Design Patterns

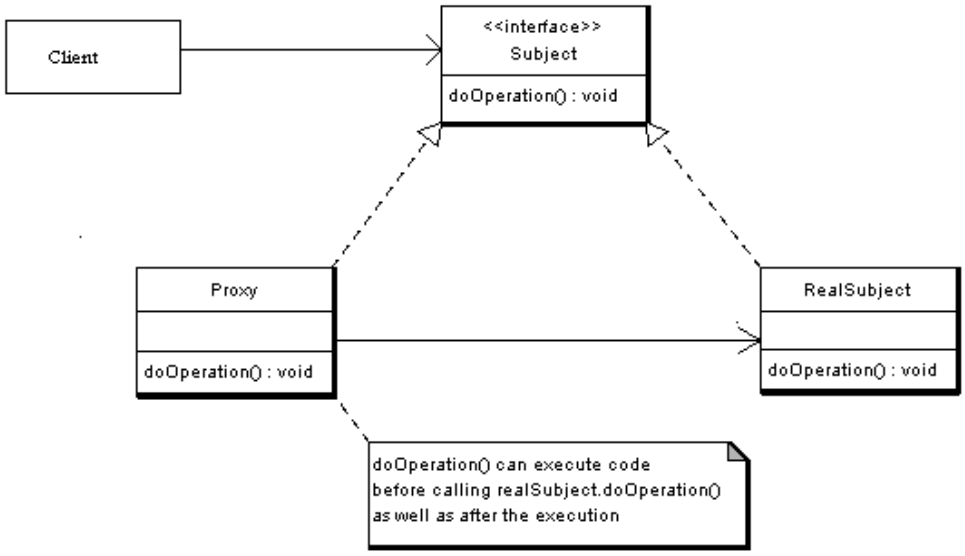
Proxy Pattern

Collin Kemner

November 1st, 2016

Introduction

The purpose of this assignment is to create an application showcasing and performing the proxy pattern correctly. The application used for this assignment is a calculator which can add, subtract, multiply, and divide two numbers that are inputted on separate screens once specific buttons are pressed. After the mathematical operation is completed, the final answer is then posted to the original Windows Form in a designated text box.



UML Diagram

According to the website oodesign.com, the proxy pattern "provide(s) a placeholder for an object to control references to it". The Subject provides an interface implemented by the RealSubject and the proxy so it can be used anywhere where the RealSubject is used. The RealSubject class is the real object that the proxy represents while the Proxy class maintains the reference that allows the Proxy to access the RealSubject, controls access to the RealSubject and its creation/deletion, implements the same interface implemented by the RealSubject so that the Proxy can be substituted for the RealSubject, and other responsibilities.

Code Analysis

Subject Interface

```
public interface Subject
    {
        double Add(double x, double y);
        double Subtract(double x, double y);
        double Multiply(double x, double y);
        double Divide(double x, double y);
    }
```

The first bit of code to be analyzed in the program is the Subject interface. The Subject defines the four mathematical operations that will be used in the application, though the actual contents of the operations are not defined in the subject. The purpose is the Subject is so all four mathematical operations can be implemented later on in the RealSubject.

RealSubject Class

```
public class RealSubject : Subject
    {
        public double Add(double x, double y)
        {
            return x + y;
        }
        public double Subtract(double x, double y)
        {
            return x - y;
        }
        public double Multiply(double x, double y)
        {
            return x * y;
        }
        public double Divide(double x, double y)
        {
            return x / y;
        }

    }
```

As stated above, the RealSubject is what implements the mathematical operations. For example, the method Add takes in two numbers (x and y) and directly implements the adding operation by the `return x + y;` statement. This is the same for the other operations as seen on the side. The RealSubject extends the Subject so that the methods can be properly implemented.

Proxy Class

```
public class Proxy : Subject
    {
        private RealSubject _rsubject = new RealSubject();

        public double Add(double x, double y)
        {
            return _rsubject.Add(x, y);
        }
        public double Subtract(double x, double y)
        {
            return _rsubject.Subtract(x, y);
        }
        public double Multiply(double x, double y)
        {
            return _rsubject.Multiply(x, y);
        }
        public double Divide(double x, double y)
        {
            return _rsubject.Divide(x, y);
        }

    }
```

Next is the Proxy class. A reference to the RealSubject is created so that the Proxy can access it and use it to perform the mathematical operations. In the Add method, a double value of the final answer of the addition of numbers x and y are returned to Form1 when it is called. This is the same for the other three methods. The Proxy extends the Subject as well so that it uses the same interface as the RealSubject.

