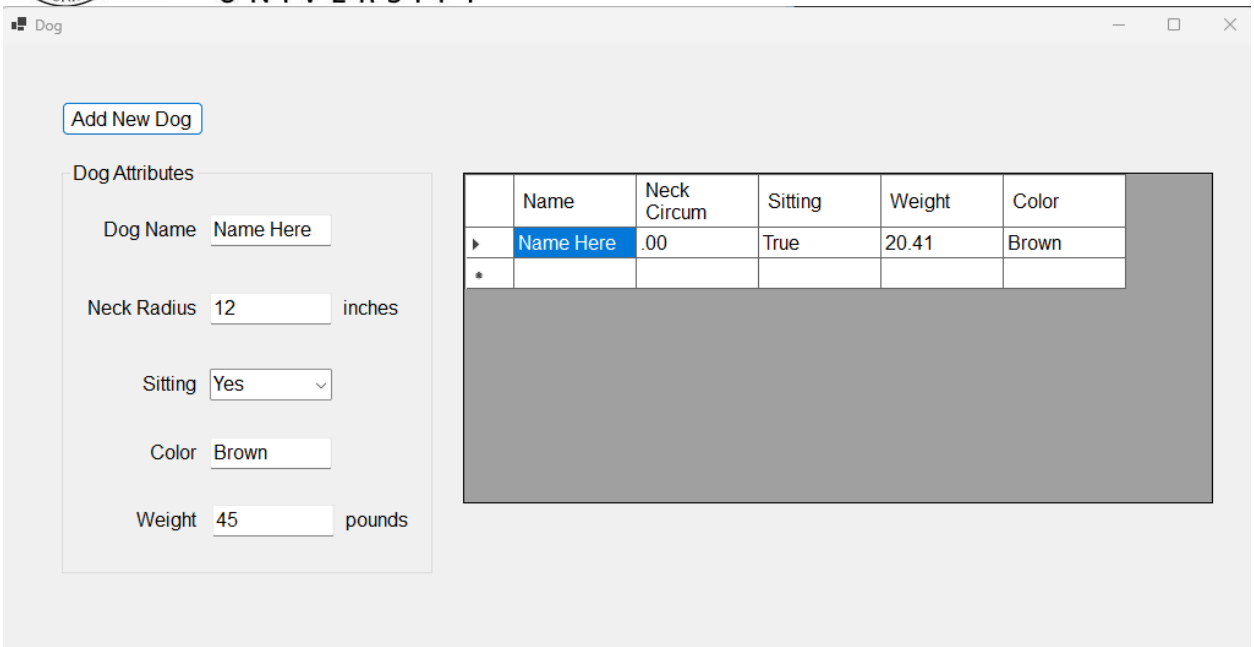

Figure 51: End Result with Data Grid View populated.

13. Submit the Activity as described in the digital classroom.

Part 2

Dice Class

Overview

Roll Two Dice: Implement Programming Problem 9 is found in Chapter 5 of the textbook.

Submit the Activity as described in the digital classroom.