Design Patterns

Command-Adapter Pattern

Collin Kemner

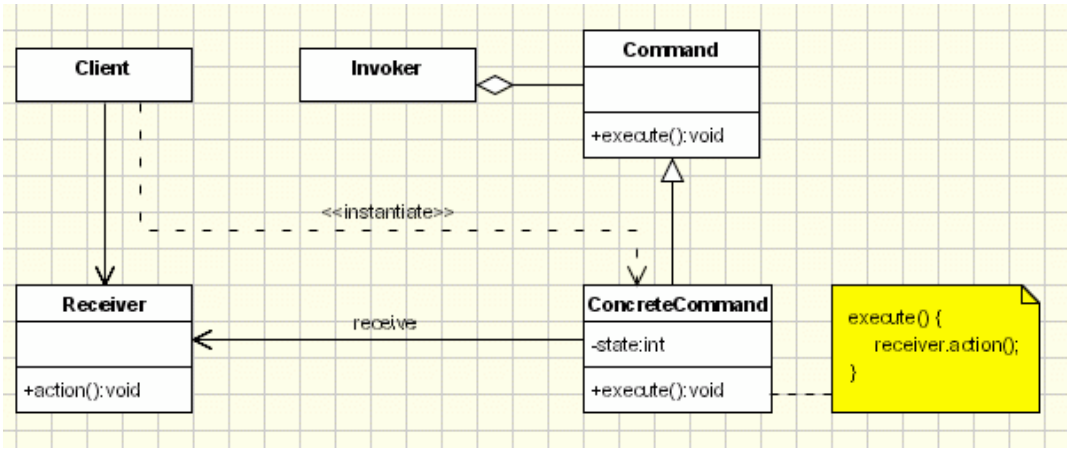
October 13th, 2016

Introduction

This assignment will showcase two patterns working alongside each other for one particular computer application. Those two patterns are the command pattern and the adapter pattern. According to oodesign.com, the command pattern "encapsulate[s] a request in an object, allows the parameterization of clients with different requests, and allows saving the requests in a queue" while the adapter pattern "convert[s] the interface of a class into another interface [the] client expects and it lets the classes work together that could not otherwise because of incompatible interfaces." In the case of the application for this assignment, the command pattern will hold commands for a calculator which will do work on two particular numbers (addition, subtraction, multiplication, and division.) The adapter pattern will then be able to take the double numbers and change them to integers as they are considered doubles as default.

Command Pattern UML Diagram

The UML diagram includes four distinct classes. The command class defines an interface which will execute an operation while the concrete command class extends the command class and implements the execute method by invoking the operations on the receiver class. The receiver class knows how to perform the operations needed to be performed and the invoker class is what initiates the request to be carried out. When the client needs a command to be executed, the invoker takes the



command and places it in the queue while the concrete command class then sends the result to the receiver class.

Command Pattern Code

Command Class

```
public abstract class Command
{
    protected Receiver _receiver = null;

    public Command(Receiver receiver)
    {
        _receiver = receiver;
    }
    public abstract double Execute();
}
```

The first class is the command class which is abstract. A "Command" constructor takes in the "Receiver" interface as a parameter. The "Execute" command is also defined underneath the constructor which will help execute the program.

Concrete Comands

```
    public class AddCommand : Command
    {
        public AddCommand(Receiver receiver)
            :base(receiver)
        {

        }
        public override double Execute()
        {
            _receiver.SetAction(MathActions.Add);
            return _receiver.GetResult();
        }
    }

    public class SubtractCommand : Command
    {
        public SubtractCommand(Receiver receiver)
            :base(receiver)
        {

        }
        public override double Execute()
        {
            _receiver.SetAction(MathActions.Subtract);
            return _receiver.GetResult();
        }
    }

    public class MultiplicationCommand : Command
    {
        public MultiplicationCommand(Receiver receiver)
            :base(receiver)
        {
```

