Tori Lentz

Dave Retterer

Design Patterns

November 21, 2016

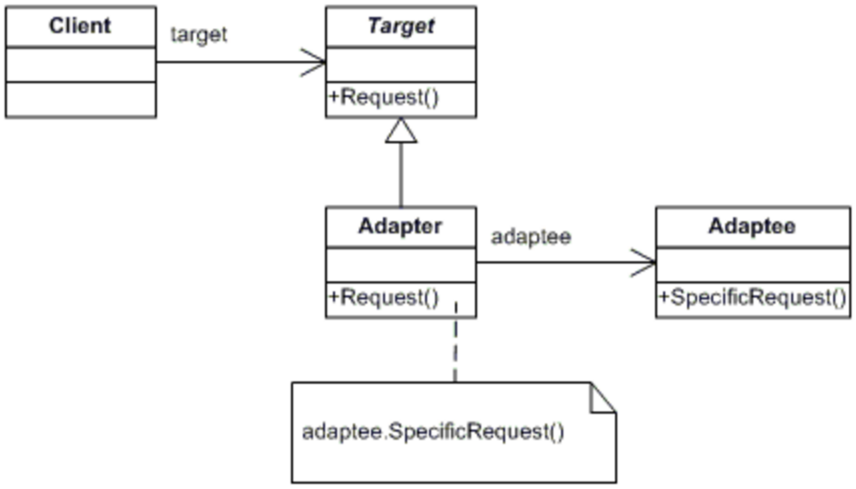
Command and Adapter Patterns

Introduction

The purpose of this assignment was to design and implement the command and adapter patterns. The main purpose of the adapter pattern is to gain the ability to convert from one interface to another. For this assignment, the idea is to convert some interface from the command pattern into a different interface.

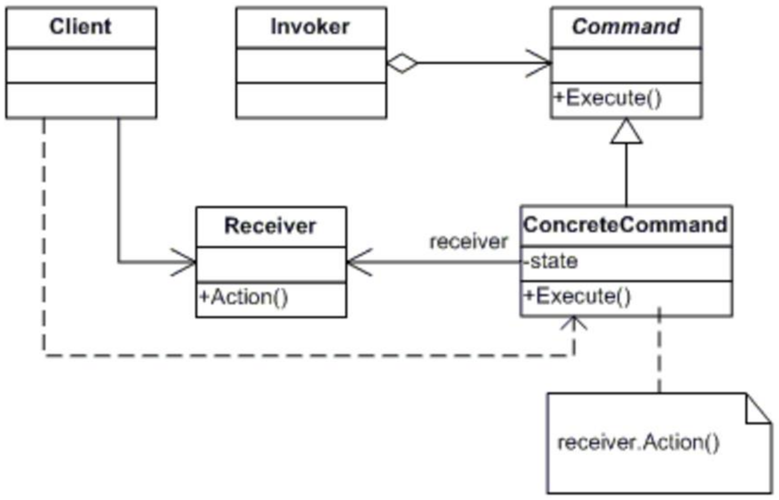
UML Diagram

Adapter Pattern



The first UML diagram depicts the adapter pattern. This pattern consists of four different classes: the client, the target, the adapter and the adaptee. The client class collaborates with different objects that conform to the Target interface. Next is the Target class which defines the domain specific interface that the client class will use. Thirdly is the adapter class which will adapt its interface to that of the target interface. Finally, is the adaptee class which defines the interface that needs to be adapted.

Command Pattern



The second UML diagram shows the command pattern. This pattern consists of five different classes: the client, the receiver, the command, the concrete command and the invoker. The client class creates a concrete command object and sets its receiver. The receiver knows how to perform all the operations for this pattern. The command class declares an interface for executing the operations. The concrete command class extends the interface of the command class and implements an execute method. This class also defines a link between the receiver class and the action to be performed. Finally, is the invoker class which asks the command class to carry out the request that is being made.

<u>Code and Description</u>

Target

```
public interface ITarget
{
    // Used when the Command/Adapter pattern is enabled
    // Interface for New Client calls
    string AdapterRedo();
    string AdapterUndo();
    string AdapterCompute(char @operator, short operand);
}
```

This is my target class. This class sets up the operations that I will use in my adapter pattern and also sets up the client interface.

