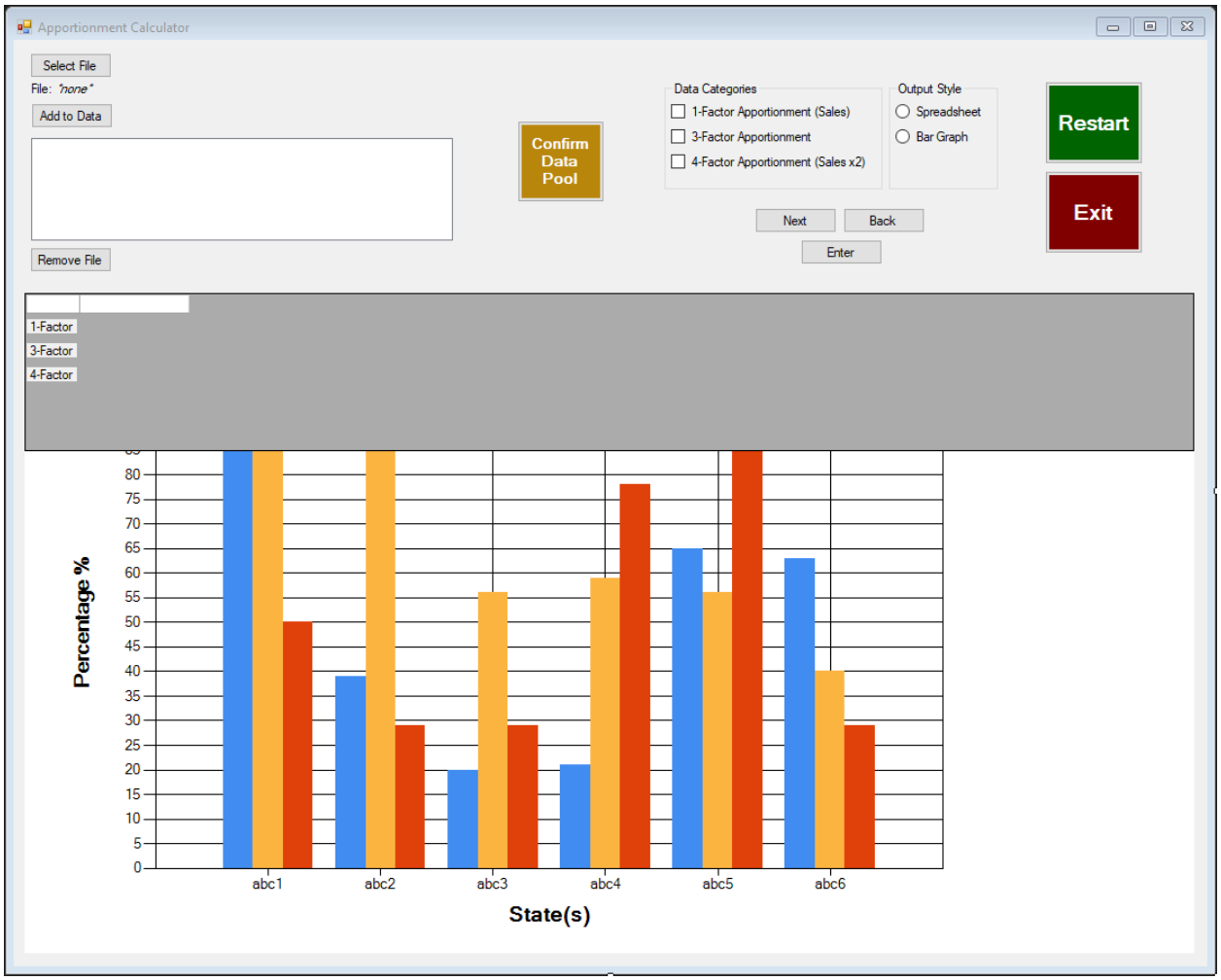
Criterion E: Product development

Advanced techniques used to address the client's requirements:

- CSV File Handling
- Class Datapoint Array Building to store File Data
- Loops, If-Then, Else Conditions
- Graphical User Interface Form Elements

Product structure



This is the structure of the product that will be distributed. The form elements at the top are compact and flow from left to right in the order of client selection. The two graphical elements at the bottom have a large enough space to be clearly seen by the client. The program is exported as an executable file, therefore the client will be able to open it with a simple click. The overall structure of the program is straightforward so the client can understand how the program flows without an explanation.