These classes collectively are what make up the concrete class. Each mathematical operation has its own class and set of instructions. For example, the add command defines the execute method by setting an action to the "add action", which originates from the MathActions enum in the Receiver class. That action that has been received (in this case, the add action), is then returned. This process is done for the other three mathematical operations. These classes are called in the Client class to choose which operation that needs to be done on the two numbers.

```
        }
        public override double Execute()
        {
            _receiver.SetAction(MathActions.Multiply);
            return _receiver.GetResult();
        }
    }

    public class DivideCommand : Command
    {
        public DivideCommand(Receiver receiver)
            :base(receiver)
        {

        }
        public override double Execute()
        {
            _receiver.SetAction(MathActions.Divide);
            return _receiver.GetResult();
        }
    }
```

Receiver Interface and MathActions

```
public enum MathActions
    {
        Add,
        Subtract,
        Multiply,
        Divide
    }
    public interface Receiver
    {
        void SetAction(MathActions action);
        double GetResult();
    }
```

This is the file the contains the Receiver interface and the MathActions enum. The enum contains definitions for the four different mathematical operations. This enum is called whenever one of the operations needs to be defined or called. The Receiver interface defines the SetAction method which sets the mathematical operation that is called. The integer method GetResult is also defined, which is the method used in the Invoker class to conduct the mathematical operations.

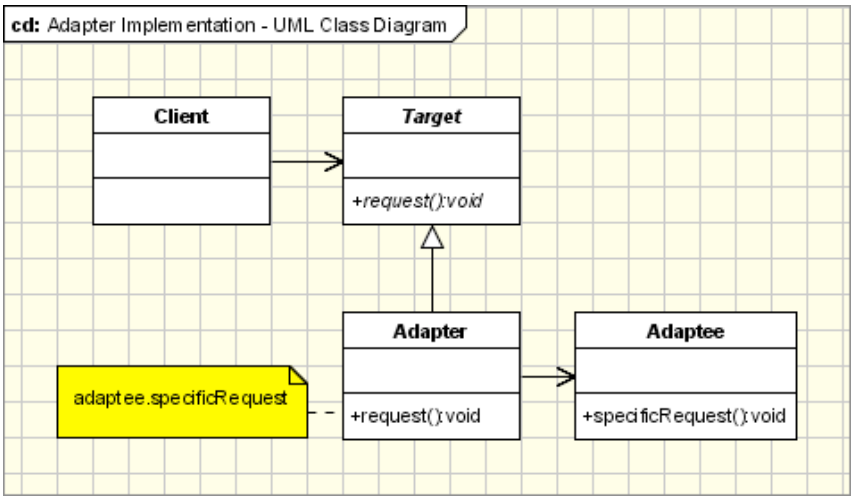
Invoker Class

```
public class Invoker : Receiver   //calculator
    {
        double _x;
        double _y;

        MathActions currentAction;

        public Invoker (double x, double y)   //construct
        {
            _x = x;
            _y = y;
        }

        public Invoker()
        {
            // TODO: Complete member initialization
        }

        public void SetAction(MathActions action)
        {
            currentAction = action;
        }

        public double GetResult()
        {
            double result;
            if(currentAction == MathActions.Add)
            {
                result = _x + _y;
            }
            else if (currentAction == MathActions.Subtract)
            {
                result = _x - _y;
            }
            else if (currentAction == MathActions.Multiply)
            {
                result = _x * _y;
            }
            else
            {
                result = _x / _y;
            }

            return result;
        }
    }
```

The invoker class extends the receiver class and sets the enum MathActions to the variable currentAction which is then used in the GetResult integer method. An if-then-else statement is used to determine which math operation to undertake. For example, if currentActions (the enum) is set to Add, addition is the operation that is chosen and two numbers (_x and _y) are then added together and the result is returned through the integer variable “result”. The Invoker method takes two doubles (x and y) from the Client and sets them equal to _x and _y respectively. This is what will carry the numbers from the Client, into the operation, and then ultimately saved as a result.

Adapter Pattern UML Diagram

The UML diagram for the Adapter pattern is pretty simple in comparison to the diagrams of other patterns. According to oodesign.com, the Target class defines a domain-specific interface that the client uses, the Adaptee defines an existing interface that needs adapting, and the Adapter adapts the interface Adaptee to the Target interface. In other words, the Adaptee intercepts a value (in this case, a double value) and sends it over to the Adapter, which uses the interface in the Target to do the needed conversion. After all of that, the Adapter returns the newly converted value to the Client.



