Design Patterns
Prof. Retterer

Jeremy Martin
10/6/16

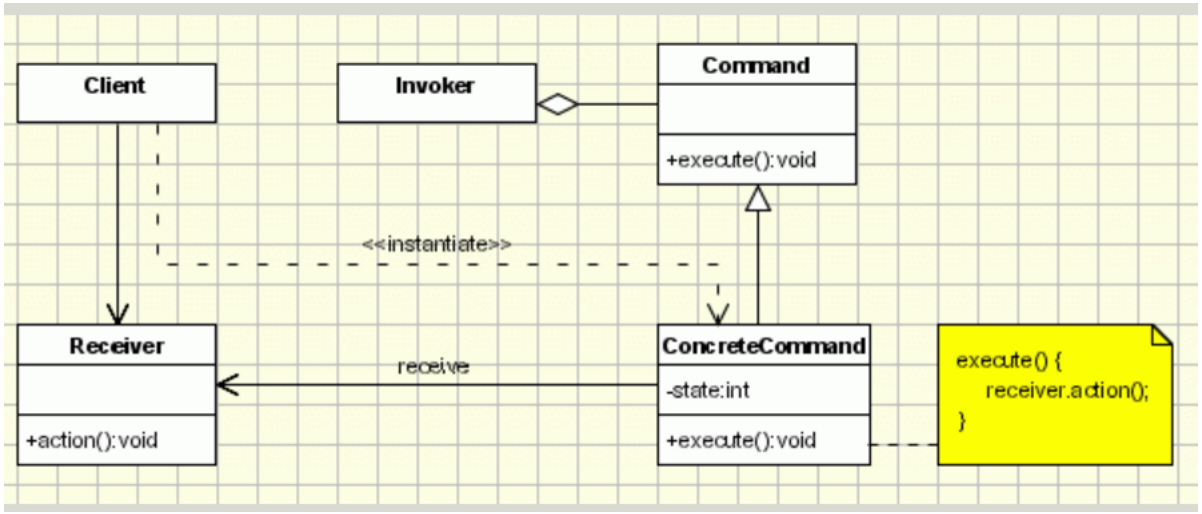
Command Pattern

Introduction

For this project we were told to make a program that demonstrated the Command pattern. For my project, I decided to make a program that allowed you to redo a move that you made chess, essentially simulating that you never took your hand off the piece in real life.

UML Diagram

To the right you will see the UML diagram for the Command pattern.



Now I will show you the code that was used to make my project work the way that it does.

The first pieces of code deal with the Command and ConcreteCommand classes.

```
public abstract class Command
    {
        public abstract void execute(string piece,string start,string end);
        public abstract void unexecute(string piece, string start);
    }

public class ConcreteCommand : Command
    {
        Reciever rec = new Reciever();

        public override void execute(string piece,string start,string end)
        {
            rec.action(piece, start, end);
        }

        public override void unexecute(string piece,string start)
        {
            rec.redo(piece, start);
        }
    }
```

The next piece of code comes from the Receiver class.

```
public class Reciever
```

```
    {
        Form2 f2 = new Form2();

        public void action(string piece,string start,string end)
        {
            string MovePiece = "The " + piece + " has been moved to " + end + "
from " + start + Environment.NewLine;
            f2.m_tbLog.Text += MovePiece;
            f2.Visible = true;
        }

        public void redo(string piece,string start)
        {
           string RemovePiece = "The " + piece + " has been returned to " + start
+ Environment.NewLine;
           f2.m_tbLog.Text += RemovePiece;
           f2.Visible = true;
        }
    }
```

I ended up making 2 forms for this project but the second form is only used to display the text box for the movement log, therefore there isn't any code for it. The last piece of code deals with Form1.

```
public partial class Form1 : Form
    {
        string Piece;
        string Start;
        string End;
        ConcreteCommand com;

        public Form1()
        {
            InitializeComponent();
            com = new ConcreteCommand();
        }

        private void m_btnMove_Click(object sender, EventArgs e)
        {
            Piece = m_tbPiece.Text;
            Start = m_tbStart.Text;
            End = m_tbEnd.Text;
            com.execute(Piece,Start,End);
        }

        private void m_tbUndo_Click(object sender, EventArgs e)
        {
            Piece = m_tbPiece.Text;
            Start = m_tbStart.Text;
            com.unexecute(Piece, Start);
        }

        private void m_tbRedo_Click(object sender, EventArgs e)
        {
            Piece = m_tbPiece.Text;
            Start = m_tbStart.Text;
            End = m_tbEnd.Text;
            com.execute(Piece, Start, End);
```

```
        }
    }
```

Conclusion

Overall, this was a pretty easy pattern to implement. There weren't any really tricky parts that I had trouble dealing with. The adapter pattern is the next pattern that we have to deal with, unfortunately I couldn't find a way to implement it into this code anywhere, there wasn't any need for it in this code.