Template Method Pattern Assignment

Design Patterns

Jacob Hartman

8 December 2016

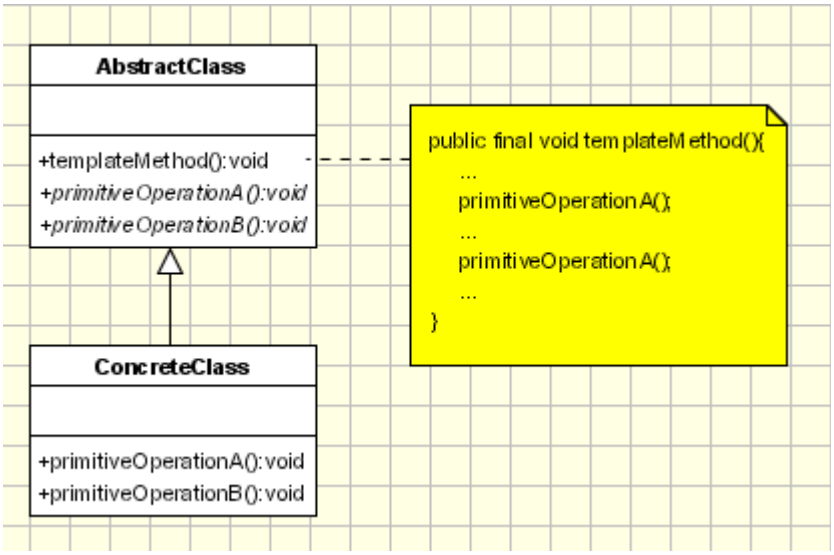
## Introduction

This assignment requires the creation of an application demonstrating the template method pattern. The template method pattern is designed to allow specific parts of an algorithm to be defined in a polymorphic manner while other parts of the algorithm remain the same.

The application I wrote for this assignment is a probability set counter. It allows the user to input a count for the input set and a number of elements to choose. It then lets the user select whether to calculate the number of possible permutations or combinations. Once all the options have been set, the result is shown to the user. Previous results are also tracked.

## UML Diagram

*image taken from http://www.oodesign.com/template-method-pattern.html*




Above is the UML diagram for the template method pattern. It shows the need for an abstract class and a concrete class which implements the abstract class. The templateMethod method is defined in the abstract class, but it uses two methods within it which are defined by the concrete class.

My abstract class is called SetCounter. In order to demonstrate the pattern's polymorphic nature, I created two concrete classes called Permutation and Combination. My template method is called calculate, and my primitive operations are getNumerator and getDenominator.

Narrative

The class I started with was the abstract class, SetCounter.

```
public abstract class SetCounter // this is the AbstractClass
{
    public int calculate(int n, int r) // this is the templateMethod
    {
        if (n < 0 || r > n)
        {
            return -1;
        }
        return getNumerator(n)/getDenominator(n, r);
    }
    public abstract int getNumerator(int n); // this is primitiveOperationA
    public abstract int getDenominator(int n, int r); // this is primitiveOperationB
```

The calculate method is defined in this class, utilizing the two primitive operations in its body. The two primitive operation method signatures are found here, but their implementation is left for later.

```
    public int factorial(int n)
    {
        if (n < 0)
        {
            return 1;
        }
        else if (n == 0)
        {
            return 1;
        }
        else
        {
            int a = n;
            n--;
            while (n > 0)
            {
                a *= n;
                n--;
            }
            return a;
        }
    }
}
```

An additional method, factorial, is defined in this class. Its use is important to the algorithm. It functions the same as the factorial function in mathematics (#!). Next, I defined the two concrete classes.

```
public class Permutation : SetCounter // this is a ConcreteClass
{
    public override int getNumerator(int n)
```

```
        {
            return factorial(n);
        }

        public override int getDenominator(int n, int r)
        {
            return factorial(n - r);
        }
    }
```

The permutation class gets the numerator and denominator based on the most commonly used algorithm for calculating the number of possible permutations.

```
    public class Combination : SetCounter // this is a ConcreteClass
    {
        public override int getNumerator(int n)
        {
            return factorial(n);
        }

        public override int getDenominator(int n, int r)
        {
            return (factorial(r) * factorial(n - r));
        }
    }
```

The combination's algorithm is very similar, but the calculation of the denominator is done differently. This version of the algorithm ignores order, focusing only on which elements are present. Lastly, I wrote code in the form to make use of the subsystems.

