Command Pattern and Adapter Pattern

Design Patterns

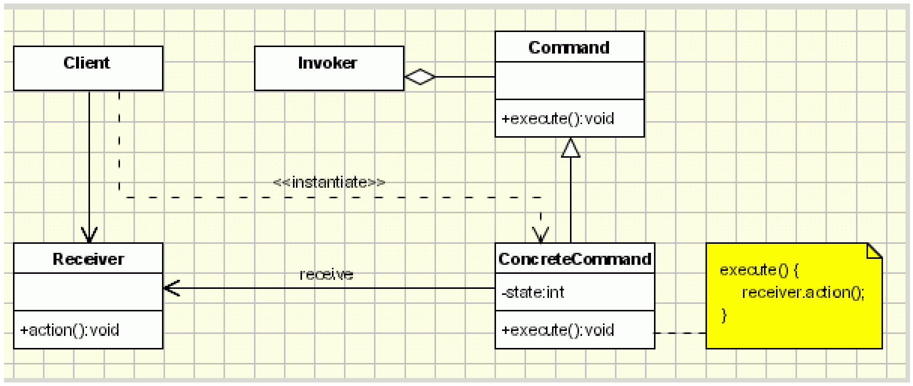
Christopher Doseck

10/7/2016

Introduction

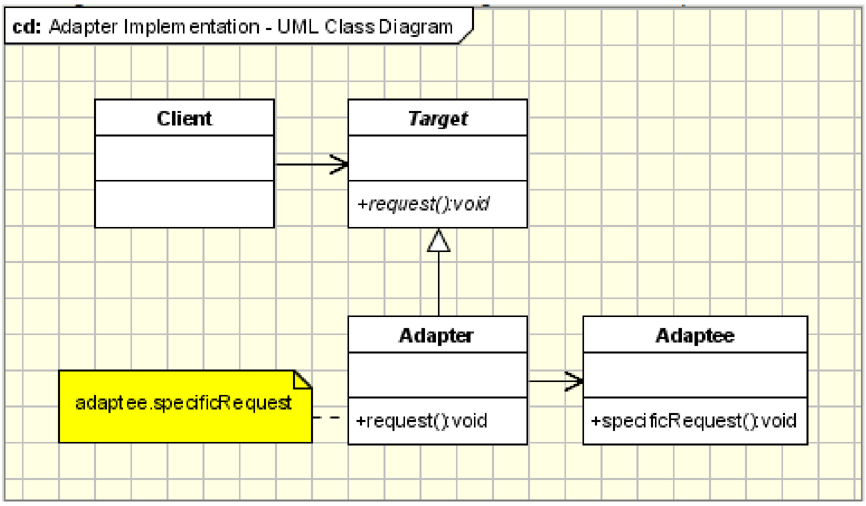
This assignment is an application that I created to show how the command pattern and the adapter pattern work. In this application I am adapting a class that changes a string into an integer into a class that changes a string into a number. I am also using the command pattern to add that number to a text box, and having it able to be undone and redone.

The UML Diagram for Command Pattern



The UML Diagram for the command pattern, shown on the right, shows the classes that are needed to have the requirements. I had the command interface as the command interface, the concrete command as the concrete command, and the rich text box as the receiver.

The UML Diagram for Adapter Pattern



The UML Diagram for the adapter pattern, shown on the right, shows the classes that are needed to have the requirements. I used the IStringToInt class as the interface of the adaptee, and StringToInt as the concrete class of the adaptee, which inherits from the IStringToInt. The adapter class is implemented in the NumberAdapter class, inheriting from the StringToNumber class, which adapts the StringToInt class into a StringToNumber interface. The NumberGetter class inherits the StringToNumber class, and has an object of the NumberAdapter class, which makes it the Target class in the diagram. Below is a table with the descriptions of each of the classes.

