Strategy Pattern

Design Patterns

Christopher Doseck

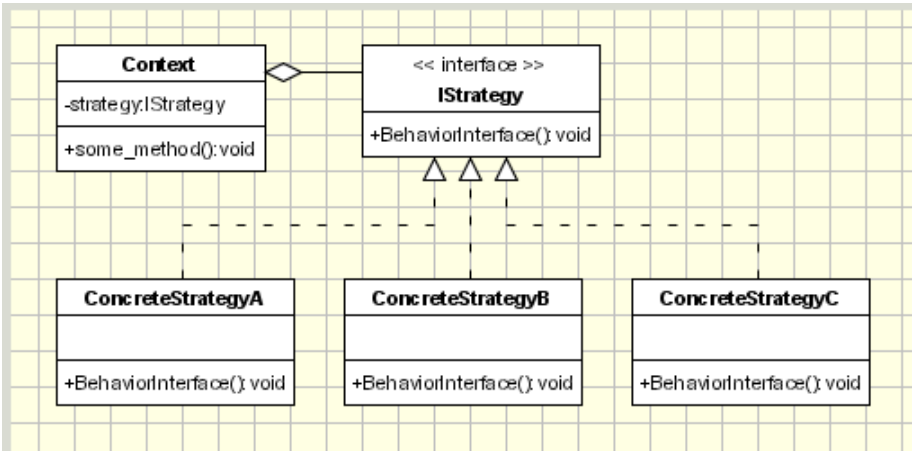
10/21/2016

Introduction

This assignment is an application that I created to show how the strategy pattern works. In this application I am using mathematical operations to represent the strategy. I use add, subtract, multiply, divide, and exponential to represent the concrete strategy.

The UML Diagram for Strategy

The UML Diagram for the strategy pattern, shown on the right, shows the classes that are needed to have the requirements. I had the Operations class as the Strategy interface, each of the different mathematical operations as concrete classes, and the Operate class as the Context class. The table below shows how all of the classes were used.



Operation	This is the interface that all of the operations inherit from.
Add	This class is a concrete strategy that represents addition. It inherits from the operation interface.
Subtract	This class is a concrete strategy that represents subtraction. It inherits from the operation interface.
Multiply	This class is a concrete strategy that represents multiplication. It inherits from the operation interface.
Divide	This class is a concrete strategy that represents division. It inherits from the operation interface.
Exponent	This class is a concrete strategy that represents exponentials. It inherits from the operation interface.
Operate	This class uses the strategy interface and makes it so that one of the concrete strategies is used.
Form1	This is the client that shows the strategy pattern in action.

Narrative

```
public interface Operation
{
    double doOperation(double a, double b);
}

public class Add : Operation
{
    public double doOperation(double a, double b)
    {
        return a + b;
    }

    public override string ToString()
    {
        return "Plus";
    }
}

public class Subtract : Operation
{
    public double doOperation(double a, double b)
    {
        return a - b;
    }

    public override string ToString()
    {
        return "Minus";
    }
}

public class Multiply : Operation
{
    public double doOperation(double a, double b)
    {
        return a * b;
    }

    public override string ToString()
    {
        return "Times";
    }
}

public class Division : Operation
{
    public double doOperation(double a, double b)
    {
        return a / b;
    }

    public override string ToString()
    {
        return "Divided By";
    }
```

This is the interface that all of the operations inherit from.

These are all of the concrete strategies. They all extend from the Operation interface. All of them override the doOperation() method and do their mathematical operation. They also override the ToString() method to return what operation they do.

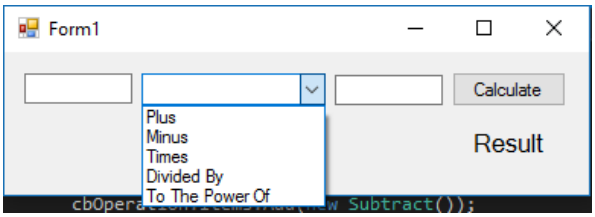
```
}

public class Exponent : Operation
{
    public double doOperation(double a, double b)
    {
        return Math.Pow(a, b);
    }

    public override string ToString()
    {
        return "To The Power Of";
    }
}

public class Operate
{
    Operation operation;

    public double calculate(double a, double b)
    {
        return operation.doOperation(a, b);
    }

    public void setOperation(Operation op)
    {
        this.operation = op;
    }
}
```

This class uses the Operation interface to be able to call the doOperation() method without having to know what child's method is being used. The setOperation() method sets the operation interface to a specific child class. The calculate() method does the operation's doOperation() method. This will get called after the setOperation() so that the child is known.

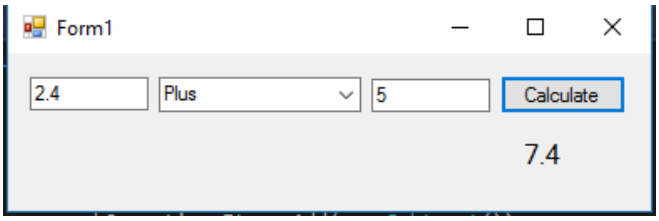
```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        cbOperation.Items.Add(new Add());
        cbOperation.Items.Add(new Subtract());
        cbOperation.Items.Add(new Multiply());
        cbOperation.Items.Add(new Division());
        cbOperation.Items.Add(new Exponent());
    }

    private void btnCalculate_Click(object sender, EventArgs e)
    {
        Operate calculation = new Operate();
        calculation.setOperation((Operation)cbOperation.SelectedItem);
        double numOne = Double.Parse(tbNumOne.Text);
        double numTwo = Double.Parse(tbNumTwo.Text);
        double result = calculation.calculate(numOne, numTwo);
        labelResult.Text = result.ToString();
    }
}
```

This populates the combo box with one of each of the children of the strategy interface.

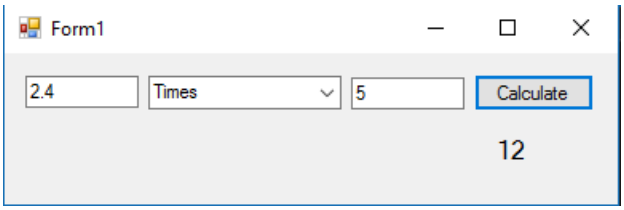
This is executed when the calculate button is pressed. It creates a new Operate. Then it calls the setOperation() method and passes the selected Operation. After that, it calls the calculate() method and passes the numbers from the two numbers from the text boxes and puts the result in a label.



Initial setup of the form.



Two numbers added together.



Two numbers multiplied together.

Conclusion

I found this project to be rather easy. There were no large struggles that I had when writing this program. The strategy pattern is very useful because it allows you to define a family of algorithms, encapsulate them, and make them interchangeable. This allows you to treat the same method in different classes, which do different things, in the same way.