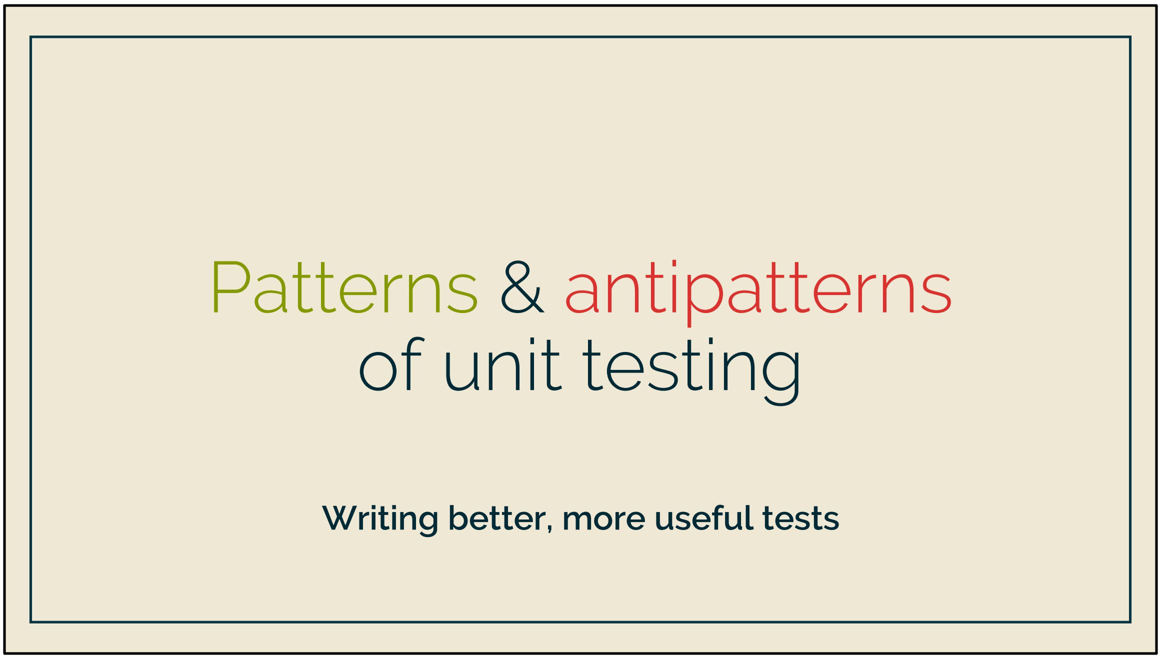
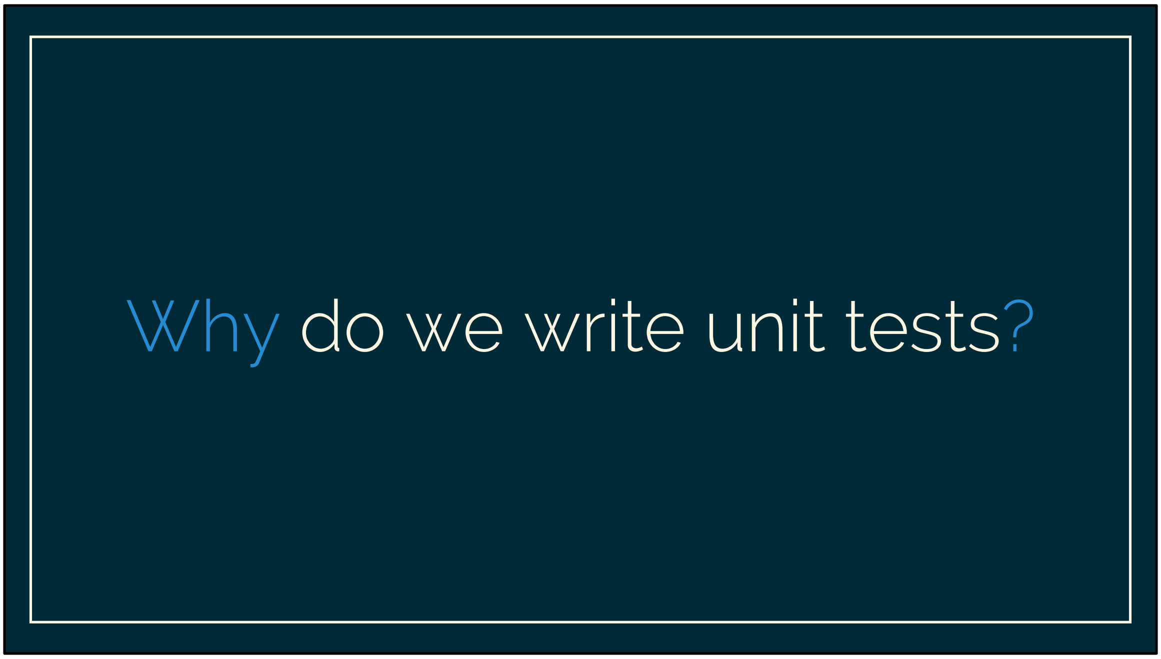


Patterns & antipatterns of unit testing

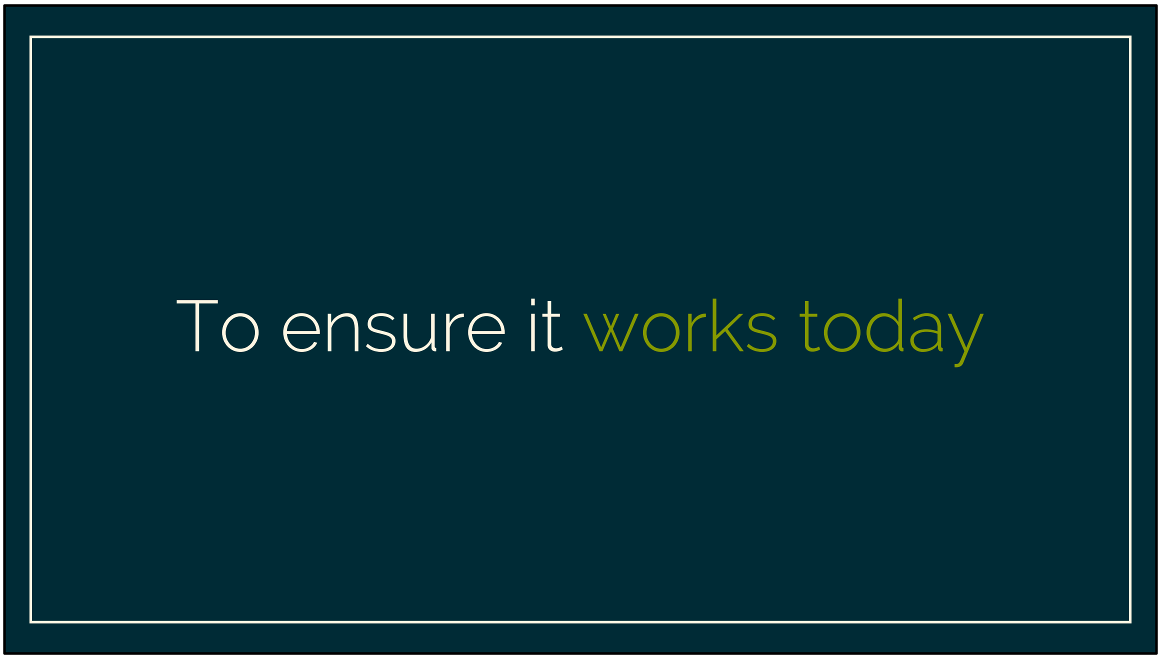
Writing better, more useful tests

Aimed at anyone using Unit Testing – whether beginner or advanced
Examples are written in C# but should be easily understood
Based on my experiences working on long lived codebases

Lets start by setting some context



Why do we write unit tests?