Form1	This is the form in which all of the changes are shown via a GUI.
Command	This is an interface for the command pattern, used to show what methods are needed.
Concrete Command	This class is the class for commands, it has an execute method and an undo method. It uses the NumberGetter class to change a string into a number.
IStringToInt	This class is the adaptee interface in the adapter pattern. It is used to sho w what methods are needed in child classes.
StringToInt	This class is the concrete adaptee class. It converts a string into and int through a method with the signature in the IStringToInt interface, which it inherits from.
StringToNumber	This is the interface used by the target and the adapter. This is what the IStringToInt and StringToInt are getting adapted into.
NumberAdapter	This is a class that inherits from the StringToNumber interface. It contains an object of the StringToInt class, which is stored in an interface of IStringToInt. This is the class that adapts the IStringToInt interface into the StringToNumber interface. This is the adapter class in the UML Diagram.
NumberGetter	This class also inherits from the StringToNumber interface. This class sets up the rest of the method that is in StringToNumber. It represents the target class in the UML Diagram.

Narrative

```
interface Command
{
    void execute();
    void undo();
}
```

This is the command interface. It has and execute and undo function.

```
public class ConcreteCommand : Command
{
    private double addNum;
    private double previousNum;
    private string previousText;
    private RichTextBox textBox;
    private NumberGetter numGetter = new NumberGetter();
    public ConcreteCommand(RichTextBox tb, string text)
    {
        this.textBox = tb;
```

This is the concrete command which inherits from the command interface. It stores the number to be added, the previous number, and the previous text from the textbox, as well as the textbox. When the constructor gets called, all of this information is stored. The NumberGetter class is used to turn the string into a number, in this case a double.

```
        this.previousNum = numGetter.getNumber(tb.Text, tb.Text.Contains("."));
        this.addNum = numGetter.getNumber(text, text.Contains("."));
        this.previousText = tb.Text;
    }

    public void execute()
    {
        textBox.Text = (previousNum + addNum).ToString();
    }

    public void undo()
    {
        textBox.Text = previousNum.ToString();
    }
}
```

When it is executed, it prints out the previous number from the box added to the number to be added to that number.

When the undo method is executed, it puts the text in the box back to the previous number.

```
public interface IStringToInt
{
    int getInt(string text);
}

public interface StringToNumber
{
    double getNumber(string text, bool hasPeriod);
}
```

These are the two interfaces for the adapter pattern, they are described more in the table in the previous section.

```
public class StringToInt : IStringToInt
{
    public int getInt(string text)
    {
        return Int32.Parse(text);
    }
}
```

This is the StringToInt class, as described in the table above. It implements the getInt() method, by parsing the text and returning the int from that.

```
public class NumberAdapter : StringToNumber
{
    IStringToInt strToInt = new StringToInt();

    public double getNumber(string text, bool hasPeriod)
    {
        if (!hasPeriod)
        {
            return strToInt.getInt(text);
        }
        else
            return 0;
    }
}
```

This is the NumberAdapter class which serves as the Adapter Class. It inherits from the StringToNumber interface. It has an object of the StringToInt class, typed as its parent. The NumberAdapter implements the StringToNumber method of getNumber by returning the IStringToInt method of getInt(), if hasPeriod is false.

```
public class NumberGetter : StringToNumber
{
    NumberAdapter numAdapt = new NumberAdapter();
    public double getNumber(string text, bool hasPeriod)
    {
        if (hasPeriod) {
            return Double.Parse(text);
        }
```

