2. Open-closed principle

In this chapter, we will cover the second principle from the SOLID acronym – open-closed principle. We will do so by looking at some examples in C#. However, the general concepts will be applicable to any other object-oriented language.

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

https://github.com/fiodarsazanavets/Dotnet-CSharp-SOLID-demo/tree/master/2-open-closed-principle

What is open-closed principle

Open-closed principle states that every atomic unit of code, such as class, module or function, should be open for extension, but closed for modification. The principle was coined in 1988 by Bertrand Meyer.

Essentially, what it means is that, once written, the unit of code should be unchangeable, unless some errors are detected in it. However, it should also be written in such a way that additional functionality can be attached to it in the future if requirements are changed or expanded. This can be achieved by common features of object-oriented programming, such as inheritance and abstraction.

Unlike the single responsibility principle which almost everyone agrees with, the open-closed principle has its critics. It’s almost always obvious how the single responsibility principle can be implemented and what benefits it will provide. However, trying to foresee where the requirements may change in the future and designing your classes in such a way that most of them would be met without having to alter your existing code is often seen as an unnecessary overhead that doesn't provide any benefits.

And the way the software is written has moved on quite a bit since 1988. If back then the process of deploying a new software version was tedious, long, and expensive, many systems of today take minutes to deploy. And the whole process can be done on demand with a click of a mouse.

And with agile software development practices being adopted everywhere in the industry, requirements change all the time. Quite often, the changes are radical. Whole sections of code get removed and replaced regularly. So, why design for adherence to the open-close principle if the component that you are writing is quite likely to be ditched very soon?

Although these objections are valid, the open-closed principle still provides its benefits. Here is an example.

Implementing the open-closed principle in C#

In the previous chapter, we had a C# example of the single responsibility principle. We have written an application that reads the text from any text file and converts it into HTML by enclosing every paragraph in `p` tags.

After all of the refactoring, we have ended up with three classes, which are as follows. `FileProcessor` class that reads the input file and saves the output into an HTML file:

```
using System.Web;

namespace TextToHtmlConvertor;

public class FileProcessor(
    string fullFilePath)
{
    public string ReadAllText()
    {
        return HttpUtility
            .HtmlEncode(
                File.ReadAllText(
                    fullFilePath));
    }

    public void WriteToFile(string text)
    {
        string outputFilePath =
            Path.GetDirectoryName(fullFilePath) +
            Path.DirectorySeparatorChar +
            Path.GetFileNameWithoutExtension(
                fullFilePath) +
            ".html";

        using StreamWriter file = new(outputFilePath);
        file.Write(text);
    }
}
```

`TextProcessor` class that processes the text from the input file:

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

namespace TextToHtmlConvertor;

public partial class TextProcessor
{
    private readonly FileProcessor fileProcessor;

    public TextProcessor(
        FileProcessor fileProcessor)
    {
        this.fileProcessor = fileProcessor;
    }

    public void ConvertText()
    {
        var inputText =
            fileProcessor.ReadAllText();

        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/>");
        fileProcessor.WriteToFile(
            sb.ToString());
    }

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

And the `Program.cs` file that serves as the entry point into the application looks like this:

```
using TextToHtmlConvertor;

try
{
    Console.WriteLine(
        "Please specify the file to convert to HTML.");
    var fullFilePath =
        Console.ReadLine() ?? string.Empty;
    var fileProcessor =
        new FileProcessor(fullFilePath);
    var textProcessor =
        new TextProcessor(fileProcessor);
    textProcessor.ConvertText();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

Console.WriteLine("Press any key to exit.");
Console.ReadKey();
```

So far, so good. The application is doing exactly what the requirements say and every element of the application serves its own purpose. But we know that HTML doesn't just consist of paragraphs, right?

So, while we are only being asked to read paragraphs and apply HTML formatting to them, it's not difficult to imagine that we may be asked in the future to expand the functionality to be able to produce much richer HTML output.

In this case, we will have no choice but to rewrite our code. And although the impact of these changes in such a small application would be negligible, what if we had to do it to a much larger application?

We would definitely need to rewrite our unit tests that cover the class, which we may not have enough time to do. So, if we had good code coverage to start with, a tight deadline to deliver new requirements may force us to ditch a few unit tests, and therefore increase the risk of accidentally introducing defects.

What if we had existing services calling into our software that aren't part of the same code repository? What if we don't even know those exist? Now, some of these may break due to receiving unexpected results and we may not find out about it until it all has been deployed into production.

So, to prevent these things from happening, we can refactor our code as follows.

