Nathan Bemus

Design Patterns

Command Pattern / Adapter Pattern

October 7, 2016

Command Pattern / Adapter Pattern

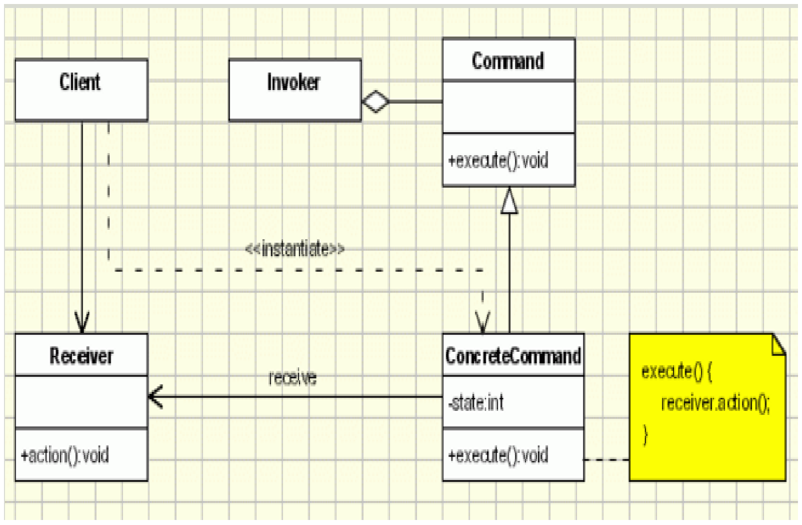
Introduction:

The purpose of this assignment was to demonstrate the command pattern and the adapter pattern. The command pattern

According to oodesign.com, it states that command design pattern encapsulates commands in objects allowing us to issue requests without knowing the requested command or object. My program demonstrates this by changing the font color with a RandomColor command.

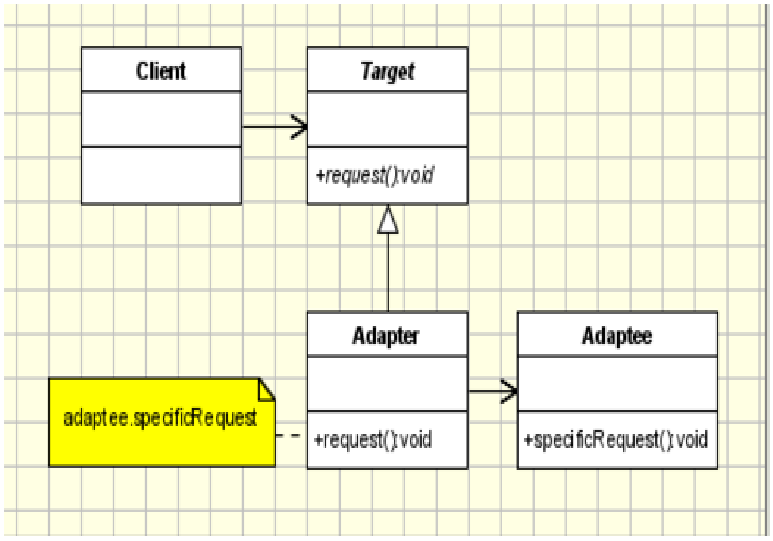
According to oodesign.com, it states that the intent is to convert the interface of a class into another interface that is expected. My program demonstrates this by allowing the user change the font type of the text box after typing in one of the key numbers and clicking a button.

UML Diagram Analysis (Command):



This diagram shows that client class calls upon the Receiver class and the ConcreteCommand class. The ConcreteCommand class gets the command from the Command class and returns it to the Receiver class. The Invoker class asks the Command class to carry out the requested command.

UML Diagram Analysis (Adapter):



The diagram shows that the client class calls upon the Target class which uses a request function. The Target defines the domain specific interface that the Client class uses. The Adapter class adapts the Adaptee class interface to the Target Class interface.

Program Walkthrough

Command Class

```
public abstract class Command
    {
        public abstract void Execute();
    }
```

This section declares an interface for creating a command. It defines a public abstract class that has one function. The function is a public abstract void that is called Execute.

ConcreteRandom Class

```
public class ConcreteRandom : Command
    {
        private Receiver _receiver;

        public ConcreteRandom(Receiver receiver)
        {
            _receiver = receiver;
        }

        public void SetCommand(Receiver receiver)
        {
            this._receiver = receiver;
        }

        public override void Execute()
        {
            _receiver.RandomColor();
        }
    }
```

The class starts off with calling command through inheritance. It then declares a private object called _receiver. It then creates a constructor.

