Jack Raney

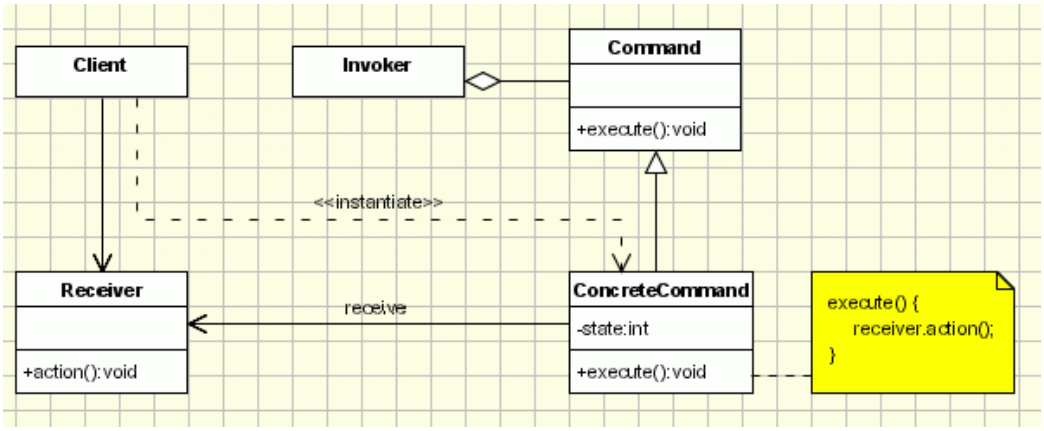
Design Patterns

10/6/2016

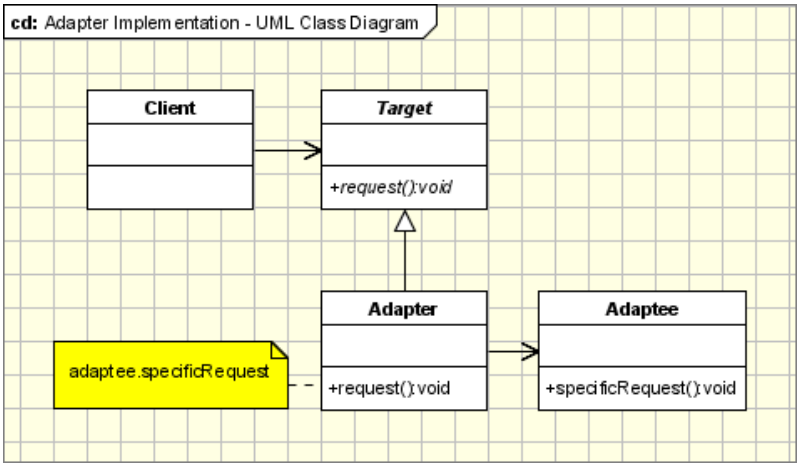
The Observer and Adapter Patterns

This paper will be showcasing the Observer and Adapter design patterns. According to oodesign.com, the Observer Pattern is used to "encapsulate a request in an object… [allow] the parameterization of clients with different requests, [and allow] saving the requests in a queue." The same source states that the Adapter Pattern "[converts] the interface of a class into another interface clients expect" and that the "adapter lets classes work together, that could not otherwise because of incompatible interfaces". The UML diagrams for these patterns are shown below.

