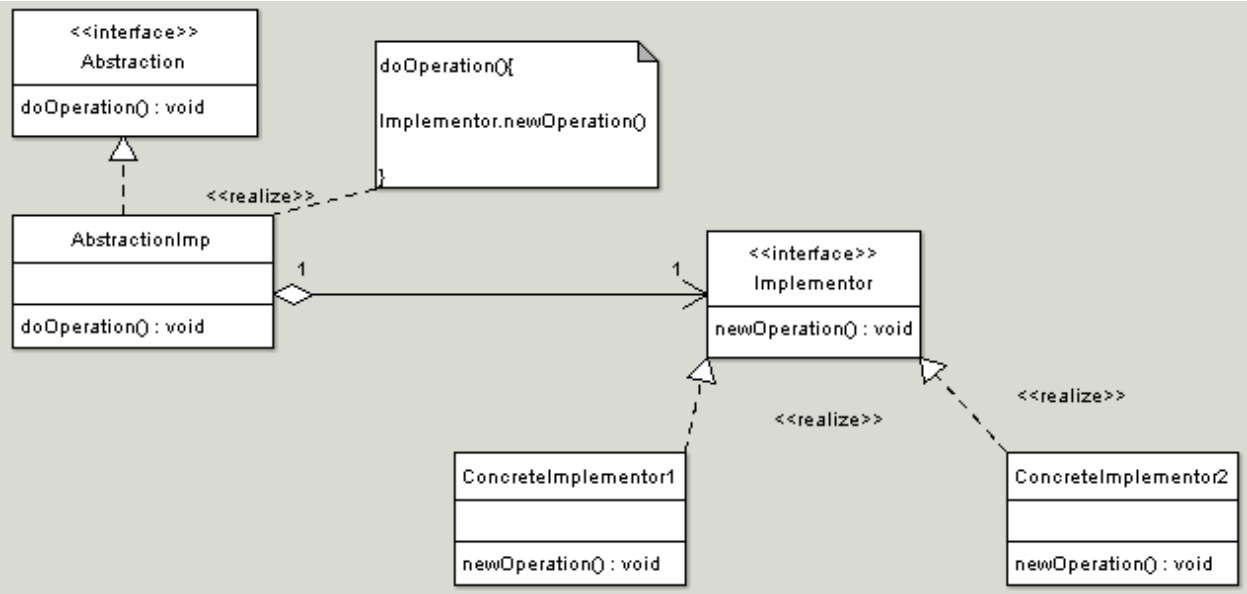
The Bridge Pattern
Design Patterns
Andreya Grzegorzewski – Ohio Northern University
Fall 2016-2017

Introduction

This assignment requires an application that makes use of the bridge pattern. This pattern is intended to decouple the abstraction from the implementation so that one can change without affecting the other. My app, inspired by my Electric Circuits class, calculates the impedance of two elements in series or in parallel. The user provides two complex numbers in polar or rectangular form, selects whether they are in series or in parallel, and clicks the “Calculate” button. The user can also select whether the result should be displayed in polar or rectangular form. Then, the result appears in a text box.

The UML Diagram



Above is the UML diagram for the bridge pattern, courtesy of oodesign.com. The pattern begins with an Abstraction interface, which allows for several different implementations of the abstraction. One implementation is shown with the AbstractionImp class. The AbstractionImp class holds a pointer to an Implementer object, which can be initialized as a ConcreteImplementor1 or a ConcreteImplementor2. AbstractionImp calls methods from the Implementor interface, which are implemented in different ways in each of the ConcreteImplementor classes. On the next page is a table displaying the classes I used to implement this pattern.

Abstraction	My Abstraction interface is called ImpedanceMath. It provides prototypes for the operations that will be shown in the form: it has one signature for getParallelImpedance and another for getSeriesImpedance.
AbstractionImp	The Form1 class serves as my AbstractionImp class. It defines the methods prototyped in the ImpedanceMath interface.
Implementor	My Implementor interface is called ComplexNumber. It contains prototypes for the basic mathematical operations (add, subtract, multiply, and divide). It also contains two methods for conversion between polar and rectangular forms.
ConcreteImplementor1	The first ConcreteImplementor class is called Rectangular. Its mathematical operation methods return Rectangular numbers – that is, numbers in the form of x + yj, where x is real and y is imaginary.
ConcreteImplementor2	My second ConcreteImplementor class is called Polar. Its methods return Polar numbers – that is, numbers in the form of r<Θ, where r is the magnitude of the number and theta is the angle in degrees. These two implementations of ComplexNumber are consistent with the complex numbers used in my circuits class.