NumberGetter inherits from the StringToNumber interface. It has an object of the NumberAdapter class. When it implements the getNumber() method from StringToNumber, if hasPeriod is true, then it parses for a double and returns that. If it is not true, then it calls the getNumberMethod from the NumberAdapter class.

```
            else
            {
                return numAdapt.getNumber(text, hasPeriod);
            }
        }
    }

    public partial class Form1 : Form
    {
        private Stack<ConcreteCommand> undoCommands = new Stack<ConcreteCommand>();
        private Stack<ConcreteCommand> redoCommands = new Stack<ConcreteCommand>();

        public Form1()
        {
            InitializeComponent();
            btnUndo.Enabled = false;
            btnRedo.Enabled = false;
            tbAllText.Text = 0.ToString();
        }

        private void btnAddText_Click(object sender, EventArgs e)
        {
            undoCommands.Push(new ConcreteCommand(tbAllText, tbTextInput.Text));
            undoCommands.Peek().execute();
            enableButtons();
        }

        private void btnUndo_Click(object sender, EventArgs e)
        {
            ConcreteCommand command = undoCommands.Pop();
            redoCommands.Push(command);
            command.undo();
            enableButtons();
        }

        private void btnRedo_Click(object sender, EventArgs e)
        {
            ConcreteCommand command = redoCommands.Pop();
            undoCommands.Push(command);
            command.execute();
            enableButtons();
        }

        private void enableButtons()
        {
            if (undoCommands.Count == 0)
                btnUndo.Enabled = false;
            else
                btnUndo.Enabled = true;
            if (redoCommands.Count == 0)
                btnRedo.Enabled = false;
            else
                btnRedo.Enabled = true;
        }
    }
```

These are the stacks of undo commands and redo commands.

When the form is created, it sets the undo and redo buttons to disabled, and the textbox to have a zero in it.

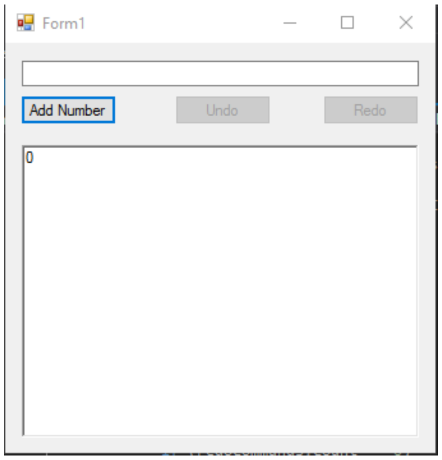
When the add text button is clicked, it adds a new command with the textbox, and the text to be added to the undo commands and then executes the command. Then it calls the enable buttons method.

When the undo button is clicked, it pops the command off the undo button stack and pushes it onto the redo stack, then calls the undo method. It then calls the enable buttons method.

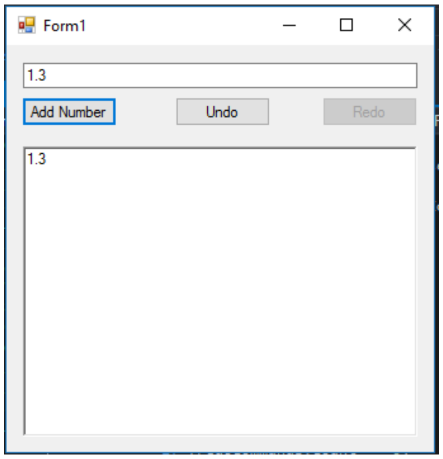
When the redo button is clicked, it pops off the command from the redo stack and pushes it onto the undo stack. It then calls the enable buttons method.

The enable buttons method checks to see if there are any commands in each stack, if there are, it enables the button for that stack, if not, it disables the button for that stack.

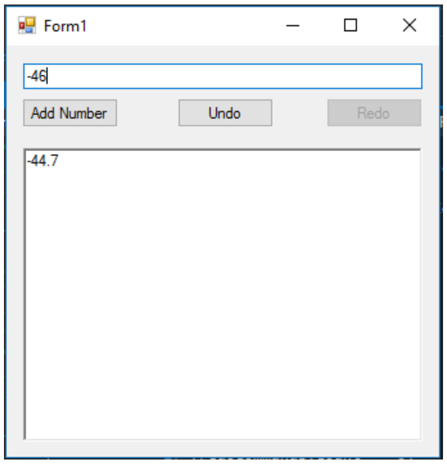
This is the initial setup, notice both buttons are disabled.