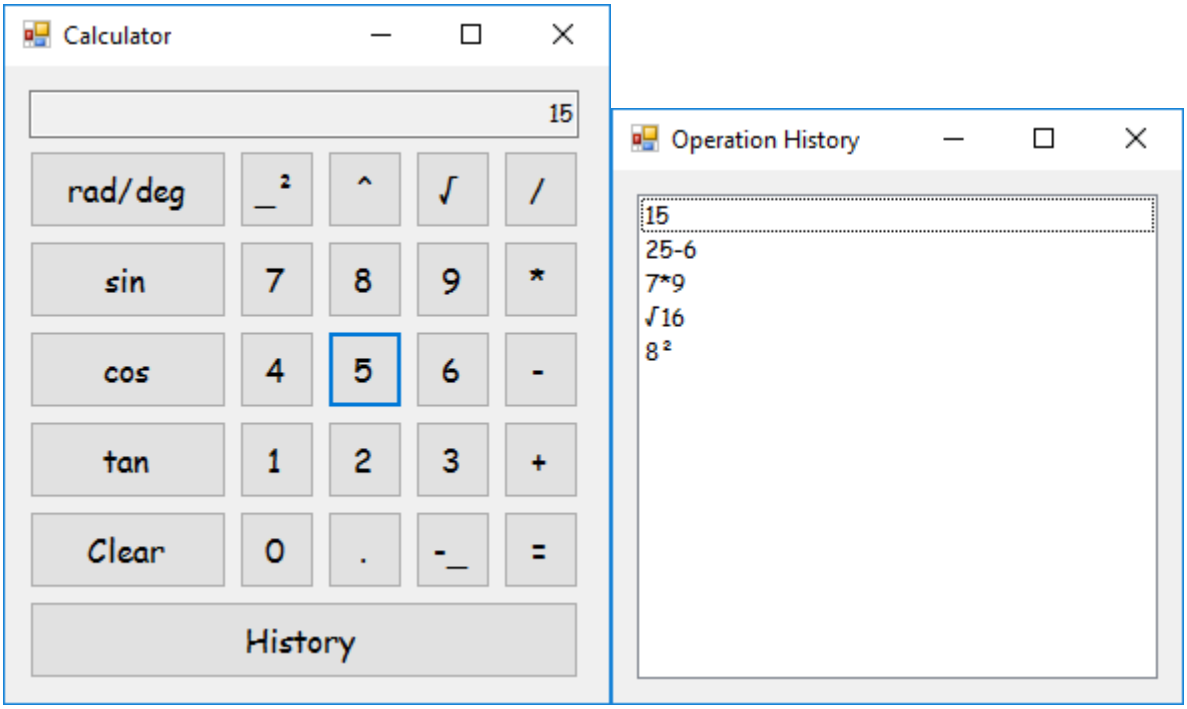
Command Pattern



Adapter Pattern



I created the program Calculator.exe as an example of both patterns at work:



The Code

```
public class Handler   //This is the Invoker
    {
        private List<string> operList = new List<string>();

        public List<string> getList()
        {
            return operList;
        }

        public double executeCommand(Operation oper)
        {
            operList.Add(oper.ToString());
            return oper.operate();
        }
    }
```

Handler.cs contains an aggregate property that holds the operations in their string format. The actual operations are not being stored because I only display the operations within the aggregate; this makes that process simpler. executeCommand adds a ToString'ed operation to the aggregate and returns the solution to the operation.

```
public abstract class Operation //This is the parent command
    {
        protected double num1;
        protected double num2;
        protected int oper;

        public abstract double operate();

        public void setOperation(double number1, int operation, double number2)
        {
            num1 = number1;
            num2 = number2;
            oper = operation;
        }
    }
```

The Operation class is an abstract parent that contains two doubles for the numbers to be operated on and an integer that represents the arithmetic operation (more on that later). All three are protected so that child classes contain them.
operate() is only a signature here, and will be detailed in Operation's child classes.

```
public class SingleOperation : Operation    //This is a concrete command
    {
        private Calculator calc = new Calculator();

        public SingleOperation(Calculator c)
        {
            calc = c;
        }

        public override double operate()
        {
            return calc.solve(num1, oper);
        }

        public override string ToString()
        {
            switch (oper)
            {
                case 6:  return num1 + "\u00b2";
                case 7:  return "\u221a" + num1;
                case 8:  return "sin" + num1;
                case 9:  return "sin" + num1;
                case 10: return "cos" + num1;
                case 11: return "cos" + num1;
                default: return "tan" + num1;
            }
        }
    }
```

SingleOperation is one of two child classes to Operation. It is constructed with a Calculator object passed into it that it uses to do the arithmetic of the operation it stores.
In this child, operate() calls the overloaded method in the Calculator class that uses only the first double.
ToString's value depends on which arithmetic operation is being performed.

```
public class DoubleOperation : Operation    //This is a concrete command
    {
        private Calculator calc;

        public DoubleOperation(Calculator c)
        {
            calc = c;
        }

        public override double operate()
        {
            return calc.solve(num1, oper, num2);
        }

        public override string ToString()
        {
            switch (oper)
            {
                case 0: return  num1 + "";
                case 1: return  num1 + "+" + num2;
                case 2: return  num1 + "-" + num2;
                case 3: return  num1 + "*" + num2;
                case 4: return  num1 + "/" + num2;
                default: return num1 + "^" + num2;
            }
        }
    }
```