Form1

```
public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        AddProxy addproxy = new AddProxy();
        SubtractProxy subproxy = new SubtractProxy();
        MultiplyProxy multproxy = new MultiplyProxy();
        DivideProxy divproxy = new DivideProxy();

        private void m_btnAdd_Click(object sender, EventArgs e)
        {
            addproxy.addEventHandler += new
AddProxy.AddEventHandler(operationproxy_operationEventHandler);
            addproxy.StartPosition = FormStartPosition.CenterScreen;
            addproxy.Show();

            m_tbAnswer.Text = "";
        }

        private void m_btnSubtract_Click(object sender, EventArgs e)
        {
            subproxy.subtractEventHandler += new
SubtractProxy.SubtractEventHandler(operationproxy_operationEventHandler);
            subproxy.StartPosition = FormStartPosition.CenterScreen;
            subproxy.Show();

            m_tbAnswer.Text = "";
```

```
        }

        private void m_btnMultiply_Click(object sender, EventArgs e)
        {
            multproxy.multiplyEventHandler += new
MultiplyProxy.MultiplyEventHandler(operationproxy_operationEventHandler);
            multproxy.StartPosition = FormStartPosition.CenterScreen;
            multproxy.Show(this);

            m_tbAnswer.Text = "";
        }

        private void m_btnDivide_Click(object sender, EventArgs e)
        {
            divproxy.divideEventHandler += new
DivideProxy.DivideEventHandler(operationproxy_operationEventHandler);
            divproxy.StartPosition = FormStartPosition.CenterScreen;
            divproxy.Show(this);

            m_tbAnswer.Text = "";
        }

        private void operationproxy_operationEventHandler(object sender,
OperationEventArgs a)
        {
            string newValue;

            newValue = a.m_answer;
            m_tbAnswer.Text = newValue;
        }
    }
```

Next is Form1, which will act as the caller of the different math proxies. References to the proxies are made so that its specific event handler can be subscribed as well as so that the proxy's form will appear in the middle of the screen. For example, since you can easily see the Divide button click event, when the Divide button on the Form is clicked, the divide event handler is subscribed to the DivideProxy class and a void for posting the mathematical operation's result to the main Form is set as its parameter. Next, the DivideProxy's form is set to the middle of the screen when it is opened and the Answer text box is cleared if there happens to be any text already inside it. This happens for the three other click events. The void at the bottom of the code is the void which will be activated once the event of the mathematical operation takes place. It sets a string to the answer of the operation (referenced from the OperationEventArgs) and is posted to the answer text box. Again, this happens with the other three proxies.

The Proxies

```
public partial class AddProxy : Form
    {
        public AddProxy()
        {
            InitializeComponent();
        }

        public delegate void AddEventHandler(object sender, OperationEventArgs a);
        public AddEventHandler addEventHandler;

        public Proxy proxy = new Proxy();

        public void m_btnAddNumbers_Click(object sender, EventArgs e)
        {
            double x = Convert.ToDouble(m_tbAddX.Text);
            double y = Convert.ToDouble(m_tbAddY.Text);

            double d_answer = proxy.Add(x, y);
            string s_answer = Convert.ToString(d_answer);

            if (m_tbAddX.Text.Length != null && m_tbAddY.Text.Length != null)
            {
                if (addEventHandler != null)
                    addEventHandler(this, new OperationEventArgs(s_answer));
            }
        }
    }
```