Narrative and Code

I started writing my code by defining the ComplexNumber abstract class, which serves as the Implementor interface described in the UML diagram. This class contains four abstract methods, called add, subtract, multiply, and divide.

```
public abstract class ComplexNumber // This is the Implementor interface
{
    public abstract ComplexNumber add(ComplexNumber num1, ComplexNumber num2);
    public abstract ComplexNumber subtract(ComplexNumber num1, ComplexNumber num2);
    public abstract ComplexNumber multiply(ComplexNumber num1, ComplexNumber num2);
    public abstract ComplexNumber divide(ComplexNumber num1, ComplexNumber num2);
```

Then, I created the Rectangular and Polar classes, which implement the four methods described above as well as some of their own methods. I started with Rectangular by declaring the class, two variables, and a constructor:

```
public class Rectangular : ComplexNumber // This is a concrete implementor
{
    double real;
    double imaginary;

    public Rectangular(double real, double imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
```

```
        }
```

Next, I provided getters for the real and imaginary variables and overrode the ToString method.

```
public double getReal()
{
    return real;
}

public double getImaginary()
{
    return imaginary;
}


public override string ToString()
{
    string result = "";

    // If we have a real number...
    if (real != 0)
    {
        result += real.ToString("0.##"); // use "0.##" formatting for two decimal places

        // Handle negative and positive formatting of the imaginary component
        if (imaginary < 0)
            result += " - " + (imaginary * -1).ToString("0.##") + "j";
        else if (imaginary > 0)
            result += " + " + imaginary.ToString("0.##") + "j";
        return result;
    }

    // If we have an imaginary component but not a real component...
    else if (imaginary != 0)
        result += imaginary.ToString("0.##") + "j";

    // Otherwise, the number is 0
    if (result == "")
        result = "0";

    return result;
}
```

Then, I made a function called cast that takes a ComplexNumber object and returns a Rectangular object in its place. In order to do this, I implemented a method called convertToRectangular in the ComplexNumber class in case I needed to use this method elsewhere. These two methods are both shown below.

```
        public Rectangular convertToRectangular(Polar p)
        {
            double real = p.getR() * Math.Cos(p.getTheta() / 180 * Math.PI);
            double imaginary = p.getR() * Math.Sin(p.getTheta() / 180 * Math.PI);
            return new Rectangular(real, imaginary);
```

```
}

private Rectangular cast(ComplexNumber num)
{
    if (num is Polar)
        return convertToRectangular((Polar)num);
    else return (Rectangular)num;
}
```

Finally, I implemented the methods defined in the ComplexNumber class. I started with add and subtract:

```
public override ComplexNumber add(ComplexNumber num1, ComplexNumber num2)
{
    Rectangular rect1 = cast(num1);
    Rectangular rect2 = cast(num2);

    double resultReal = rect1.real + rect2.real;
    double resultImaginary = rect1.imaginary + rect2.imaginary;

    return new Rectangular(resultReal, resultImaginary);
}

public override ComplexNumber subtract(ComplexNumber num1, ComplexNumber num2)
{
    Rectangular rect1 = cast(num1);
    Rectangular rect2 = cast(num2);

    double resultReal = rect1.real - rect2.real;
    double resultImaginary = rect1.imaginary - rect2.imaginary;

    return new Rectangular(resultReal, resultImaginary);
}
```

Then I implemented multiply, which uses the “foil” strategy often used to multiply two binomials.

```
public override ComplexNumber multiply(ComplexNumber num1, ComplexNumber num2)
{
    Rectangular rect1 = cast(num1);
    Rectangular rect2 = cast(num2);

    double f = rect1.real * rect2.real;
    double o = rect1.real * rect2.imaginary;
    double i = rect1.imaginary * rect2.real;
    double l = rect1.imaginary * rect2.imaginary;

    return new Rectangular(f - l, o + i);
}
```

There is no algorithm defined for dividing complex numbers in rectangular form, so I had to call the Polar class’s divide method.

```
        public override ComplexNumber divide(ComplexNumber num1, ComplexNumber num2)
        {
            ComplexNumber p = new Polar(0, 0);
            Polar num = (Polar)p.divide(num1, num2);
            return cast(num);
        }
```

