4. Interface segregation principle

In this chapter, we will have a look at the letter “I” of the SOLID acronym: the interface segregation principle.

The full solution demonstrated in this chapter is available via the following link:

https://github.com/fiodarsazanavets/Dotnet-CSharp-SOLID-demo/tree/master/4-interface-segregation-principle

What is interface segregation principle

In object-oriented programming, interfaces are used to define signatures of methods and properties without specifying the exact logic inside of them. Essentially, they act as a contract that a class must adhere to. If a class implements any particular interface, it must contain all components defined by the interface as its public members.

The interface segregation principle states that if any particular interface member is not intended to be implemented by any of the classes that implement the interface, it must not be in the interface. It is closely related to the single responsibility principle by making sure that only the absolutely essential functionality is covered by the interface and the class that implements it.

And now, we will see why interface segregation is important.

Importance of interface segregation

We will continue with the same code that we ended up with after we implemented the Liskov substitution principle in the previous chapter.

So, we have a base class called `TextProcessor` that converts paragraphs in the input text into HTML paragraphs by applying relevant tags to them.

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

namespace TextToHtmlConvertor;

public partial class TextProcessor
{
    public virtual string ConvertText(string inputText)
    {
        var paragraphs = MyRegex()
            .Split(inputText)
            .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/>");
```

```
26
27          return sb.ToString();
28      }
29
30      [GeneratedRegex(@"(\r\n?|\n)")]
31      private static partial Regex MyRegex();
32  }
```

We also have a derived class, `MdTextProcessor`, that adds a new processing capability – the ability to detect specific Markdown symbols and convert them into corresponding HTML tags. To make sure that any places in the code that accept the original `TextProcessor` class don't start to behave differently if an instance of `MdTextProcessor` is passed instead, the additional functionality is implemented via a new method – `ConvertMdText`.

```
1   namespace TextToHtmlConvertor;
2
3   public class MdTextProcessor(
4       Dictionary<string, (string, string)> tagsToReplace) :
5       TextProcessor
6   {
7       public override string ConvertText(string inputText)
8       {
9           var processedText = base.ConvertText(inputText);
10
11          foreach (var key in tagsToReplace.Keys)
12          {
13              var replacementTags = tagsToReplace[key];
14
15              if (CountStringOccurrences(
16                  processedText, key) % 2 == 0)
17                  processedText =
18                      ApplyTagReplacement(
19                          processedText,
```

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

        return processedText;
    }

    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(inputTag, currentIndex)) != -1)
        {
            count++;
```

```
55
56              if (count % 2 != 0)
57              {
58                  var prepend =
59                      outputOpeningTag;
60                  text =
61                      text.Insert(currentIndex, prepend);
62                  currentIndex +=
63                      prepend.Length + inputTag.Length;
64              }
65              else
66              {
67                  var append =
68                      outputClosingTag;
69                  text =
70                      text.Insert(currentIndex, append);
71                  currentIndex +=
72                      append.Length + inputTag.Length;
73              }
74          }
75
76          return text.Replace(inputTag, string.Empty);
77      }
78  }
```

Currently, neither of the classes implements any interfaces. But we may want to be able to use similar functionality in other classes. Perhaps, we would want to convert input text into different formats, not just HTML. Perhaps, we want to simply be able to mock the functionality in some unit tests. So, it would be beneficial to us to get our classes to implement some interfaces.

