5. Dependency inversion principle

Finally, we've reached the last of the SOLID principles - dependency inversion principle.

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

https://github.com/fiodarsazanavets/Dotnet-CSharp-SOLID-demo/tree/master/5-dependency-inversion-principle

What is dependency inversion principle

Dependency inversion principle states that a higher-level object should never depend on a concrete implementation of a lower-level object. Both should depend on abstractions. But what does it actually mean, you may ask?

Any object-oriented language will have a way of specifying a contract to which any concrete class or module should adhere. Usually, this is known as an interface.

Interface is something that defines the signature of all the public members that the class must have, but, unlike a class, an interface doesn't have any logic inside of those members. It doesn't even allow you to define a method body to put the logic in.

But as well as being a contract that defines the accessible surface area of a class, an interface can be used as a data type in variables

and parameters. When used in such a way, it can accept an instance of absolutely any class that implements the interface.

And this is where dependency inversion comes from. Instead of passing a concrete class into your methods and constructors, you pass the interface that the class in question implements.

The class that accepts an interface as its dependency is a higher-level class than the dependency. And passing interface is done because your higher level class doesn't really care what logic will be executed inside of its dependency if any given method is called on the dependency. All it cares about is that a method with a specific name and signature exists inside the dependency.

Why the dependency inversion principle is important

We will take our code from where we left it in the previous article about the interface segregation principle.

So, we have the `TextProcessor` class that modifies input text by converting it into HTML paragraphs:

```
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();
}
```

It implements the following interface:

```
namespace TextToHtmlConvertor;


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

We have a more advanced version of `TextProcessor` that also converts MD tags into corresponding HTML elements. It's called `MdTextProcessor` and it's derived from the original `TextProcessor`:

```
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)
        {
            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)
        {
```

```
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);
```

```
                currentIndex +=
                    append.Length + inputTag.Length;
            }
        }

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

It implements the following interface:

```
namespace TextToHtmlConvertor;


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

We have the `FileProcessor` class that manages files:

```
using System.Web;

namespace TextToHtmlConvertor;

public class FileProcessor(string fullFilePath) : IFilePr\
ocessor
{
    private readonly string fullFilePath = fullFilePath;

    public string ReadAllText()
    {
        return HttpUtility.HtmlEncode(File.ReadAllText(fu\
llFilePath));
    }
```

```
15
16      public void WriteToFile(string text)
17      {
18          var outputFilePath = Path.GetDirectoryName(fullFi\
19  lePath) +
20              Path.DirectorySeparatorChar +
21              Path.GetFileNameWithoutExtension(fullFilePath\
22  ) + ".html";
23
24              using var file = new StreamWriter(outputFileP\
25  ath);
26              file.Write(text);
27      }
28  }
```

It implements the following interface:

```
1  namespace TextToHtmlConvertor;
2
3  public interface IFileProcessor
4  {
5     string ReadAllText();
6     void WriteToFile(string text);
7  }
```

And we have the `Program.cs` file which coordinates the entire logic:

```
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();
```

Having all the logic inside the `Program.cs` file is probably not the best way of doing things. It's meant to be purely an entry point

for the application. Since it's a console application, providing input from the console and output to it is also acceptable. However, having it to coordinate text conversion logic between separate classes is probably not something we want to do.

So, we have moved our logic into a separate class that coordinates the text conversion process and we called it `TextConversionCoordinator`:

```
namespace TextToHtmlConvertor;

public class TextConversionCoordinator(
    FileProcessor fileProcessor,
    MdTextProcessor textProcessor)
{
    public ConversionStatus ConvertText()
    {
        var status = new ConversionStatus();
        string inputText;
        try
        {
            inputText = fileProcessor.ReadAllText();
            status.TextExtractedFromFile = true;
        }
        catch (Exception ex)
        {
            status.Errors.Add(ex.Message);
            return status;
        }

        string outputText;
        try
        {
            outputText = textProcessor
                .ConvertMdText(inputText);

```

```
28              if (outputText != inputText)
29                  status.TextConverted = true;
30          }
31          catch (Exception ex)
32          {
33              status.Errors.Add(ex.Message);
34              return status;
35          }
36
37          try
38          {
39              fileProcessor.WriteToFile(outputText);
40              status.OutputFileSaved = true;
41          }
42          catch (Exception ex)
43          {
44              status.Errors.Add(ex.Message);
45              return status;
46          }
47
48          return status;
49      }
50  }
```

It has a single method and returns conversation status object, so we can see which parts of the process have succeeded:

```
namespace TextToHtmlConvertor;

public class ConversionStatus
{
   public bool TextExtractedFromFile { get; set; }
   public bool TextConverted { get; set; }
   public bool OutputFileSaved { get; set; }
   public List<string> Errors { get; set; } = new List<st\
ring>();
}
```

And our `Program.cs` file becomes 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 coordinator =
        new TextConversionCoordinator(
```

```
23              fileProcessor, textProcessor);
24      var status = coordinator.ConvertText();
25
26      Console.WriteLine(
27          $"Text extracted from file: {status.TextExtracted\
28  FromFile}");
29      Console.WriteLine(
30          $"Text converted: {status.TextConverted}");
31      Console.WriteLine(
32          $"Output file saved: {status.OutputFileSaved}");
33
34      if (status.Errors.Count > 0)
35      {
36          Console.WriteLine(
37              "The following errors occurred during the con\
38  version:");
39          Console.WriteLine(string.Empty);
40
41          foreach (var error in status.Errors)
42              Console.WriteLine(error);
43
44      }
45  }
46  catch (Exception ex)
47  {
48      Console.WriteLine(ex.Message);
49  }
50
51  Console.WriteLine("Press any key to exit.");
52  Console.ReadKey();
```

Currently, this will work, but it's almost impossible to write unit tests against it. You won't just be unit-testing the method. If you run it, you will run your entire application logic.

And you depend on a specific file to be actually present in a specific

location. Otherwise, you won't be able to emulate the successful scenario reliably. The folder structure on different environments where your tests run will be different. And you don't want to ship some test files with your repository and run some environment-specific setup and tear-down scripts. Remember that those output files need to be managed too?

Luckily, your `TextConversionCoordinator` class doesn't care where the file comes from. All it cares about is that `FileProcessor` returns some text that can be then passed to `MdTextProcessor`. And it doesn't care how exactly `MdTextProcessor` does its conversion. All it cares about is that the conversion has happened, so it can set the right values in the status object it's about to return.

Whenever you are unit-testing a method, there are two things you primarily are concerned about:

1. Whether the method produces expected outputs based on known inputs.
2. Whether all the expected methods on the dependencies were called.

And both of these can be easily established if we change the constructor parameters to accept the concrete implementations.

So, we change `TextConversionCoordinator` class as follows:

```
1  namespace TextToHtmlConvertor;
2
3  public class TextConversionCoordinator(
4      FileProcessor fileProcessor,
5      MdTextProcessor textProcessor)
6  {
7      public ConversionStatus ConvertText()
8      {
9          var status = new ConversionStatus();
```

```
10          string inputText;
11          try
12          {
13              inputText = fileProcessor.ReadAllText();
14              status.TextExtractedFromFile = true;
15          }
16          catch (Exception ex)
17          {
18              status.Errors.Add(ex.Message);
19              return status;
20          }
21
22          string outputText;
23          try
24          {
25              outputText = textProcessor
26                  .ConvertMdText(inputText);
27
28              if (outputText != inputText)
29                  status.TextConverted = true;
30          }
31          catch (Exception ex)
32          {
33              status.Errors.Add(ex.Message);
34              return status;
35          }
36
37          try
38          {
39              fileProcessor.WriteToFile(outputText);
40              status.OutputFileSaved = true;
41          }
42          catch (Exception ex)
43          {
44              status.Errors.Add(ex.Message);
```

```
45              return status;
46          }
47
48          return status;
49      }
50  }
```

If you now compile and run the program, it will work in exactly the same way as it did before. However, now we can write unit tests for it very easily.