DoubleOperation is very similar SingleOperation. The differences are that operate() calls Calculator's other solve method that uses both doubles and ToString's values are for different commands.

```
public class Calculator //This is the receiver for the Command Pattern
                        //It is also the client for the Adapter Pattern
    {
        private RadTrig rad = new RadTrig();
        private DegTrig deg;

        public void setDeg()
        {
            deg = new DegTrig(rad);
        }

        public Calculator()
        {
            setDeg();
        }

        public double solve(double num1, int oper, double num2)
        {
            switch (oper)
            {
                case 0: return num1;
                case 1: return num1 + num2;
                case 2: return num1 - num2;
                case 3: return num1 * num2;
                case 4: return num1 / num2;
                default: return Math.Pow(num1, num2);
            }
        }

        public double solve(double num, int oper)
        {
            switch (oper)
            {
                case 6:  return Math.Pow(num, 2);
                case 7:  return Math.Sqrt(num);
                case 8:  return rad.sin(num);
                case 9:  return deg.sin(num);
                case 10: return rad.cos(num);
                case 11: return deg.cos(num);
                case 12: return rad.tan(num);
                default: return deg.tan(num);
            }
        }
    }

public abstract class TrigAdapter   //This is the abstract adapter (target) class
    {
        public const double PI = Math.PI;

        public abstract double sin(double num);
        public abstract double cos(double num);
        public abstract double tan(double num);
    }
```

Calculator is used to do the actual arithmetic with the Operation objects. It also holds the classes used to do trigonometry, which are part of the adapter pattern.

TrigAdapter is the abstract parent to the adapter class of this program.

```
public class DegTrig : TrigAdapter  //This is the concrete adapter class
    {
        private RadTrig rad = new RadTrig();

        const double CONVERTER = PI / 180;

        public DegTrig(RadTrig r)
        {
            rad = r;
        }

        public override double sin(double num)
        {
            return rad.sin(num * CONVERTER);
        }

        public override double cos(double num)
        {
            return rad.cos(num * CONVERTER);
        }

        public override double tan(double num)
        {
            return rad.tan(num * CONVERTER);
        }
    }
```

DegTrig is the adapter for RadTrig; it converts the methods from RadTrig (which takes inputs in radians) into methods that input degrees, which is easier to look at and understand by most people, as degrees are taught much earlier than radians in schooling.

```
public class RadTrig    //The adaptee class
    {
        public double sin(double num)
        {
            return Math.Sin(num);
        }

        public double cos(double num)
        {
            return Math.Cos(num);
        }

        public double tan(double num)
        {
            return Math.Tan(num);
        }
    }
```

See above.

```
public partial class Form1 : Form   //This is the Command Pattern client
    {
        const int PLUS    = 1;
        const int MINUS   = 2;
        const int MULT    = 3;
        const int DIV     = 4;
        const int POW     = 5;
        const int SQR     = 6;
        const int SQRT    = 7;
        const int RAD_SIN = 8;
        const int DEG_SIN = 9;
        const int RAD_COS = 10;
        const int DEG_COS = 11;
        const int RAD_TAN = 12;
        const int DEG_TAN = 13;

        double num1;
        int num1Int;
        double num1Dec;

        double num2;
        int num2Int;
        double num2Dec;

        double result;

        int decPlace;

        int oper;

        bool isNum1;
        bool isInt;
        bool isNeg;
        bool isRad;
        bool willRepeat;

        Calculator calc;
        SingleOperation sOper;
        DoubleOperation dOper;
        Handler handler;

        public Form1()
        {
            InitializeComponent();
            reset();

            calc = new Calculator();
            sOper = new SingleOperation(calc);
            dOper = new DoubleOperation(calc);
            handler = new Handler();

            isRad = true;
        }
```

These constants represent the different arithmetic operations.

The values entered into the calculator are input one digit at a time, so I split it into two halves: the integer before the decimal point and the double value afterwards.

The result double is what the value of each operation is stored into. decPlace is a counter that determines what decimal place the doubles' decimal values are currently at.
oper is the int that determines which operation is going to be performed.
Boolean values keep track of which of the two doubles is being entered, whether the double is negative, if trig functions are using degrees or radians, and if the equals button will repeat the previous operation.
A calculator, the two Operations, and the handler are all instantiated within the form's constructor.