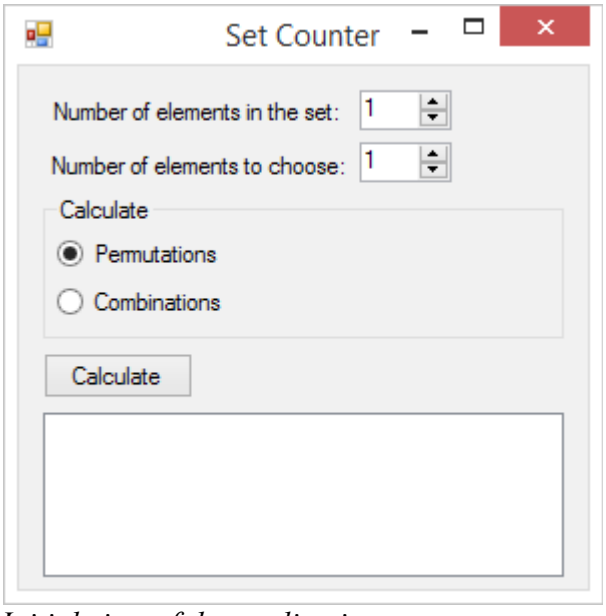
```
    public partial class Form1 : Form
    {
        SetCounter counter;
        public Form1()
        {
            InitializeComponent();
            counter = new Permutation();
        }

        private void rb_permutation_CheckedChanged(object sender, EventArgs e)
        {
            if (rb_permutation.Checked && !(counter is Permutation))
            {
                counter = new Permutation();
            }
        }

        private void rb_combinations_CheckedChanged(object sender, EventArgs e)
        {
            if (rb_combinations.Checked && !(counter is Combination))
            {
                counter = new Combination();
            }
        }
```

There are two radio buttons on the form that allow the user to select between permutations and combinations. Clicking a radio button updates the state of the counter, which is a SetCounter object. The counter object is set to the class which can calculate what the user has requested. This way, the

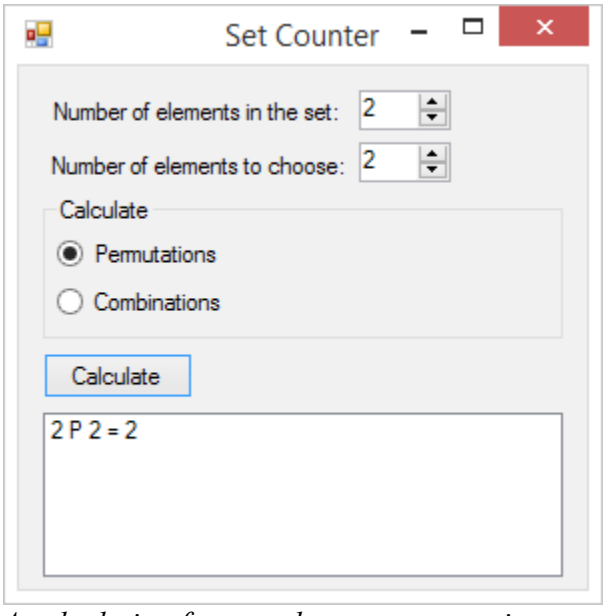
calculate method can be called in a polymorphic manner.

```
        private void btn_calculate_Click(object sender, EventArgs e)
        {
            int n = (int)spin_setcount.Value;
            int r = (int)spin_choosecount.Value;
            int result = counter.calculate(n, r);
            if (result == -1)
            {
                MessageBox.Show("The number of elements to choose must not be greater than" +
                    " the set size!");
            }
            else
            {
                string s;
                if (counter is Permutation)
                {
                    s = "P";
                }
                else
                {
                    s = "C";
                }
                string entry = "" + n + " " + s + " " + r + " = " + result;
                lb_result.Items.Insert(0, entry);
            }
        }
    }
```

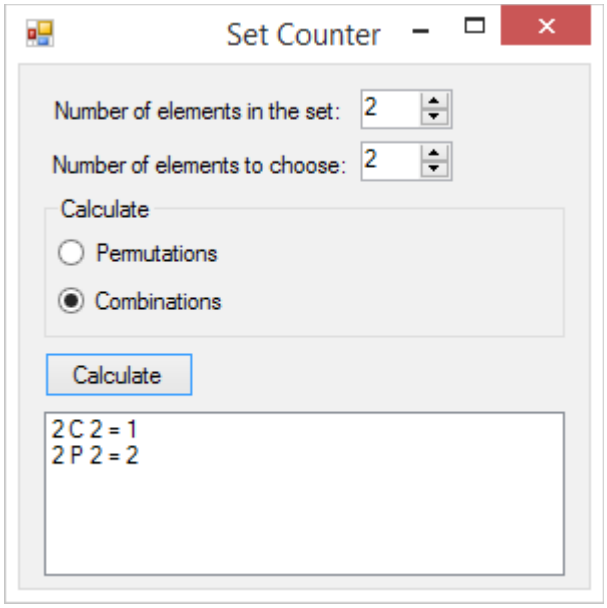
There are two numeric input boxes on the form. One holds the number of elements in the set, and the other holds the number of elements to choose. When the calculate button is clicked, these values are retrieved and the result is calculated. If the result indicates incorrect input, indicated by a return value of -1, a message box warns the user. Otherwise, a string representing the calculation is created and added to the top of a list box on the form. Below are some screenshots of the application in use.