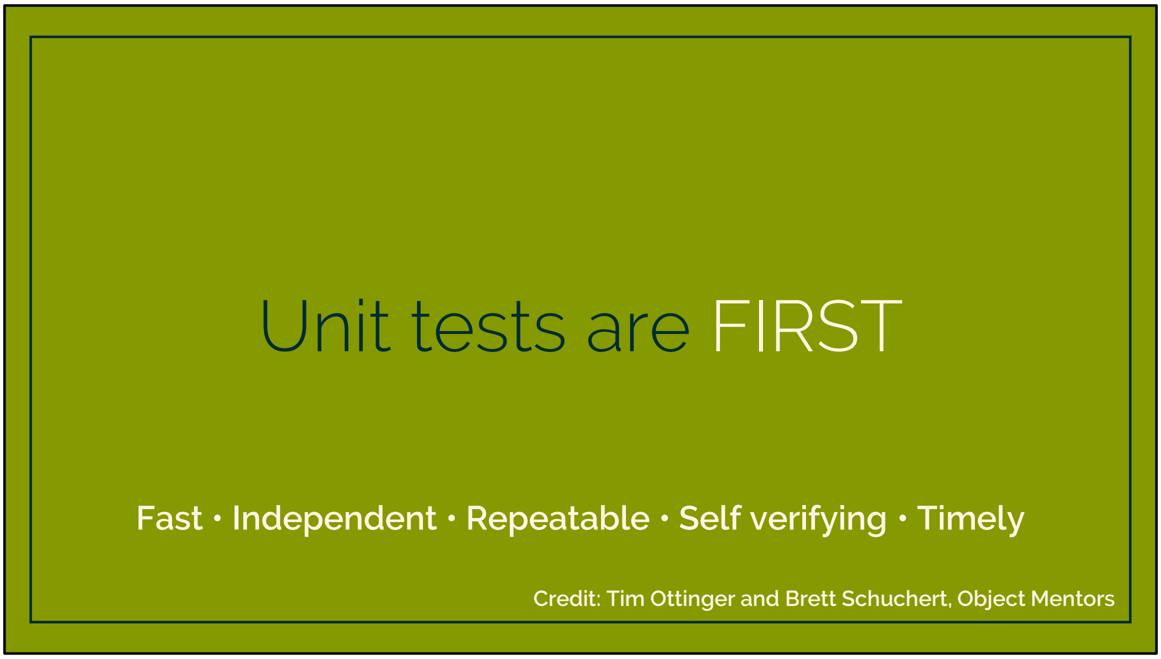


To ensure it works today

When we write it the first time

To ensure it continues to work

As we – or other developers in the future - make improvements and fix bugs

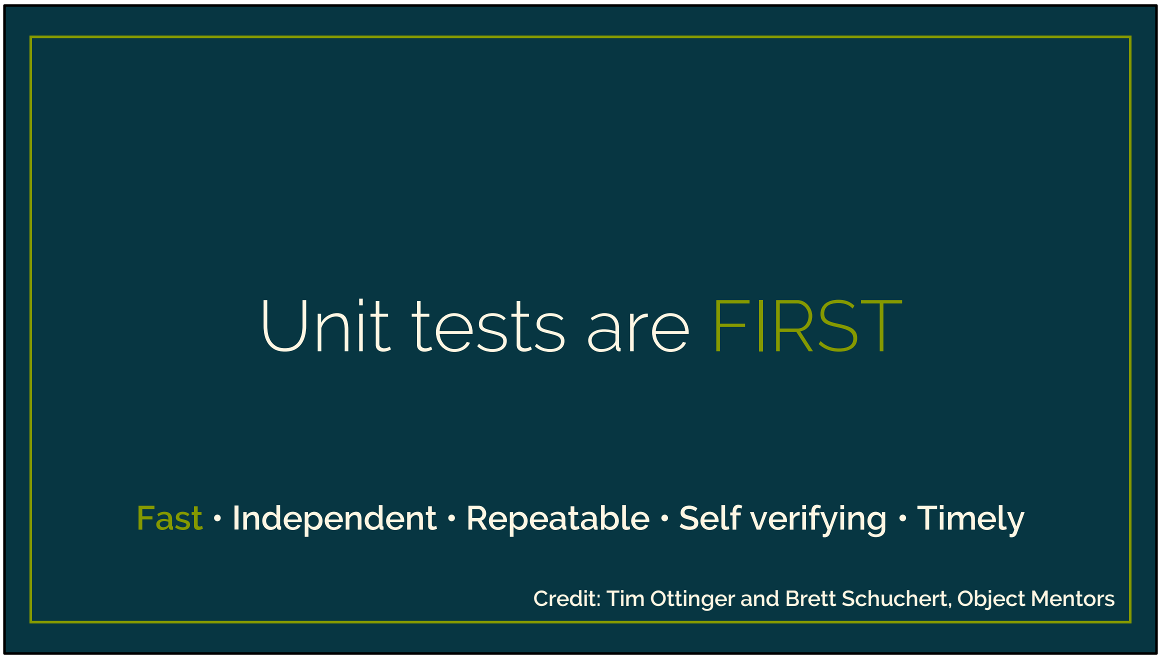


Unit tests are FIRST

Fast • Independent • Repeatable • Self verifying • Timely

Credit: Tim Ottinger and Brett Schuchert, Object Mentors

A useful mnemonic for tests that are useful

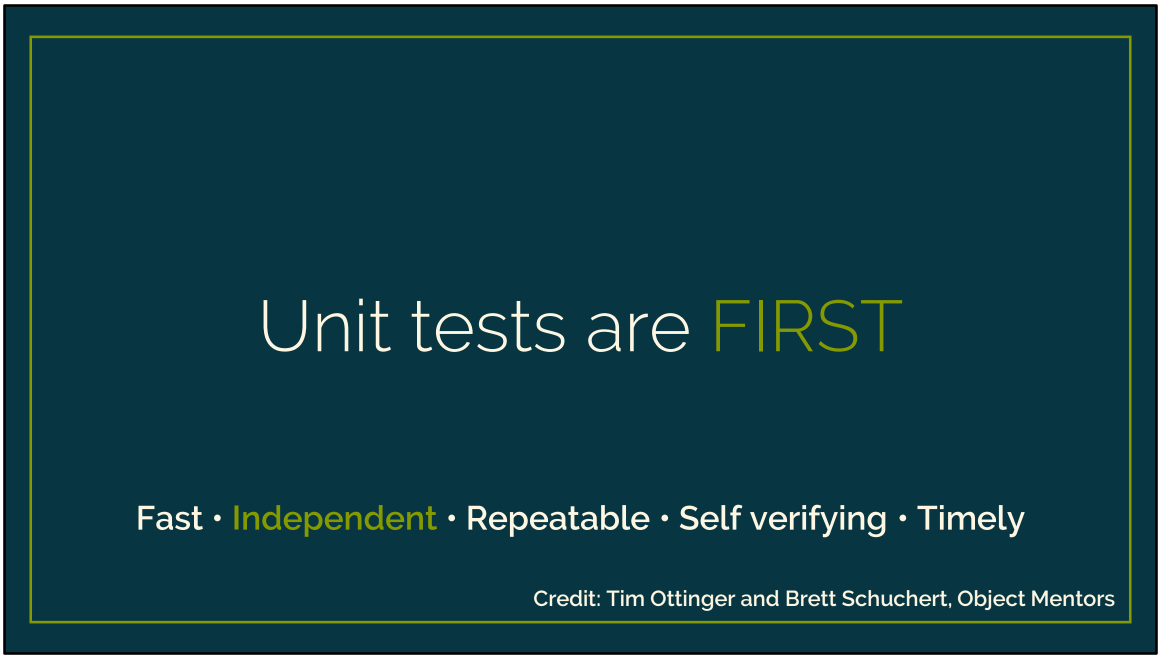


Unit tests are FIRST

Fast • Independent • Repeatable • Self verifying • Timely

Credit: Tim Ottinger and Brett Schuchert, Object Mentors

Fast - Performance is a feature;
Fast enough that developers will routinely run them.
Ideally, these should run in memory, within the single process, without external dependencies (database, file system, cache, webserver etc …)

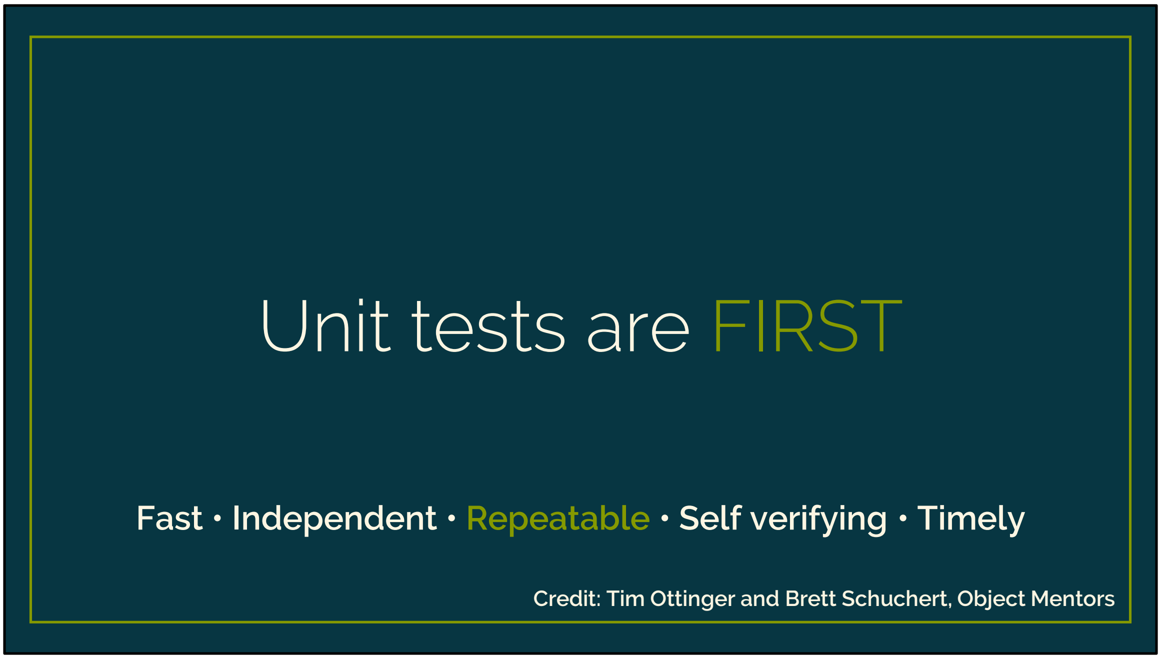


Unit tests are FIRST

Fast • Independent • Repeatable • Self verifying • Timely

Credit: Tim Ottinger and Brett Schuchert, Object Mentors

Independent (aka Isolated) – can be used in any order or subset
Don’t want to create dependencies between tests – these dependencies are usually invisible, and therefore brittle and easily broken;
Some test frameworks (like xUnit) can run tests in parallel, so you might end up with tests that only sometimes pass

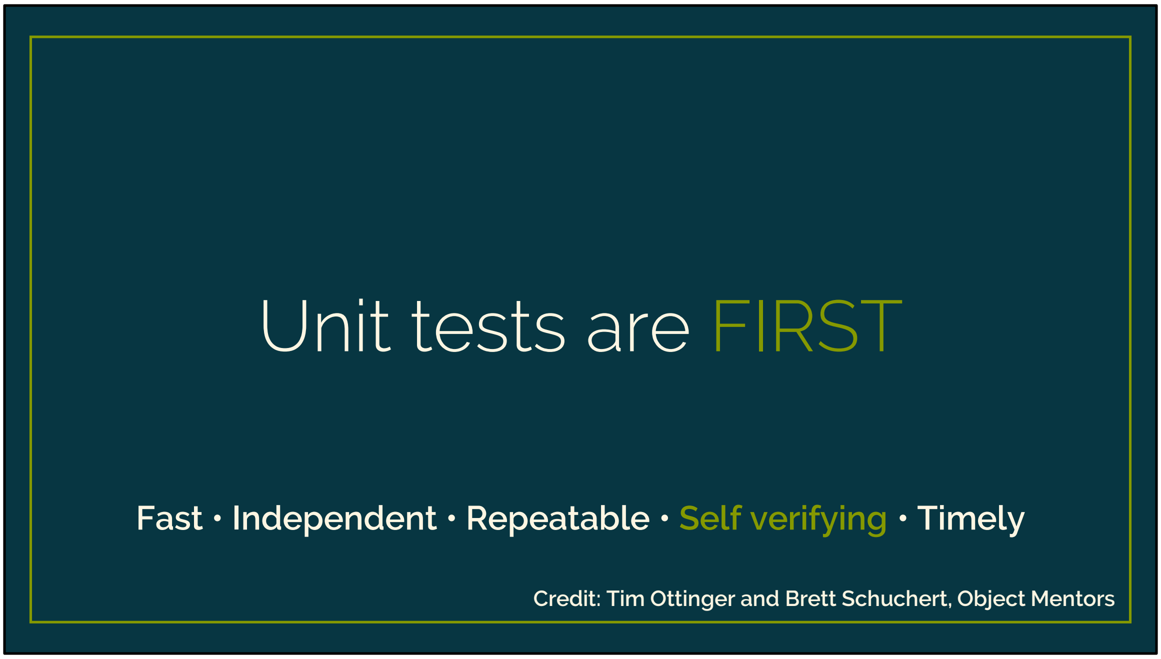


Unit tests are FIRST

Fast • Independent • Repeatable • Self verifying • Timely

Credit: Tim Ottinger and Brett Schuchert, Object Mentors

Repeatable (aka Reliable) - consistency is vital; unreliable tests will be ignored or deleted by other developers