Adapter Pattern Code

Adaptee Class

```
public class Adaptee
    {
        public int DoubleNumber1(double a)
        {
            int doublevalue = System.Convert.ToInt32(a);
            return doublevalue;
        }

    }
```

The Adaptee class is the class that intercepts the double value from an outside source. The double value that is passed into the "DoubleNumber1" method is then converted into an integer and then is set to the integer variable "doublevalue" and is then returned.

Target Interface

```
public interface Target
    {
        int Request(double i);
    }
```

The Target interface is the simplest piece of code in the whole application. It consists of an integer method, named “Request”, which takes a double value (that double value being the value that was passed into the Adaptee class).

Adapter Class

```
public class Adapter : Target
    {
        private Adaptee adaptee = new Adaptee();

        public int Request(double a)
        {
            int intvalue = adaptee.DoubleNumber1(a);
            return intvalue;
        }
    }
```

The Adapter class extends the Target interface in order to use the integer method. A private instance of the Adaptee is defined, which is used so that the “intvalue” integer variable can be set equal to the integer value from the “DoubleNumber1” method. That integer value is the value that was converted from double to integer. The “intvalue” value is then returned, which will be forwarded to wherever it is called.

The Client

```
public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void Form1_Load(object sender, EventArgs e)
        {

        }

        #region Commands, Adapter, Bools

        Receiver calculation = null;
        Command command = null;
        AddCommand addCommand = null;
        SubtractCommand subCommand = null;
        MultiplicationCommand multCommand = null;
        DivideCommand divCommand = null;

        Adapter first = new Adapter();

        private bool IntCheckChecked = false;
        private bool DoubleCheckChecked = false;

        private bool AddButtonPushed = false;
        private bool SubtractButtonPushed = false;
        private bool MultiplyButtonPushed = false;
        private bool DivideButtonPushed = false;

        #endregion
```

The Client's code is very long and extensive, so it will be explained part by part and not all at once. Starting off, the first bit of interesting code is the region of commands, the adapter instance, and a plethora of bools. The commands area are all instances of the mathematic commands and an instance from the Command class. That is what makes referencing the different commands possible. Next, there is an instance of the Adapter class. This is so that the Adapter class can be referenced and can be fed a double value so it can be converted to an integer. Then there are a number of bools set to false. The top two bools are for when a radio button that tells which number type to use is checked. The other four bools are to help determine when a specific mathematical operation button is pushed on the application.

```
#region Integer/Double Decision
private void m_rbIntegers_CheckedChanged(object sender, EventArgs e)
{
    IntCheckChecked = true;
    DoubleCheckChecked = false;
}

private void m_rbDoubles_CheckedChanged(object sender, EventArgs e)
{
    string s_x = m_tbNumber1.Text;
    double x = StringToDouble.Conversion(s_x.ToString());
    string s_y = m_tbNumber2.Text;
    double y = StringToDouble.Conversion(s_y.ToString());

    calculation = new Invoker(x, y);

    addCommand = new AddCommand(calculation);
    subCommand = new SubtractCommand(calculation);
    multCommand = new MultiplicationCommand(calculation);
    divCommand = new DivideCommand(calculation);

    DoubleCheckChecked = true;
    IntCheckChecked = false;
}
#endregion
```

The next region is where the press of one of the radio buttons determines which type of value to use. When the integer radio button is checked, the bool for checking if the integer radio button has been checked (IntCheckChecked) is set to true. The checking bool for the double value's radio button (DoubleCheckChecked) is set to false so there is no possible way that switching between the two value types will glitch the program out in any way. That is all the code for the integer radio button, but the reason for there being little code in that section will be explained later on.