With my Rectangular class completely implemented, I moved on to Polar. I started by again creating two variables and a constructor.

```
public class Polar : ComplexNumber // this is a concrete implementor
    {
        double r; // The magnitude of the number as represented by a line
        double theta; // The angle the line makes with the positive x axis

        public Polar(double r, double theta)
        {
            this.r = r;

            while (theta > 180)
                theta -= 360;
            while (theta < -180)
                theta += 360;

            this.theta = theta;
        }
```

Next, I created two getters and overrode ToString:

```
        public override string ToString()
        {
            if (r == 0)
                return "0";

            return r.ToString("0.##") + " <" + theta.ToString("0.##");
        }

        public double getR()
        {
            return r;
        }

        public double getTheta()
        {
            return theta;
        }
```

Anticipating the need for a convertToPolar method in ComplexNumber and a cast method in Polar, I created both.

```
        public Polar convertToPolar(Rectangular rect)
        {
            double real = rect.getReal();
            double imag = rect.getImaginary();
```

```
        double r = Math.Sqrt(real * real + imag * imag);
        double theta = Math.Atan(imag / real) * 180 / Math.PI;

        return new Polar(r, theta);
    }

    private Polar cast(ComplexNumber num)
    {
        if (num is Polar)
            return (Polar)num;
        else return convertToPolar((Rectangular)num);
    }
```

Finally, I moved on to implementing the abstract methods. I started with multiply and divide, which are very simple with polar numbers.

```
    public override ComplexNumber multiply(ComplexNumber num1, ComplexNumber num2)
    {
        Polar p1 = cast(num1);
        Polar p2 = cast(num2);

        double r = p1.r * p2.r;
        double theta = p1.theta + p2.theta;

        // Want theta in the range of [-180, 180]
        while (theta > 180)
            theta -= 360;

        while (theta < -180)
            theta += 360;

        return new Polar(r, theta);
    }

    public override ComplexNumber divide(ComplexNumber num1, ComplexNumber num2)
    {
        Polar p1 = cast(num1);
        Polar p2 = cast(num2);

        double theta = p1.theta - p2.theta;
        double r = p1.r / p2.r;

        while (theta > 180)
            theta -= 360;

        while (theta < -180)
            theta += 360;

        return new Polar(r, theta);
    }
```

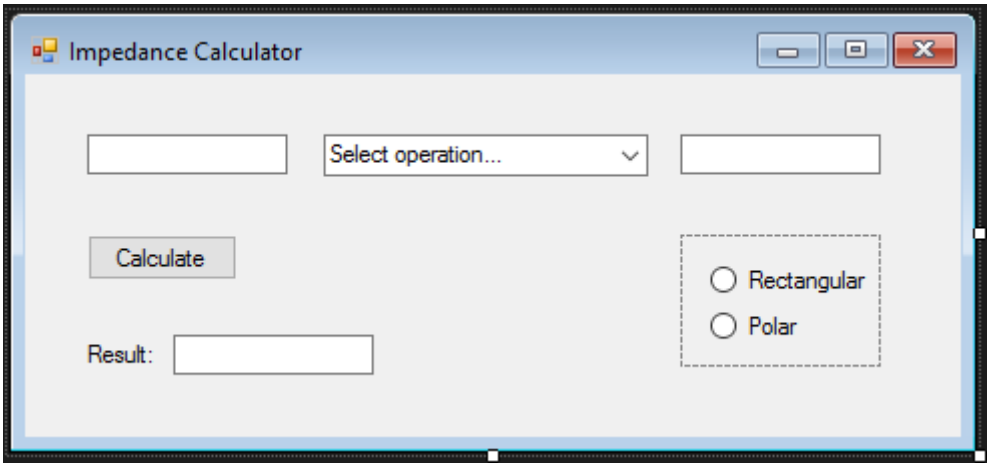
As there are no methods defined for adding and subtracting polar numbers, I had to call the Rectangular add and subtract methods and cast the result for the remaining two methods.

```
public override ComplexNumber add(ComplexNumber num1, ComplexNumber num2)
{
    ComplexNumber r = new Rectangular(0, 0);
    Rectangular num = (Rectangular)r.add(num1, num2);
    return cast(num);
}

public override ComplexNumber subtract(ComplexNumber num1, ComplexNumber num2)
{
    ComplexNumber r = new Rectangular(0, 0);
    Rectangular num = (Rectangular)r.subtract(num1, num2);
    return cast(num);
}
```