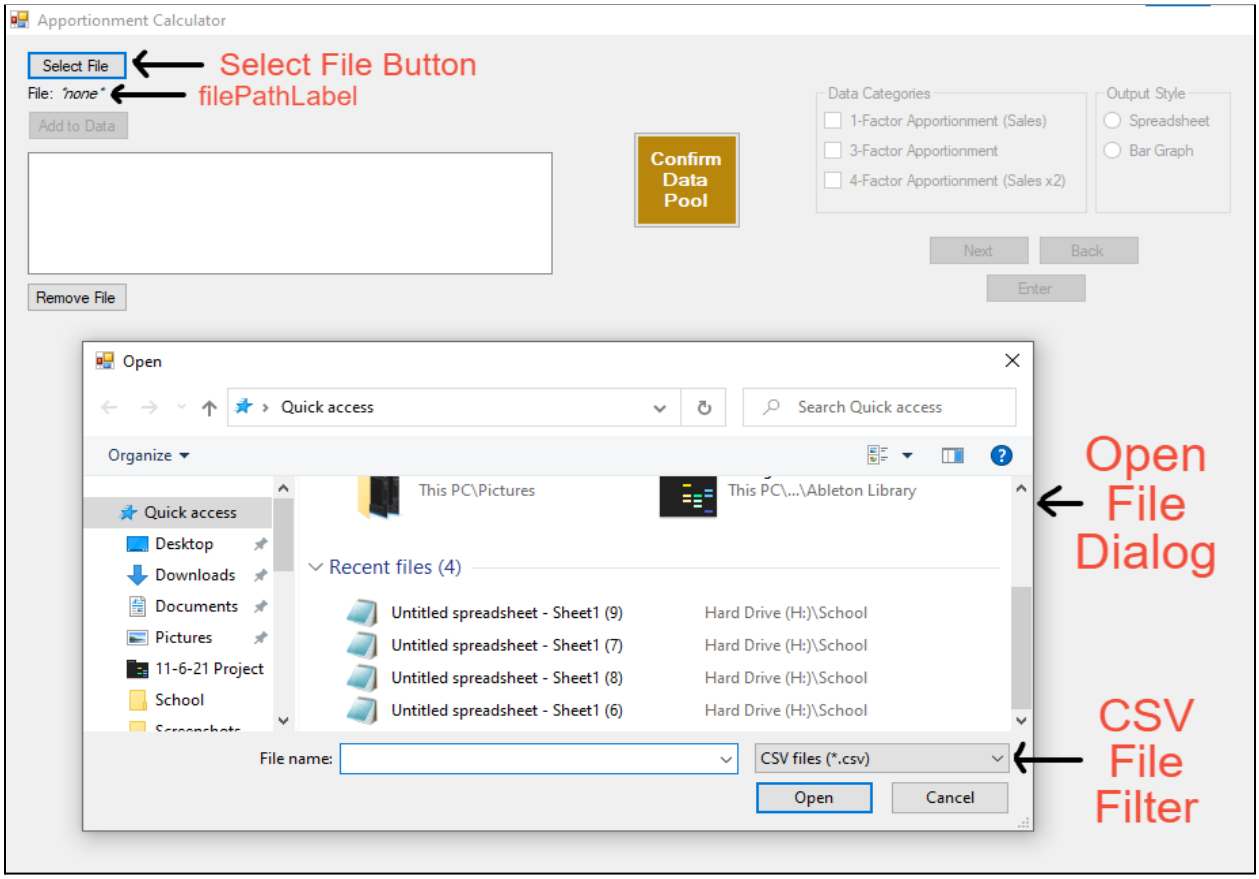
The structure and organization of the program

*(*Images are always below the corresponding text. All yellow source code indicates parts that were used and unaltered)*

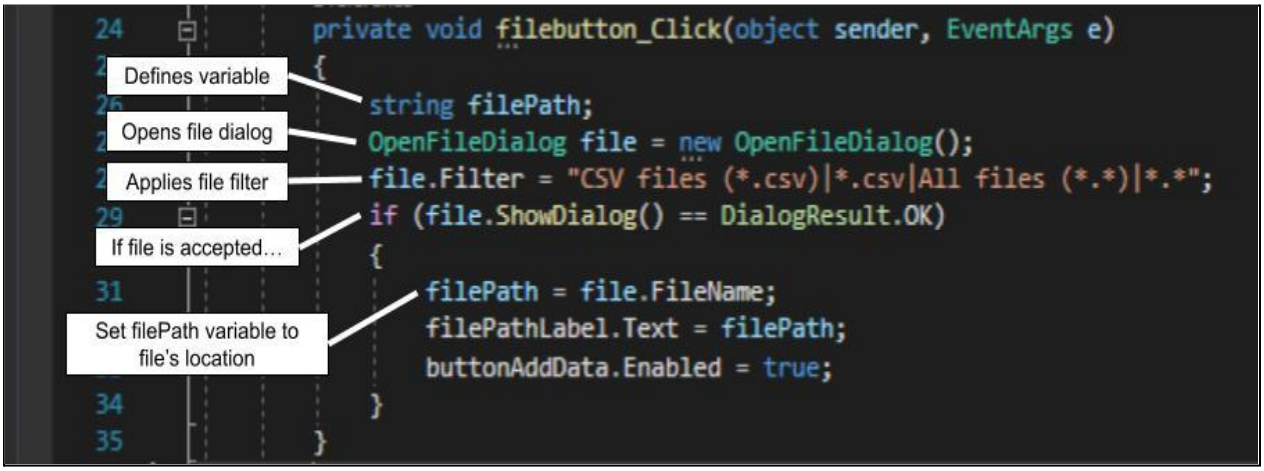
Technique: CSV File Handling

A button was added to Form 1 with the label “Select File.” When this button is pressed, it is programmed to open up the client’s file library and prompt them to select a file. A filter was added to restrict the file type the client can select to only CSV files. A string variable “filePath” is defined, which stores the location on the client’s computer of the file they select. A text label “filePathLabel” was added underneath the Select File button, whose text is set to display the file location string after the client selects a file. *(*See Images Below)*

