3. Liskov substitution principle

In this chapter, we will cover the Liskov substitution principle. I will explain why this principle is important and will provide an example of its usage in C#.

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

https://github.com/fiodarsazanavets/Dotnet-CSharp-SOLID-demo/tree/master/3-liskov-substitution-principle

What is Liskov substitution principle

Liskov substitution principle was initially introduced by Barbara Liskov, an American computer scientist, in 1987. The principle states that if you substitute a sub-class with any of its derived classes, the behavior of the program should not change.

This principle was introduced specifically with inheritance in mind, which is an integral feature of object-oriented programming. Inheritance allows you to extend the functionality of classes or modules (depending on what programming language you use). So, if you need a class with some new functionality that is closely related to what you already have in a different class, you can just inherit it from the existing class instead of creating a completely new one.

When inheritance is applied, any object-oriented programming language will allow you to insert an object that has the derived class as its data type into a variable or parameter that expects an object of the subclass. For example, if you had a base class called `Car`, you could create another class that inherits from it and is called

`SportsCar`. In this case, an instance of `SportsCar` is also `Car`, just like it would have been in real life. Therefore a variable or parameter that is of a type `Car` would be able to be set to an instance of `SportsCar`.

And this is where a potential problem arises. There may be a method or a property inside the original `Car` class that uses some specific behavior and other places in the code have been written to expect that specific behavior from the instances of `Car` objects. However, inheritance allows those behaviors to be completely overridden.

If the derived class overrides some of the properties and methods of the subclass and modifies their behavior, then passing an instance of the derived class into the places that expect the subclass may cause unintended consequences. And this is exactly the problem that the Liskov substitution principle was designed to address.

Implementing Liskov substitution principle in C#

In our previous article where we covered the open-closed principle, we have ended up with a solution that reads the textual content of a file, encloses every paragraph in the `p` HTML tags, and makes a conversion of certain Markdown markers into equivalent HTML tags.

And this is what we have ended up with.

We have the `TextProcessor` base class that performs the basic processing of paragraphs in the text that has been passed to it:

```
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();
}
```

And we have a class that derives from it, which is called `MdTextProcessor`. It overrides the `ConvertText` method by adding some processing steps to it. Basically, it checks the text for specific

Markdown markers and replaces them with corresponding HTML tags. Both the markers and the tags are fully configurable via a dictionary.

```
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);
    }
}
```

This structure implements the open-closed principle quite well, but it doesn't implement the Liskov substitution principle. Although the overridden `ConvertText` makes a call to the original method in the base class and calling this method on the derived class will still process the paragraphs, the method implements some additional logic, which may produce completely unexpected results. I will demonstrate this via a unit test.

So, this is a test I have written to validate the basic functionality of the original `ConvertText` method. We initialize an instance of the `TextProcessor` object in the constructor, pass some arbitrary input text, and then check whether the expected output text has been produced.

```
using TextToHtmlConvertor;
using Xunit;


namespace TextToHtmlConvertorTests;


public class TextProcessorTests
{
   private readonly TextProcessor textProcessor;
```

```


    public TextProcessorTests()
    {
        textProcessor = new TextProcessor();
    }


    [Fact]
    public void CanConvertText()
    {
        var originalText = "This is the first paragraph. I\
t has * and *.\r\n" +
            "This is the second paragraph. It has ** and *\
*.";
        var expectedSting = "<p>This is the first paragrap\
h. It has * and *.</p>\r\n" +
        "<p>This is the second paragraph. It has ** and **\
.</p>\r\n" +
        "<br/>\r\n";
        Assert.Equal(expectedSting, textProcessor.ConvertT\
ext(originalText));
    }
}
```

Please note that we have deliberately inserted some symbols into the input text that have a special meaning in Markdown document format. However, `TextProcessor` on its own is completely agnostic of Markdown, so those symbols are expected to be ignored. The test will therefore happily pass.

As our `textProcessor` variable is of type `TextProcessor`, it will happily be set to an instance of `MdTextProcessor`. So, without modifying our test method in any way or changing the data type of the `textProcessor` variable, we can assign an instance of `MdTextProcessor` to the variable:

```
using TextToHtmlConvertor;
using Xunit;


namespace TextToHtmlConvertorTests;


public class TextProcessorTests
{
   private readonly TextProcessor textProcessor;
   public TextProcessorTests()
   {
       var tagsToReplace =
           new Dictionary<string, (string, string)>
           {
               { "**", ("<strong>", "</strong>") },
               { "*", ("<em>", "</em>") },
               { "~~", ("<del>", "</del>") }
           };
       textProcessor = new MdTextProcessor(tagsToReplace);
   }


   [Fact]
   public void CanConvertText()
   {
       var originalText = "This is the first paragraph. I\
t has * and *.\r\n" +
           "This is the second paragraph. It has ** and *\
*.";
       var expectedSting = "<p>This is the first paragrap\
h. It has * and *.</p>\r\n" +
           "<p>This is the second paragraph. It has ** an\
d **.</p>\r\n" +
           "<br/>\r\n";
```

```
        Assert.Equal(expectedSting, textProcessor.ConvertT\
ext(originalText));
    }
}
```

The test will now fail. The output from the `ConvertText` method will change, as those Markdown symbols will be converted to HTML tags. And this is exactly how other places in your code may end up behaving differently from how they were intended to behave.

However, there is a very easy way of addressing this issue. If we go back to our `MdTextProcessor` class and change the override of the `ConvertText` method into a new method that I called `ConvertMdText` without changing any of its content, our test will, once again, pass.

```
namespace TextToHtmlConvertor;

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

We still have our code structured in accordance with the single responsibility principle, as the method is purely responsible for converting text and nothing else. We still have 100% saturation, as the new method fully relies on the existing functionality from the base class, so our inheritance wasn't pointless.

We are still acting in accordance with the open-closed principle, but we no longer violate the Liskov substitution principle. Every instance of the derived class will have the base class functionality inherited, but all of the existing functionality will work exactly as it did in the base class. So, using objects made from derived classes will not break any existing functionality that relies on the base class.

Conclusion

The Liskov substitution principle is a pattern of coding that will help to prevent unintended functionality from being introduced into the code when you extend existing classes via inheritance.

However, certain language features may give less experienced developers an impression that it's OK to write code in violation of this principle. For example, the `virtual` keyword in C# may seem like it's even encouraging people to ignore this principle. And sure enough, when an abstract method is overridden, nothing will break, as the original method didn't have any implementation details. But a virtual method would have already had some logic inside of it; therefore overriding it will change the behavior and would probably violate the Liskov substitution principle.

The important thing to note, however, is that a principle is not the same as a law. While a law is something that should always be applied, a principle should be applied in the majority of cases. And sometimes there are situations where violating a certain principle makes sense.

Also, overriding virtual methods in C# won't necessarily violate Liskov substitution principle. The principle will only be violated if the output behavior of the overridden method changes. Otherwise, if the override merely changes the class variables that are used in other methods that are only relevant to the derived class, Liskov substitution principle will not be violated.

So, whenever you need to decide whether or not to override a virtual method in C#, use common sense. If you are confident that none of the components that rely on the base class functionality will be broken, then go ahead and override the method, especially if it seems to be the most convenient thing to do. But try to apply the Liskov substitution principle as much as you can.