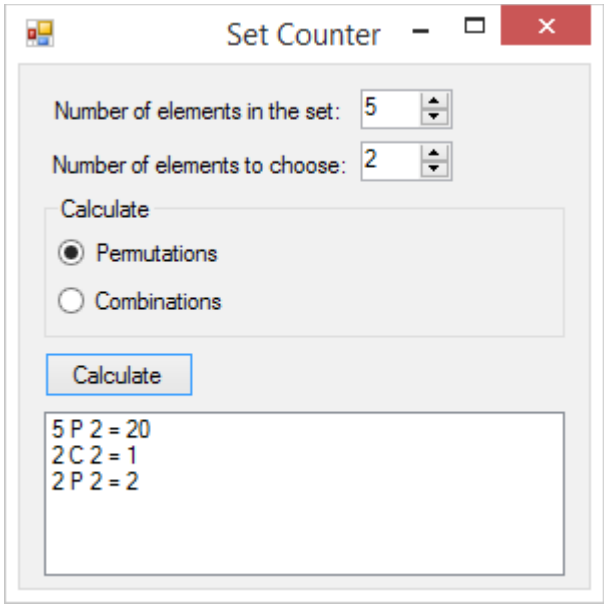

*Initial view of the application.*



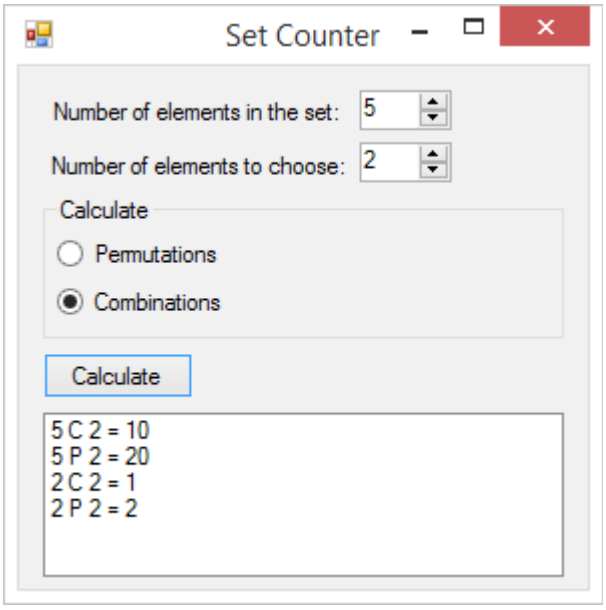

*A calculation for two element permutations from a set of two.*



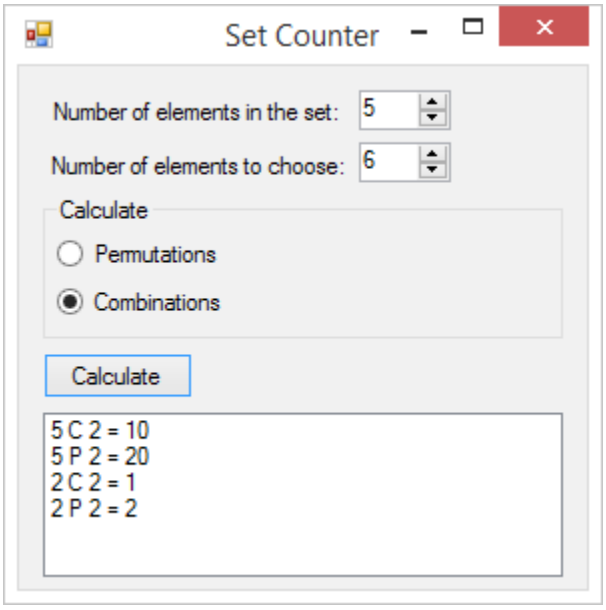

*The same set has two permutations, but only one combination.*



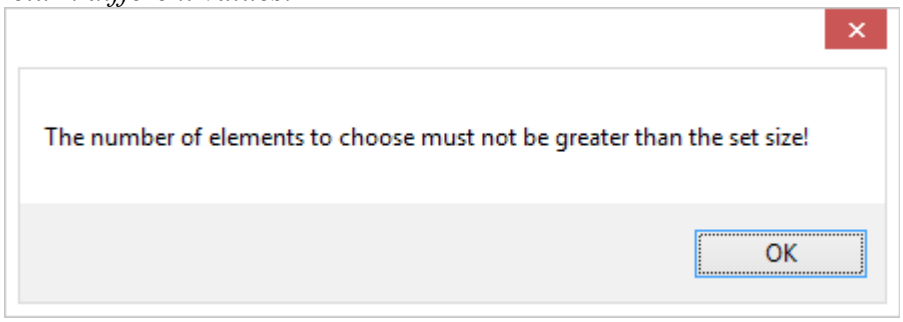

*Another example calculation.*




*The calculations for permutations and combinations return different values.*




*When incorrect values are input...*




*...A message box informs the user of the error.*

Observations

The template method pattern is simple, but it can be very useful. It aids in the use of polymorphic code, which increases the potential for code to be reusable. Its simplicity makes it very versatile. I feel that my submission fits the assignment very well. It follows the UML diagram in demonstrating the pattern, and it even demonstrates the polymorphic nature of the pattern. I expect I will use this pattern a lot in the future.