Adapter

```
public class AdapterUser // Used when the Command/Adapter pattern is enabled
{
    User user = new User();

    public string AdapterUndo()
    {
        //User _user = new User();
        return user.Undo(Convert.ToInt16(1));
    }
    public string AdapterRedo()
    {
        //User _user = new User();
```

Here is where I set up my adaptations. In this class I am transforming parameters into data types and values that the command pattern invoker is expecting.

```
            return user.Redo(Convert.ToInt16(1));
        }
        public string AdapterCompute(char @operator, short operand)
        {
            //User _user = new User();
            return user.Compute(@operator, operand);
        }
    }
```

Receiver

```
    public class Calculator
    {
        private int _curr = 0;

        public string Operation(char @operator, int operand)
        {
            switch (@operator)
            {
                case '+': _curr += operand; break;
                case '-': _curr -= operand; break;
                case '*': _curr *= operand; break;
                case '/': _curr /= operand; break;
            }

            return _curr.ToString();
        }
    }
```

This is my receiver class which will be used in the command pattern. Here I set up my operations for my calculator.

Command

```
    public abstract class Command
    {
        public abstract string Execute();
        public abstract string UnExecute();
    }
```

This is my command class where I first define my abstract definitions for my methods.

Concrete Command

```
        public class CalculatorCommand : Command
    {
        public char _operator;
        public int _operand;
        public Calculator _calculator;

    //    // Constructor
        public CalculatorCommand(Calculator calculator, char @operator, int operand)
        {
            this._calculator = calculator;
            this._operator = @operator;
            this._operand = operand;
        }

    //    // Gets operator
        public char Operator
        {
            set { _operator = value; }
        }
```

My concrete command class further defines my methods and will then tell my program what it needs to do.

```
    //    // Get operand
        public int Operand
        {
            set { _operand = value; }
        }

    //    // Execute new command
        public override string Execute()
        {
            return _calculator.Operation(_operator, _operand);
        }

    //    // Unexecute last command
        public override string UnExecute()
        {
           return _calculator.Operation(Undo(_operator), _operand);
        }

    //    // Returns opposite operator for given operator
        private char Undo(char @operator)
        {
            switch (@operator)
            {
                case '+': return '-';
                case '-': return '+';
                case '*': return '/';
                case '/': return '*';
                default:
                    throw new
            ArgumentException("@operator");
            }
        }
    }
```

Invoker / Adaptee

```
    public class User // Used as the Adaptee class when Command/Adapter pattern is
enabled
    {
        // Initializers
        private Calculator _calculator = new Calculator();
        private List<Command> _commands = new List<Command>();
        private int _current = 0;

        public string Redo(int levels)
        {
            string returnValue = "";

            // Perform redo operations
            for (int i = 0; i < levels; i++)
            {
                if (_current <= _commands.Count - 1)
                {
                    Command command = _commands[_current++];
                    returnValue = command.Execute();
                }
            }
            return returnValue;
```

Finally, I have my invoker class which is also the same as my adaptee class. Within this class is where the two patterns are finally connected and, it is also where all the operations that the methods call upon are defined.

```
        }

        public string Undo(int levels)
        {
            string returnValue = "";

            // Perform undo operations
            for (int i = 0; i < levels; i++)
            {
                if (_current > 0)
                {
                    Command command = _commands[--_current] as Command;
                    returnValue = command.UnExecute();
                }
            }
            return returnValue;
        }

        public string Compute(char @operator, int operand)
        {
            // Create command operation and execute it
            string returnValue = "";
            Command command = new CalculatorCommand(
              _calculator, @operator, operand);
            returnValue = command.Execute();

            // Add command to undo list
            _commands.Add(command);
            _current++;
            return returnValue;
        }
    }
```

Form

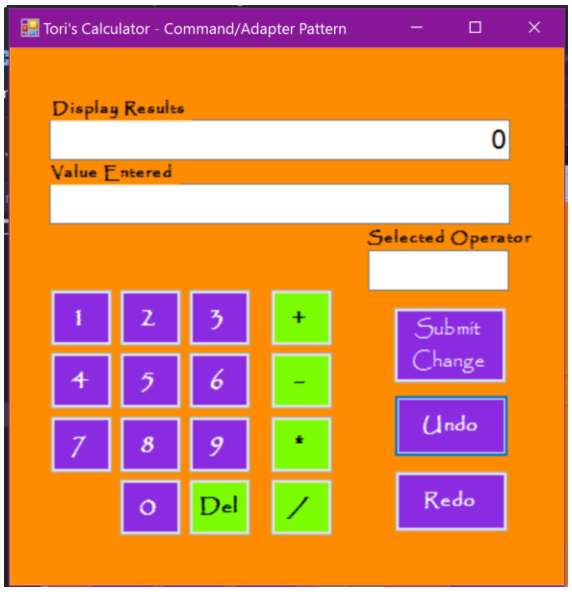
```
    public partial class Form1 : Form
    {
        private int undoCount = 0;
        private int redoCount = 0;
        private int submitCount = 0;

        // Code for Command pattern
        // Uncomment User and comment out AdapterUser
when running the Command pattern
        //private User user = new User();

        // Code for Command/Adapter patterns combined
        // Comment out User and uncomment AdapterUser
when running the Command/Adapter combined pattern
        public AdapterUser adaptee = new AdapterUser();

        public Form1()
        {
            InitializeComponent();

        }

        private void btn_OperatorPlus_Click(object
sender, EventArgs e)
```