Code Outcome:



Program Code:



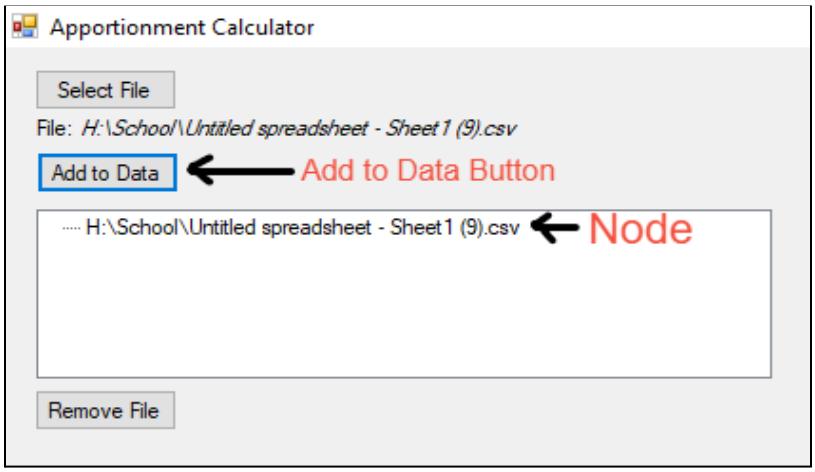
Source Code:

```
string path;
OpenFileDialog file = new OpenFileDialog();
if (file.ShowDialog() == DialogResult.OK)
{
    path = file.FileName;
}
```

*harryovers. “Visual Studio File Selector.” *Stack Overflow*. Last modified September 1, 2009. Accessed October 18, 2021. https://stackoverflow.com/questions/1671127/visual-studio-file-selector/1671137.

Another button was added as well as a list tree. This button, labeled “Add to Data”, creates a new node in the list titled with the file path the client selected. *(*See Images Below)*

Code Outcome:

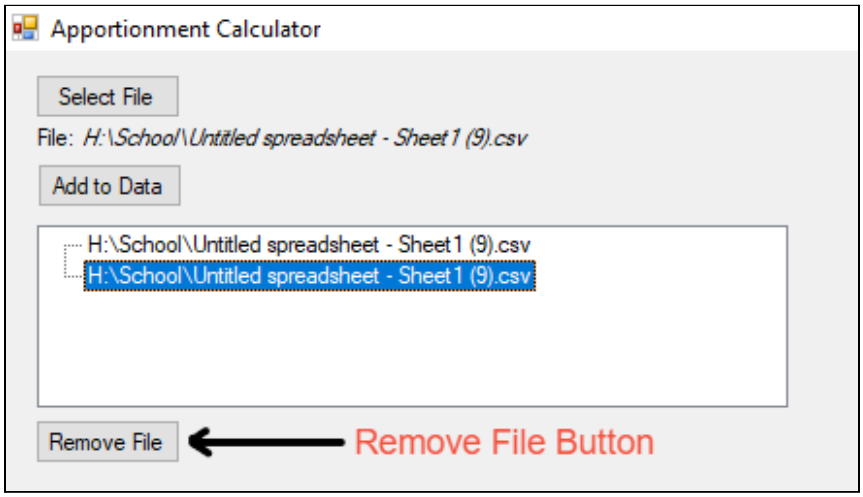


Program Code:

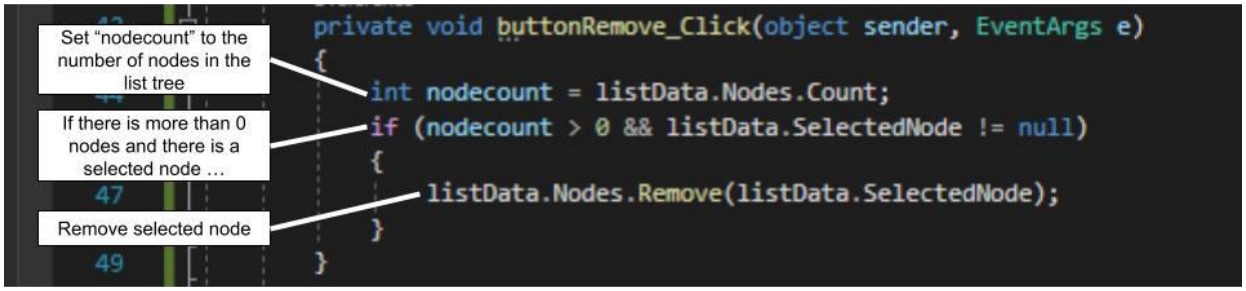
```
36          private void buttonAddData_Click(object sender, EventArgs e)
37          {
38              string filePath = filePathLabel.Text;
39              listData.Nodes.Add(filePath);
40          }
```

A button was also added underneath to remove a node from the list if the client selects a node. *(*See Images Below)*

Code Outcome:

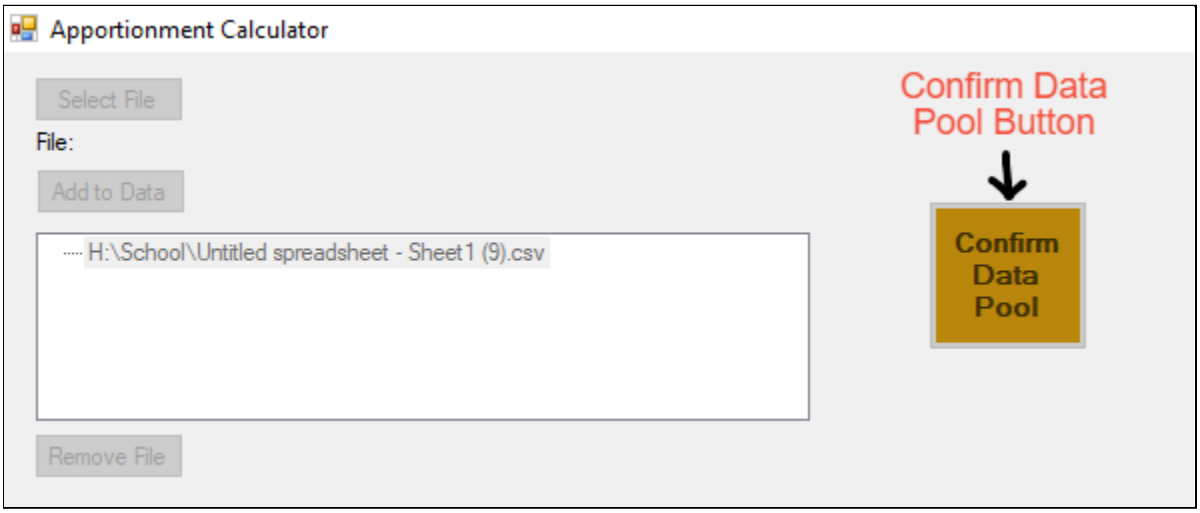


Program Code:

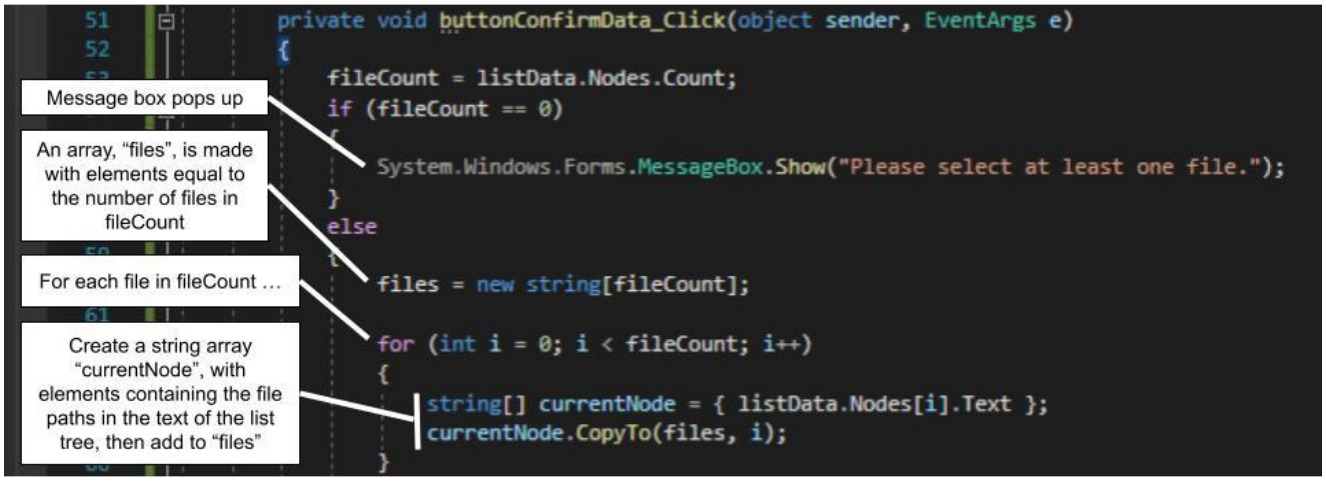


A Confirm Data Pool button was added, that when clicked, confirms the client has at least one file, then creates an array containing the file path nodes as strings so the files can be read for future data manipulation. *(*See Images Below)*

Code Outcome:



Program Code:



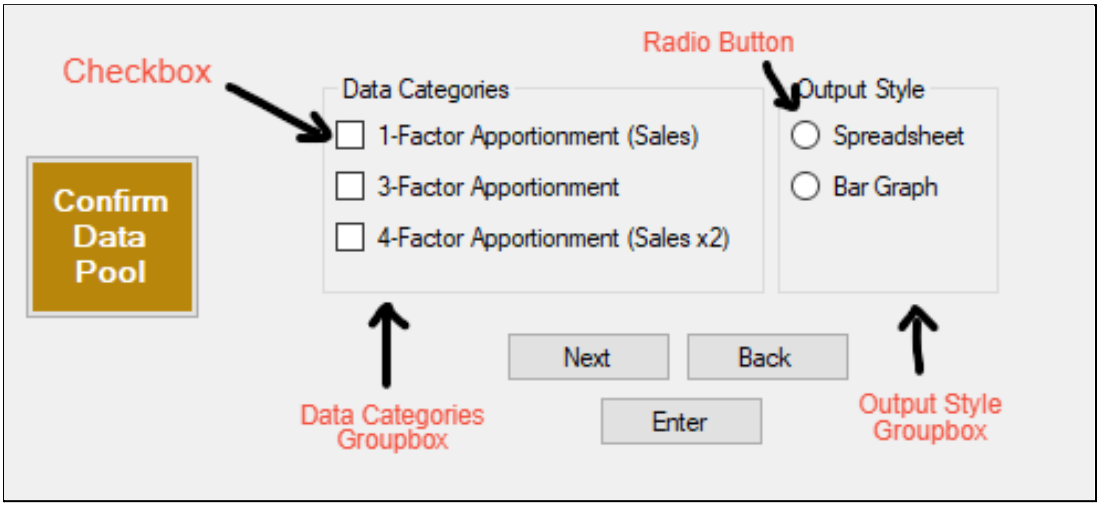
This technique of handling CSV files within the program is crucial for my client. Keith works with exclusively spreadsheet files in his profession, therefore all the unsorted data he receives is put in a spreadsheet. In order to reduce the amount of work my client has to do, utilizing the technique of file handling is crucial. By allowing my client to simply input particular files and then having the program handle it, he does not have to manually input data, which will save my client time and makes the program much easier to use.