Lastly for this area is the event for when the double radio button is selected. As double values are the default of the application, this is where part of the command pattern takes place. First, the two values that will have done on them are converted from strings to type double (saved respectively to variables "x" and "y"). Then, through the Receiver (its instance is called "calculation"), the x and y values are sent into the Invoker class. The instances of all four of the mathematical operation's classes are set to names that apply to the Client's class (so that they can be referenced and have work done in them). Also, like the integer radio button event, the radio button bool variables are mentioned, although the bool for the double button is set to true and the bool for the integer button is set to false.

```
#region Click Events For Commands
private void m_btnAdd_Click(object sender, EventArgs e)
{
    command = addCommand;
    AddButtonPushed = true;

    SubtractButtonPushed = false;
    MultiplyButtonPushed = false;
    DivideButtonPushed = false;
}

private void m_btnSubtract_Click(object sender, EventArgs e)
{
    command = subCommand;
    SubtractButtonPushed = true;

    AddButtonPushed = false;
    MultiplyButtonPushed = false;
    DivideButtonPushed = false;
}

private void m_btnMultiply_Click(object sender, EventArgs e)
{
    command = multCommand;
    MultiplyButtonPushed = true;

    AddButtonPushed = false;
    SubtractButtonPushed = false;
    DivideButtonPushed = false;
}

private void m_btnDivide_Click(object sender, EventArgs e)
{
    command = divCommand;
    DivideButtonPushed = true;

    AddButtonPushed = false;
    SubtractButtonPushed = false;
    MultiplyButtonPushed = false;
}
#endregion
```

The next area of the Client is where the Command pattern is actually set into motion. All four of the mathematical operations have their own button. When, say, the add button is pressed, the command from the Command class is set to the "addCommand" concrete command class. That is what points out which command to use, in this case, the command which undertakes an addition operation. After that is complete, the "AddButtonPushed" bool is set to true and the other three bools are set to false. The purpose of the bools are to help the Adapter pattern work, which the next region on the Client will show.

```
    #region Calculation Click Command

    private void m_btnDoCalculation_Click(object sender, EventArgs e)
    {
        if (IntCheckChecked == true)
        {
            string s_x = m_tbNumber1.Text;
            double x = StringToDouble.Conversion(s_x.ToString());
            string s_y = m_tbNumber2.Text;
            double y = StringToDouble.Conversion(s_y.ToString());

            if (AddButtonPushed == true)
            {
                double EndResultDouble = x + y;
                int i_ER = first.Request(EndResultDouble);
                string s_ER = i_ER.ToString();
                m_tbFinalNumber.Text = s_ER;
            }
            if (SubtractButtonPushed == true)
            {
                double EndResultDouble = x - y;
                int i_ER = first.Request(EndResultDouble);
                string s_ER = i_ER.ToString();
                m_tbFinalNumber.Text = s_ER;
            }
            if (MultiplyButtonPushed == true)
            {
                double EndResultDouble = x * y;
                int i_ER = first.Request(EndResultDouble);
                string s_ER = i_ER.ToString();
                m_tbFinalNumber.Text = s_ER;
            }
            if (DivideButtonPushed == true)
            {
                double EndResultDouble = x / y;
                int i_ER = first.Request(EndResultDouble);
                string s_ER = i_ER.ToString();
                m_tbFinalNumber.Text = s_ER;
            }
        }
        else if (DoubleCheckChecked == true)
        {
            m_tbFinalNumber.Text = command.Execute().ToString("0.00");
        }
    }
    #endregion
}
```

This last area of the Client is the event for when the “DoCalculaton” button is pressed, which executes the Command pattern and also helps execute the Adapter pattern. First off, there is an if statement which has a parameter for if “IntCheckChecked” is true, or in other words, if the radio button for type int has been selected. If it is selected, then the numbers from the text boxes on the application are set to doubles. Then, there are four if statements corresponding to the four mathematical operations and which button was pressed (which is decided by which bool value is set to true). If the “AddButtonPushed” bool is set to true, the sum of the x and y values are set to a single double value. That value is then converted to int by sending it to the Adapter (by using the line “first.Request(EndResultDouble)”).

Through the Adapter, Adaptee, and Target interface, the double value is converted to type integer and is subsequently converted again to type string (s_ER). Then, finally, the string value is then sent into the final number text box (m_tbFinalNumber). If the double radio button is checked, there is simply one line that sends the value to the text box. That is because it utilizes the Command pattern and all the work was already done. So, the “Execute()” method is used to fire the command and the decimal points for the double value is set to two decimals.

