

## The Template Method Pattern

### Design Patterns

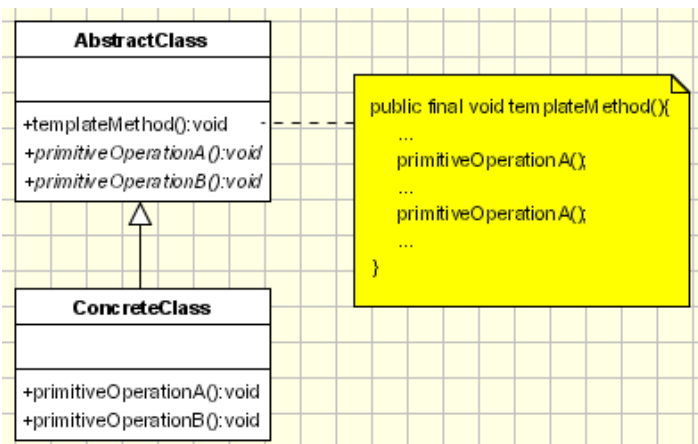
Andrey Grzegorzewski – Ohio Northern University

Fall 2016-2017

### Introduction

This assignment requires an application that makes use of the template method pattern. This pattern is used when a method has a similar structure every time it is used, but one or more parts of its implementation vary depending on the circumstances. My app finds the root mean square value of a waveform given its expression, start time, and end time. Calculating the root mean square involves the same basic procedure every time: the RMS value is equal to the square root of one over the time period times the integral of the square of the wave. However, the process of finding the integral of the square of the expression varies depending on the type of expression.

### The UML Diagram



To the left is the UML diagram for the template method pattern, courtesy of oodesign.com. The diagram shows an abstract class containing multiple methods. The templateMethod is concrete and calls the abstract primitiveOperation methods in its implementation. The primitiveOperations are defined elsewhere in concrete classes that inherit from the abstract class.

These methods are implemented differently based on the needs of the individual concrete classes. Below is a table showing the classes I used to implement this pattern.

<i>AbstractClass</i>	I have two abstract classes; the first is called Waveform, and the second is called TrigWaveform. Waveform has a concrete template method called getRMS, while TrigWaveform inherits from Waveform and has two different template methods. They both share an abstract primitive operation called getIntegral, which is defined in the concrete classes.
ConcreteClass	My concrete classes are SinWaveform, CosWaveform, and LinearWaveform. Each of these classes implements the getIntegral method differently. SinWaveform and CosWaveform inherit from TrigWaveform, while LinearWaveform inherits only from Waveform.

## Narrative and Code

I began working on this app by designing the form, which is shown to the left. In the form, the user can insert a waveform expression, a start time, and an end time into three separate text boxes, then click the Calculate button. Upon clicking the button, the root mean square value will appear at the bottom right of the form. Acceptable input formats for the waveform expression include  $n \cdot x$ ,  $n \cdot \cos(m \cdot x)$ , and  $n \cdot \sin(m \cdot x)$ , where  $m$  and  $n$  are

optional decimal or integer coefficients.

Once I created my form, I wrote the code for the Waveform class. It contains the variables necessary to calculate the RMS value as well as a constructor, the primitive operation, and the template method.

```
public abstract class Waveform // This is an AbstractClass
{
    protected double startPoint; // The start time of one period
    protected double endPoint;   // The end time of one period
    protected string expression; // The expression of the wave

    public Waveform(double startPoint, double endPoint, string expression)
    {
        this.startPoint = startPoint;
        this.endPoint = endPoint;
        this.expression = expression;
    }

    // This is the primitive operation.
    // It returns the definite integral of the square of the expression.
    public abstract double getIntegral();

    public double getRMS() // This is the template method
    {
        // The equation for the RMS value is the square root of one over
        // the time interval times the integral of the square of the expression.
        double integral = getIntegral();
        return Math.Sqrt(integral / (endPoint - startPoint));
    }
}
```

