1. Single responsibility principle

In this chapter, we will focus on the first of these principles – single responsibility principle. I will explain its importance and provide some examples of its usage in C# code.

Incidentally, as well as being the first principle in the abbreviation, it is also the one that is the easiest to grasp, the easiest to implement, and the easiest to explain. Arguably, it is also the most important principle on the list. So, let's go ahead and find out what it is and why you, as a developer, absolutely must know it.

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

https://github.com/fiodarsazanavets/Dotnet-CSharp-SOLID-demo/tree/master/1-single-responsibility-principle

What is single responsibility principle

Single responsibility principle states that, for every self-contained unit of code (which is, usually, a class), there should be one and only one reason to change. In more simple language, this means that any given class should be responsible for only one specific functionality.

Basically, your code should be structured like a car engine. Even though the whole engine acts as a single unit, it consists of many components, each playing a specific role. A spark plug exists only to ignite the fuel vapors. A cam belt is there only to synchronize the rotation of the crankshaft and the camshafts and so on.

Each of these components is atomic (unsplittable), self-contained, and can be easily replaced. So should be each of your classes.

Clean Code, a book that every developer should read, provides an excellent and easy-to-digest explanation of what the single responsibility principle is and provides some examples of it in Java. What I will do now is explain why the single responsibility principle is so important by providing some C# examples.

The importance of the single responsibility principle

Those who are familiar with C# will recognize that every C# application has a file that serves as an entry point, which is usually called `Program.cs`.

Now, let's imagine a basic console application that will read an input text from any specified text file, will wrap every paragraph in HTML `p` tags, and will save the output in a new HTML file in the same folder that the input file came from. If we are to put entire logic into a single class, it would look something similar to this:

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

try
{
    Console.WriteLine(
        "Please specify the file to convert to HTML.");
    var fullFilePath =
        Console.ReadLine() ?? string.Empty;
    var inputText =
        ReadAllText(fullFilePath);
    var paragraphs =
```

```
13          Regex
14              .Split(inputText, @"(\r\n?|\n)")
15              .Where(p => p
16                  .Any(char.IsLetterOrDigit));
17      var sb = new StringBuilder();
18
19      foreach (var paragraph in paragraphs)
20      {
21          if (paragraph.Length == 0)
22              continue;
23
24          sb.AppendLine($"<p>{paragraph}</p>");
25      }
26
27      sb.AppendLine("<br/>");
28      WriteToFile(
29          fullFilePath, sb.ToString());
30  }
31  catch (Exception ex)
32  {
33      Console.WriteLine(ex.Message);
34  }
35  Console.WriteLine("Press any key to exit.");
36  Console.ReadKey();
37
38  string ReadAllText(string fullFilePath)
39  {
40      return System.Web.HttpUtility
41          .HtmlEncode(File.ReadAllText(fullFilePath));
42  }
43
44  void WriteToFile(
45      string fullFilePath, string text)
46  {
47      var outputFilePath =
```

```
48          Path.GetDirectoryName(fullFilePath) +
49          Path.DirectorySeparatorChar +
50          Path.GetFileNameWithoutExtension(fullFilePath) +
51          ".html";
52      using StreamWriter file =
53          new(outputFilePath);
54      file.Write(text);
55  }
```

This code will work, but it will be relatively difficult to modify. Because everything is in one place, the code will take longer to read than it could have been. Of course, this is just a simple example, but what if you had a real-life console application with way more complicated functionality all in one place?

It is crucially important that you understand the code before you make any changes to it. Otherwise, you will inadvertently introduce bugs. So, you will, pretty much, have to read the entire class, even if only a tiny subset of it is responsible for a particular functionality that you are interested in. Otherwise, how would you know if there is nothing else in the class that will be affected by your changes?

Imagine another scenario. Two developers are working on the same file, but are making changes to completely different pieces of functionality within it. Once they are ready to merge their changes, there is a merge conflict. It's an absolute nightmare to resolve because each of the developers is only familiar with his own set of changes and isn't aware of how to resolve the conflict with the changes made by another developer.

The single responsibility principle exists precisely to eliminate these kinds of problems. In our example, we can apply the single responsibility principle by splitting our code into separate classes, so as well as having `Program.cs` file, we will also have `FileProcessor` and `TextProcessor` classes.

The content of our `FileProcessor` class will be as follows:

```
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);
    }
}
```

The content of the `TextProcessor` class will be as follows:

```
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>");
        }

```

```
36          sb.AppendLine("<br/>");
37          fileProcessor.WriteToFile(sb.ToString());
38      }
39
40      [GeneratedRegex(@"(\r\n?|\n)")]
41      private static partial Regex MyRegex();
42  }
```

And this is what remains of our original `Program.cs` file:

