44. Template Method

The Template Method design pattern is all about using a combination of concrete and abstract logic inside an abstract class to create multiple concrete instances of it. Template Method would be defined inside an abstract class, so it cannot be used directly. It will have to be overridden by a concrete implementation of such a class.

It's not to be confused with using interfaces or pure abstract classes. What makes the Template Method unique is that some pieces of logic inside of it are already concrete. Typically, this is achieved by having a concrete public method that calls for various protected methods. However, some or all of these protected methods would be abstract, which makes the behavior of the class unique to each of its implementations.

Template Method design pattern can be summarized as follows:

- There is an abstract class with a non-abstract public method that calls one or more abstract protected methods.
- Any concrete implementation of this class will need to override the abstract protected methods, giving them implementation-specific behavior.
- As a result, the overall behavior of the public method that calls those abstract methods will be unique based on implementation.

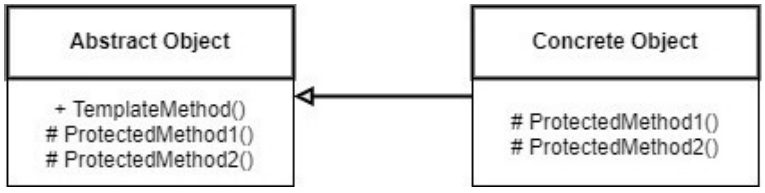

Figure 44.1 - Template Method UML diagram

We will now go through an example implementation of the Template Method. The complete solution can be found via the link below:

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Behavioral_Patterns/Template-Method

Prerequisites

In order to be able to implement the code samples below, you need the following installed on your machine:

- .NET 8 SDK (or newer)
- A suitable IDE or a code editor (Visual Studio, Visual Studio Code, JetBrains Rider)

Template Method implementation example

In this example, we will be building an application that is capable of converting text from one format to another. We will use the console application as our project type. The first thing that we will do is add the following abstract class, which contains the template method:

```
using System.Text;
using System.Text.RegularExpressions;

namespace Template_Method_Demo;

internal abstract partial class
    AbstractTextToHtmlConverter
{
        protected static string
        ProcessParagraphs(string text)
    {
        var paragraphs =
            MyRegex()
            .Split(text)
            .Where(p => p.Any(
                char.IsLetterOrDigit));

        var sb = new StringBuilder();

        foreach (var paragraph in paragraphs)
        {
            if (paragraph.Length == 0)
                continue;

            sb.AppendLine(
                $"<p>{paragraph}</p>");
        }

        sb.AppendLine("<br/>");

        return sb.ToString();
    }

    protected abstract string
        ApplyPostProcessing(string text);
```

```
36
37      public string ConvertText(
38          string text)
39      {
40          text = ProcessParagraphs(text);
41          return ApplyPostProcessing(text);
42      }
43
44      [GeneratedRegex(@"(\r\n?|\n)")]
45      private static partial Regex MyRegex();
46  }
```

As you can see, we have a public method called `ConvertText`, which has some pre-defined logic. This method calls two protected methods: `ProcessParagraphs` and `ApplyPostProcessing`. The former method also has some pre-defined content. However, the latter one is abstract, so it can have any implementation.

What we are doing in this class is converting some basic text to HTML. Let's now create a class that inherits from this abstract class. We will call it `BasicTextToHtmlConverter`. In this implementation, there is no real post-processing happening. We are just returning the input text without any changes.

```
1   namespace Template_Method_Demo;
2
3   internal class BasicTextToHtmlConverter :
4       AbstractTextToHtmlConverter
5   {
6           protected override string
7           ApplyPostProcessing(string text)
8       {
9           return text;
10      }
11  }
```