Next are the proxies. These proxies are all specialized to its specific mathematical operation and they all have their own event handlers so that the main form knows which event has been fired. In the AddProxy form code, the signature and event handler are both defined. The signature uses OperationEventArgs rather than the normal EventArgs arguments as the OperationEventArgs is used to pass the mathematical answer back to the main form. A reference of the Proxy class is defined. Then, a button click event is created in which two double values (x and y) are converted to double from their original string type (as they originate from text boxes in the form). These two values are then sent to the Proxy class' method "Add" and the resulting value is saved to the double variable "d_answer". Then, a new string, s_answer, is defined as d_answer converted to string. After all of this is said and done, the event handler then fires the new event and uses the OperationEventArgs in the parameter of the event handler and also uses the string version of the mathematical result as the OperationEventArgs' parameter. That will allow the main form to access the result from the event being fired. This same thing happens with the three other proxies.

```
public partial class SubtractProxy : Form
    {
        public SubtractProxy()
        {
            InitializeComponent();
        }

        public delegate void SubtractEventHandler(object sender, OperationEventArgs a);
        public SubtractEventHandler subtractEventHandler;

        public Proxy proxy = new Proxy();

        public void m_btnSubtractNumbers_Click(object sender, EventArgs e)
        {
            double x = Convert.ToDouble(m_tbSubtractX.Text);
            double y = Convert.ToDouble(m_tbSubtractY.Text);

            double d_answer = proxy.Subtract(x, y);
            string s_answer = Convert.ToString(d_answer);

            if (m_tbSubtractX.Text.Length != null && m_tbSubtractY.Text.Length != null)
            {
                if (subtractEventHandler != null)
                    subtractEventHandler(this, new OperationEventArgs(s_answer));
            }
        }
    }

public partial class MultiplyProxy : Form
    {
        public MultiplyProxy()
        {
            InitializeComponent();
        }

        public delegate void MultiplyEventHandler(object sender, OperationEventArgs a);
        public MultiplyEventHandler multiplyEventHandler;

        public Proxy proxy = new Proxy();

        private void m_btnMultiplyNumbers_Click(object sender, EventArgs e)
        {
            double x = Convert.ToDouble(m_tbMultiplyX.Text);
            double y = Convert.ToDouble(m_tbMultiplyY.Text);

            double d_answer = proxy.Multiply(x, y);
            string s_answer = Convert.ToString(d_answer);

            if (m_tbMultiplyX.Text.Length != null && m_tbMultiplyY.Text.Length != null)
            {
                if (multiplyEventHandler != null)
                    multiplyEventHandler(this, new OperationEventArgs(s_answer));
            }
        }
    }
```

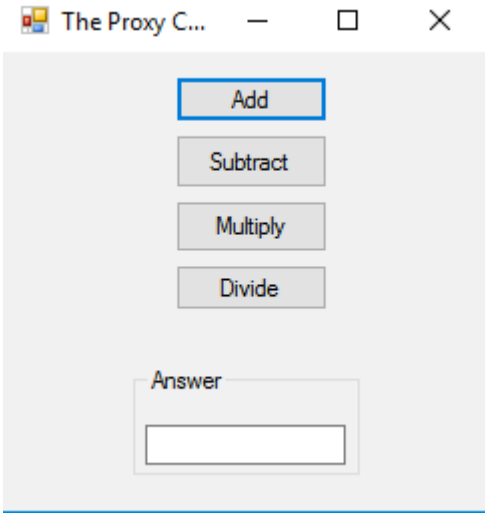
```
public partial class DivideProxy : Form
    {
        public DivideProxy()
        {
            InitializeComponent();
        }

        public delegate void DivideEventHandler(object sender, OperationEventArgs a);
        public DivideEventHandler divideEventHandler;

        public Proxy proxy = new Proxy();

        private void m_btnDivideNumbers_Click(object sender, EventArgs e)
        {
            double x = Convert.ToDouble(m_tbDivideX.Text);
            double y = Convert.ToDouble(m_tbDivideY.Text);

            double d_answer = proxy.Divide(x, y);
            string s_answer = Convert.ToString(d_answer);

            if (m_tbDivideX.Text.Length != null && m_tbDivideY.Text.Length != null)
            {
                if (divideEventHandler != null)
                    divideEventHandler(this, new OperationEventArgs(s_answer));
            }
        }
    }
```