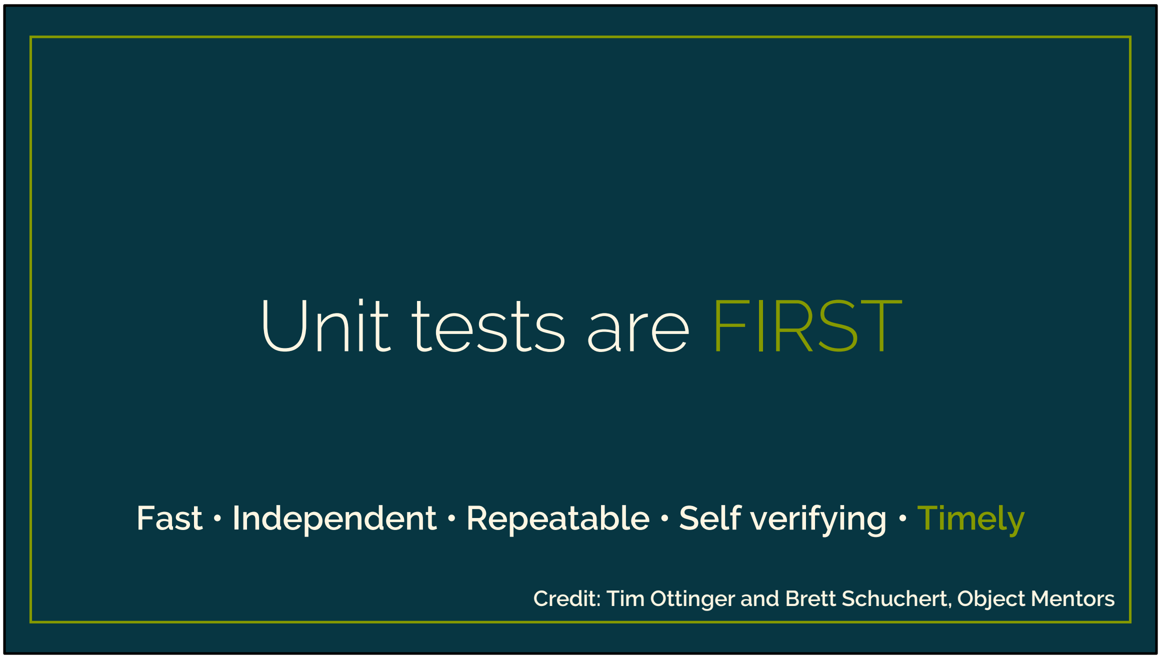


Unit tests are FIRST

Fast • Independent • Repeatable • Self verifying • Timely

Credit: Tim Ottinger and Brett Schuchert, Object Mentors

Self verifying – Pass or fail should be obvious.
Shouldn’t need to query a database, read a log file or fire up a web page to see if the tests passed or failed.

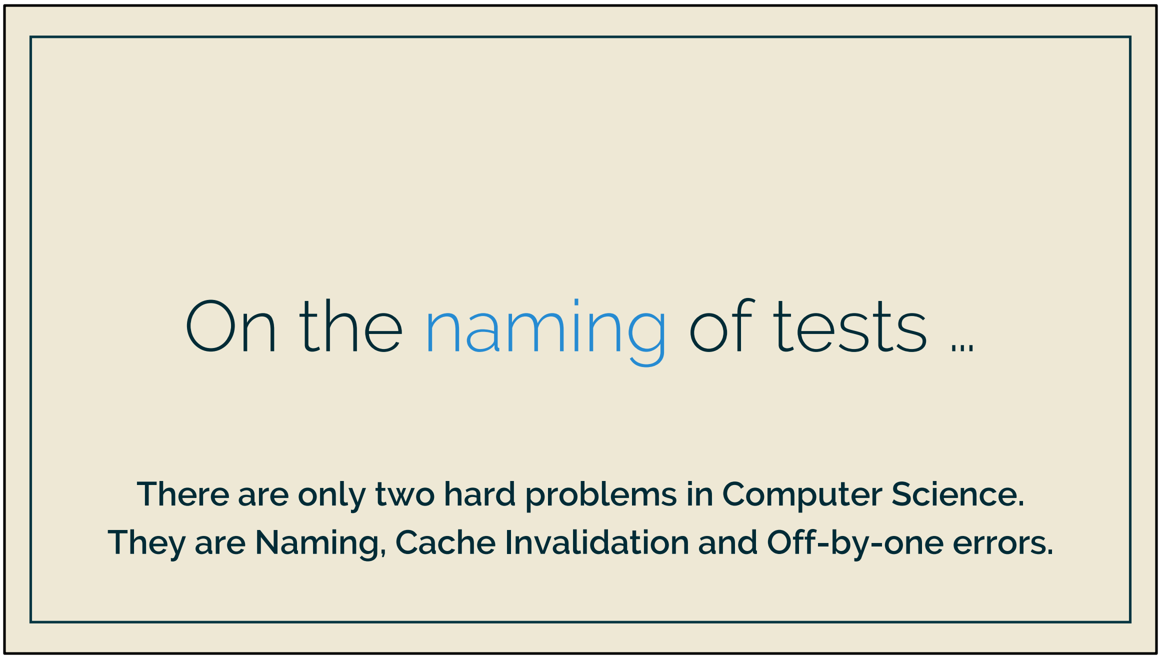


Unit tests are FIRST

Fast • Independent • Repeatable • Self verifying • Timely

Credit: Tim Ottinger and Brett Schuchert, Object Mentors

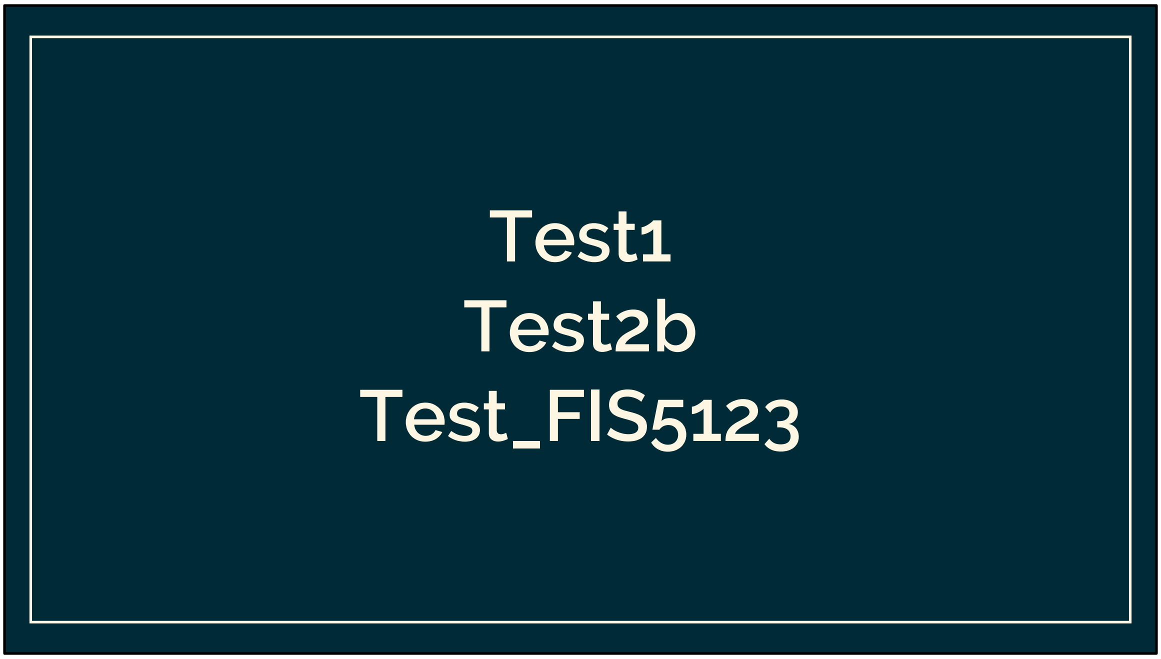
Timely - written just before the code they test.
Good intentions to write tests after the fact (almost) never come to pass.
Write the tests first so you think first about the external interface of the thing you're writing
Tests written afterwards are usually very bound to the specific implementation



On the naming of tests …

**There are only two hard problems in Computer Science.
They are Naming, Cache Invalidation and Off-by-one errors.**

Names are the first thing you encounter
And will remain after everything else is lost
So they should be as good as possible
(This applies to more than just tests)

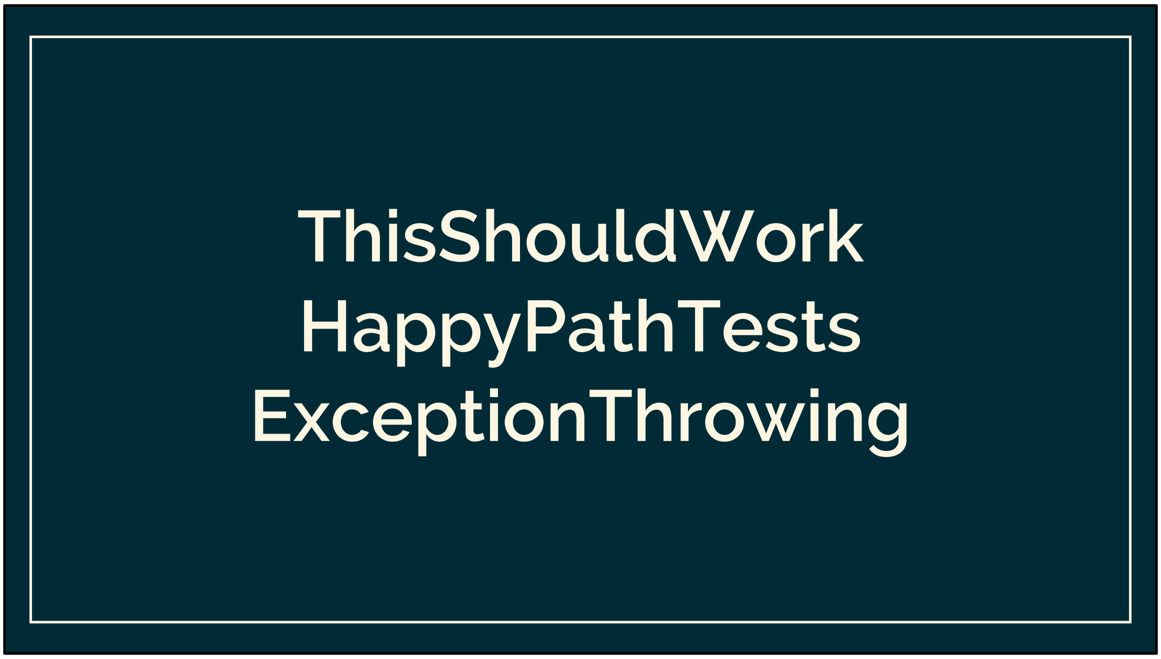


Test1
Test2b
Test_FIS5123

Actual test names from real code

Test1 - numbers aren’t informative

Test2b – You might guess that this has something to do with Test2a … but there was no Test2a
---
Test_FIS5123 - tests with References to bug tracking tools also aren’t informative – they force people to look elsewhere to find out what the test is supposed to do. And what if the other system is offline, or the dev doesn’t have access or the issue has been archived



