MiniSpec

How-to Author a Testing Framework in .NET

Rebecca Taylor

Contents

Introduction

Defining the API

Before we begin implementation, we need to decide what we want the end result to *look like*.

What will the experience of authoring tests be like for developers?

Conventional Testing Styles

Developers who have experience authoring tests will likely have used one or more *testing styles*.

There are different schools of thought on what tests should *look like*.

xUnit, Behavior-Driven Development (BDD), Gherkin

The most common testing *syntax styles* are: xUnit, Behavior-Driven Development, and Gherkin.

> Note: Behavior-Driven Development is a software *process*, not a code syntax.
> However, similar *syntax styles* have emerged over the years for these different testing paradigms.

xUnit

xUnit-style syntax typically…

- Uses built-in language constructs for defining "Test Fixtures" (*groups of tests*) and "Tests"
- Provides `setUp` and `tearDown` functions for test setup and cleanup.
- Uses "Assertions" implemented as functions accepting 2 parameters: "Expected" and "Actual"

```
class DogTests {
    Dog dog;
    SetUp() { dog = new Dog(); }
    TestBark() {
        AssertEqual("Woof!", dog.Bark());
    }
}
```

Behavior-Driven Development

BDD-style syntax typically…

- Places an emphasis on using natural language, e.g. `describe("Dog").it("can bark!")`
- Provides `before` and `after` functions for test setup and cleanup.
- Uses natural language for "Expectations", e.g. `x.ShouldEqual()` or `Expect(x).toEqual()`

```
Dog dog;
Describe("Dog", () => {
  Before(() => { dog = new Dog(); });
  It("can bark", () => {
    Expect(dog.Bark()).ToEqual("Woof!");
  });
});
```

Gherkin (aka Cucumber)

From Wikipedia:

> "Cucumber is a software tool that supports behavior-driven development (BDD)."
>
> "Gherkin is the language that Cucumber uses to define test cases."

Gherkin is another BDD testing syntax which places an emphasis on using natural language.

Rather than defining tests in programming code, Gherkin uses a plain text syntax:

```
Feature: Dog
  Scenario: Barking
    Given a dog
    When the dog barks
    Then the output should be "Woof!"
```

Testing libraries for Gherkin allow you to write an interpreter for your Gherkin code:

```
[Then("the output should be \"(.*)\"")]
public void ThenTheOutputShouldBe(string value) {
  Output.Should().Equal(value);
}
```

Choosing a Style to Implement

So, which style(s) should we support with our MiniSpec testing framework project?

You can implement whatever you like! Whatever syntax your heart desires <3

In this book, we will be implementing:

- xUnit syntax where each test is represented by a C# method
- We will embrace the top-level statement support in C# 9 (*just for fun!*)
- We will provide an optional `Expect()` method for assertions

MiniSpec Syntax

```
// Simple tests may simply return a Boolean:
bool TestAnotherThing => 1 == 2;

// Developers may optionally include our Expect() method.
using static MiniSpec.Expect;

// Expect() can be used with simple one-line tests:
bool TestMoreThings => Expect(Foo).ToEqual("Bar");

// Or define full methods (Note: using a class is optional)
void MyTest() {
  Expect(TheAnswer).ToEqual(42);
}

// Support for setup and teardown functionality
void SetUp() { /* do something */ }
void TearDown() { /* do something */ }

// Tests may also be grouped within a class
class MyTests {
  bool PassingTest => true;

  // Or even grouped within a method
  static void Group() {
    bool LocalTestFunction() => Expect("This Syntax").To.Work.OK;
  }
}
```

Test-Driven Test Development

We will test-drive the development of our testing framework (*test-driven test development!*)

As we're using Test-Driven Development (TDD), the first thing we need is a *failing test*!

Writing a Red Test

We will be using Behavior-Driven Development, so we'll start off by testing some behavior.