Technique: Class Datapoint Array Building to store File Data

Two group boxes were added to Form 1 with the labels of "Data Categories" and "Output Style". Inside the Data Categories group box, three checkboxes were added for each of the three apportionment types my client uses. Inside the other group box, two radio buttons were added. There are radio buttons in the second because only one option should be selected for

output style. Multiple apportionments can be calculated at the same time if needed, therefore checkboxes are more suitable for the Data Categories one. *(*See Image Below)*

Program View:



Two buttons, Next and Back, were added in order to allow the client to change their choices or files quickly instead of exiting and restarting the program. An integer, nextCount, is defined to keep track of what selection stage the client is on. *(*See Images Below)*

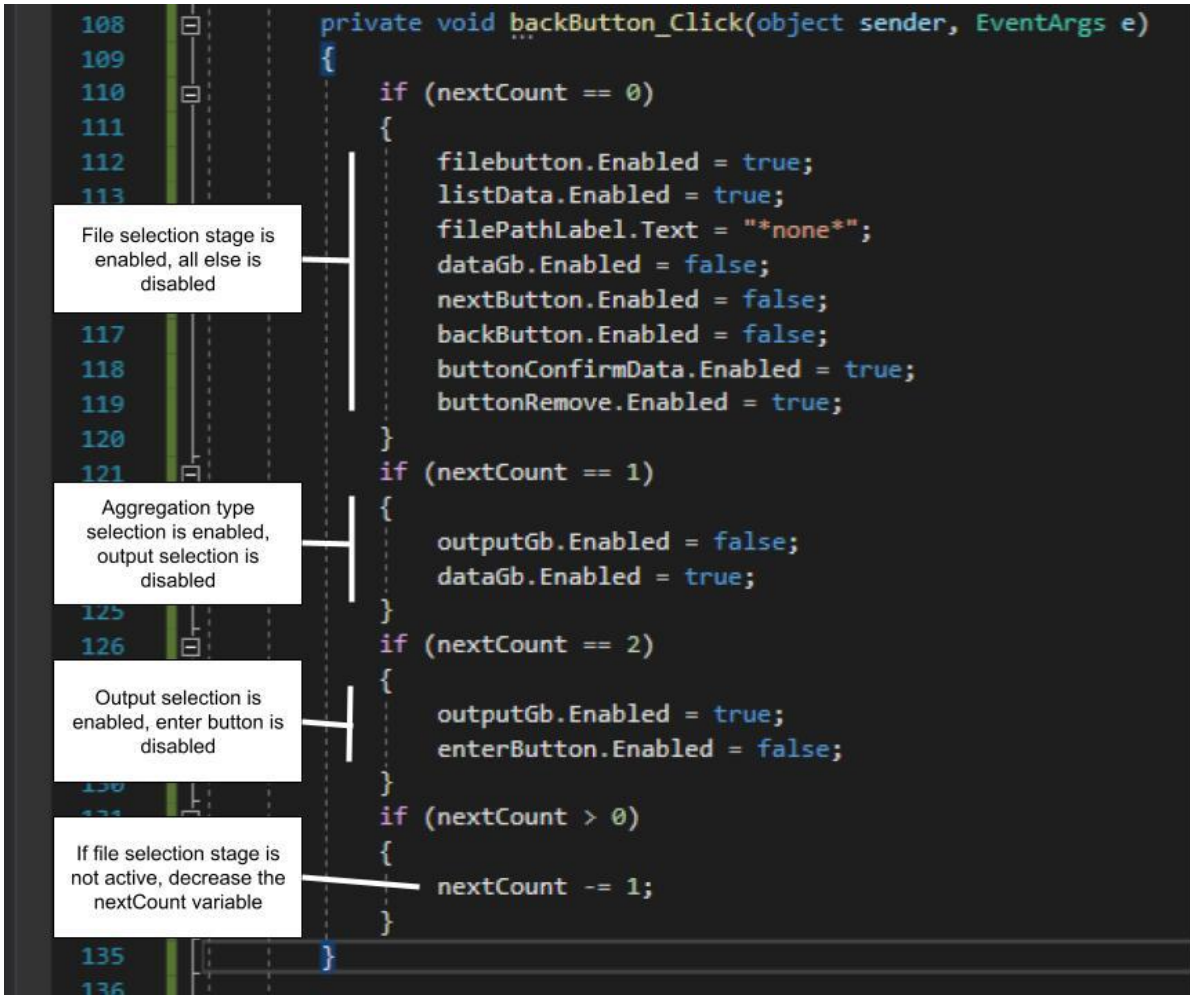
Program Code (Next Button):

```
private void nextButton_Click(object sender, EventArgs e)
{
    if (spreadRb.Checked == false && barRb.Checked == false && nextCount == 1)
    {
        System.Windows.Forms.MessageBox.Show("Please select an output type.");
    }
    else
    {
        if (nextCount == 1)
        {
            outputGb.Enabled = false;
            enterButton.Enabled = true;
            nextCount += 1;
        }
    }
    if (box1Factor.Checked == false && box3Factor.Checked == false && box4Factor.Checked == false && nextCount == 0)
    {
        System.Windows.Forms.MessageBox.Show("Please select at least one data category.");
    }
    else
    {
        if (nextCount == 0)
        {
            outputGb.Enabled = true;
            dataGb.Enabled = false;
            nextCount += 1;
        }
    }
}
```