Receiver Class:

```
public class Receiver
    {
        public Color fontcolor = System.Drawing.Color.Black;
        public void RandomColor()
        {
            Random random = new Random();
            int randomNumber = random.Next(0, 7);

            switch (randomNumber)
            {
                case 0:
                    fontcolor = System.Drawing.Color.Red;
                    break;
                case 1:
                    fontcolor = System.Drawing.Color.Orange;
                    break;
                case 2:
                    fontcolor = System.Drawing.Color.Yellow;
                    break;
                case 3:
                    fontcolor = System.Drawing.Color.Green;
                    break;
                case 4:
                    fontcolor = System.Drawing.Color.Blue;
                    break;
                case 5:
                    fontcolor = System.Drawing.Color.Pink;
                    break;
                default:
                    fontcolor = System.Drawing.Color.Black;
```

The class initializes a Color object and calls it fontcolor. The class then creates a random number generator and that creates a number between 0 – 6. Then there is a switch case statement that determines what color the font is.

```
                    break;
                }
            }
```

Invoker Class:

```
public class Invoker
    {
        private Command _command;

        public void SetCommand(Command command)
        {
            this._command = command;
        }

        public void ExecuteCommand()
        {
            _command.Execute();
        }
    }
```

This class asks the command class to carry out the request. The class initializes a private Command object called _command. It then has two functions called SetCommand and ExecuteCommand. The function SetCommand sets the command object to the command selected by the user. The ExecuteCommand class invokes the command to fire.

Target Class:

```
public abstract class Target
    {
        public Font font;
        public abstract void Request(int number);
    }
```

This class creates a font object called font. Then the class calls a function called request.

Adapter Class:

```
public class Adapter : Target
    {
        public Font fonttype;
        public override void Request(int adaptee)
        {
            switch (adaptee)
            {
                case 0:
                    fonttype = new Font("Times New Roman", 12.0f);
                    break;
                case 1:
                    fonttype = new Font("Courier New", 12.0f);
                    break;
                case 2:
                    fonttype = new Font("Comic Sans", 12.0f);
                    break;
                case 3:
                    fonttype = new Font("Georgia", 12.0f);
```

This class inherits the functions from Target. The function uses a switch case to determine which font type to change to. The switch case uses the value passed in from the Target Class

```
                    break;
                case 4:
                    fonttype = new Font("Castellar", 12.0f);
                    break;
                case 5:
                    fonttype = new Font("Wingdings", 12.0f);
                    break;
                default:
                    fonttype = new Font("Calibri", 12.0f);
                    break;

            }
        }
```

Form1 (Main Class):

After initializing the components I create two button click events. The first event is for the random text color. This event declares the classes and the invoker object calls the SetCommand and the ExecuteCommand. After the command is executed the textbox is updated. The Change font click event is set up similarly in the form but is executed differently in the classes. The text in the textbox is converted into an integer and the textbox is updated to display the new font.

```
    public partial class Form1 : Form
    {

        public Form1()
        {
            InitializeComponent();
        }

        private void btnColorize_Click(object sender, EventArgs e)
        {
            Receiver receiver = new Receiver();
            ConcreteRandom random = new ConcreteRandom(receiver);
            Invoker invoker = new Invoker();
            invoker.SetCommand(random);
            invoker.ExecuteCommand();
            tbxColor.ForeColor = receiver.fontcolor;
        }

        private void btnChangeFont_Click(object sender, EventArgs e)
        {
            Adaptee adaptee = new Adaptee();
            Adapter adapter = new Adapter();
            adapter.Request(Int32.Parse(tbxColor.Text));
            tbxColor.Font = adapter.fonttype;
        }
    }
```

Conclusion:

The program is an example of both the command pattern and the adapter pattern. Adapter pattern changes the interface to what the user wants, and the command pattern encapsulates a command in an object to keep the contents intact from the user.

Reflection:

Overall this was an interesting program to write and was semi enjoyable to code. The hardest part for this assignment was to come up with an idea that fit both parameters of the command pattern and the adapter pattern and finding tiny issues. I think I have a general understanding on how to use both patterns but I still am not sure about the adapter pattern.