Project Setup

Create an project folder somewhere. This is where you'll be writing the test framework.

```
mkdir MiniSpec
cd MiniSpec
```

> Consider making the folder a git repository to save changes as you walk thru this book:
>
> ```
> git init
> ```

Let's create a test project and write tests *pretending* that MiniSpec already works:

```
dotnet new console -n MyTests
```

> *A new `console` projects? Wait. What? Why in the… what? So: only console projects support the new top-level statements in C# 9, so let's define tests in a console project! This will be an optional feature and, well, it's just neato and I'd like to try it out! Let's have fun.*

This will create a new project folder `MyTests/`. Let's go there and write our first test!

We'll create a file containing 2 xUnit-style tests, one which should fail and the other should pass.

Rename the generated `Program.cs` file to `Tests.cs` and replace its content with the following:

Example `Tests.cs` File

```
void TestShouldPass() {
  // Do nothing
}

void TestShouldFail() {
  throw new System.Exception("Kaboom!");
}
```

That's it. No `using` statements. Just a tiny file with 2 methods. They're not even `public`.

Now, we have two options:

- Write **implementation code** to *run these two tests* and **print** out the results
- Write **integration test** which *runs these two tests* and verifies the results are **printed** correctly.

Either approach is valid. We can treat our new `Tests.cs` *as a failing test*, conceptually.

But let's go ahead and setup a real integration test which we can add to during development!

Integration Tests

Back in the root of our project folder, let's create a project using an *existing* .NET testing framework.

At the time of writing, there are a many choices to choose from: `xUnit`, `NUnit`, `MSTest`, and more.

To make this tutorial easier for most developers out there, let's use the most popular one: `xUnit`

Let's make a new `xUnit` test project now by running this command from the *root project folder*:

```
dotnet new xunit -n MiniSpec.Specs
```

This will create a new project folder `MiniSpec.Specs/`. Let's go there and write an integration test!

We'll create a test which:

- Runs `minispec.exe` with the `MyTests.dll` DLL assembly provided as an argument
- Asserts that the output contains text which indicates that `TestShouldPass()` passed
- Asserts that the output contains text which indicates that `TestShouldFail()` failed

> What is `minispec.exe`? It doesn't exist yet, but that's the program we'll make to run tests!

Rename `UnitTest1.cs` to `IntegrationTest.cs` and replace its content with the following:

IntegrationTest.cs

```
using Xunit;

public class IntegrationTest {

    [Fact]
    public void ExpectedSpecsPassAndFail() {
        // Arrange
        var minispecExe = System.IO.File.Exists("minispec.exe") ?
            "minispec.exe" : "minispec"; // No .exe extension on Linux

        using var minispec = new System.Diagnostics.Process {
            StartInfo = {
                RedirectStandardOutput = true, // Get the STDOUT
                RedirectStandardError = true,  // Get the STDERR
                FileName = minispecExe,
                Arguments = "MyTests.dll"
            }
        };

        // Act
        minispec.Start();
        minispec.WaitForExit();
        var stdout = minispec.StandardOutput.ReadToEnd();
        var stderr = minispec.StandardError.ReadToEnd();
        var output = $"{stdout}{stderr}";
        minispec.Kill();

        // Assert
        Assert.Contains("PASS TestShouldPass", output);
        Assert.Contains("FAIL TestShouldFail", output);
        Assert.Contains("Kaboom!", output);
    }
}
```

Review

So, what's happening here?

- We assume that there will be a `minispec.exe` executable (*or simply `minispec` on Linux*).
- We invoke the `minispec.exe` process passing the DLL with our defined tests as an argument.
- We read STDOUT and STDERR from the process result, i.e. all of the program's console output.
- *STDOUT and STDERR are combined because we don't currently care which the results output to.*
- We look for expected messages in the output, e.g. `PASS [testname]` or `FAIL [testname]`

> We're totally making up some of these things as we go along, e.g. the `PASS`/`FAIL` messages. This is how TDD works. We just need to make it fail, then pass, then we can change it later!

Making it Go Green

Our goal now is to make the test pass.

Is our goal to fully implement the testing framework? **No.**

Using TDD our goal now is *simply* to do whatever we need to do to make the test pass.