The interfaces can be mocked up, so the methods on them return expected values. And this will enable us to test the logic of the method in isolation from any other components:

```
1   using Moq;
2   using TextToHtmlConvertor;
3   using Xunit;
4
5   namespace TextToHtmlConvertorTests;
6
7   public class TextConversionCoordinatorTests
8   {
9       private readonly TextConversionCoordinator coordinato\
10  r;
11      private readonly Mock<IFileProcessor> fileProcessorMo\
12  q;
13      private readonly Mock<IMdTextProcessor> textProcessor\
14  Moq;
15
16      public TextConversionCoordinatorTests()
17      {
18          fileProcessorMoq =
19              new Mock<IFileProcessor>();
20          textProcessorMoq =
21              new Mock<IMdTextProcessor>();
```

```
22          coordinator =
23              new TextConversionCoordinator(
24                  fileProcessorMoq.Object, textProcessorMoq\
25  .Object);
26      }
27
28
29      // This is a scenario that tests TextConversionCoordi\
30  nator under the normal circumstances.
31      // The dependency methods have been set up for succes\
32  sful conversion.
33      [Fact]
34      public void CanProcessText()
35      {
36          fileProcessorMoq
37              .Setup(p => p.ReadAllText())
38              .Returns("input");
39          textProcessorMoq
40              .Setup(p => p.ConvertMdText("input"))
41              .Returns("altered input");
42
43          var status = coordinator.ConvertText();
44
45          Assert.True(status.TextExtractedFromFile);
46          Assert.True(status.TextConverted);
47          Assert.True(status.OutputFileSaved);
48          Assert.Empty(status.Errors);
49      }
50
51      // This is a scenario that tests TextConversionCoordi\
52  nator where the text hasn't been changed.
53      // The dependency methods have been set up accordingl\
54  y.
55      [Fact]
56      public void CanDetectUnconvertedText()
```

```
57      {
58          fileProcessorMoq
59              .Setup(p => p.ReadAllText())
60              .Returns("input");
61          textProcessorMoq
62              .Setup(p => p.ConvertMdText("input"))
63              .Returns("input");
64
65          var status = coordinator.ConvertText();
66
67          Assert.True(status.TextExtractedFromFile);
68          Assert.False(status.TextConverted);
69          Assert.True(status.OutputFileSaved);
70          Assert.Empty(status.Errors);
71      }
72
73      // This is a scenario that tests TextConversionCoordi\
74  nator where the text hasn't been read.
75      // The dependency methods have been set up accordingl\
76  y.
77      [Fact]
78      public void CanDetectUnsuccessfulRead()
79      {
80          fileProcessorMoq
81              .Setup(p => p.ReadAllText())
82              .Throws(new Exception("Read error occurred.")\
83  );
84
85          var status = coordinator.ConvertText();
86
87          Assert.False(status.TextExtractedFromFile);
88          Assert.False(status.TextConverted);
89          Assert.False(status.OutputFileSaved);
90          Assert.Single(status.Errors);
91          Assert.Equal(
```

```
92              "Read error occurred.", status.Errors.First()\
93  );
94      }
95
96      // This is a scenario that tests TextConversionCoordi\
97  nator where an attempt to convert the text thrown an erro\
98  r.
99      // The dependency methods have been set up accordingl\
100 y.
101     [Fact]
102     public void CanDetectUnsuccessfulConvert()
103     {
104         fileProcessorMoq
105             .Setup(p => p.ReadAllText())
106             .Returns("input");
107         textProcessorMoq
108             .Setup(p => p.ConvertMdText("input"))
109             .Throws(new Exception("Convert error occurred\
110 ."));
111
112         var status = coordinator.ConvertText();
113
114         Assert.True(status.TextExtractedFromFile);
115         Assert.False(status.TextConverted);
116         Assert.False(status.OutputFileSaved);
117         Assert.Single(status.Errors);
118         Assert.Equal(
119             "Convert error occurred.", status.Errors.Firs\
120 t());
121     }
122
123     // This is a scenario that tests TextConversionCoordi\
124 nator where an attempt to save the file thrown an error.
125     // The dependency methods have been set up accordingl\
126 y.
```

```
127     [Fact]
128     public void CanDetectUnsuccessfulSave()
129     {
130         fileProcessorMoq
131             .Setup(p => p.ReadAllText())
132             .Returns("input");
133         textProcessorMoq
134             .Setup(p => p.ConvertMdText("input"))
135             .Returns("altered input");
136         fileProcessorMoq
137             .Setup(p => p.WriteToFile("altered input"))
138             .Throws(new Exception("Unable to save file.")\
139 );
140 
141         var status = coordinator.ConvertText();
142 
143         Assert.True(status.TextExtractedFromFile);
144         Assert.True(status.TextConverted);
145         Assert.False(status.OutputFileSaved);
146         Assert.Single(status.Errors);
147         Assert.Equal(
148             "Unable to save file.", status.Errors.First()\
149 );
150     }
151 }
```

What we are doing here is checking whether the outputs are as expected and whether specific methods on specific dependencies are being called. I have used the `Moq` NuGet package to verify the latter. This is also the library that allowed me to mock interfaces and set up return values from the methods.

Due to dependency inversion, we were able to write unit tests to literally cover every possible scenario of what `TextConversionCoordinator` can do.

Dependency inversion is not only useful in unit tests

Although I have given unit tests as an example of why the dependency inversion principle is important, the importance of this principle goes far beyond unit tests.

In our working program, we could pass an implementation of `IFileProcessor` that, instead of reading a file on the disk, reads a file from a web location. After all, `TextConversionCoordinator` doesn't care which file the text was extracted from, only that the text was extracted.

So, dependency inversion will add flexibility to our program. If the external logic changes, `TextConversionCoordinator` will be able to handle it just the same. No changes will need to be applied to this class.

The opposite of the dependency inversion principle is tight coupling. And when this occurs, the flexibility in your program will disappear. This is why tight coupling must be avoided.