My form class is my client and within this class I set up my form with buttons to make it look like a calculator. Each button click concatenates into my new value text box. The delete button clears the new value text box. The submit button takes an operator and a new value and makes a call to compute. The undo passes in a single step undo and will undo the layers one at a time. The redo button works the same as the undo button. I have one on load event which initializes the display text box to zero. I also call my methods from my patterns within this class. I built the calculator using the command pattern and all this code still exists in my program for testing, should I ever need it, and has been commented out.

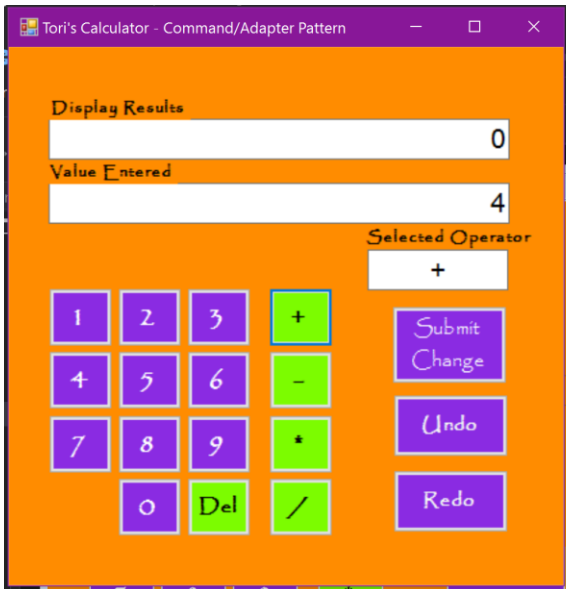
```
{
    tb_Operator.Text = "+";
}

private void btn_OperatorMinus_Click(object sender, EventArgs e)
{
    tb_Operator.Text = "-";
}

private void btn_OperatorMultiply_Click(object sender, EventArgs e)
{
    tb_Operator.Text = "*";
}

private void btn_OperatorDivide_Click(object sender, EventArgs e)
{
    tb_Operator.Text = "/";
}

private void button13_Click(object sender, EventArgs e)
{
    tb_NewValue.Text = tb_NewValue.Text + "0";
}

private void btn_Value1_Click(object sender, EventArgs e)
{
    tb_NewValue.Text = tb_NewValue.Text + "1";
}

private void btn_Value2_Click(object sender, EventArgs e)
{
    tb_NewValue.Text = tb_NewValue.Text + "2";
}

private void btn_Value3_Click(object sender, EventArgs e)
{
    tb_NewValue.Text = tb_NewValue.Text + "3";
}

private void btn_Value4_Click(object sender, EventArgs e)
{
    tb_NewValue.Text = tb_NewValue.Text + "4";
}

private void btn_Value5_Click(object sender, EventArgs e)
{
    tb_NewValue.Text = tb_NewValue.Text + "5";
}

private void btn_Value6_Click(object sender, EventArgs e)
{
    tb_NewValue.Text = tb_NewValue.Text + "6";
}

private void btn_Value7_Click(object sender, EventArgs e)
{
    tb_NewValue.Text = tb_NewValue.Text + "7";
}
```

```
        private void btn_Value8_Click(object sender, EventArgs e)
        {
            tb_NewValue.Text = tb_NewValue.Text + "8";
        }

        private void btn_Value9_Click(object sender, EventArgs e)
        {
            tb_NewValue.Text = tb_NewValue.Text + "9";
        }

        private void btn_ValueDecimalPoint_Click(object sender, EventArgs e)
        {
            tb_NewValue.Text = tb_NewValue.Text + ".";
        }

        private void btn_ValueDelete_Click(object sender, EventArgs e)
        {
            tb_NewValue.Text = "";
        }

        private void btn_SubmitChange_Click(object sender, EventArgs e)
        {
            if (tb_NewValue.Text != "")
            {
                if (tb_Operator.Text != "")
                {
                    // Call for Command pattern code
                    //tb_Display.Text = user.Compute(Convert.ToChar(tb_Operator.Text),
Convert.ToInt16(tb_NewValue.Text));

                    // Call for Command/Adapter patterns combined code
                    tb_Display.Text =
adaptee.AdapterCompute(Convert.ToChar(tb_Operator.Text),
Convert.ToInt16(tb_NewValue.Text));
                    tb_NewValue.Text = "";
                    tb_Operator.Text = "";
                    submitCount = submitCount+1;
                }
                else {
                    MessageBox.Show("Operator must be selected before submitting
request.");
                }
            }
            else
            {
                MessageBox.Show("Value must be entered before submitting request.");
            }
        }

        private void btn_Redo_Click(object sender, EventArgs e)
        {
            if(undoCount > 0)
            {
                // Call for Command pattern code
                //tb_Display.Text = user.Redo(1);

                // Call for Command/Adapter patterns combined code
                tb_Display.Text = adaptee.AdapterRedo();

                redoCount = redoCount+1;
            }
            else
            {
                MessageBox.Show("There is nothing to Redo.");
            }
        }

        private void btn_Undo_Click_1(object sender, EventArgs e)
        {
            if(submitCount > 0)
            {
                // Call for Command pattern code
                //tb_Display.Text = user.Undo(1);

                // Call for Command/Adapter patterns combined code
                tb_Display.Text = adaptee.AdapterUndo();
                undoCount = undoCount+1;
            }
            else
            {
                MessageBox.Show("There is nothing to Undo.");
            }
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Call for Command pattern code
            //tb_Display.Text = user.Compute('+', 0);

            // Call for Command/Adapter patterns combined code
            tb_Display.Text = adaptee.AdapterCompute('+', 0);
        }
    }
```