MiniSpec Project

Back in the root of our project folder, let's create a new project for `minispec.exe`.

Let's make a new `console` project by running this command from the *root project folder*:

```
dotnet new console -n MiniSpec
```

MiniSpec Solution

While we're here in the root project folder, let's create a Solution to make building simpler.

We'll add all of projects which we've created so far: `MyTests`, `MiniSpec.Specs`, and `MiniSpec`

```
dotnet new sln
dotnet sln add MyTests
dotnet sln add MiniSpec.Specs
dotnet sln add MiniSpec
```

If you'd ever like to build all projects at once, now you can run `dotnet build` from this folder.

`minispec.exe`

Build the new `MiniSpec` console project by running `dotnet build` from the `MiniSpec` folder.

If you look in the generated `bin/Debug/*/` folder, you should now see a `MiniSpec.exe` file.

We'd like to make one *minor correction* now and rename the generated executable to `minispec.exe`

We can do this by specifying `<AssemblyName>minispec</AssemblyName>` in the `.csproj` file.

Update `MiniSpec.csproj` to the following:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

Rebuild the project with `dotnet build` and you will see `minispec.exe` in `bin/Debug/*/`

Great! That's the filename we specified in `IntegrationTest.cs`. Let's try running that now!

Run the Integration Test

Back in the `MiniSpec.Specs` project, add project references for `MiniSpec` and `MyTests`:

```
cd MiniSpec.Specs/
dotnet add reference ../MiniSpec
dotnet add reference ../MyTests
```

Now run the tests with `dotnet test` (*excerpt below*)

```
IntegrationTest.ExpectedSpecsPassAndFail [FAIL]
  Failed IntegrationTest.ExpectedSpecsPassAndFail]
  Error Message:
   Assert.Contains() Failure
Not found: PASS TestShouldPass    <---- What We Expected
In value:  Hello World!           <---- Actual Value
  Stack Trace:
     at IntegrationTest.ExpectedSpecsPassAndFail()
Failed!  - Failed:     1, Passed:     0, Skipped:     0, Total:     1
```

Ah ha! The test looked for `"PASS TestShouldPass"` but found `"Hello World!"`

This is fabulous, it means that `minispec.exe` is running correctly!

Take a look at the generated `Program.cs` in the new `MiniSpec` project:

```
using System;

namespace MiniSpec
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

This is where the `"Hello World!"` value is coming from.

Update MiniSpec `Program.cs`

Try updating `MiniSpec/Program.cs` to the following:

```
using System;

Console.WriteLine($"Received Args: {string.Join(", ", args)}");
```

> Where's the `Main` method?
> C# 9 supports top-level statements *used in one file* to define your main program more easily.

And now, still from `MiniSpec.Specs/`, run `dotnet test` again to see the change:

```
$ dotnet test
...
Not found: PASS TestShouldPass
In value:  Received Args: MyTests.dll
...
```

Wonderful. Ok. Our program runs. It gets a list of DLLs. Now let's run the tests in the DLLs!

Discovering Tests in DLLs

Our `minispec.exe` program is currently seeing a list of paths to DLL files.

Let's load the provided DLLs and find our defined test methods inside of them!

Get List of Methods in DLL

First, let's update the test to *print out a list of methods* from the provided DLL.

Update `MiniSpec/Program.cs` to the following:

```
using System;
using System.Reflection;
using System.Runtime.Loader;

foreach (var dll in args) {
    Console.WriteLine($"Loading {dll}");
    var dllPath = System.IO.Path.GetFullPath(dll);
    var assembly = AssemblyLoadContext.Default.LoadFromAssemblyPath(
       dllPath);

    foreach (var type in assembly.GetTypes()) {
        Console.WriteLine($"Found type: {type}");
        foreach (var method in type.GetMethods(BindingFlags.NonPublic |
           BindingFlags.Instance))
            Console.WriteLine($"Instance Method: {method.Name}");
        foreach (var method in type.GetMethods(BindingFlags.NonPublic |
           BindingFlags.Static))
            Console.WriteLine($"Static Method: {method.Name}");
    }
}
```

Review

- Load any argument as a .NET DLL assembly
- Loop over every defined type in the assembly (`args` *is available to top-level statements*)
- Loop over every instance method on the type (*and print out the method name*)
- Loop over every static method on the type (*and print out the method name*)

Run the tests again with `dotnet test` (*excerpt below*)