Our `TextProcessor` class will become this:

```csharp
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/>");

        return sb.ToString();
    }

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

We have now completely separated file-processing logic from it. The main method of the class, `ConvertText`, now takes the input text as a parameter and returns the formatted output text. Otherwise, the logic inside of it is the same as it was before. All it does is split the input text into paragraphs and enclose each one of them in the `p` tag. And to allow us to expand this functionality if requirements ever change, it was made virtual.

Our `Program.cs` file now looks like this:

```
using TextToHtmlConvertor;

try
{
    Console.WriteLine(
        "Please specify the file to convert to HTML.");
    var fullFilePath =
        Console.ReadLine() ?? string.Empty;
    var fileProcessor =
        new FileProcessor(fullFilePath);

    var textProcessor =
        new TextProcessor();
    var inputText =
        fileProcessor.ReadAllText();
    var outputText =
        textProcessor.ConvertText(inputText);
    fileProcessor.WriteToFile(outputText);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

Console.WriteLine("Press any key to exit.");
Console.ReadKey();
```

We are now calling `FileProcessor` methods from within this class.

Otherwise, the output will be exactly the same.

Now, one day, we are told that our application needs to be able to recognize Markdown (MD) emphasis markers in the text, which include bold, italic, and strikethrough. These will be converted into their equivalent HTML markup.

So, in order to do this, all you have to do is add another class that inherits from TextProcessor. We'll call it `MdTextProcessor`:

```
namespace TextToHtmlConvertor;

public class MdTextProcessor(
    Dictionary<string, (string, string)> tagsToReplace) :
    TextProcessor
{
    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;
    }

    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++;

            if (count % 2 != 0)
            {
                var prepend =
                    outputOpeningTag;
                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);
    }
}
```

In its constructor, the class receives a dictionary of tuples containing two string values. The key in the dictionary is the Markdown emphasis marker, while the value contains the opening HTML tag and closing HTML tag. The code inside the overridden `ConvertText` method calls the original `ConvertText` method from its base class and then looks up all instances of each emphasis marker in the text. It then ensures that the number of those is even (otherwise it would be an incorrectly formatted Markdown content) and replaces them with opening and closing HTML tags.

Now, our `Program.cs` file will look like this:

```
using TextToHtmlConvertor;

try
{
    Console.WriteLine(
        "Please specify the file to convert to HTML.");
    var fullFilePath =
        Console.ReadLine() ?? string.Empty;
    var fileProcessor =
        new FileProcessor(fullFilePath);
    var tagsToReplace =
        new Dictionary<string, (string, string)>
    {
        { "**", ("<strong>", "</strong>") },
        { "*", ("<em>", "</em>") },
        { "~~", ("<del>", "</del>") }
    };

    var textProcessor =
        new MdTextProcessor(tagsToReplace);
    var inputText =
        fileProcessor.ReadAllText();
    var outputText =
        textProcessor.ConvertText(inputText);
    fileProcessor.WriteToFile(outputText);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

Console.WriteLine("Press any key to exit.");
Console.ReadKey();
```

The dictionary is something we pass into `MdTextProcessor` from the outside, so we needed to initialize it here. And now our textProcessor variable is of type MdTextProcessor rather than TextProcessor. The rest of the code has remained unchanged.

So, if we had any existing unit tests on the `ConvertText` method of the `TextProcessor` class, they would not be affected at all. Likewise, if any external application uses `TextProcessor`, it will work just like it did before after our code update. Therefore we have added new capabilities without breaking any of the existing functionality at all.

Also, there is another example of how we can future-proof our code. The requirements of what special text markers our application must recognize may change, so we have turned it into easily changeable data. Now, `MdTextProcessor` doesn't have to be altered.

Also, although we could simply use one opening HTML tag as the value in the dictionary and then just create a closing tag on the go by inserting a slash character into it, we have defined opening and closing tags explicitly. Again, what if the requirements state that we need to add various attributes to the opening HTML tags? What if certain key values will correspond to nested tags? It would be difficult to foresee all possible scenarios beforehand, so the easiest thing we could do is make it explicit to cover any of such scenarios.

Conclusion

Open-closed principle is a useful thing to know, as it will substantially minimize the impact of any changes to your application code. If your software is designed with this principle in mind, then future modifications to any one of your code components will not cause the need to modify any other components and assess the impact on any external applications.

However, unlike the single responsibility principle, which should be followed almost like a law, there are situations where applying the open-closed principle has more cons than pros. For example, when designing a component, you will have to think of any potential changes to the requirements in the future. This, sometimes, is counterproductive, especially when your code is quite likely to be radically restructured at some point.

So, while you still need to spend some time thinking about what new functionality may be added to your new class in the future, use common sense. Considering the most obvious changes is often sufficient.

Also, although adding new public methods to the existing class without modifying the existing methods would, strictly speaking, violate the open-closed principle, it will not cause the most common problems that the open-closed principle is designed to address. So, in most cases, it is completely fine to do so instead of creating even more classes that expand your inheritance hierarchy.

However, in this case, if your code is intended to be used by external software, always make sure that you increment the version of your library. If you don't, then the updated external software that relies on the new methods will get broken if it accidentally receives the old version of the library with the same version number. However, any software development team absolutely must have a versioning strategy in place and most of them do, so this problem is expected to be extremely rare.