You can see how testing this DLL would be a manual effort - make a change, build the project, load the DLL, call each method (perhaps with multiple sets of test data), and figure out if the output is correct or not. This is where automated testing comes into play.

Make sure you're back in the `mycalculator` directory and create a new test project.

```
PS C:\Tools\mycalculator> dotnet new xunit -n Calculator.Tests
The template "xUnit Test Project" was created successfully.
```

> xUnit is a free unit testing framework.

As before, add the test project to the main solution.

```
PS C:\Tools\mycalculator> dotnet sln add .\Calculator.Tests\
Project `Calculator.Tests\Calculator.Tests.csproj` added to the solution.
```

Move into the `Calculator.Tests` directory and add a reference to the main Calculator project.

```
PS C:\Tools\mycalculator> cd .\Calculator.Tests\

PS C:\Tools\mycalculator> dotnet add reference ..\Calculator\Calculator.csproj
Reference `..\Calculator\Calculator.csproj` added to the project.
```

Rename `UnitTest1.cs` to `CalculatorTests.cs` and open it for editing.

```
PS C:\Tools\mycalculator\Calculator.Tests> ls

Directory: C:\Tools\mycalculator\Calculator.Tests

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----          14/11/2022    14:53                obj
-a---          14/11/2022    14:53           1006 Calculator.Tests.csproj
-a---          14/11/2022    14:53            112 UnitTest1.cs
-a---          14/11/2022    14:53             19 Usings.cs

PS C:\Tools\mycalculator\Calculator.Tests> move .\UnitTest1.cs .\CalculatorTests.cs
PS C:\Tools\mycalculator\Calculator.Tests> code .
```

To keep the code short, I'm only going to implement tests for the `Add` method.

```
namespace Calculator.Tests;

public class CalculatorTests
{
    [Theory]
    [InlineData(5, 5, 10)]
    [InlineData(-5, -5, -10)]
    [InlineData(-5, 5, 0)]
    [InlineData(int.MinValue, -1, int.MaxValue)]
    public void AddTests(int num1, int num2, int expected)
    {
        var calculator = new Calculator();
        var result = calculator.Add(num1, num2);

        Assert.Equal(expected, result);
    }
}
```

I don't want to dive too deep into what's happening, but here's a brief explanation. The body of `AddTests` is quite self-explanatory - we instantiate a new instance of `Calculator`, call the `Add` method and capture the output in a variable called `result`. The value of `result` is then compared against the value of `expected`.

If they match, the test passes. Otherwise, it will fail.

Furthermore, `AddTests` takes in `num1`, `num2` and `expected`, which are provided by the `InlineData` declarations above it. This means this test will run a total of 4 times, each time with a different set of values.

To execute the tests, run `dotnet test`.

```
PS C:\Tools\mycalculator\Calculator.Tests> dotnet test

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:     4, Skipped:     0, Total:     4, Duration: 6 ms - Calculator.Tests.dll (net6.0)
```

We can see that all 4 tests passed, and the process is much faster and more accurate than manual tests. As you can imaging, this benefit increases the more complex the project becomes.

Commit this project to GitLab.

```
PS C:\Tools\mycalculator> git add . && git commit -m "create calculator tests"
PS C:\Tools\mycalculator> git push -u origin main
```