45. Visitor

Visitor is a design pattern that allows you to modify the behavior of existing objects when you cannot modify these objects directly. Essentially, you allow these objects to be "visited" by an external object. This object would invoke one of the public methods on the original object. But either before or after the call, it will perform some additional actions.

The Visitor design pattern can be summarized as follows:

- The original object is known as **Component**.
- **Visitor** object has a method that would call a specific method on a specific implementation of **Component** while performing some additional actions.
- **Component** implementation has been extended to accept a **Visitor** object.
- When a **Visitor** object is accepted by a **Component** object, the specific method on **Visitor** object gets triggered.

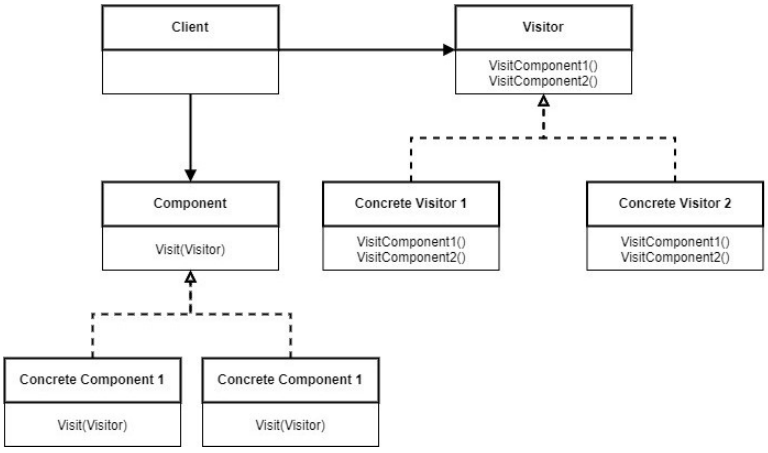

Figure 45.1 - Visitor UML diagram

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

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

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)

Visitor implementation example

Imagine that we had a library that is capable of converting between some basic HTML and plain text. However, we then realize that it

doesn't deal with all HTML element types. And then we also realize that we don't want to just deal with plain text. We want to convert between HTML and Markdown.

But the problem is that we can't change the original functionality, as this will break some components of the system that are already using it. So, we call the Visitor design pattern to our rescue.

We will create a console application project. Then we will add the following interface to it to enable our existing classes to accept a **Visitor**.

```
namespace Visitor_Demo;

internal interface IComponent
{
    string Accept(
        IVisitor visitor,
        string text);
}
```

To fix the compiler error that complains about the absence of the `IVisitor` type, we add this type. And it will look as follows:

```
namespace Visitor_Demo;

internal interface IVisitor
{
    string VisitTextToHtmlConverter(
        TextToHtmlConverter component,
        string text);
    string VisitHtmlToTextConverter(
        HtmlToTextConverter component,
        string text);
}
```

In this **Visitor** interface, we have methods that will allow us to visit two concrete **Component** objects. And we will now add both of these objects.

The `HtmlToTextConverter` class will have the following definition:

```
namespace Visitor_Demo;

internal class HtmlToTextConverter :
    IComponent
{
    public string Accept(
        IVisitor visitor,
        string text)
    {
        return visitor
            .VisitHtmlToTextConverter(
                this, text);
    }

    public string ProcessParagraphs(
        string text)
    {
        return text
            .Replace("<p>", "")
            .Replace("</p>", "\n")
            .Replace("<br/>", "");
    }
}
```

And `TextToHtmlConverter` will look like this:

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

namespace Visitor_Demo;

internal partial class TextToHtmlConverter :
    IComponent
{
    public string Accept(
        IVisitor visitor,
        string text)
    {
        return visitor
        .VisitTextToHtmlConverter(
            this, text);
    }

    public 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();
    }

    [GeneratedRegex(@"(\r\n?|\n)")]
    private static partial Regex MyRegex();
}
```

As you can see, these convertors only know how to deal with `p` and `br` HTML elements. But we want them both to be able to deal with `strong`, `em`, and `del` elements too. Also, we want the converters to be able to convert between these elements and corresponding MD markers. And to apply this functionality, we will add the following **Visitor** implementation:

```
namespace Visitor_Demo;

internal class MdConverterVisitor :
    IVisitor
{
    public string VisitTextToHtmlConverter(
        TextToHtmlConverter component,
        string text)
    {
        text = component
        .ProcessParagraphs(text);

        var tagsToReplace =
            new Dictionary<string, (string, string)>
        {
            { "**", ("<strong>", "</strong>") },
            { "*", ("<em>", "</em>") },
```