And then we will add another implementation of this abstract class, which assumes that the input text is Markdown (MD). In this case, the post-processing step will search for MD markers in the text and replace them with appropriate HTML tags.

```
namespace Template_Method_Demo;

internal class MdToHtmlConverter :
    AbstractTextToHtmlConverter
{
    private readonly Dictionary
        <string, (string, string)> tagsToReplace;

    public MdToHtmlConverter()
    {
        tagsToReplace =
            new Dictionary<string, (string, string)>
        {
            { "**", ("<strong>", "</strong>") },
            { "*", ("<em>", "</em>") },
            { "~~", ("<del>", "</del>") }
        };

    }

    protected override string
        ApplyPostProcessing(string text)
    {
        foreach (var key in tagsToReplace.Keys)
        {
            var replacementTags = tagsToReplace[key];

            if (CountStringOccurrences(
                text, key) % 2 == 0)
                text =
                    ApplyTagReplacement(
```

```
                        text,
                        key,
                        replacementTags.Item1,
                        replacementTags.Item2);
        }

        return text;
    }

    private static int CountStringOccurrences(
        string text, string pattern)
    {
        int count = 0;
        int currentIndex = 0;
        while ((currentIndex = text
            .IndexOf(
                pattern, currentIndex)) != -1)
        {
            currentIndex += pattern.Length;
            count++;
        }
        return count;
    }

    private static string ApplyTagReplacement(
        string text,
        string inputTag,
        string outputOpeningTag,
        string outputClosingTag)
    {
        int count = 0;
        int currentIndex = 0;

        while ((currentIndex =
            text.IndexOf(
```

```
67                  inputTag,
68                  currentIndex)) != -1)
69          {
70              count++;
71
72              if (count % 2 != 0)
73              {
74                  var prepend =
75                      outputOpeningTag;
76                  text = text.Insert(
77                      currentIndex, prepend);
78                  currentIndex +=
79                      prepend.Length +
80                      inputTag.Length;
81              }
82              else
83              {
84                  var append = outputClosingTag;
85                  text =
86                      text.Insert(currentIndex, append);
87                  currentIndex +=
88                      append.Length + inputTag.Length;
89              }
90          }
91
92          return text.Replace(inputTag, string.Empty);
93      }
94  }
```

Now, we can test both implementations. To do so, we will replace the content of the `Program.cs` file with the following:

```
using Template_Method_Demo;

var inputText = @"This is the *first* paragraph.

This is the **second** paragraph.

This is the ~~third~~ paragraph.";

Console.WriteLine(
    "Text after using basic converter:");
var basicTextConverter =
    new BasicTextToHtmlConverter();
Console.WriteLine(
    basicTextConverter
        .ConvertText(inputText));

Console.WriteLine(
    "Text after using MD converter:");
var mdToHtmlConverter =
    new MdToHtmlConverter();
Console.WriteLine(
    mdToHtmlConverter
    .ConvertText(inputText));

Console.ReadKey();
```

And, as expected, `BasicTextToHtmlConverter` has left all the MD markers intact. The `MdToHtmlConverter`, on the other hand, has correctly identified them all and replaced each with a corresponding HTML tag.

```
Text after using basic converter:
<p>This is the *first* paragraph.</p>
<p>This is the **second** paragraph.</p>
<p>This is the ~~third~~ paragraph.</p>
<br/>

Text after using MD converter:
<p>This is the <em>first</em> paragraph.</p>
<p>This is the <strong>second</strong> paragraph.</p>
<p>This is the <del>third</del> paragraph.</p>
<br/>
```

Figure 44.2 - Different outcomes with different implementations of Template Method

This concludes the overview of the Template Method design pattern. Let's now summarize its benefits.

Benefits of using Template Method

The main benefits of using Template Method are as follows:

- It's easy to define a structure for related algorithms.
- Open-closed principle is well maintained, as the original class doesn't have to change.
- You can reuse code in related algorithms without having to write similar steps.

Now we will have a look at the things to watch out for while using the Template Method.

Caveats of using Template Method

One thing you need to be mindful of while using the Template Method is making sure that you don't violate the Liskov substitution principle, as it's an easy principle to accidentally violate while using this design pattern.

Also, you need to make sure that the process inside the Template Method doesn't have too many steps. Otherwise, you will be violating the single responsibility principle and perhaps even making your code more difficult to maintain overall.