This is all the code for the Implementor portion of my bridge pattern. With this done, I moved on to the Abstraction portion. I started by creating my Abstraction interface, which is called ImpedanceMath. It provides two prototypes for methods that find impedance.

```
interface ImpedanceMath // This is the Abstraction interface
{
    ComplexNumber getParallelImpedance(ComplexNumber num1, ComplexNumber num2);
    ComplexNumber getSeriesImpedance(ComplexNumber num1, ComplexNumber num2);
}
```

Finally, I moved on to my form, which serves as the abstraction implementation. I started by designing it. A screenshot of my form from the design view of VisualStudio is shown below.



In the left and right text boxes, the user inputs a complex number of whatever form he chooses. The user then selects whether the equivalent impedance should be found as though those two elements are in series or in parallel. When the user clicks the calculate button, the resulting number is displayed in the bottom text box. The user can choose whether the result should be displayed in rectangular or polar form with the radio buttons on the right side of the form.

With my form designed, I started writing code for the form. I began by creating a pointer to a ComplexNumber object and initializing it to a Rectangular object. Because I did this, I marked the Rectangular radio button as checked. I also added two string items to my combo box.

```
public partial class Form1 : Form, ImpedanceMath // this is the abstraction
   implementation
{
    ComplexNumber cn;
    public Form1()
    {
        InitializeComponent();

        rectangularRB.Checked = true;
        cn = new Rectangular(0, 0);
        operationCB.Items.Add("is in series with");
        operationCB.Items.Add("is in parallel with");
    }
```

Next, I added methods to handle when the user clicks a radio button. This simply overwrites the object currently held in cn.

```
    private void rectangularRB_CheckedChanged(object sender, EventArgs e)
    {
        cn = new Rectangular(0, 0);
    }

    private void polarRB_CheckedChanged(object sender, EventArgs e)
    {
        cn = new Polar(0, 0);
    }
```

Next, I created a very long and ugly method called parseTbText that takes a string representation of complex number and returns the appropriate Rectangular or Polar object. The method is as follows.

```
private ComplexNumber parseTbText(string text)
{
    char[] textArray = text.ToCharArray();

    // If the number is polar...
```

```
if (text.Contains("<"))
{
    // Find the end index of r
    int stopIndexOfR = 0;
    for (int i = 0; i < text.Length; i++)
    {
         if (textArray[i] == '<' || textArray[i] == ' ')
         {
             stopIndexOfR = i - 1;
             break;
         }
    }

    // Create r by using parse and substring
    double r = Double.Parse(text.Substring(0, stopIndexOfR + 1));

    // Find the start index of theta
    int startIndexOfTheta = text.Length;
    for (int i = stopIndexOfR + 1; i < text.Length; i++)
    {
        if (textArray[i] != '<' && textArray[i] != ' ')
        {
            startIndexOfTheta = i;
            break;
        }
    }

    // Create theta
    double theta = Double.Parse(text.Substring(startIndexOfTheta));

    // Create and return a Polar object with r and theta
    Polar result = new Polar(r, theta);
    return result;
}

// Otherwise, the number is in rectangular form
else
{
    double real = 0;
    double imaginary = 0;

    // If the number doesn't have a j in it, it's just a real number
    if (!text.Contains("j"))
        real = Double.Parse(text);

    // If the text doesn't have a space and does contain a j, it's just an imaginary
      component
    else if (!text.Contains(" "))
        imaginary = Double.Parse(text.Substring(0, text.Length - 1));

    // Otherwise, we have real and imaginary.
    else
    {
        // Find the stop index of the real portion
        int stopIndexOfReal = 0;
        for (int i = 0; i < text.Length; i++)
        {
             if (textArray[i] == ' ' || textArray[i] == '+' || textArray[i] == '-')
```

```
                {
                    stopIndexOfReal = i;
                    break;
                }
            }

            // Create the real number
            real = Double.Parse(text.Substring(0, stopIndexOfReal + 1));

            // Find the start index of the imaginary component
            if (text.Contains("j"))
            {
                int startIndexOfImaginary = -1;
                for (int i = stopIndexOfReal; i < text.Length; i++)
                {
                    if(textArray[i] != ' ' && textArray[i] != '+' && textArray[i] != ' ')
                    {
                        startIndexOfImaginary = i;
                        break;
                    }
                }

                // Create the imaginary component
                if (startIndexOfImaginary != -1)
                    imaginary = Double.Parse(text.Substring(startIndexOfImaginary,
                      text.Length - startIndexOfImaginary - 1));

                // Check to see if the imaginary part should be negative
                if (text.Substring(1, text.Length - 1).Contains("-"))
                    imaginary *= -1;
            }
        }

        // Create and return the Rectangular object
        Rectangular result = new Rectangular(real, imaginary);
        return result;
    }
}
```