If no option is selected for output type, request an output type. Else, enabled enter button and disable the output options.

If no option is selected for aggregation type, request at least one. Else, enabled output selection and disable aggregation type selection.

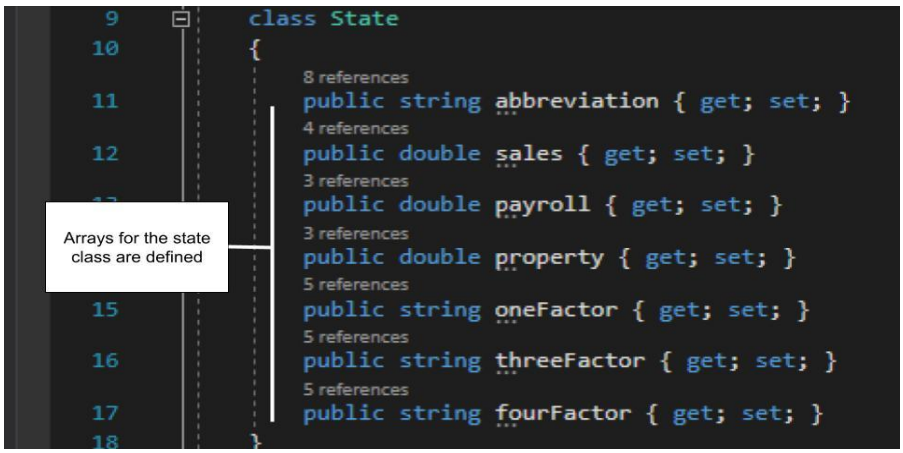
Program Code (Back Button):




```
private void backButton_Click(object sender, EventArgs e)
{
    if (nextCount == 0)
    {
        filebutton.Enabled = true;
        listData.Enabled = true;
        filePathLabel.Text = "*none*";
        dataGb.Enabled = false;
        nextButton.Enabled = false;
        backButton.Enabled = false;
        buttonConfirmData.Enabled = true;
        buttonRemove.Enabled = true;
    }
    if (nextCount == 1)
    {
        outputGb.Enabled = false;
        dataGb.Enabled = true;
    }
    if (nextCount == 2)
    {
        outputGb.Enabled = true;
        enterButton.Enabled = false;
    }
    if (nextCount > 0)
    {
        nextCount -= 1;
    }
}
```

Another button is also added to Form 1 with the label “Enter”. This button will create a data class of arrays of data needed from the files based on the selections of the client. A class called “State” was created to house arrays of data elements needed by the client for each State in the United States. *(*See Image Below)*

Program Code:



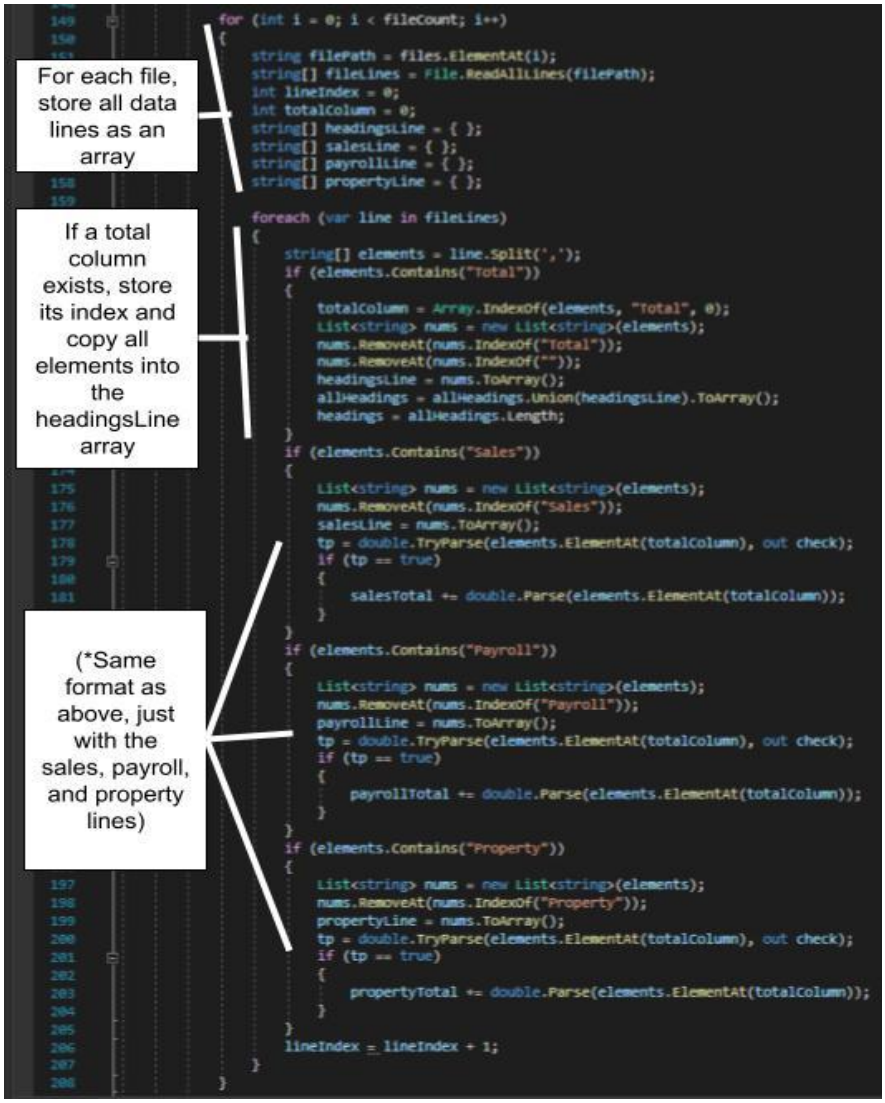

```
class State
{
    8 references
    public string abbreviation { get; set; }
    4 references
    public double sales { get; set; }
    3 references
    public double payroll { get; set; }
    3 references
    public double property { get; set; }
    5 references
    public string oneFactor { get; set; }
    5 references
    public string threeFactor { get; set; }
    5 references
    public string fourFactor { get; set; }
}
```

Creating a class of arrays of the data is crucial because the data in the spreadsheet needs to be redefined for the program in order for it to be usable since multiple spreadsheets need to be aggregated. A class of arrays is also the best technique for this purpose because foreach loops can be called that execute for each state in the class, which keeps the program simple and time-efficient.

Technique: Loops, If-Then, Else Conditions

Inside of the enterbutton_click function, loops will be used to create the arrays of data from the files. The first is a loop that executes based on the number of files. For each file, the lines are read and a foreach loop scans all lines in the file to create a running total of headings, sales, payroll, and property. Other loops are used to repeat processes, like creating sums of each category. *(*See Image Below)*

Program Code:




```
for (int i = 0; i < fileCount; i++)
{
    string filePath = files.ElementAt(i);
    string[] fileLines = File.ReadAllLines(filePath);
    int lineIndex = 0;
    int totalColumn = 0;
    string[] headingsLine = { };
    string[] salesLine = { };
    string[] payrollLine = { };
    string[] propertyLine = { };

    foreach (var line in fileLines)
    {
        string[] elements = line.Split(',');
        if (elements.Contains("Total"))
        {
            totalColumn = Array.IndexOf(elements, "Total", 0);
            List<string> nums = new List<string>(elements);
            nums.RemoveAt(nums.IndexOf("Total"));
            nums.RemoveAt(nums.IndexOf(""));
            headingsLine = nums.ToArray();
            allHeadings = allHeadings.Union(headingsLine).ToArray();
            headings = allHeadings.Length;
        }
        if (elements.Contains("Sales"))
        {
            List<string> nums = new List<string>(elements);
            nums.RemoveAt(nums.IndexOf("Sales"));
            salesLine = nums.ToArray();
            tp = double.TryParse(elements.ElementAt(totalColumn), out check);
            if (tp == true)
            {
                salesTotal += double.Parse(elements.ElementAt(totalColumn));
            }
        }
        if (elements.Contains("Payroll"))
        {
            List<string> nums = new List<string>(elements);
            nums.RemoveAt(nums.IndexOf("Payroll"));
            payrollLine = nums.ToArray();
            tp = double.TryParse(elements.ElementAt(totalColumn), out check);
            if (tp == true)
            {
                payrollTotal += double.Parse(elements.ElementAt(totalColumn));
            }
        }
        if (elements.Contains("Property"))
        {
            List<string> nums = new List<string>(elements);
            nums.RemoveAt(nums.IndexOf("Property"));
            propertyLine = nums.ToArray();
            tp = double.TryParse(elements.ElementAt(totalColumn), out check);
            if (tp == true)
            {
                propertyTotal += double.Parse(elements.ElementAt(totalColumn));
            }
        }
        lineIndex = lineIndex + 1;
    }
}
```

If-then statements are used to determine which apportionment types and which output style to perform based on the client selection. *(*See Image Below)*