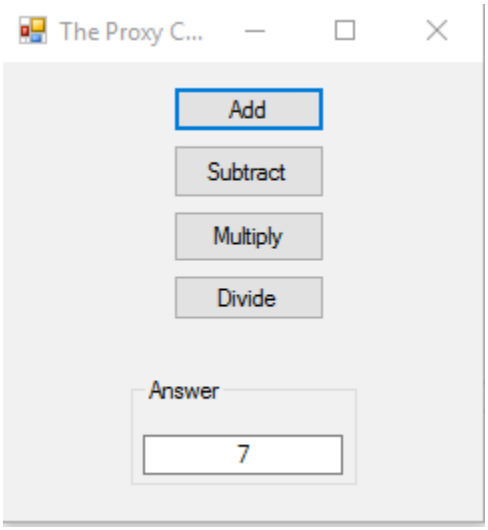
OperationEventArgs: New Event Arguments

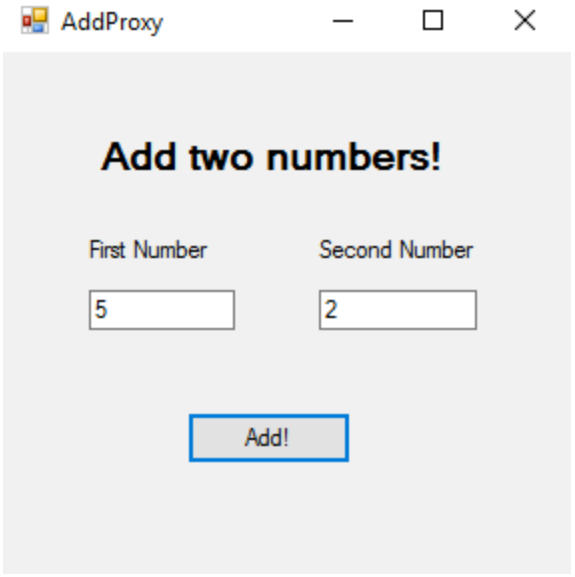
```
public class OperationEventArgs : EventArgs
    {
        private string _answer;

            public string m_answer
            {
                get { return _answer; }
                set { _answer = value; }
            }

            public OperationEventArgs(string answer)
            {
                _answer = answer;
            }
    }
```

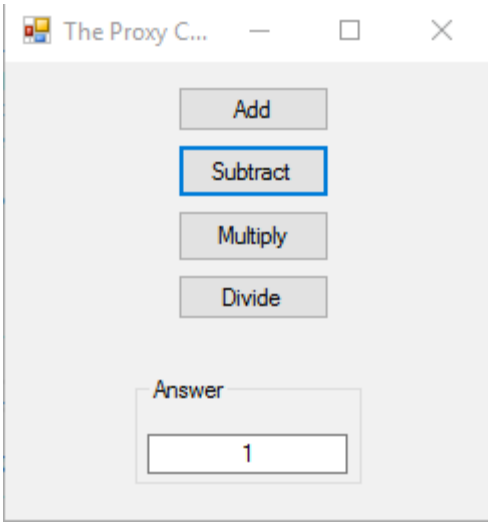
This set of event arguments is what carries the mathematical operation's answer to the main form. It extends the normal event arguments (EventArgs) and stores the answer in the string _answer. The method OperationEventArgs then takes a string, "answer", as a parameter and then sets it equal to the original answer string. When the OperationEventArgs is called, the constructor m_answer is mentioned and is what will send the value over.

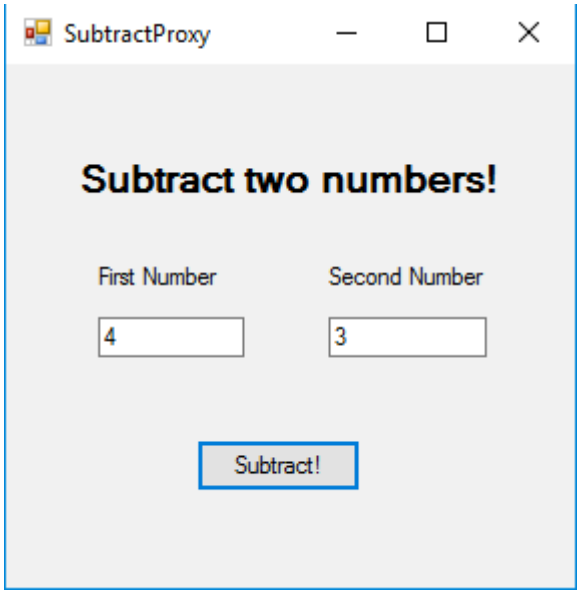
Screenshots of the Application Working as Described