```
Not found: PASS TestShouldPass
In value:  Loading MyTests.dll
Found type: <Program>$
Instance Method: MemberwiseClone
Instance Method: Finalize
Static Method: <Main>$
Static Method: <<Main>$>g__TestShouldPass|0_0
Static Method: <<Main>$>g__TestShouldFail|0_1
```

The test is still failing (*"Not found: PASS TestShouldPass"*) but we can see new output, which is good!

Even though we did not *explicitly* define it, C# 9 added a `<Program>` class for us.

As you would expect from a console application, this class has a static `<Main>` method.

And it looks like we found the test methods which we defined as top-level statements too!

> Huh. `<<Main>$>g__TestShouldPass|0_0`. I guess *that's* how local methods are represented.

Running Tests in DLLs

What now? Well, remember our goal? *"do whatever we need to do to make the test pass"*

Let's be naive and simply run every static method we find with `Test` in the name.

Update `MiniSpec/Program.cs` to the following:

```
using System;
using System.Linq;
using System.Reflection;
using System.Runtime.Loader;

foreach (var dll in args) {
    var dllPath = System.IO.Path.GetFullPath(dll);
    var assembly = AssemblyLoadContext.Default.LoadFromAssemblyPath(
       dllPath);
    foreach (var type in assembly.GetTypes()) {
        var testMethods = type.GetMethods(BindingFlags.NonPublic |
           BindingFlags.Static)
            .Where(m => m.Name.Contains("Test"));
        foreach (var method in testMethods) {
            try {
```

```
                method.Invoke(null, null);
                Console.WriteLine($"PASS {method.Name}");
            } catch (Exception e) {
                Console.WriteLine($"FAIL {method.Name}");
                Console.WriteLine($"ERROR {e.Message}");
            }
        }
    }
}
```

Run the tests again with `dotnet test` (*excerpt below*)

```
Not found: PASS TestShouldPass
In value:  PASS <<Main>$>g__TestShouldPass|0_0
FAIL <<Main>$>g__TestShouldFail|0_1
ERROR Exception has been thrown by the target of an invocation.
```

Yikes, we tried but a few things are incorrect which we need to fix.

- Name of the test is showing up as `<<Main>$>g__TestShouldPass|0_0`
- ^— this should be: `TestShouldPass`
- Exception message only says *Exception has been thrown by the target of an invocation*
- ^— this should be `Kaboom!`

Fix `Program.cs`

Update `MiniSpec/Program.cs` to the following:

```
using System;
using System.Linq;
using System.Reflection;
using System.Runtime.Loader;
using System.Text.RegularExpressions;

foreach (var dll in args) {
    var dllPath = System.IO.Path.GetFullPath(dll);
    var assembly = AssemblyLoadContext.Default.LoadFromAssemblyPath(
       dllPath);
    foreach (var type in assembly.GetTypes()) {
        var testMethods = type.GetMethods(BindingFlags.NonPublic |
           BindingFlags.Static)
```

```
            .Where(m => m.Name.Contains("Test"));
        foreach (var method in testMethods) {
            var displayName = method.Name;
            if (Regex.IsMatch(displayName, @"[^\w]"))
                displayName =
                    Regex.Match(displayName, @"Test([\w]+)").Value;
            try {
                method.Invoke(null, null);
                Console.WriteLine($"PASS {displayName}");
            } catch (Exception e) {
                Console.WriteLine($"FAIL {displayName}");
                Console.WriteLine($"ERROR {e.InnerException.Message}");
            }
        }
    }
}
```

Run the tests again with `dotnet test` (*excerpt below*)

```
Passed!  - Failed:     0, Passed:     1, Skipped:     0, Total:     1
```

Phew! We did it! Green, passing tests! Goodness gracious! Hooray!

Try it yourself!