As our `MdTextProcessor` has two methods, `ConvertText`, which is inherited from TextProcessor, and `ConvertMdText`, which is its own, we can have an interface like this:

```
namespace TextToHtmlConvertor;

public interface ITextProcessor
{
   string ConvertText(string inputText);
   string ConvertMdText(string inputText);
}
```

And implement it like this:

```
namespace TextToHtmlConvertor;

public class MdTextProcessor(
    Dictionary<string, (string, string)> tagsToReplace) :
    TextProcessor, ITextProcessor
{
    public override string ConvertText(string inputText)
    {
        var processedText = base.ConvertText(inputText);

        foreach (var key in tagsToReplace.Keys)
        {
            var replacementTags = tagsToReplace[key];

            if (CountStringOccurrences(
                processedText, key) % 2 == 0)
                processedText =
                    ApplyTagReplacement(
                        processedText,
                        key,
                        replacementTags.Item1,
                        replacementTags.Item2);
        }

        return processedText;
```

```
26      }
27
28      private static int CountStringOccurrences(
29          string text, string pattern)
30      {
31          int count = 0;
32          int currentIndex = 0;
33          while ((currentIndex =
34              text.IndexOf(pattern, currentIndex)) != -1)
35          {
36              currentIndex += pattern.Length;
37              count++;
38          }
39          return count;
40      }
41
42      private static string ApplyTagReplacement(
43          string text,
44          string inputTag,
45          string outputOpeningTag,
46          string outputClosingTag)
47      {
48          int count = 0;
49          int currentIndex = 0;
50
51          while ((currentIndex =
52              text.IndexOf(inputTag, currentIndex)) != -1)
53          {
54              count++;
55
56              if (count % 2 != 0)
57              {
58                  var prepend =
59                      outputOpeningTag;
60                  text =
```

```
                    text.Insert(currentIndex, prepend);
                currentIndex +=
                    prepend.Length + inputTag.Length;
            }
            else
            {
                var append =
                    outputClosingTag;
                text =
                    text.Insert(currentIndex, append);
                currentIndex +=
                    append.Length + inputTag.Length;
            }
        }

        return text.Replace(inputTag, string.Empty);
    }
}
```

So far so good. However, there is a problem. If we want to implement this interface in the original `TextProcessor` class, we will now have to add the `ConvertMdText` method to it. However, this method is not relevant to this particular class. So, to let the programmers know that this method is not intended to be used, we get it to throw `NotImplementedException`.

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

namespace TextToHtmlConvertor;

public partial class TextProcessor : ITextProcessor
{
    public virtual string ConvertText(string inputText)
    {
```

```
10          var paragraphs = MyRegex()
11              .Split(inputText)
12              .Where(p => p.Any(
13                  char.IsLetterOrDigit));
14
15          var sb = new StringBuilder();
16
17          foreach (var paragraph in paragraphs)
18          {
19              if (paragraph.Length == 0)
20                  continue;
21
22              sb.AppendLine($"<p>{paragraph}</p>");
23          }
24
25          sb.AppendLine("<br/>");
26
27          return sb.ToString();
28      }
29
30      public string ConvertMdText(string inputText)
31      {
32         throw new NotImplementedException();
33      }
34
35      [GeneratedRegex(@"(\r\n?|\n)")]
36      private static partial Regex MyRegex();
37  }
```

This introduces a problem. We have ended up with a method that we will never use. And what if we want to create other derived classes that are designed to deal with text formats other than MD? We will then have to expand our interface and keep adding new unused methods to every class that implements it.

So, we can implement the following solution. Our base interface

will only have a single method that the base class will use:

```
namespace TextToHtmlConvertor;

public interface ITextProcessor
{
   string ConvertText(string inputText);
}
```

The good news that, in C#, we can use inheritance for interfaces, just like we can use it for classes. So, we can create another interface that will be relevant to those classes that are specific to processing of MD-formatted text:

```
namespace TextToHtmlConvertor;

public interface IMdTextProcessor : ITextProcessor
{
   string ConvertMdText(string inputText);
}
```

And so, we would implement the interface by the base class without having to add any new methods:

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

namespace TextToHtmlConvertor;

public partial class TextProcessor : ITextProcessor
{
    public virtual string ConvertText(string inputText)
    {
        var paragraphs = MyRegex()
            .Split(inputText)
```

```
            .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();
}
```