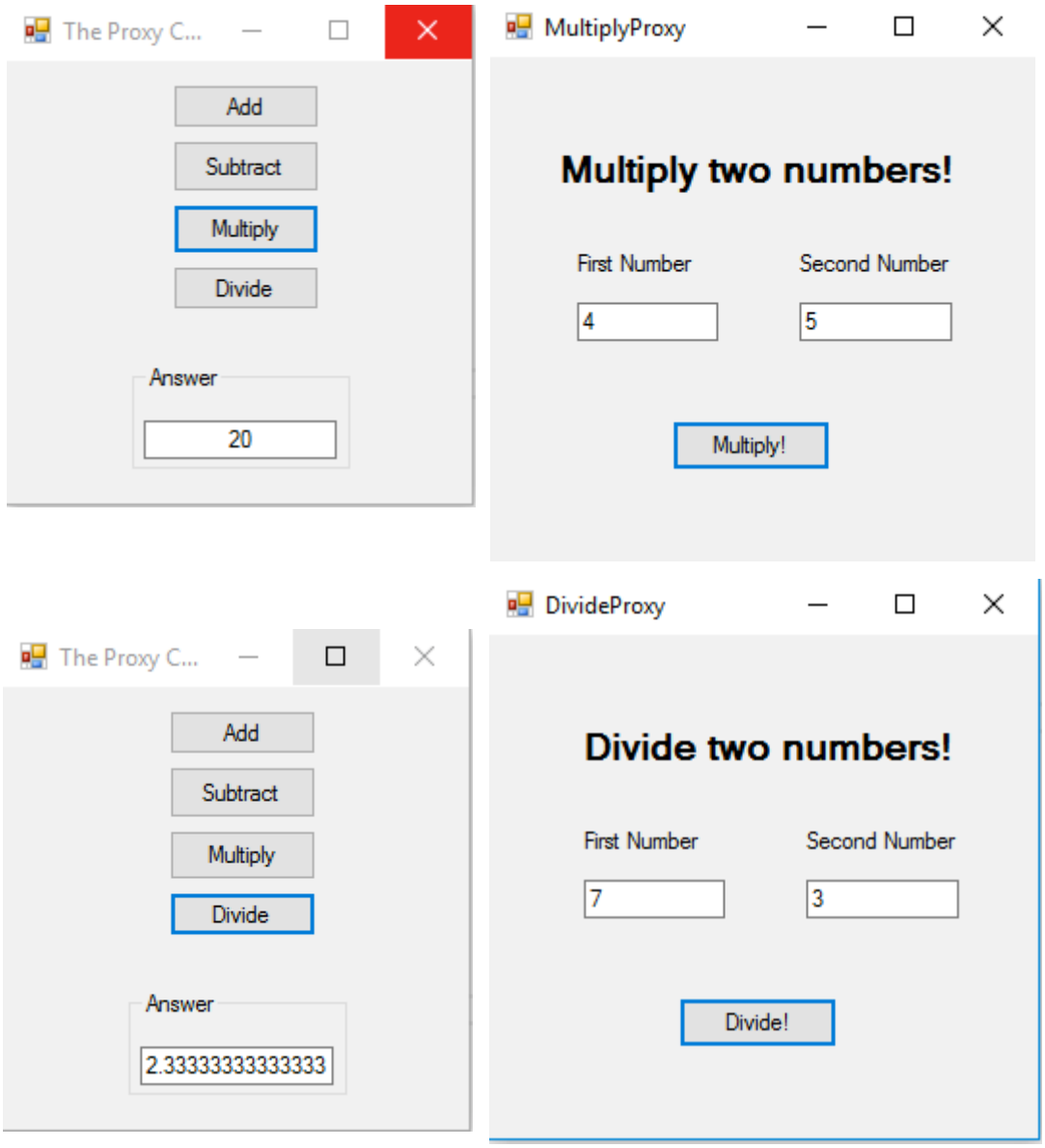












Observations/Reflections

I felt that this was a great pattern to write and that it helped me understand more how to think on my own while writing applications rather than outside help. Although I did have outside help at the beginning of the assignment, after the framework of it was laid down I started to use my intuition as to how to go about this assignment. I got a lot of the code done in the Form and the proxy's forms when I realized that I needed to use event handlers and customized event arguments similar to the observer pattern, so I brought up the observer pattern to remind myself how they worked and I utilized it in a

similar way in the Proxy pattern. Thankfully it worked out and now this assignment is completed. I do hope that this application and code accurately represents the Proxy pattern as I believe it should.