StringToDouble class

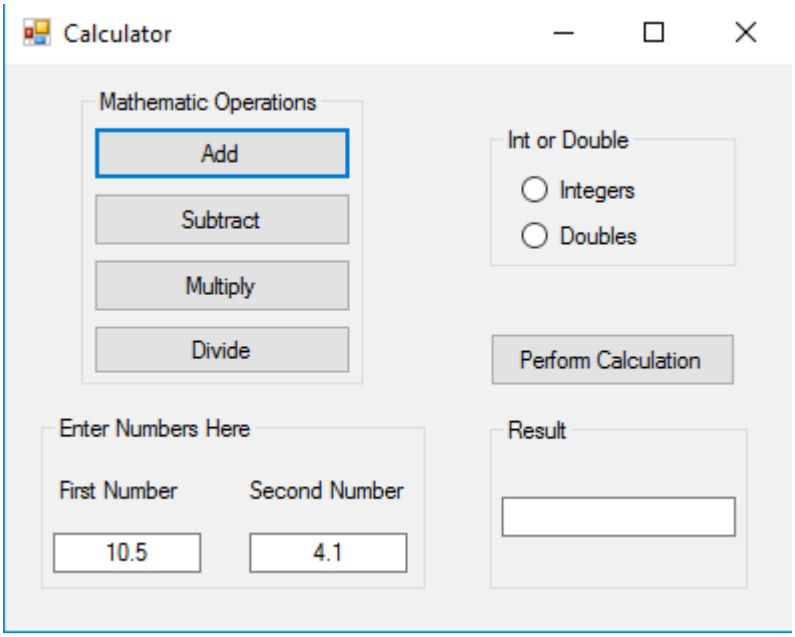
```
public static class StringToDouble
    {
        public static double Conversion(string Value)
        {
            if (Value == null)
            {
                return 0;
            }
            else
            {
                double OutVal;
                double.TryParse(Value, out OutVal);

                if (double.IsNaN(OutVal) || double.IsInfinity(OutVal))
                {
                    return 0;
                }
                return OutVal;
            }
        }

    }
```

This class is what converts the “x” and “y” numbers from the Client from strings to double. The static double takes in a string value (the x or y value) and checks to see if there is actually a string that was put through. If there is, then it is converted, set to “OutVal” and returned. When the Client calls the class, the doubles are grabbed from this class.

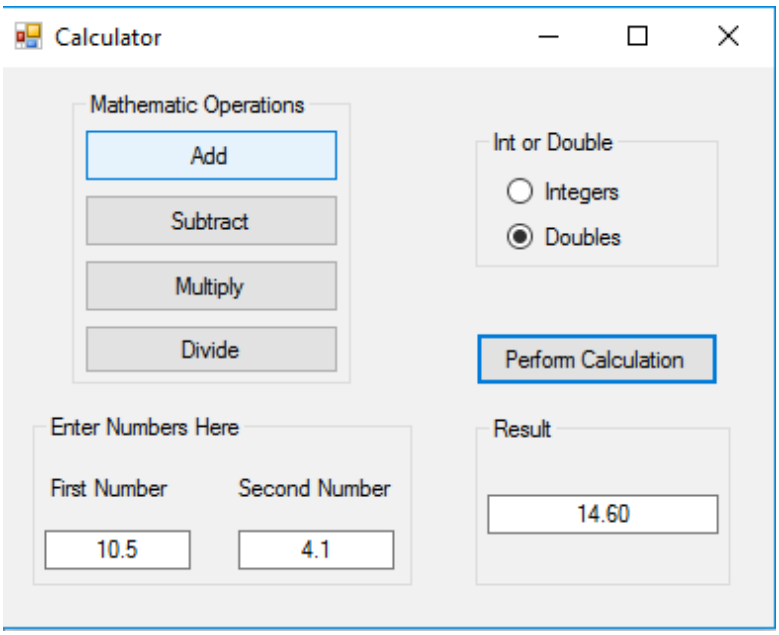
Screenshots

The following images are images of the application working to its fullest potential:

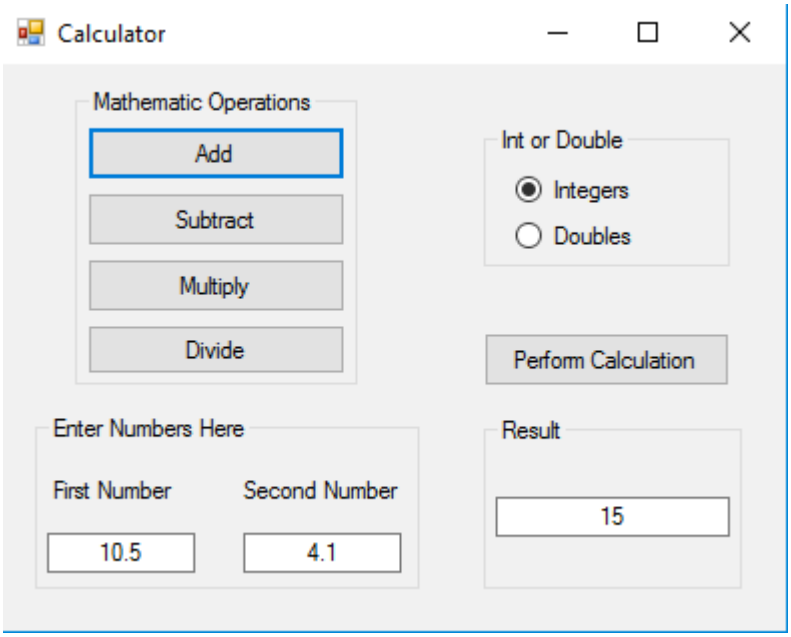
The Idle Application

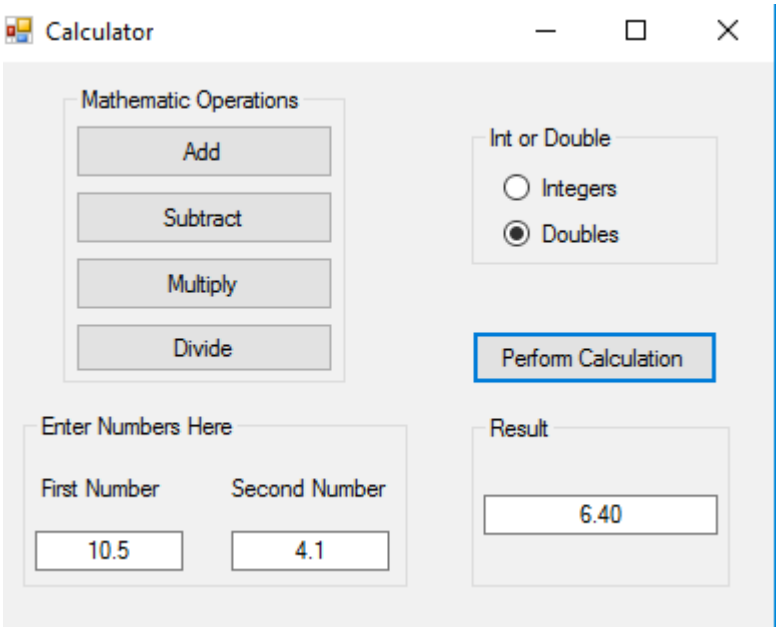


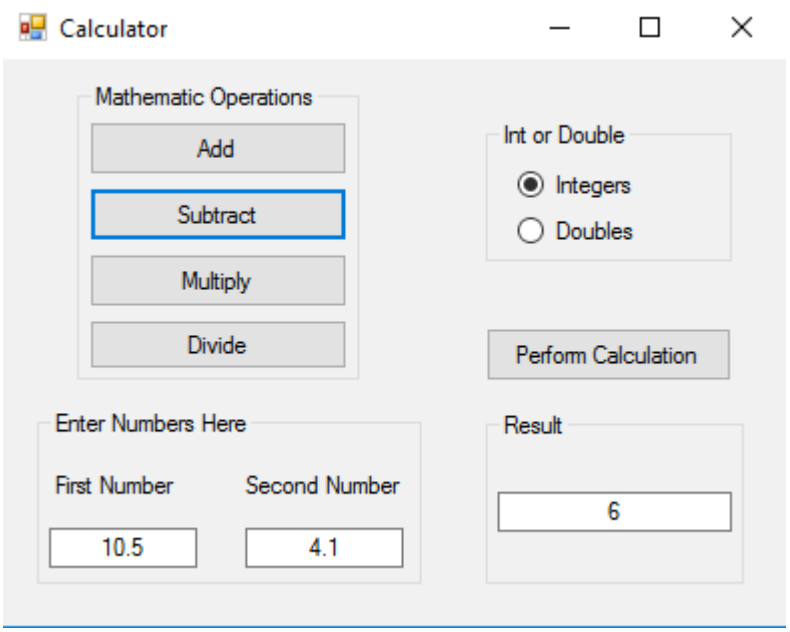
Selected to Double

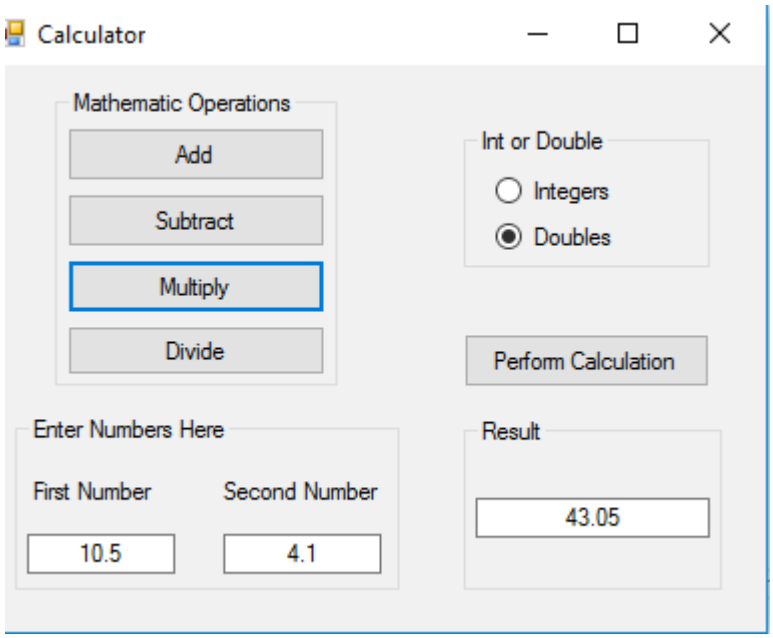


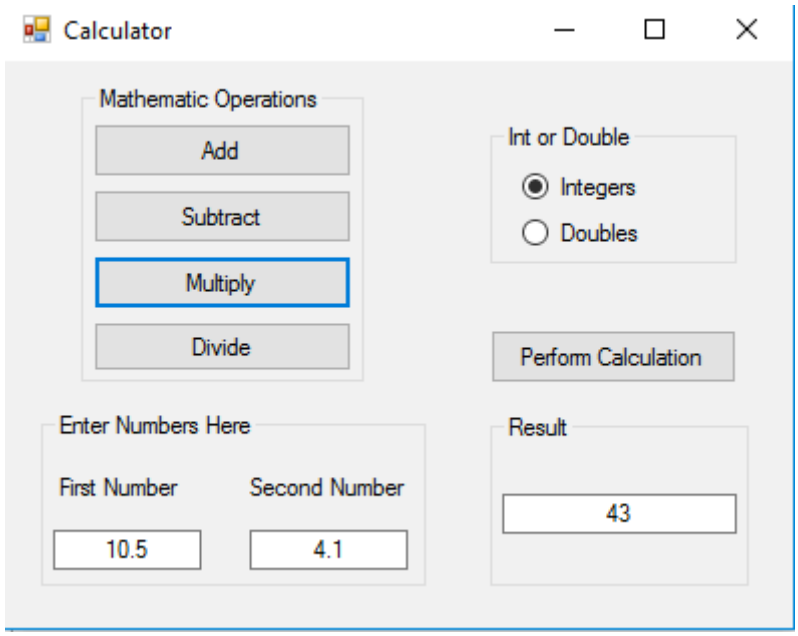
Selected to Integer

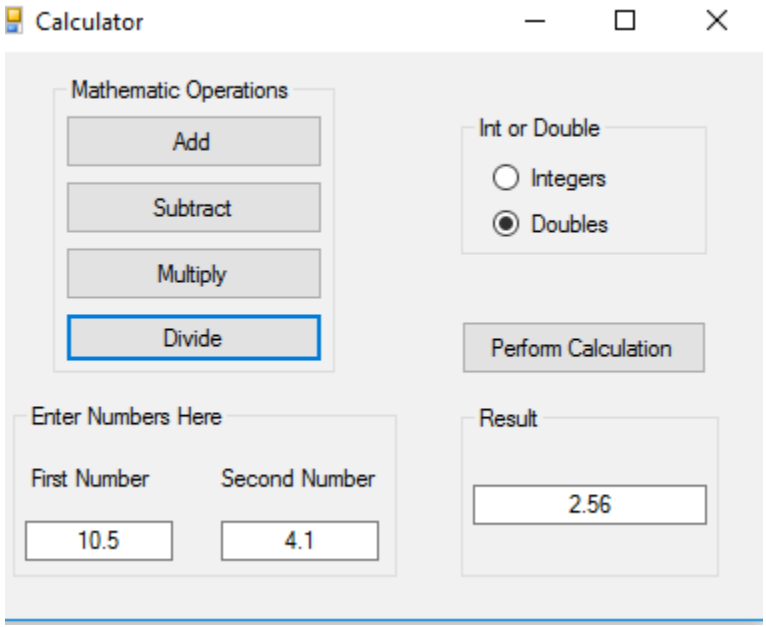


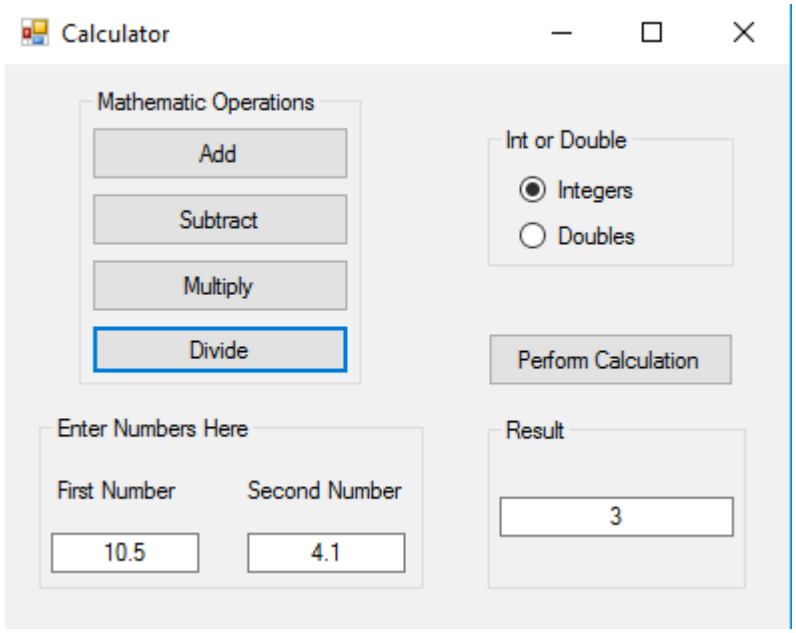





Calculator
Mathematic Operations
Add
Subtract
Multiply
Divide
Int or Double
Integers
Doubles
Perform Calculation
Enter Numbers Here
First Number
Second Number
10.5
4.1
Result
43.05


Calculator
Mathematic Operations
Add
Subtract
Multiply
Divide
Int or Double
Integers
Doubles
Perform Calculation
Enter Numbers Here
First Number
Second Number
10.5
4.1
Result
43


Calculator
Mathematic Operations
Add
Subtract
Multiply
Divide
Int or Double
Integers
Doubles
Perform Calculation
Enter Numbers Here
First Number
Second Number
10.5
4.1
Result
2.56


Calculator
Mathematic Operations
Add
Subtract
Multiply
Divide
Int or Double
Integers
Doubles
Perform Calculation
Enter Numbers Here
First Number
Second Number
10.5
4.1
Result
3

Conclusion/Observations

Overall, even though I did struggle a bit with this pattern, I did enjoy writing it; especially since there were a few times where I had to rely on my intuition as a beginner programmer and it ended up paying off. I feel that now that I did this big program, I can go on to understand programming in C# better than I ever have. I feel a lot more confident in my abilities to program than I ever have before. I feel that I understand how both of these patterns work and that I demonstrated them just fine on this application. I do apologize for turning this assignment in late as I was having some difficulties with the Adapter pattern. I also started out trying to use type int as the default type when I should've started out using type double as the default. When a value, even with decimals, is set to int, those decimals are automatically chopped off and do not represent a change to type double well, so I had to change up some of my code so that type double was the default. I'm not entirely disappointed that I turned it in late as I ended up learning more during this process.