Program Code:


```
if (spreadRb.Checked == true)
{
    foreach (State state in states)
    {
        spreadApportionment.Columns.Add(state.abbreviation, state.abbreviation);
        DataGridViewColumn column = spreadApportionment.Columns[state.abbreviation];
        column.SortMode = DataGridViewColumnSortMode.NotSortable;
        spreadApportionment.Visible = true;
    }
}
if (barRb.Checked == true)
{
    chartApportionment.Visible = true;
}
if (box1Factor.Checked)
{
    int i = 0;
    foreach (State state in states)
    {
        state.oneFactor = ((state.sales) / salesTotal).ToString("P");
        state.oneFactor = state.oneFactor.Remove(5);
        if (barRb.Checked == true)
        {
            chartApportionment.Series["1-Factor Apportionment"].Points.AddXY(state.abbreviation, state.oneFactor);
        }
        if (spreadRb.Checked == true)
        {
            string[] array = { state.oneFactor };
            array.CopyTo(row1, i);
        }
        i += 1;
    }
}
```

This technique of looping is crucial to saving time for my client. By making functions a loop, I am reducing the time it takes for the program to read and execute code, therefore I am saving valuable time for my client in his job. In addition, utilizing if-then statements saves my client time because the program can skip lines it does not need to execute.

Technique: Graphical User Interface Form Elements

The C# programming I am utilizing relies on objects, therefore the GUI of the program includes elements such as buttons, radio buttons, and list elements the client can click on. By utilizing a simple and interactive GUI, my client will have no trouble figuring out how to use the program. Another part of the GUI that I programmed was a custom bar graph and spreadsheet for output graphics. The settings for each are defined in the form designer, and the data from the State class is added programmatically in the enterbutton_click function. *(*See Images Below)*

Program Code:

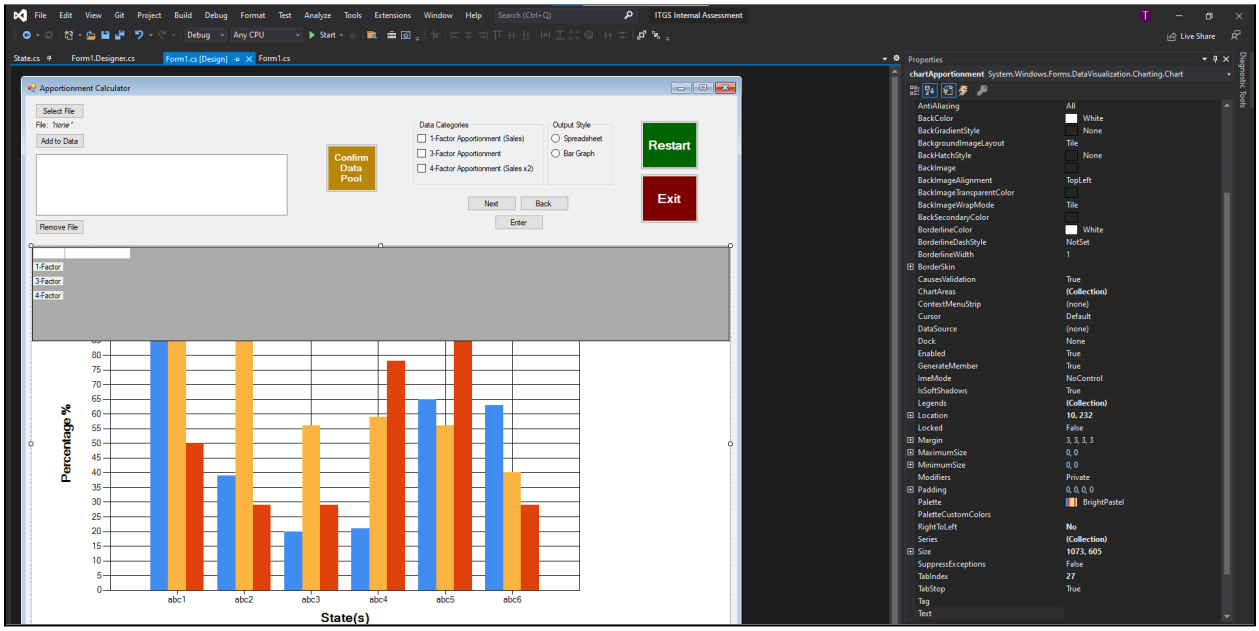
```
                if (barRb.Checked == true)
                {
                    chartApportionment.Series["4-Factor Apportionment"].Points.AddXY(state.abbreviation, state.fourFactor);
                }
                if (spreadRb.Checked == true)
                {
                    string[] array = { state.fourFactor };
                    array.CopyTo(row4, i);
                }
                i += 1;
            }
        }
        if (spreadRb.Checked == true)
        {
            spreadApportionment.Columns.Remove(Column1);
            spreadApportionment.Rows.Add(row1);
            spreadApportionment.Rows.Add(row3);
            spreadApportionment.Rows.Add(row4);
            sprlbl1.Visible = true;
            sprlbl3.Visible = true;
            sprlbl4.Visible = true;
        }
```

Add data point to bar graph

Copy data point to its row array

Add row arrays to the spreadsheet and format the spreadsheet correctly

Program GUI:



By having multiple graphical display styles for the final data, my client will be able to view the data however is most appealing or appropriate for him at the time. This element of the user interface is important because if my client cannot understand the data that the program sorts, then the program will not be saving him time in the long run.

Word Count: 1,067