ThisShouldWork
HappyPathTests
ExceptionThrowing

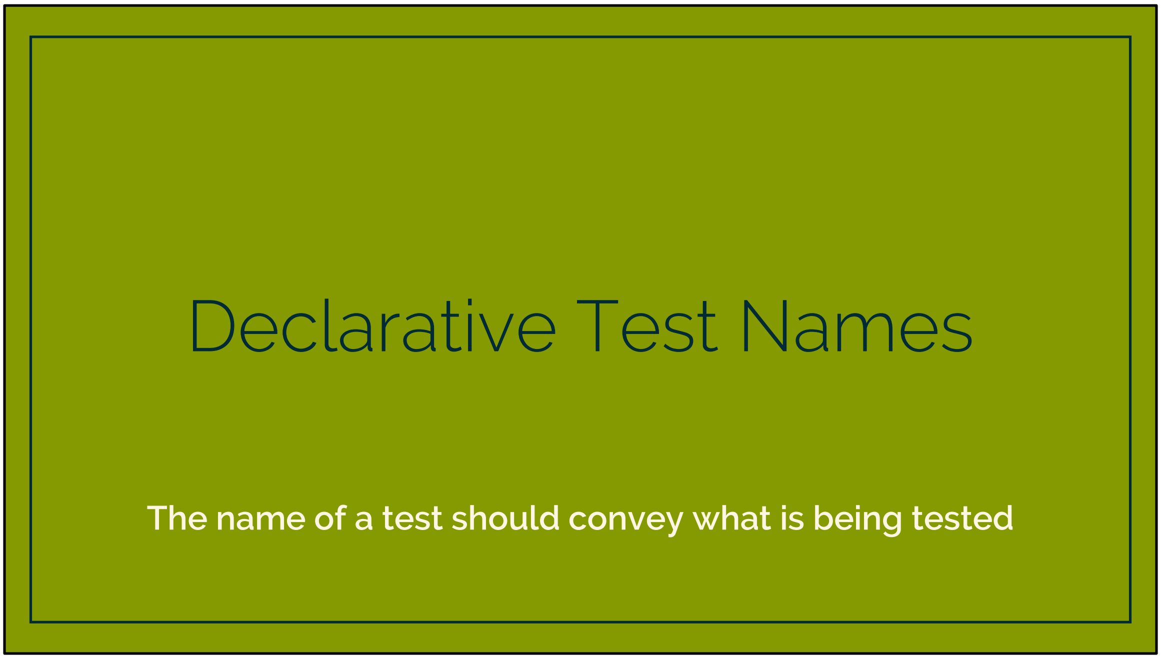
ThisShouldWork – at least it’s confident

HappyPathTests – we can guess

ExceptionThrowing – lots of try/catch blocks

Obtuse Naming

Names that tell you nothing about the test

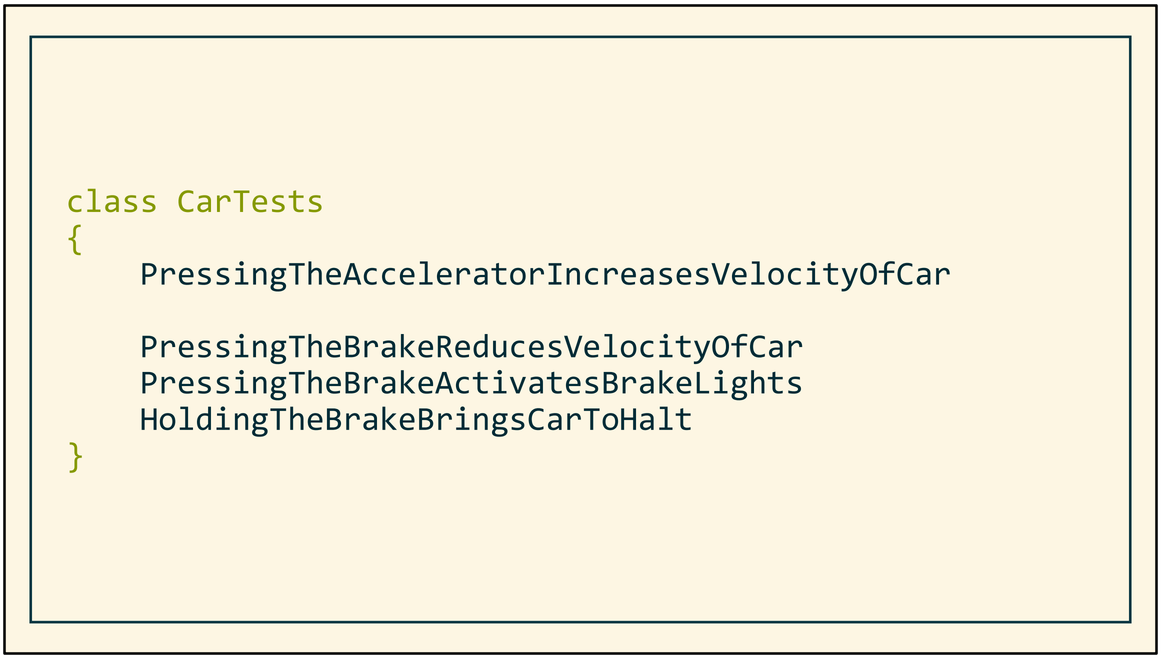


Declarative Test Names

The name of a test should convey what is being tested

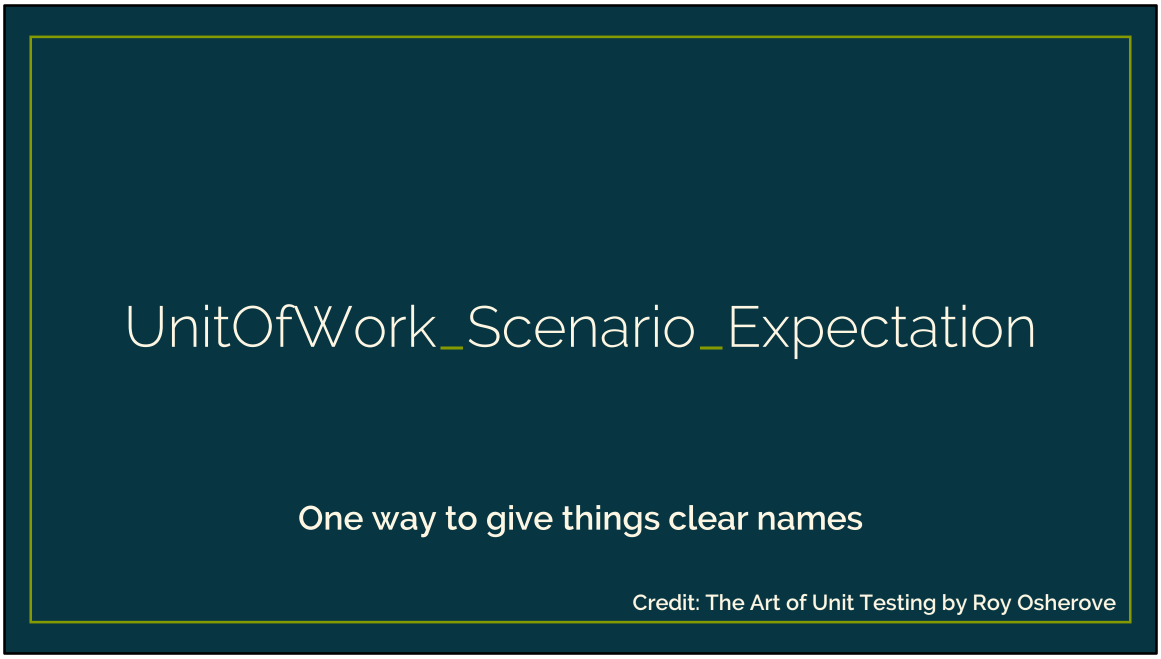
Aka: Make it obvious what the test is checking

These can end up relatively long



```
class CarTests
{
    PressingTheAcceleratorIncreasesVelocityOfCar

    PressingTheBrakeReducesVelocityOfCar
    PressingTheBrakeActivatesBrakeLights
    HoldingTheBrakeBringsCarToHalt
}
```

Accelerator – 1x
Brake – 3x



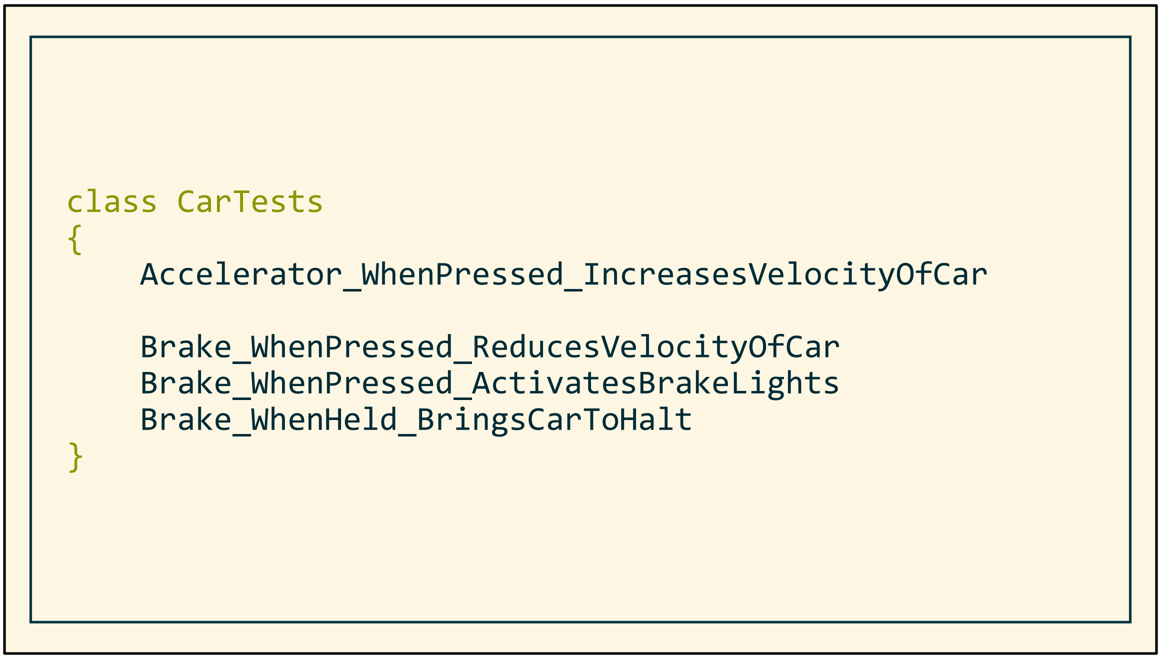
UnitOfWork_Scenario_Expectation

One way to give things clear names

Credit: The Art of Unit Testing by Roy Osherove

UnitOfWork – identify the thing being tested … method/class/feature
Scenario – The story/situation/context of the test
Expectation – what should happen if it works