Next, I created my first concrete class, called LinearWaveform. This class contains a constructor and a definition of the getIntegral method.

```
public class LinearWaveform : Waveform // This is a ConcreteClass
{
    public LinearWaveform(double startPoint, double endPoint, string expression)
        : base(startPoint, endPoint, expression) { }

    public override double getIntegral()
    {
        double coef = 1;

        // Get the coefficient - for example, in 2*x, get 2
        if (expression.Contains("*"))
        {
            int length = expression.Length - 2;
            coef = Double.Parse(expression.Substring(0, length));
        }

        coef *= coef;

        // The integral of the square of a linear expression is: coef * x^3 / 3
        return coef / 3 * ((endPoint * endPoint * endPoint) - (startPoint *
            startPoint * startPoint));
    }
}
```

Then, I started to work on the trigonometric portion of the code. I created another AbstractClass that inherits from Waveform and called it TrigWaveform. It defines two template methods that will be used by both CosWaveform and SinWaveform. It does not create a prototype for the primitive operation because the prototype has already been created in the Waveform class. Comments explain the function of each method.

```
public abstract class TrigWaveform : Waveform // This is an AbstractClass
{
    public TrigWaveform(double startPoint, double endPoint, string expression)
        : base(startPoint, endPoint, expression) { }

    // This is a template method. It gets the coefficient that appears
    // before the trig function, if there is one.
    // For example, in 3*cos(4*x), this method returns 3.
    public double getCoefOutside()
    {
        double coefOutside = 1;

        // checks to see if there is an asterisk before "sin" or "cos"
        if (expression.Substring(0, expression.IndexOf('(')).Contains("*"))
        {
            int length = expression.IndexOf('*');
            coefOutside = Double.Parse(expression.Substring(0, length));

            coefOutside *= coefOutside;
        }
    }
}
```

```

        int substringIndex = expression.IndexOf('*') + 5;
        expression = expression.Substring(substringIndex, expression.Length -
            substringIndex - 1);
        // For example, changes expression from n*cos(3*x) to 3*x
    }
    else
        expression = expression.Substring(4, expression.Length - 5);

    return coefOutside;
}

// This is a template method. It gets the coefficient inside the
// parenthesis of the trig function, if there is one.
// For example, in 3*cos(4*x), this method returns 4.
public double getCoefInside()
{
    double coefInside = 1;

    if (expression.Contains("("))
        coefInside = Double.Parse(expression.Substring(0,
            expression.IndexOf('*')));

    return coefInside;
}

// The primitive operation getIntegral does not need to be listed here since
// TrigWaveform is an abstract class. The method is inherited from Waveform and
// implemented in CosWaveform and SinWaveform.
}

```

Once I wrote the code for the TrigWaveform class, I could create the SinWaveform and CosWaveform classes, which are my two remaining ConcreteClasses. These classes are very similar to one another. They both contain a constructor, call getCoefOutside and getCoefInside, and then perform the integral calculation. However, since they are both very short, both classes will be shown here for completeness.

```

public class SinWaveform : TrigWaveform // This is a ConcreteClass
{
    public SinWaveform(double startPoint, double endPoint, string expression)
        : base(startPoint, endPoint, expression) { }

    public override double getIntegral()
    {
        double coefOutside = getCoefOutside();
        double coefInside = getCoefInside() * 2;

        // Use the identity sin^2(u) = (1 - cos(2u)) / 2
        double firstTerm = 0.5 * (endPoint - startPoint);
        double secondTerm = 0.5 / coefInside * Math.Sin(coefInside * startPoint) -
            0.5 / coefInside * Math.Sin(coefInside * endPoint);

        return coefOutside * (firstTerm + secondTerm);
    }
}

```

```

public class CosWaveform : TrigWaveform // This is a ConcreteClass
{
    public CosWaveform(double startPoint, double endPoint, string expression)
        : base(startPoint, endPoint, expression) { }

    public override double getIntegral()
    {
        double coefOutside = getCoefOutside();
        double coefInside = getCoefInside() * 2;

        // Use the identity cos^2(u) = (1 + cos(2u)) / 2
        double firstTerm = 0.5 * (endPoint - startPoint);
        double secondTerm = 0.5 / coefInside * (Math.Sin(coefInside * endPoint) -
            Math.Sin(coefInside * startPoint));

        return coefOutside * (firstTerm + secondTerm);
    }
}