```
1   using TextToHtmlConvertor;
2
3   try
4   {
5       Console.WriteLine(
6           "Please specify the file to convert to HTML.");
7       var fullFilePath =
8           Console.ReadLine() ?? string.Empty;
9       var fileProcessor =
10          new FileProcessor(fullFilePath);
11      var textProcessor =
12          new TextProcessor(fileProcessor);
13      textProcessor.ConvertText();
14  }
15  catch (Exception ex)
16  {
17      Console.WriteLine(ex.Message);
18  }
19
20  Console.WriteLine("Press any key to exit.");
21  Console.ReadKey();
```

Now, the entire text-processing logic is handled by `TextProcessor` class, while the `FileProcessor` class is solely responsible for reading from files and writing into them.

This has made your code way more manageable. First of all, if it's a specific functionality you would want to change, you will only need to modify the file that is responsible for that specific functionality and nothing else. You won't even have to know how anything else works inside the app. Secondly, if one developer is changing how the text is converted by the app, while another developer is making changes to how files are processed, their changes will not clash.

While we've split text-processing and file-processing capabilities into their own single responsibility classes, we have left a minimal amount of code inside the `Program` class, the application entry point. It is now solely responsible for launching the application, reading the user's input, and calling methods in other classes.

The concept of class cohesion

In our example, it was very clear where the responsibilities should be split. And, in most cases, the decision will be based on the same factor we have used – splitting responsibility based on atomic functional areas. In our case, the application was mainly responsible for two things – processing text and managing files; therefore we have two functional areas within it and ended up with a separate class responsible for each of these.

However, there will be situations where a clear-cut functional area would be difficult to establish. Different functionalities sometimes have very fuzzy boundaries. This is where the concept of class cohesion comes into play to help you decide which classes to split and which ones to leave as they are.

Class cohesion is a measure of how different public components of a given class are interrelated. If all public members are inter-related, then the class has maximal cohesion, while a class that doesn't have any inter-related public members has no cohesion. The best way to determine the degree of cohesion within a class is to check whether

all private class-level variables are used by all public members.

If every private class-level variable is used by every public member, then the class is known to have maximal cohesion. This is a very clear indicator that the class is atomic and shouldn't be split. The exception would be when the class can be refactored in an obvious way and the process of refactoring eliminates some or all of the cohesion within the class.

If every public member inside the class uses at least one of the private class-level variables, while the variables themselves are interdependent and are used in combination with some of the public methods, the class has less cohesion, but would probably still not be in violation of single responsibility principle.

If, however, there are some private class-level variables that are only used by some of the public members, while other private class-level variables are only ever used by a different subset of public members, the class has low cohesion. This is a good indicator that the class should probably be split into two separate classes.

Finally, if every public member is completely independent from any other public member, the class has zero cohesion. This would probably mean that every public method should go into its own separate class.

Let's have a look at the examples of cohesion above.

In our `TextProcessor` class, we only have one method, `ConvertText`. So, we don't even have to look at the cohesion. It has maximal cohesion already.

In our `FileProcessor` class, we have two methods, `ReadAllText` and `WriteToFile`. Both of these methods use the `fullFilePath` variable, which is initialized in the class constructor. So, the class also has maximal cohesion and therefore is atomic.

God Object – the opposite of single responsibility

So, you now know what the single responsibility principle is and how it benefits you as a developer. What you may be interested to know is that this principle has the opposite, which is known as God Object.

In software development, a method of doing things that is opposite to what best practices prescribe is known as anti-pattern; therefore God Object is a type of anti-pattern. It is just as important to name bad practices as it is to name good practices. If something has a name, it becomes easy to conceptualize and remember, and it is crucially important for software developers to remember what not to do.

In this case, the name perfectly describes what this object is. As you may have guessed, a God Object is a type of class that is attempting to do everything. Just like God, it is omnipotent, omniscient and omnipresent.

In our example, the first iteration of our code had `Program.cs` file containing the entire application logic, therefore it was a God Object in the context of our application. But this was just a simplistic example. In a real-life scenario, a God Object may span thousands of lines of code.

So, I don't care whether you are religious or not. Everyone is entitled to worship any deity in the privacy of their own home. Just make sure you don't put God into your code. And remember to always use the single responsibility principle.

Conclusion

In this chapter, we have covered the first and arguably the most important SOLID principle of object-oriented software development.

In the next chapter, we will have a look at how to use the open-closed principle in the context of C#.