Along with the four classes' instantiation previously mentioned, the program is "reset" and defaults to using radians upon startup.

```
private void reset()
{
    calcText.Clear();
    num1     = 0;
    num1Int  = 0;
    num1Dec  = 0;

    num2     = 0;
    num2Int  = 0;
    num2Dec  = 0;

    decPlace = 0;
    oper     = 0;

    isNum1 = true;
    isInt  = true;
    isNeg  = false;
    willRepeat = false;
}
```

The reset method clears the textbox that outputs what the user enters and the operations' values, and sets all number properties to 0. The Booleans are adjusted to when the first value is being entered

```
private void addDigit(int num)
{
    if (willRepeat)
    {
        reset();
    }

    if (isNum1)
    {
        if (isInt)
        {
            num1Int *= 10;
            num1Int += num;
        }

        else
        {
            num1Dec *= 10;
            num1Dec += num;
            decPlace++;
        }
    }

    else
    {
        if (isInt)
        {
            num2Int *= 10;
            num2Int += num;
        }

        else
        {
            num2Dec *= 10;
            num2Dec += num;
            decPlace++;
        }
    }

    calcText.Text += num;
}
```

addDigit concatenates another digit into the current value in the program's focus and outputs the new value into the textbox.

```
        private void setOper(int op)
        {
            oper = op;

            isNum1 = false;

            if (num1Dec >= 1)
            {
                num1Dec /= Math.Pow(10, decPlace);
                decPlace = 1;
            }

            isInt = true;
            isNeg = false;

            calcText.Clear();
        }
```

setOper sets the oper variable to the entered integer and switches focus from the first double to the second.

```
private void zeroButton_Click(object sender, EventArgs e)
        {
            addDigit(0);
        }
```

The numbered buttons perform addDigit, inputting their respective digits.

```
private void deciButton_Click(object sender, EventArgs e)
        {
            if (isInt)
            {
                isInt = false;
                calcText.Text += ".";
            }
        }

private void negaButton_Click(object sender, EventArgs e)
{
    if (isNum1)
    {
        num1Int = 0 - num1Int;
    }

    else
    {
        num2Int = 0 - num2Int;
    }

    if (!isNeg)
    {
        calcText.Text = "-" + calcText.Text;
        isNeg = true;
    }

    else
    {
        calcText.Text = calcText.Text.Substring(1);
        isNeg = false;
    }
}
```

The decimal button switches focus from the integer half of the value to the decimal half.

The negative button converts the value to its negative counterpart and displays the negative sign in fromt of the value within the textbox.

```
private void trigButton_Click(object sender, EventArgs e)
{
    isRad = !isRad;
}

private void clrButton_Click(object sender, EventArgs e)
{
    reset();
}
```

The Deg/Rad button switches between radians and degrees.

The clear button resets the values in the calculator.

```
private void plusButton_Click(object sender, EventArgs e)
        {
            setOper(PLUS);
        }

private void sinButton_Click(object sender, EventArgs e)
        {
            if (isRad)
                setOper(RAD_SIN);
            else
                setOper(DEG_SIN);
        }
```

Each of the operation buttons (+, ^{2}, sin) set the operation integer as their respective constants above.

The trig operations will use different constants depending on whether the calculator is set to degrees or radians.

```
private void equalButton_Click(object sender, EventArgs e)
        {
            if (!willRepeat)
            {
                num2Dec /= Math.Pow(10, decPlace);

                num1 = num1Int + num1Dec;
                num2 = num2Int + num2Dec;
                willRepeat = true;
            }
            dOper.setOperation(num1, oper, num2);
            sOper.setOperation(num1, oper, num2);

            if (oper < SQR)
                result = handler.executeCommand(dOper);

            else
                result = handler.executeCommand(sOper);

            calcText.Text = result + "";
            num1 = result;
        }
```

The equal button…

1. combines the haves of each double together
2. sets the repeat Boolean to true so the function will repeat with consecutive clicks of the equal button
3. inputs the doubles and operation integer into the operation classes
4. uses the handler to add the operations to its list and perform the operation
5. displays the result

```
private void histButton_Click(object sender, EventArgs e)
        {
            new Form2(handler).Show();
        }

public partial class Form2 : Form
    {
        List<string> list;

        public Form2(Handler h)
        {
            InitializeComponent();
            list = h.getList();

            for (int i = 0; i < list.Count; i++)
            {
                histList.Items.Add(list[i]);
            }
        }
    }
```

The history button opens and shows the second form, which displays every operation performed in a listbox.

Reflection

This is the first time I have purposefully combined more than one design pattern, so I tried to make it simple to create; I failed in that regard, as it was very frustrating to get the program working to a point that I could type a paper about it.

On a side note, I am now very used to coding for 7 hours almost non-stop.