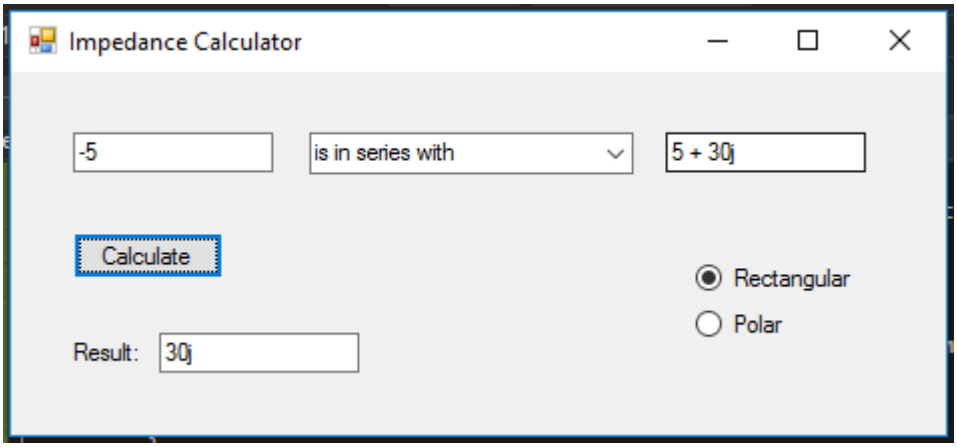
I then implemented the methods from the ImpedanceMath class.

```
        public ComplexNumber getParallelImpedance(ComplexNumber num1, ComplexNumber num2)
        {
            ComplexNumber numerator = cn.multiply(num1, num2);
            ComplexNumber denominator = cn.add(num1, num2);
            ComplexNumber result = cn.divide(numerator, denominator);
            return result;
        }

        public ComplexNumber getSeriesImpedance(ComplexNumber num1, ComplexNumber num2)
        {
            return cn.add(num1, num2);
        }
```

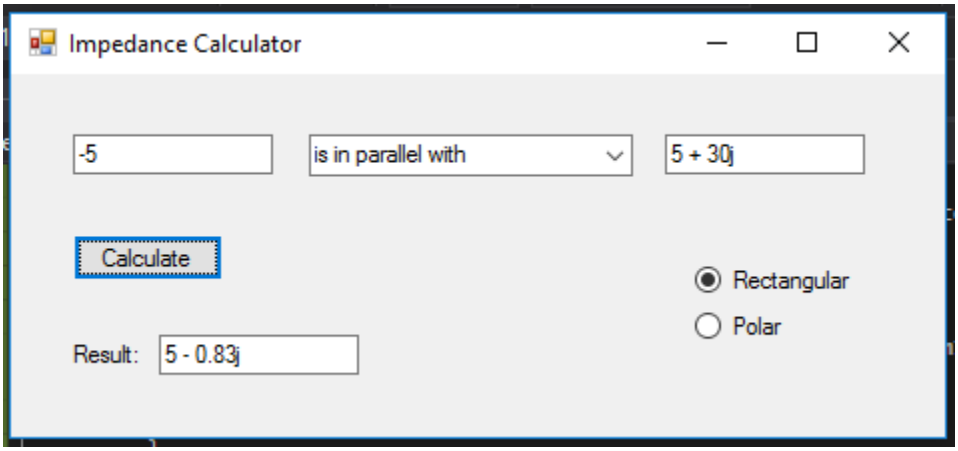
Finally, I implemented the method called when a click event is fired from the calculate button.

```
private void calculateButton_Click(object sender, EventArgs e)
{
    ComplexNumber num1 = parseTbText(numberTB1.Text);
    ComplexNumber num2 = parseTbText(numberTB2.Text);
    ComplexNumber result;

    if (operationCB.SelectedIndex == 0)
        result = getSeriesImpedance(num1, num2);
    else result = getParallelImpedance(num1, num2);

    resultTB.Text = result.ToString();
}
```