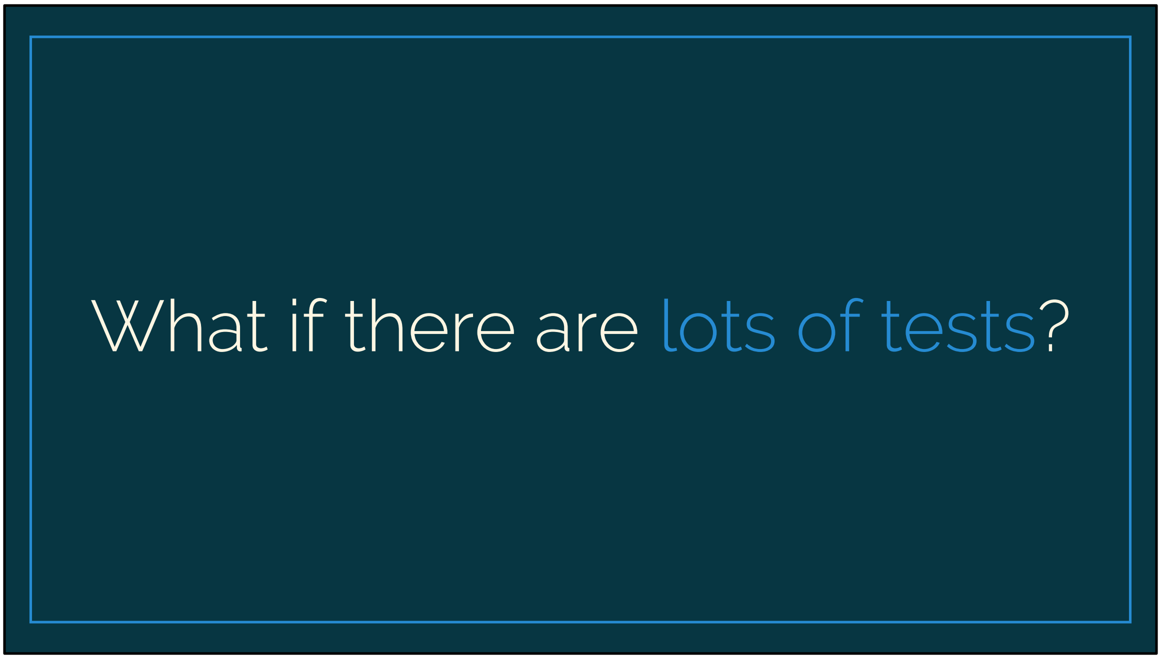
This can lead to longer names – but they’re clear, and that’s more important



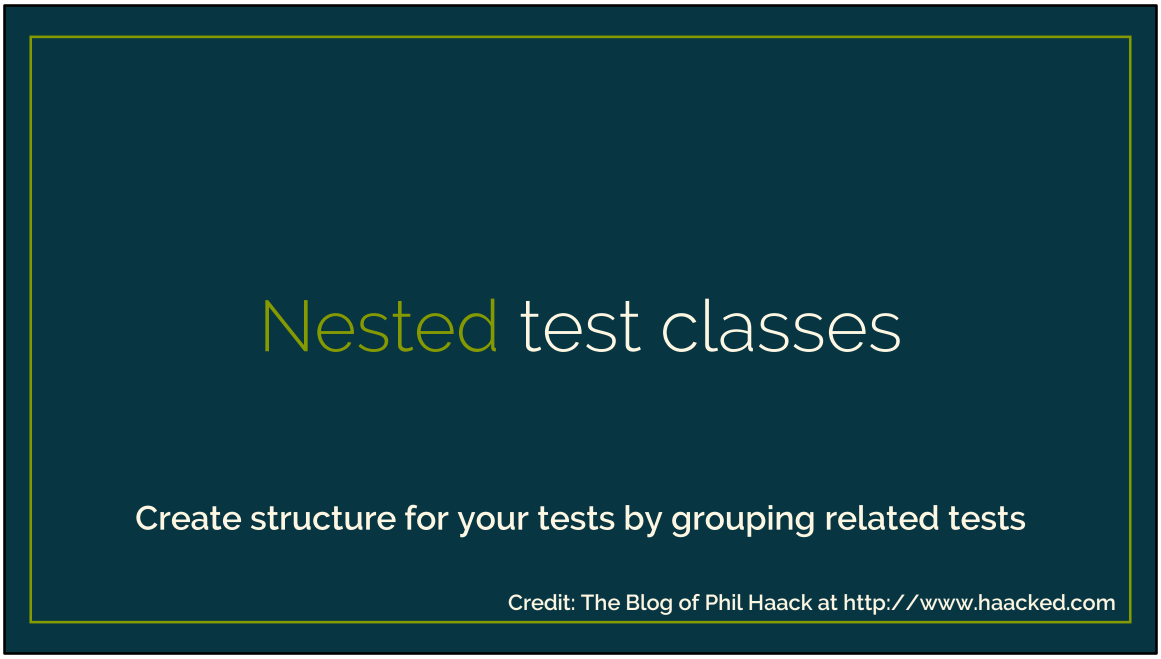
```
class CarTests
{
    Accelerator_WhenPressed_IncreasesVelocityOfCar

    Brake_WhenPressed_ReducesVelocityOfCar
    Brake_WhenPressed_ActivatesBrakeLights
    Brake_WhenHeld_BringsCarToHalt
}
```

Accelerator – 1x
Brake – 3x

When you get a lot of tests for the same thing, adding another layer of grouping can be useful



What if there are lots of tests?

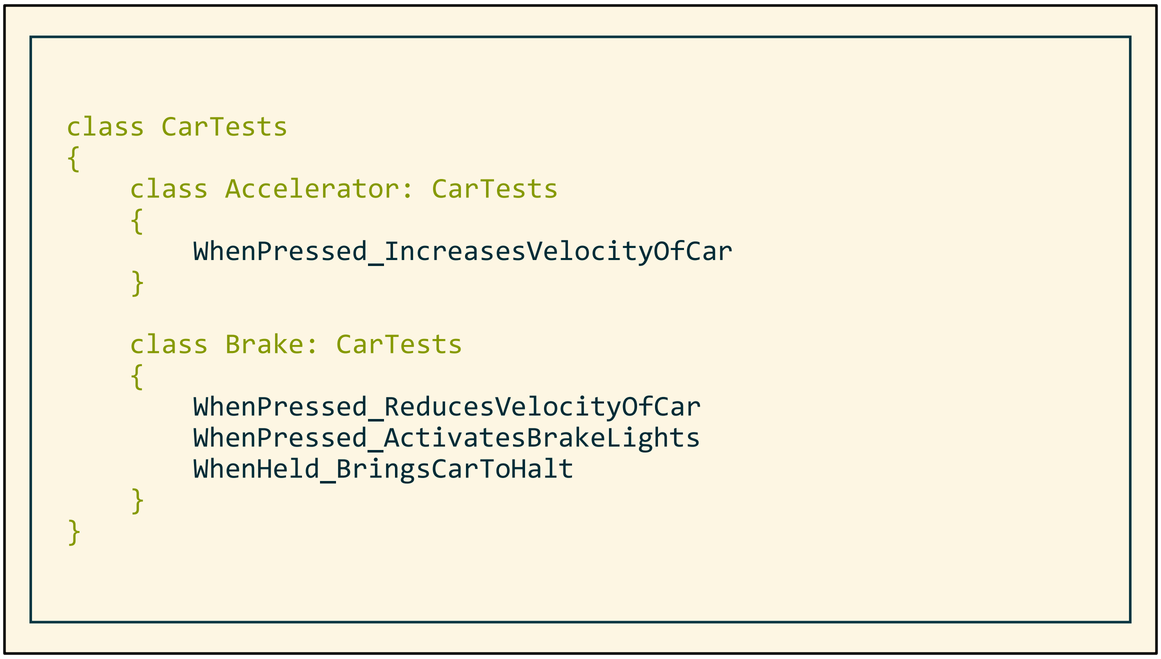


Nested test classes

Create structure for your tests by grouping related tests

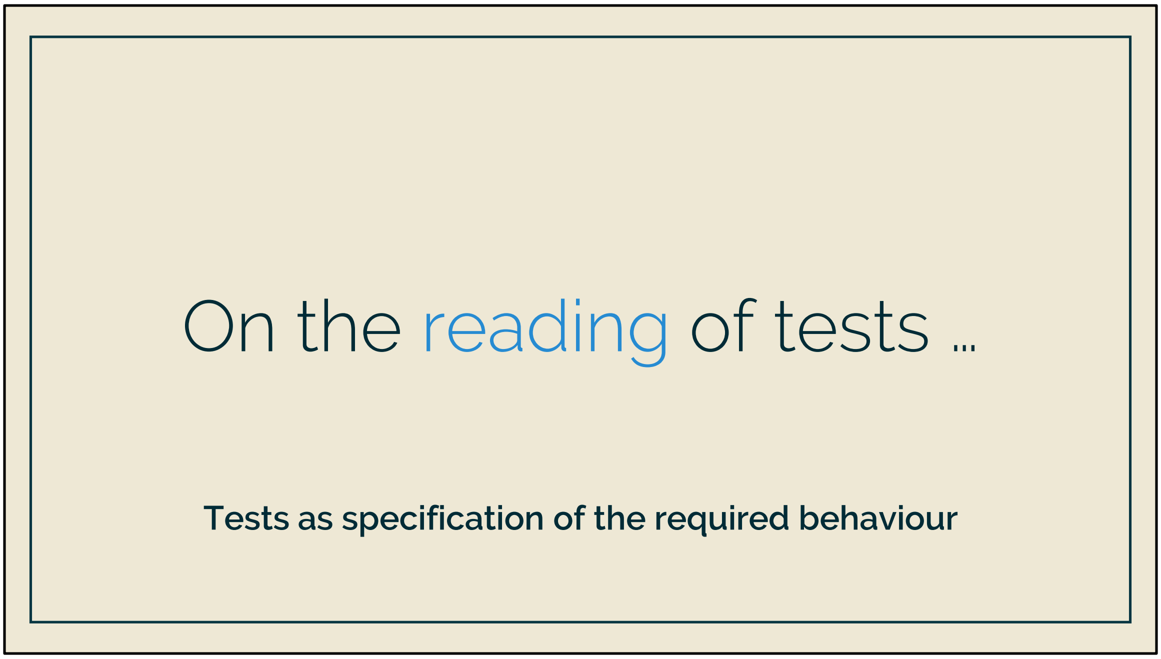
Credit: The Blog of Phil Haack at http://www.haacked.com