```
            { "~~", ("<del>", "</del>") }
        };

        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;
    }

    public string VisitHtmlToTextConverter(
        HtmlToTextConverter component,
        string text)
    {
        return component
            .ProcessParagraphs(text)
            .Replace("<strong>", "**")
            .Replace("</strong>", "**")
            .Replace("<em>", "*")
            .Replace("</em>", "*")
            .Replace("<del>", "~~")
            .Replace("</del>", "~~");
    }

    private static int CountStringOccurrences(
```

```
53          string text, string pattern)
54      {
55          int count = 0;
56          int currentIndex = 0;
57          while ((currentIndex =
58              text.IndexOf(
59                  pattern, currentIndex)) != -1)
60          {
61              currentIndex += pattern.Length;
62              count++;
63          }
64          return count;
65      }
66
67      private static string ApplyTagReplacement(
68          string text,
69          string inputTag,
70          string outputOpeningTag,
71          string outputClosingTag)
72      {
73          int count = 0;
74          int currentIndex = 0;
75
76          while ((currentIndex =
77              text.IndexOf(
78                  inputTag, currentIndex)) != -1)
79          {
80              count++;
81
82              if (count % 2 != 0)
83              {
84                  var prepend = outputOpeningTag;
85                  text = text.Insert(
86                      currentIndex, prepend);
87                  currentIndex +=
```

```
88                      prepend.Length +
89                      inputTag.Length;
90              }
91              else
92              {
93                  var append = outputClosingTag;
94                  text = text.Insert(
95                      currentIndex, append);
96                  currentIndex +=
97                      append.Length +
98                      inputTag.Length;
99              }
100         }
101
102         return text.Replace(
103             inputTag, string.Empty);
104     }
105 }
```

And that's it. We have added some new functionality to our existing **Component** objects. The only modification that we have applied to our **Component** objects was the added ability to accept a **Visitor** object. All remaining functionality remained intact.

Now, we will replace the content of our `Program.cs` class to test our **Visitor** logic.

```
using Visitor_Demo;

var mdText = @"This is **first** paragraph.

This is *second* paragraph.

This is ~~third~~ paragraph.";

var htmlText = """
<p>This is <strong>first</strong> paragraph.</p>
<p>This is <em>second</em> paragraph.</p>
<p>This is <del>third</del> paragraph.</p>
<br/>
""";

var visitor = new MdConverterVisitor();
var textToHtmlConverter =
    new TextToHtmlConverter();
var htmlToTextConverter =
    new HtmlToTextConverter();

Console.WriteLine(
    "MD text converted to HTML:");
Console.WriteLine(
    textToHtmlConverter
        .Accept(visitor, mdText));
Console.WriteLine(string.Empty);
Console.WriteLine(
    "HTML text converted to MD:");
Console.WriteLine(
    htmlToTextConverter
    .Accept(visitor, htmlText));

Console.ReadKey();
```

And, as we can see from the following screenshot, we were success-

fully able to add some new behavior.

```
MD text converted to HTML:
<p>This is <strong>first</strong> paragraph.</p>
<p>This is <em>second</em> paragraph.</p>
<p>This is <del>third</del> paragraph.</p>
<br/>


HTML text converted to MD:
This is **first** paragraph.

This is *second* paragraph.

This is ~~third~~ paragraph.
```

Figure 45.2 - The outcome of using different Visitors

The overview of the Visitor design pattern is now complete. Let's summarize the value it gives us.

Benefits of using Visitor

The Visitor design pattern has the following key benefits:

- A new functionality can be added to existing objects without modifying their existing logic.
- Open-closed principle is well maintained, as the existing objects remain close to modification.

And now let's get familiar with the things that you need to watch out for while using Visitor.

Caveats of using Visitor

So, the main caveat of using the Visitor design pattern is that the **Visitor** object doesn't have access to private and protected members of the **Component**. This makes it unsuitable for certain scenarios where some new functionality needs to be added to an existing object.

Another caveat of using Visitor is that there are times when the **Component** object would be modified. As a developer, you need to watch out for, so you can either update your corresponding **Visitor** or retire it altogether.

Finally, because a **Visitor** object may contain many methods, each corresponding to a specific **Component** implementation, you need to watch out for having too many of such methods on a single **Visitor**. Otherwise, you may end up with a God object.