And our derived class will remain clean too:

```
namespace TextToHtmlConvertor;

public class MdTextProcessor(
    Dictionary<string, (string, string)> tagsToReplace) :
    TextProcessor, IMdTextProcessor
{
    public override string ConvertText(string inputText)
    {
        var processedText = base.ConvertText(inputText);

        foreach (var key in tagsToReplace.Keys)
```

```
12          {
13              var replacementTags = tagsToReplace[key];
14
15              if (CountStringOccurrences(
16                  processedText, key) % 2 == 0)
17                  processedText =
18                      ApplyTagReplacement(
19                          processedText,
20                          key,
21                          replacementTags.Item1,
22                          replacementTags.Item2);
23          }
24
25          return processedText;
26      }
27
28      private static int CountStringOccurrences(
29          string text, string pattern)
30      {
31          int count = 0;
32          int currentIndex = 0;
33          while ((currentIndex =
34              text.IndexOf(pattern, currentIndex)) != -1)
35          {
36              currentIndex += pattern.Length;
37              count++;
38          }
39          return count;
40      }
41
42      private static string ApplyTagReplacement(
43          string text,
44          string inputTag,
45          string outputOpeningTag,
46          string outputClosingTag)
```

```
47      {
48          int count = 0;
49          int currentIndex = 0;
50
51          while ((currentIndex =
52              text.IndexOf(inputTag, currentIndex)) != -1)
53          {
54              count++;
55
56              if (count % 2 != 0)
57              {
58                  var prepend =
59                      outputOpeningTag;
60                  text =
61                      text.Insert(currentIndex, prepend);
62                  currentIndex +=
63                      prepend.Length + inputTag.Length;
64              }
65              else
66              {
67                  var append =
68                      outputClosingTag;
69                  text =
70                      text.Insert(currentIndex, append);
71                  currentIndex +=
72                      append.Length + inputTag.Length;
73              }
74          }
75
76          return text.Replace(inputTag, string.Empty);
77      }
78  }
```

When NotImplementedException is appropriate

In C#, `NotImplementedException` is an error type that is specifically designed to be thrown from the members that implement the interface, but aren't intended to be used. And it comes directly from the core system library of C#.

At first glance, it may seem that with this feature in place, the language was designed to violate the interface segregation principle. However, there are situations where leaving unused interface methods in your class will be appropriate without necessarily violating this principle.

One situation where throwing `NotImplementedException` will be appropriate if your class is a work in progress and this has been put there temporarily. You may decide all the essential members for your class ahead of time and create the interface immediately before you forget what you intend to do.

However, creating the class may take a relatively long time and you may not be able to do it all in one go. Despite this, you will already have to implement every interface member. Otherwise, your code will not compile. And gradually, you will populate every member with some valid logic.

This is precisely why Visual Studio populates all auto-generated members with `NotImplementedException` if you want to auto-implement the interface. Clicking the action button will ensure that your code will compile immediately, while you can populate all the members with whatever logic you want at your own convenience.

And, because every interface member is intended to eventually be implemented and is absolutely required for the finished product, this will not violate the interface segregation principle.

The other case is when you want to write an interface implementation to enable easy testing of your code. You may have an original class that is intended to be placed in production. This class may have various pieces of logic that you will not be able to replicate on a development machine. For example, it may be sending requests to a certain server or reading messages from a certain queue. And in that class, every method and property is absolutely required to be there.

However, there may be some pieces of functionality in the class that you would be able to run anywhere. So, you may have a version of the class that only contains those components, which will allow you to easily run certain tests on any machine.

And this is where `NotImplementedException` serves its purpose. In this context, it is there to notify the developer that, although this method or property is an absolutely essential member of the class under normal circumstances, it is not relevant in this specific context.

The alternative would be to mock those methods up. But doing so would give them logic that is completely different from how the production class behaves. Therefore, the behavior you will see in your tests will not be representative of the actual behavior of the deployed software and the tests may be pointless. The methods that are testable, however, will behave exactly as they would in production.