```
bin/Debug/*/minispec.exe bin/Debug/*/MyTests.dll
PASS TestShouldPass
FAIL TestShouldFail
ERROR Kaboom!
```

> On Linux: `./bin/Debug/*/minispec bin/Debug/*/MyTests.dll`

Red, Green, Refactor

If you wrote code different from what we have at home, *now is the time to Refactor!*

As the author, I am doing BDD (Book-Driven Development) and refactoring as I go.

At home, *it is really important not to forget the Refactor step!*

In the next section, we'll come up with a list of features to implement and walk thru them.

Planning Phase

We've created a working prototype. Now we need to decide what to make next!

Brainstorm Features

What do we want our wonderful new test framework to provide?

> This is *my personal braindump of ideas* - come up with your own ideas at home!

Command-Line Interface

- [ ] Output should show pretty colors
- [ ] `minispec` should always exit `0` on success or non-zero on failure
- [ ] `minispec --version` - *Print out the current version of minispec*
- [ ] `minispec -l/--list` - *Print out test names instead of running them*
- [ ] `minispec -m/--match [Test Name Matcher]` - *Run a subset of the tests*
- [ ] `minispec -v/--verbose` - *Print output from every test, even passing ones*
- [ ] `minispec -q/--quiet` - *Don't print anything, exit 0 on success or exit 1 on failure*
- [ ] `minispec -n/--no-local` - *Don't consider local functions when searching for tests*
- [ ] `minispec -p/--pattern` - *Provide a custom pattern used to find test methods*
- [ ] `minispec -s/--setup` - *Provide a custom pattern used to find setup methods*
- [ ] `minispec -t/--teardown` - *Provide a custom pattern used to find teardown methods*
- [ ] `minispec -f/--formatter` - *Name of output reporter formatter to use, e.g. TAP*
- [ ] `minispec -d/--dll` - *Provide a custom pattern used to auto-find DLLs*
- [ ] `minispec -c/--config` - *Provide a text configuration file (default `.minispec`)*

Syntax DSL (Domain-Specific Language)

- [ ] Support DLLS which need to load dependencies, including if there are conflicts
- [ ] Support failing if a Test method with a bool return type returns `false`

- [ ] Support running instance methods
- [ ] Invoke parent method(s) before invoking test function (*if local function*)
- [ ] Allow for some local functions within a test function *not* to be run (*use _ prefix*)
- [ ] Detect and run `SetUp` and `TearDown` methods before and after *each run* of a test case
- [ ] Determine and implement a nice way of supporting parameterized tests (DDT)
- [ ] Let `dotnet run` run the tests if you invoke `MiniSpec.Run()`

Assertions & Expectations

- [ ] Should work fine with `xUnit` and `NUnit` and `FluentAssertions` assertions
- [ ] Extensibility so it's easy to add your own `Expect()` assertions
- [ ] `Expect().ToEqual`
- [ ] `Expect().ToContain`
- [ ] `Expect().ToMatch`
- [ ] `Expect(()=> { ... }).ToFail("Kaboom!")`

Distribution

- [ ] `Expect()` should be available on its own via `MiniSpec.Expect`
- [ ] `minispec.exe` should be available on its own via `MiniSpec.Console`
- [ ] `MiniSpec` package should install both the library and the executable
- [ ] Make available via GitHub Packages
- [ ] Make available via MyGet
- [ ] Make available via NuGet

Choose Feature to Implement

Looking at the list, as it is now, it looks pretty daunting.

For the next parts of this book, you'll be able to hop around and implement whichever set of these features that you'd like to (*although some may depend on completing other sections first*).

My recommendation to you is to start by choosing one of these options:

- Something which will make you ***happy***
- Something which is ***easy*** to get done
- Something which provides the most ***value***

Make *sure* that you *test-drive* (*and don't forget the Refactor step!*).

Have fun!

Choose Your Own Adventure