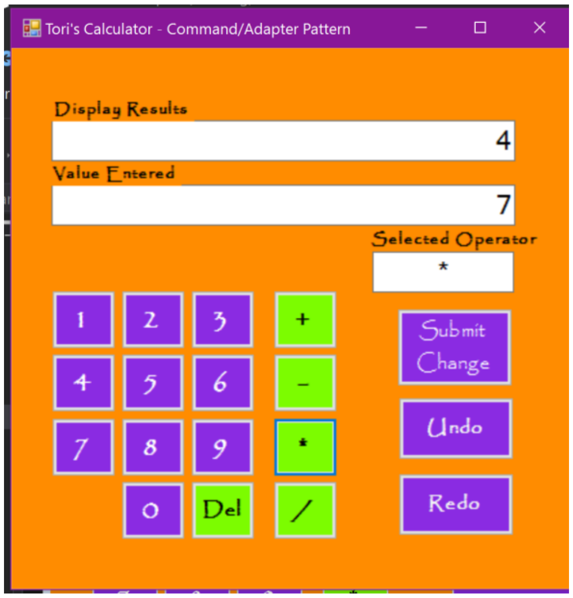
Screen Shots



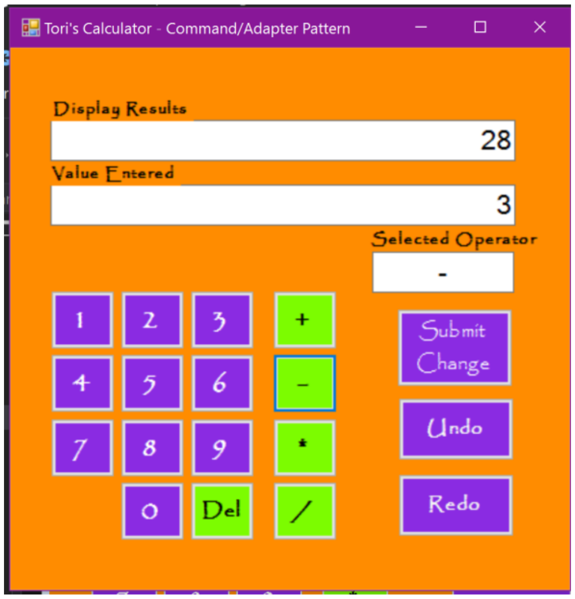
Form on load



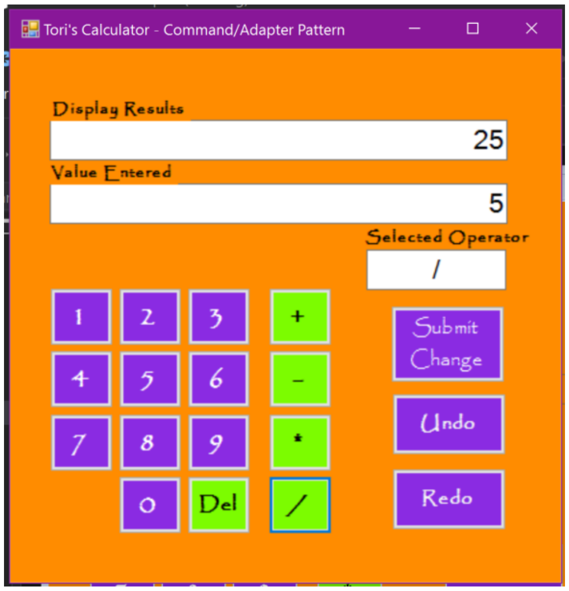
Form after number and plus sign clicked



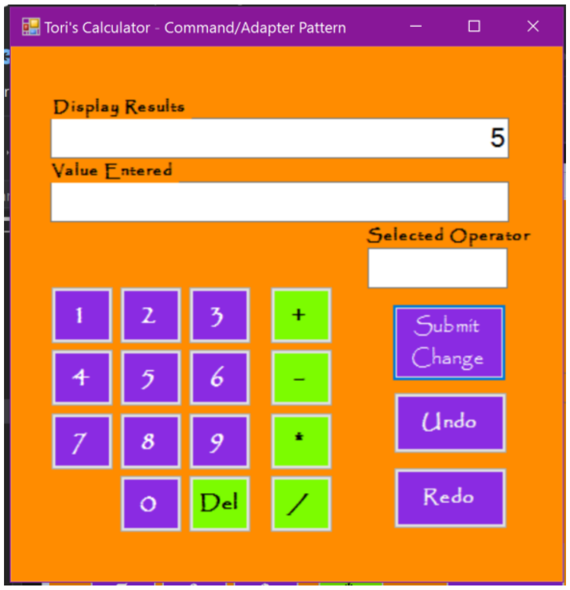
Submit change button, number button and multiply button pushed



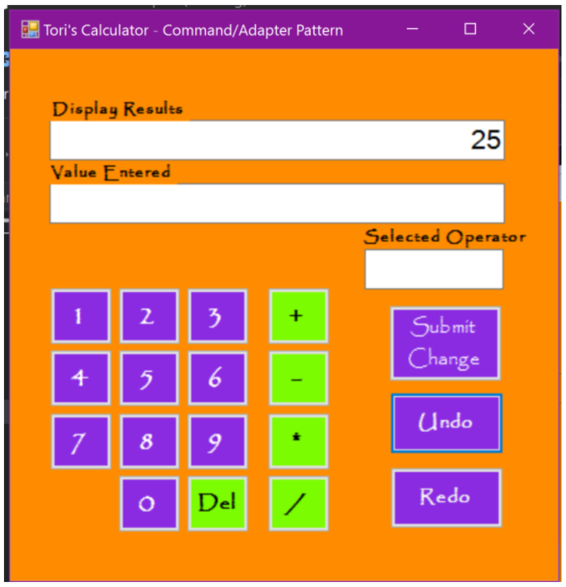
Submit change button, number button and subtract button pushed



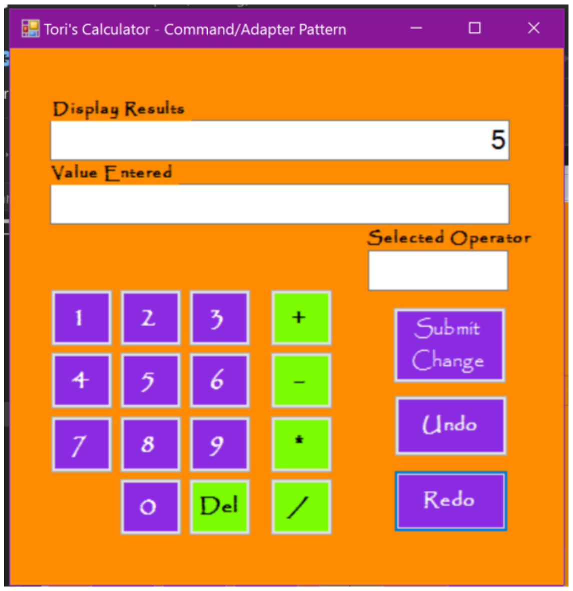
Submit change button, number button and divide button pushed



Submit change button pushed



Undo button pushed



Redo button pushed

Observations

Overall, these patterns went very well. It took me a long time to try and figure out how to do everything within these patterns but, after a while, I was finally able to figure it out. Seeing how these patterns work makes me realize how many applications they have. I know down the road I am going to use these patterns for a lot of different things because, I'm sure there will be times when I am going to need to adapt some kind of code or need to implement commands of some kind.