Typically group by the UnitOfWork



```
class CarTests
{
    class Accelerator: CarTests
    {
        WhenPressed_IncreasesVelocityOfCar
    }

    class Brake: CarTests
    {
        WhenPressed_ReducesVelocityOfCar
        WhenPressed_ActivatesBrakeLights
        WhenHeld_BringsCarToHalt
    }
}
```

The nested tests descend from CarTests to aid in sharing code



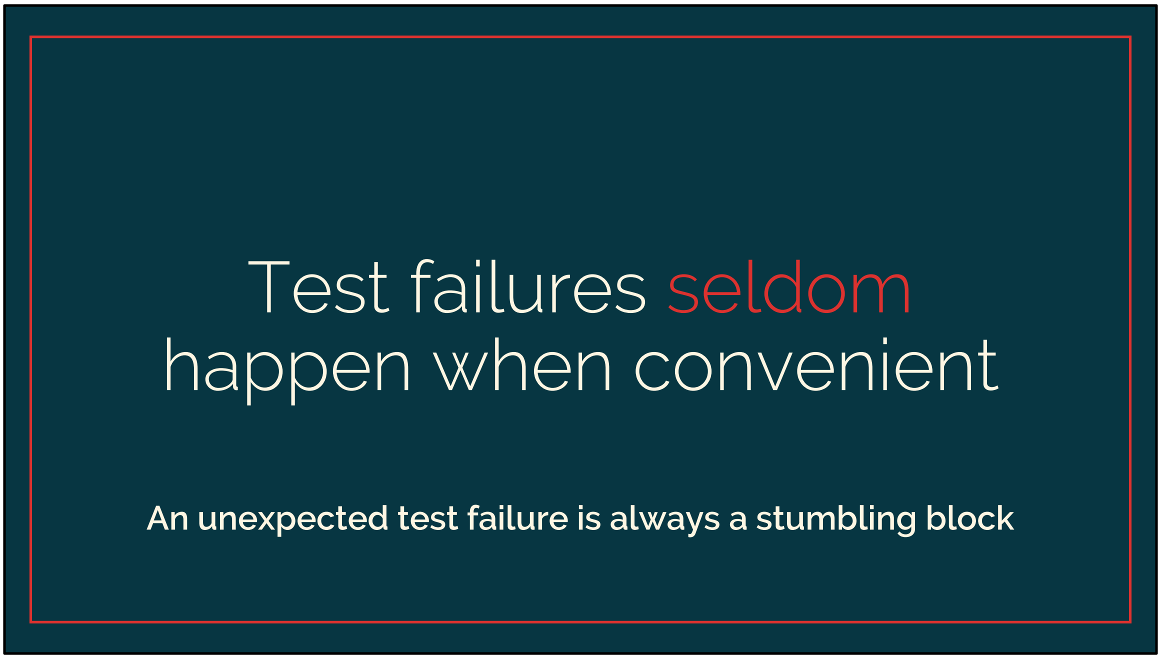
On the reading of tests …

Tests as specification of the required behaviour

It’s vital that tests are easy to read, even more so than with regular production code

---

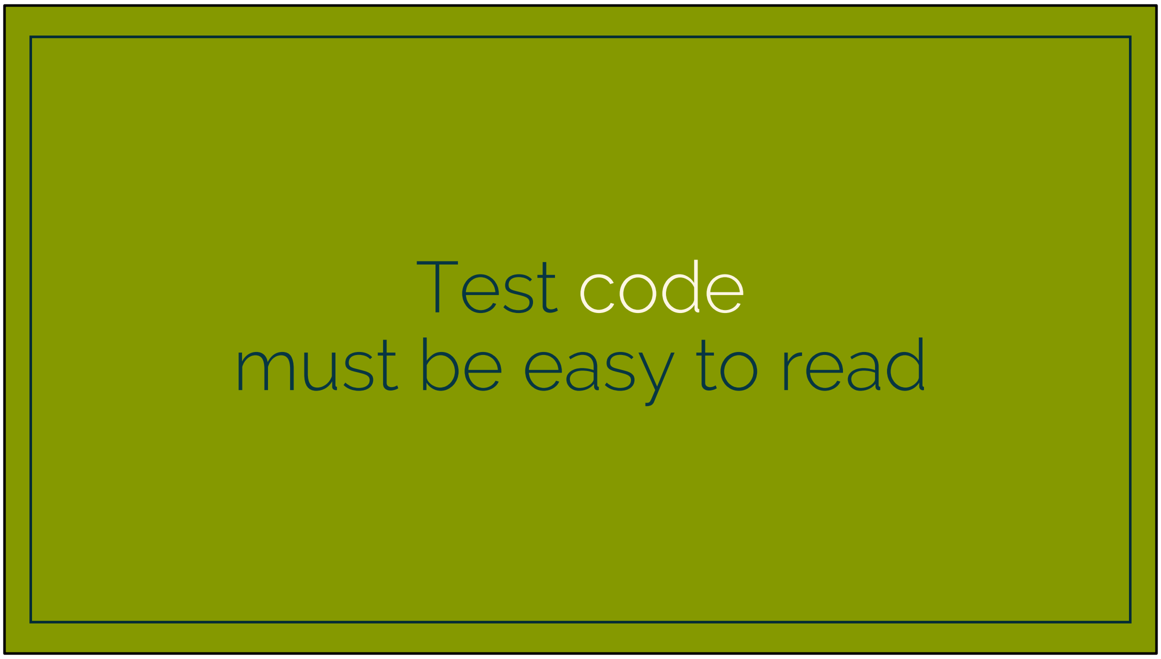
Because the simple truth is that tests never fail at a good time



Test failures seldom happen when convenient

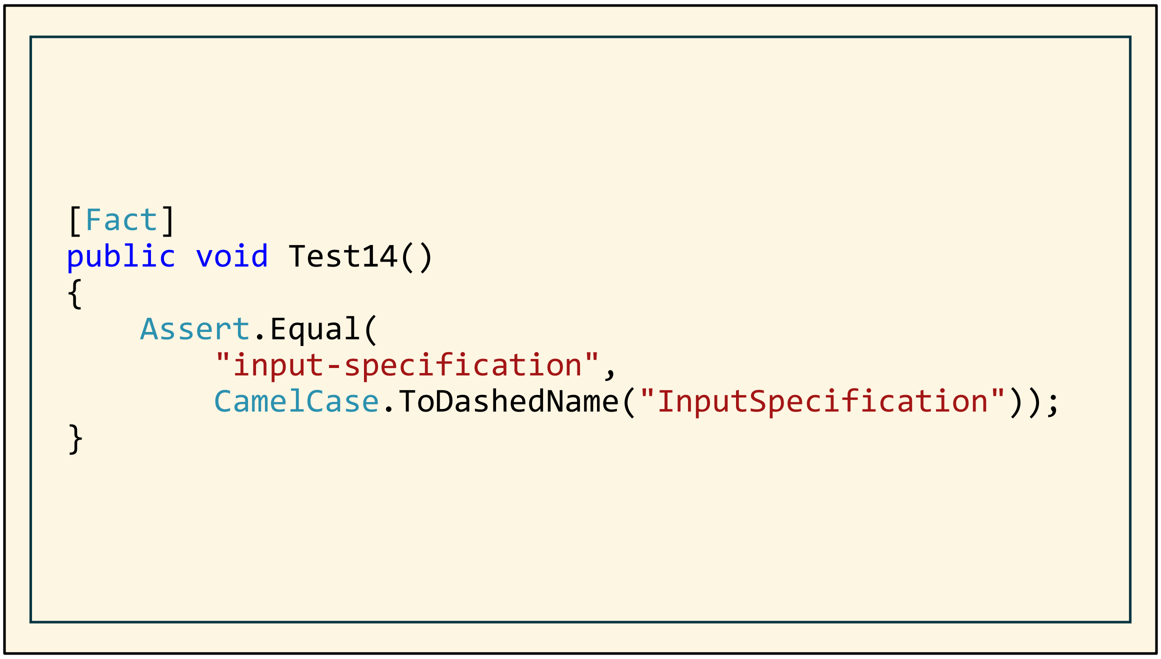
An unexpected test failure is always a stumbling block

An unexpected test failure always requires
a detour away from the developers intended goal



Test code
must be easy to read

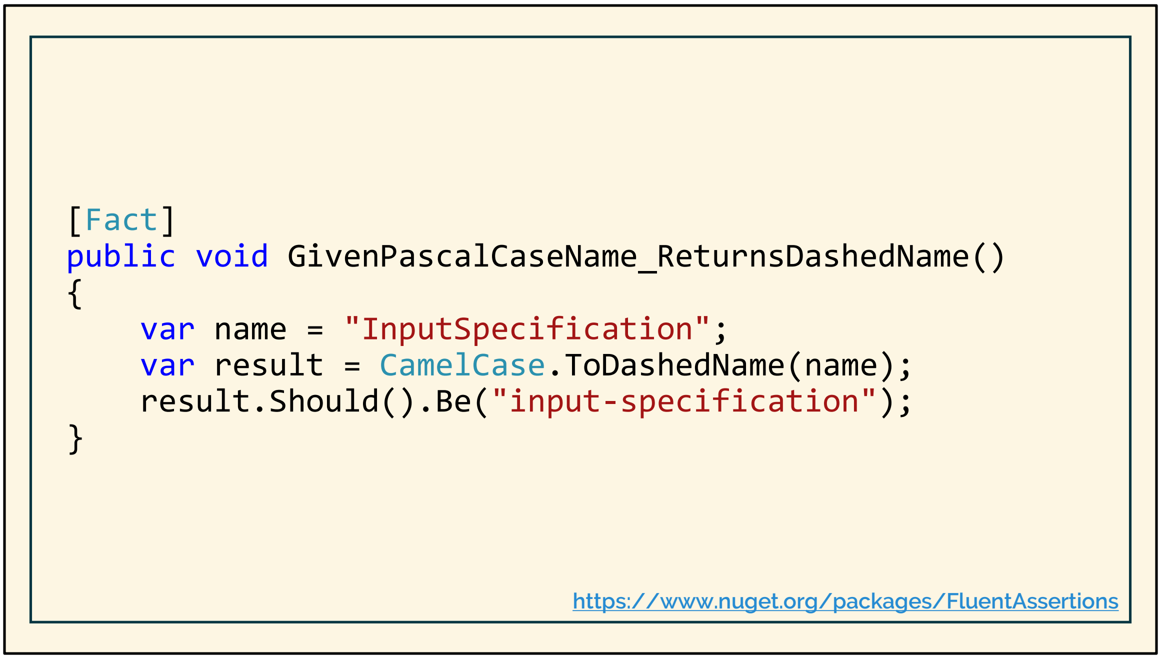
```
class CamelCase
{
    static string ToDashedName(string camelCase) { }
}
```



```
[Fact]
public void Test14()
{
    Assert.Equal(
        "input-specification",
        CamelCase.ToDashedName("InputSpecification"));
}
```

Problems with this code include:

- Poor name (the worst thing)
- Everything mashed together



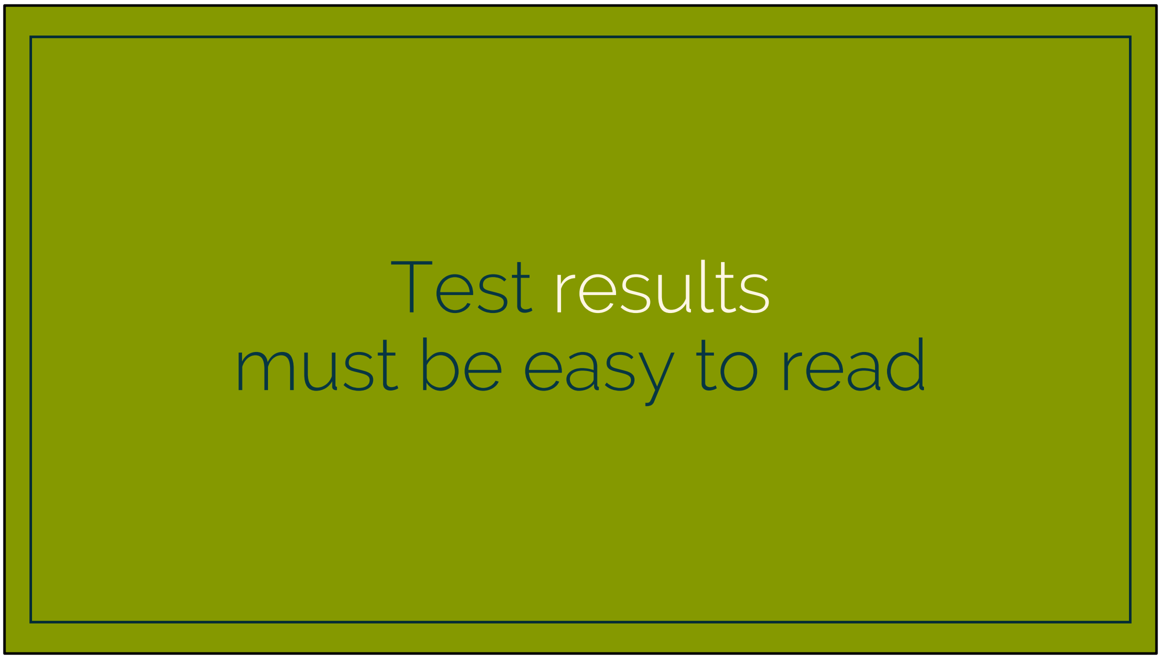
```
[Fact]
public void GivenPascalCaseName_ReturnsDashedName()
{
    var name = "InputSpecification";
    var result = CamelCase.ToDashedName(name);
    result.Should().Be("input-specification");
}
```

https://www.nuget.org/packages/FluentAssertions

Improvements include

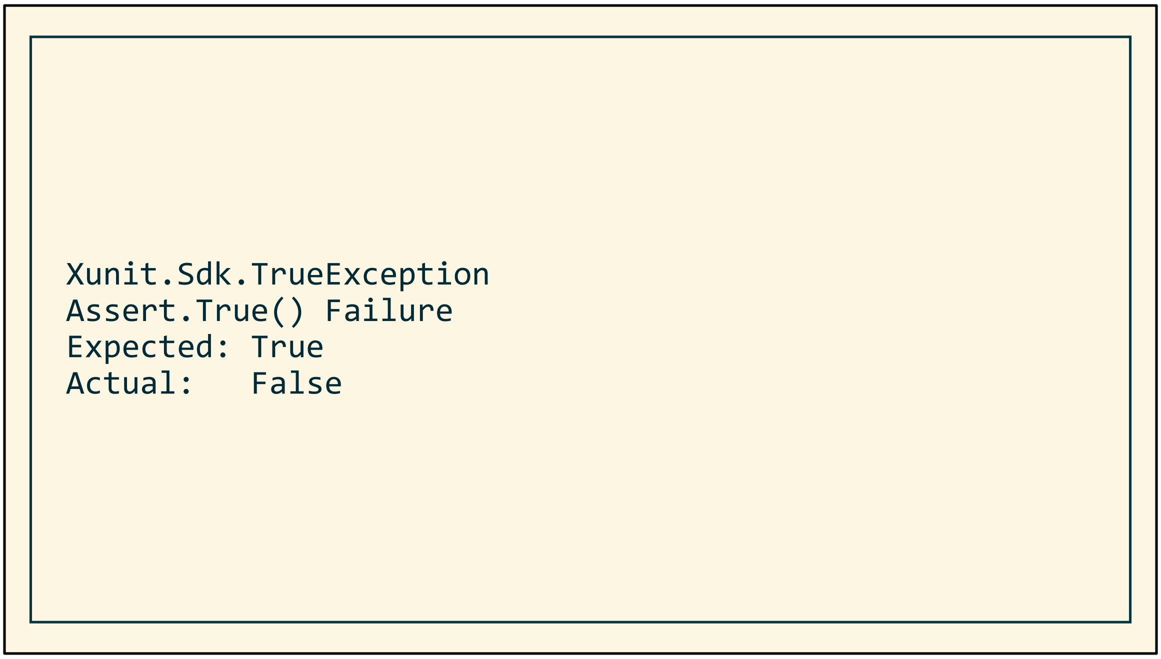
- Name that tells you what is being tested
- Separate the arrange/act/assert steps

Uses a different style of assertion



Test results
must be easy to read

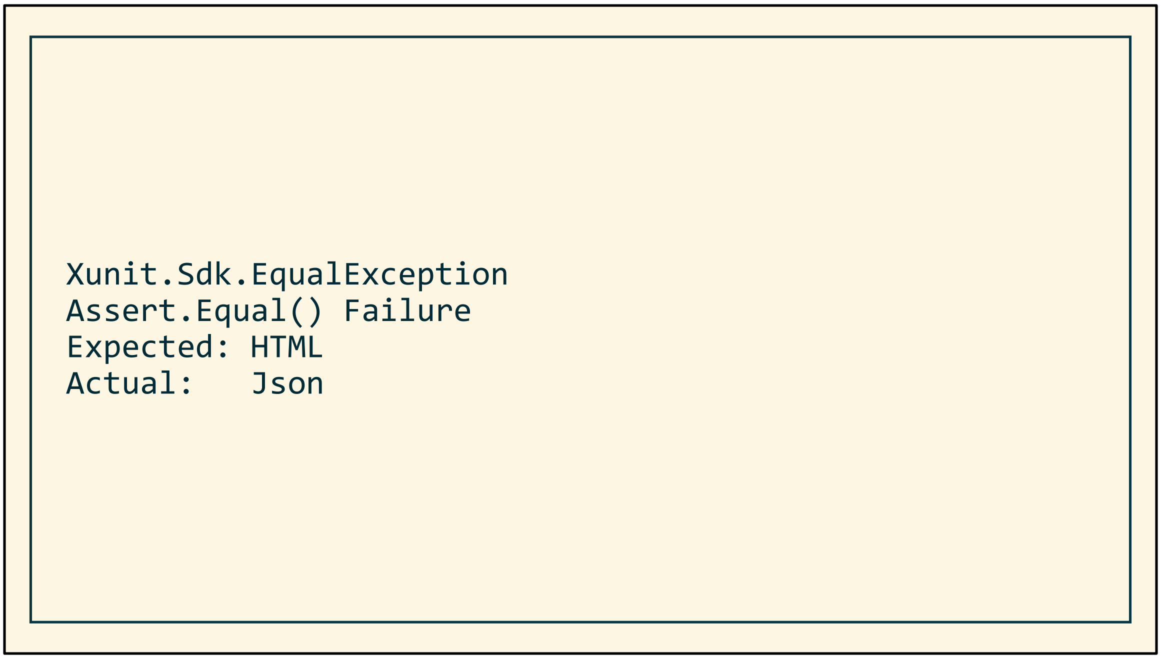
```
[Fact]
public void Equals_GivenSame_ReturnsTrue()
{
    var fileType = new FileType("HTML", "*.html");
    var same = new FileType("Json", "*.json");
    Assert.True(same.Equals(fileType));
}
```



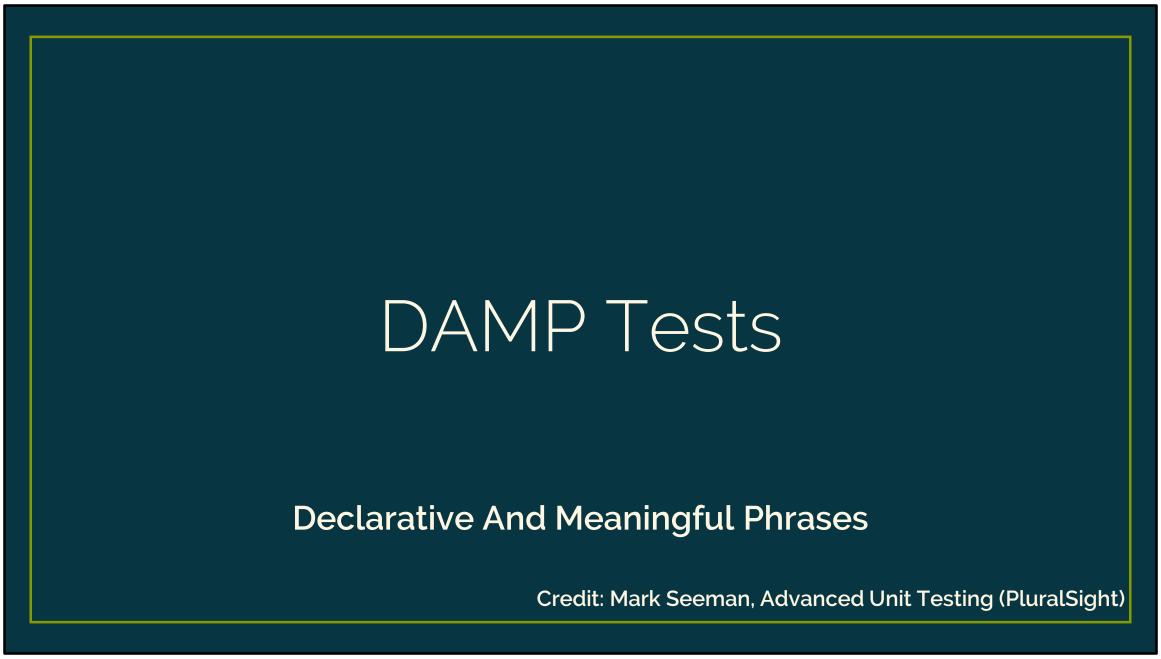
```
Xunit.Sdk.TrueException
Assert.True() Failure
Expected: True
Actual:   False
```