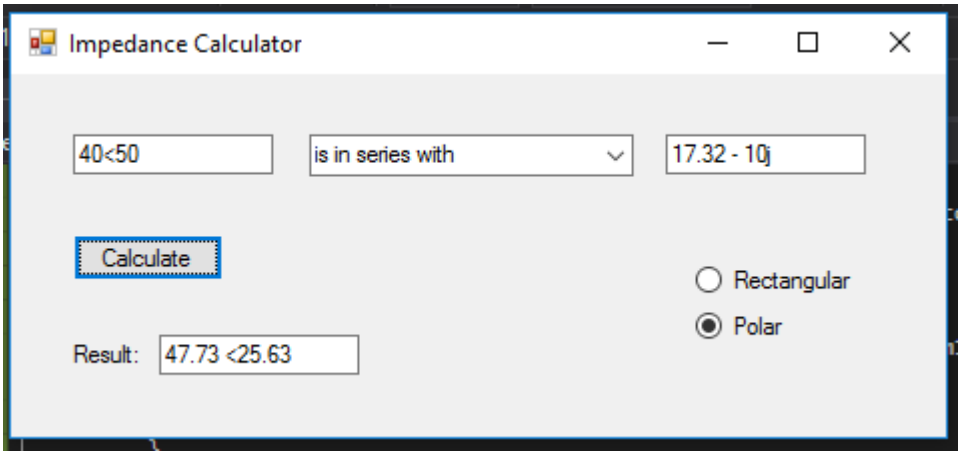
This concludes the code I wrote for this application. Following is a series of screenshots that demonstrate the use of this app.



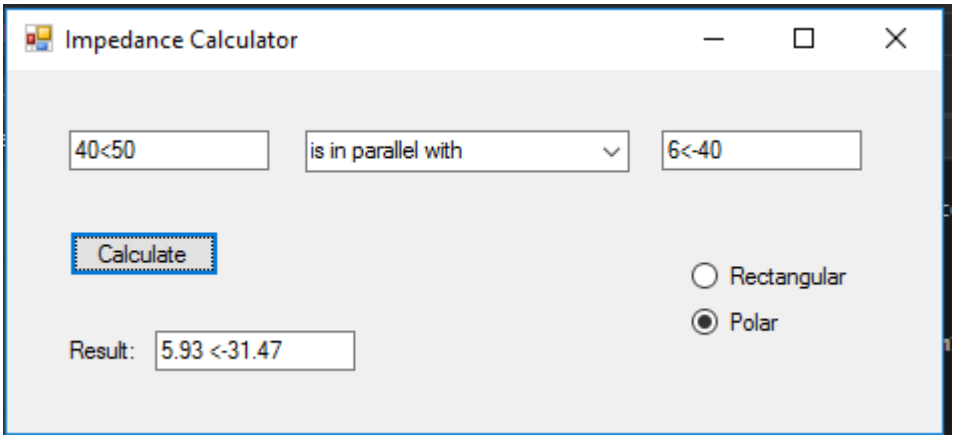
This screenshot shows what equates to a simple addition. It also shows that the parsing function implemented in the form can handle rectangular numbers without both components present.



This screenshot shows the same impedances in parallel, with a different result shown.



Here, we see that the numbers can be in two different forms in the text boxes. We also see a number displayed in polar form.



This screenshot was included for completeness: here we see two impedances in polar form in parallel with each other to produce a polar result.

Observations

This code was very fun for me to write! It was actually the most exciting thing to happen in my circuits class all year. I was excited to see another practical application of my code. I found this pattern very easy to understand, as it followed a format that I'm becoming familiar with: it reminded me of the strategy pattern, which I also understood quite well.

I had some trouble with the trig I used in my code at first because I forgot that C# performs its trig calculations in radians. This was an easy fix once I googled the Sin function. I also had some trouble checking the accuracy of my code. I had to either calculate the results by hand or find examples from my circuits book where the answer was provided. This showed me an issue that comes up in real-world coding: it isn't always so easy to verify that code is working the way it should be. In my case, the

solution was rather simple, but I could see bug-testing becoming very complicated very quickly once applications start to become bigger and more complex.

I would also like to note that although my subtract methods were never used in this code, I think they were good to include for completeness in case this app is ever extended to include more complex operations that might require subtraction.