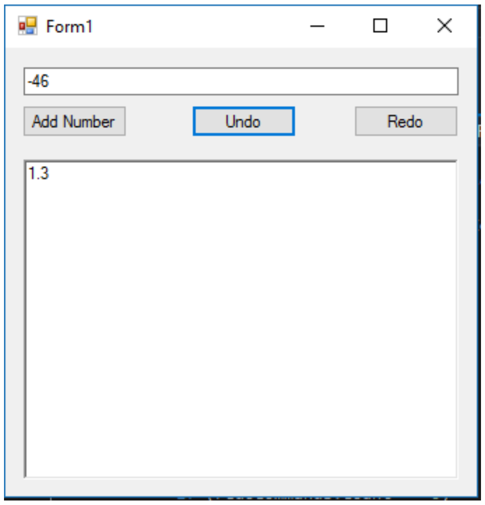
1.3 was added, undo button has become enabled.



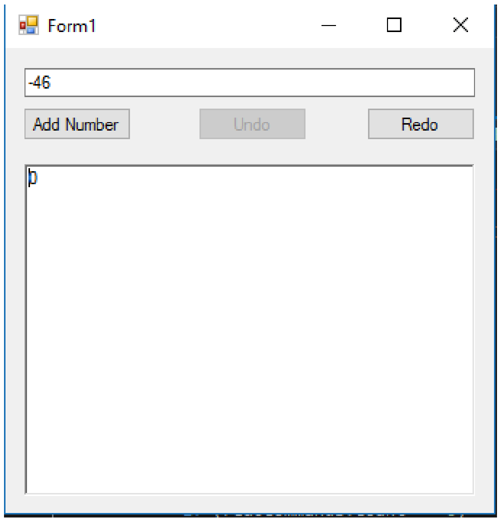
-46 was added.



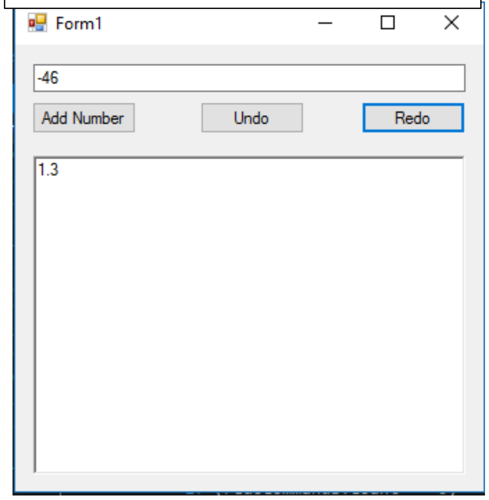
Undo was pressed, redo button has been enabled.



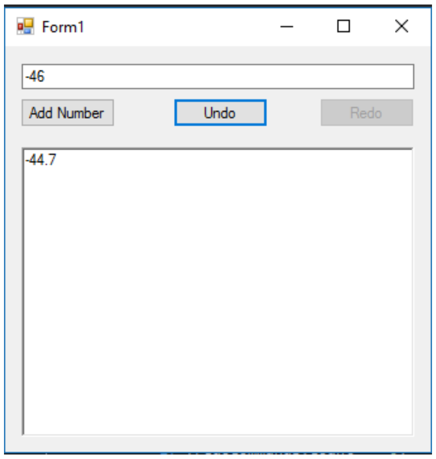
Undo pressed again, undo button disabled because stack is empty.



Redo pressed, undo button enabled again.



Redo button pressed again, no longer enabled because stack is empty.



Observations

I really enjoyed doing the command pattern part of this assignment. The command pattern was easy to do, but was extremely useful. I did not like having to do two patterns in one assignment. I thought that it took away from learning each of the patterns some by trying to get them to work together.