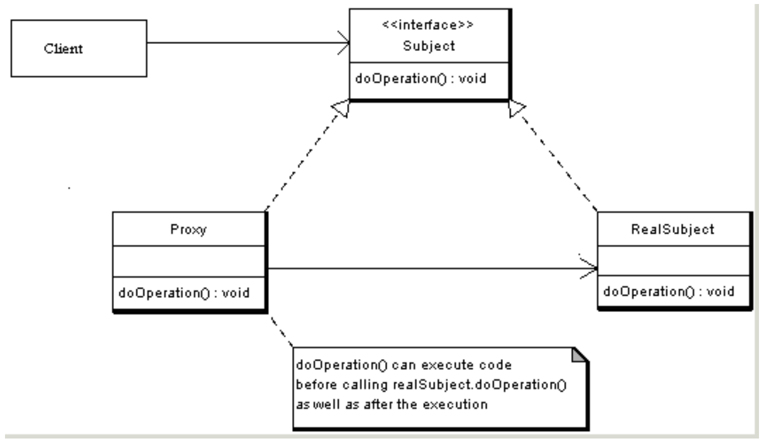
Corey McLaughlin Design Patterns 11/1/16

Introduction

The assignment this time was to demonstrate the proxy pattern. Which means I needed to create an app that goes to the proxy class which goes to a different location to get an answer. I did this in the form of making a Pokémon battle simulator. Player one and two picks a type from a dropdown list and then the simulator tells who would win in the hypothetical battle. The UML diagram to the left will show the structure I followed.


Client
<<interface>>
Subject
doOperation() : void
Proxy
doOperation() : void
RealSubject
doOperation() : void
doOperation() can execute code before calling realSubject.doOperation() as well as after the execution

The Code

```
  public abstract class Types
   {
        public abstract bool Results( string Player1, string Player2);

   }
```

Types is analogous to the subject class that is shown on the UML. Its purpose is to create the doOperation, or in this case, Results.

```
 public class RealTypes : Types
   {
        public override bool Results(string Player1, string Player2)
        {
            if((Player1 == "Fire" && Player2 == "Grass") ||
               (Player1 == "Water" && Player2 == "Fire") ||
               (Player1 == "Grass" && Player2 == "Water"))
            {
                return true;
            }
            return false;

        }
   }
```

RealTypes is my RealSubject class. It pulls from the Subject class to get the function and returns either true or false depending on the criteria.

```
public class Proxy : Types
   {
        RealTypes _realTypes = new RealTypes();
        public override bool Results(string Player1,
string Player2)
        {
            return _realTypes.Results(Player1, Player2);
        }
   }
```

This is my Proxy class. This is the class that the client will actually go to for the results of the hypothetical battle. Then the proxy will go to RealTypes, which in turns goes to Types. Then all the returns are given to the form.

```
public partial class Form1 : Form
    {

        Proxy proxy = new Proxy();

        public Form1()
        {
            InitializeComponent();
            cbPlayer1.Items.Insert(0, "Fire");
            cbPlayer1.Items.Insert(1, "Water");
            cbPlayer1.Items.Insert(2, "Grass");
            cbPlayer2.Items.Insert(0, "Fire");
            cbPlayer2.Items.Insert(1, "Water");
            cbPlayer2.Items.Insert(2, "Grass");
        }

        private void btnGo_Click(object sender, EventArgs e)
        {
            if (cbPlayer1.Text == "Pick A Type")
            {
                MessageBox.Show("PICK A TYPE YOU UNDERSIZED BANANA HAMMOCK");

            }
            else if (cbPlayer2.Text == "Pick A Type")
            {
                MessageBox.Show("PICK A TYPE YOU UNDERSIZED BANANA HAMMOCK");
            }
            else if (cbPlayer1.SelectedIndex.Equals(cbPlayer2.SelectedIndex))
            {
                txtResults.Text = "Well this is awkward";
            }
            else
            {
                if(proxy.Results(cbPlayer1.Text,cbPlayer2.Text))
                {
                    txtResults.Text = "Player One would win.";
                }
                else
                {
                    txtResults.Text = "Player Two would win.";
                }
            }

        }
    }
```

This is the form. Here I populate the list and make if statements that check to make sure that a choice is made. It also checks to see if a draw occurs. If neither of these things happen, then it checks to see if the win conditions set by RealTypes are met via the Proxy class.

Observations

This pattern was pretty simple once I read up on it on the internet. I can't see when I would use it normally. But I can see why it is useful to know about it and how to use it. Also, I apologize for having message boxes yell at the user. I was full of angst. Getting the form to realize that a selection wasn't made was hell.