```

Finally, I wrote my form code. In the form code, I create a Waveform object and write the code for the Calculate button's click event handler.

```

public partial class Form1 : Form
{
    Waveform waveform;

    public Form1()
    {
        InitializeComponent();
    }

    private void calculateButton_Click(object sender, EventArgs e)
    {
        // Get the variables for the constructor
        double startPoint = Double.Parse(startTB.Text);
        double endPoint = Double.Parse(endTB.Text);
        string expression = waveformTB.Text;

        // Instantiate waveform depending on the type of expression
        if (expression.Contains("cos"))
            waveform = new CosWaveform(startPoint, endPoint, expression);
        else if (expression.Contains("sin"))
            waveform = new SinWaveform(startPoint, endPoint, expression);
        else
            waveform = new LinearWaveform(startPoint, endPoint, expression);

        // Set label text to the result
        rmsLabel.Text = String.Format("{0:N2}", waveform.getRMS());
    }
}

```

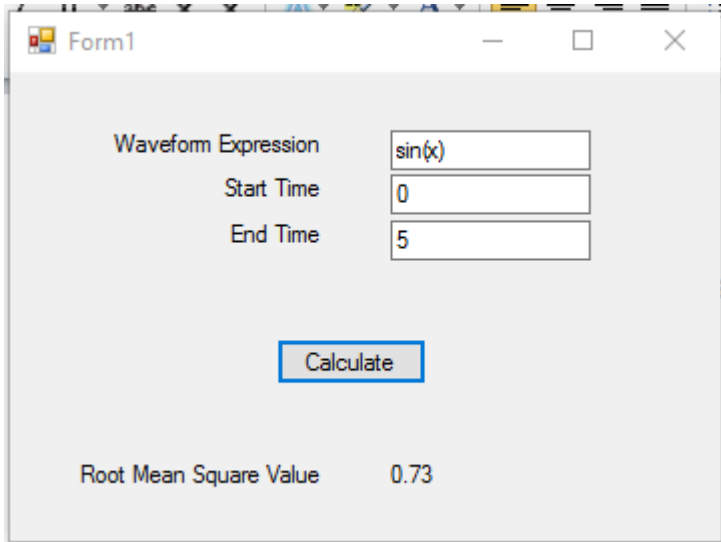
This concludes the code that I wrote for this app. On the next page are several screenshots of my app in action.

A screenshot of a Windows application window titled "Form1". The window has a standard Windows title bar with minimize, maximize, and close buttons. The main area of the form is light gray. It contains three input fields: "Waveform Expression" with the text "2\*x", "Start Time" with the text "0", and "End Time" with the text "5". Below these fields is a button labeled "Calculate". At the bottom of the form, the text "Root Mean Square Value" is followed by the value "5.77".

Here, the LinearWaveform class was used.

A screenshot of a Windows application window titled "Form1". The window has a standard Windows title bar with minimize, maximize, and close buttons. The main area of the form is light gray. It contains three input fields: "Waveform Expression" with the text "3\*cos(3\*x)", "Start Time" with the text "0", and "End Time" with the text "5". Below these fields is a button labeled "Calculate". At the bottom of the form, the text "Root Mean Square Value" is followed by the value "2.09".

Here, the CosWaveform and TrigWaveform classes were used.



Form1

Waveform Expression

Start Time

End Time

Root Mean Square Value 0.73

Finally, the SinWaveform and TrigWaveform classes were used. Here, we can also see that my app can handle calculations whether or not there are coefficients involved.

### Observations

This pattern was very easy for me to understand and I almost immediately thought of a use for it. Many things are done by repeating the same basic procedure over and over with slight differences somewhere in the middle. In fact, I could make a template pattern for many different types of problems in my Circuits class. Initially, I did not include the TrigWaveform class in my code, but I immediately saw the use for it when I started to write my CosWaveform and SinWaveform classes and saw the repetition. I thought it was very interesting to have nested template patterns; using the TrigWaveform class simplified the code a lot for my CosWaveform and SinWaveform classes and helped me further understand inheritance and the template method pattern.