This test failure tells you nothing about why the test failed

```
[Fact]
public void Equals_GivenSame_ReturnsTrue()
{
    var fileType = new FileType("HTML", "*.html");
    var same = new FileType("Json", "*.json");
    Assert.Equal(fileType, same);
}
```



```
Xunit.Sdk.EqualException
Assert.Equal() Failure
Expected: HTML
Actual:   Json
```

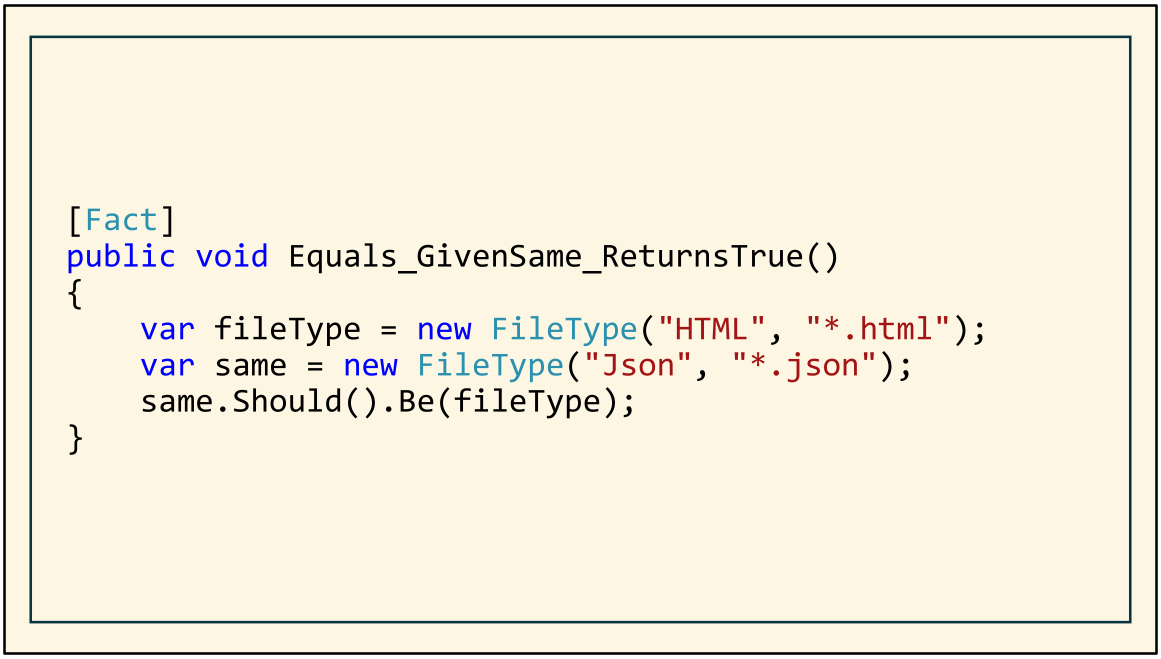


DAMP Tests

Declarative And Meaningful Phrases

Credit: Mark Seeman, Advanced Unit Testing (PluralSight)

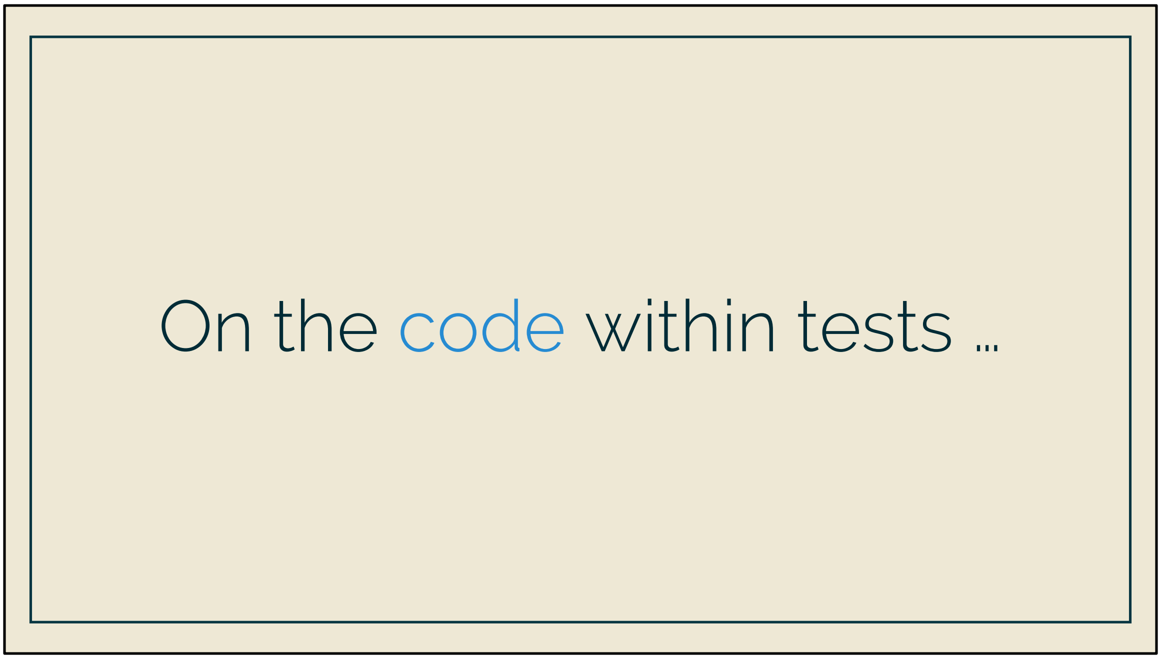
Don’t make people decipher what the tests are doing



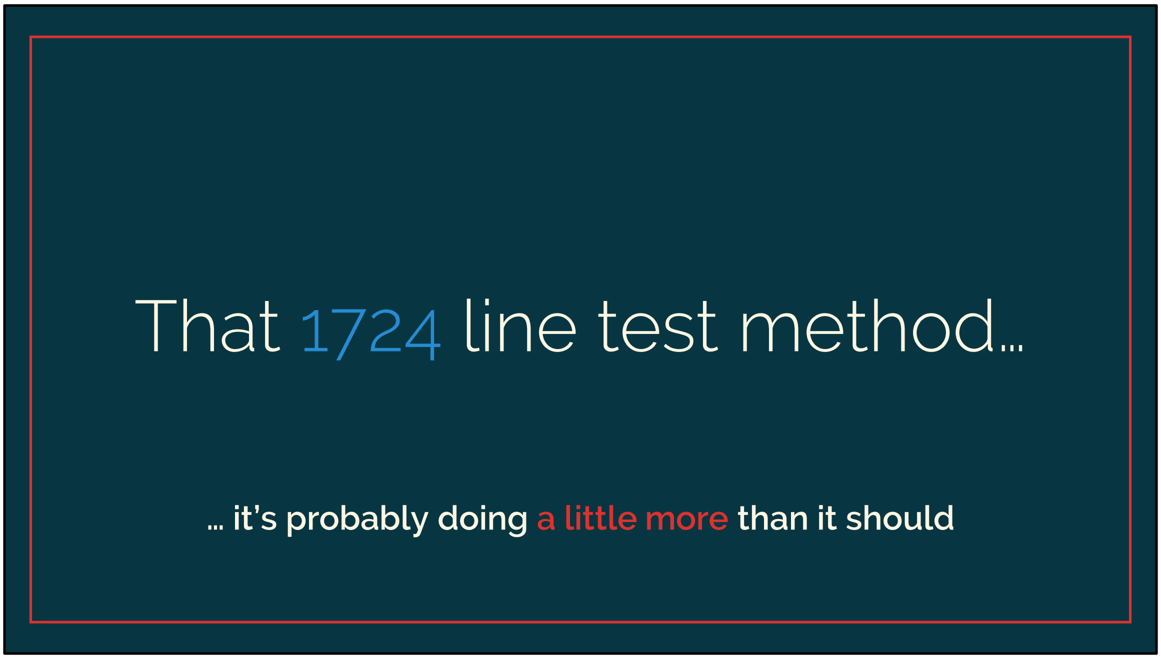
```
[Fact]
public void Equals_GivenSame_ReturnsTrue()
{
    var fileType = new FileType("HTML", "*.html");
    var same = new FileType("Json", "*.json");
    same.Should().Be(fileType);
}
```

The fluent style of the assertion makes this easier to read
Coincidentally, the library I used for doing this is called Fluent Assertions, but there are other choices too

```
Xunit.Sdk.XunitException
Expected object to be HTML, but found Json.
```



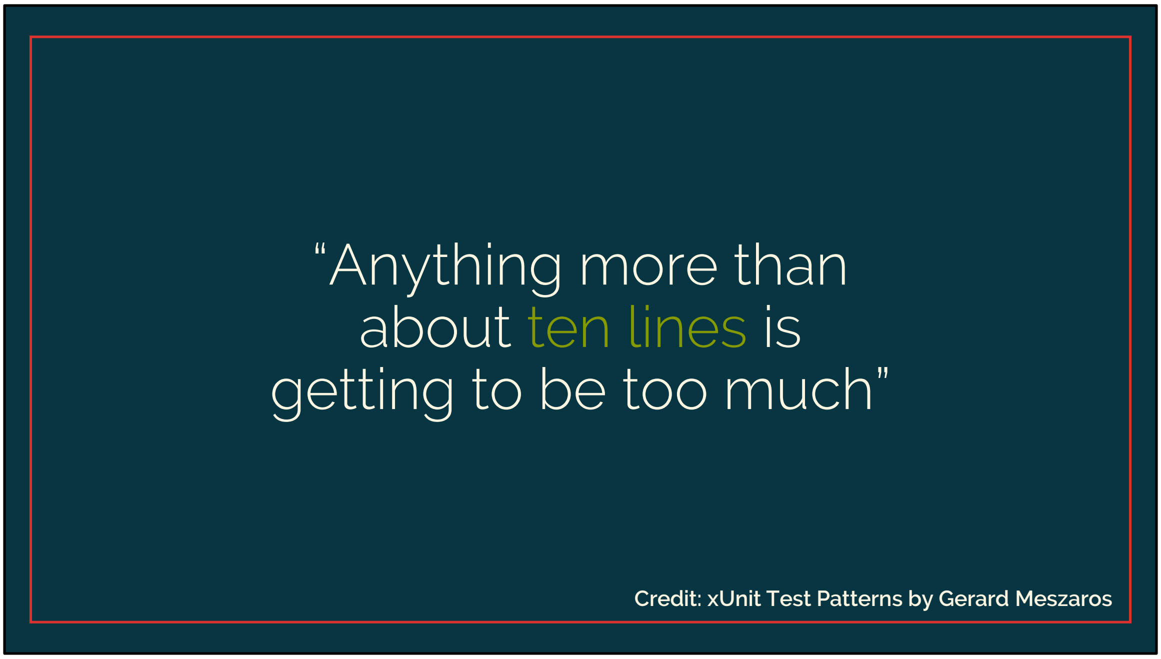
On the code within tests …



That 1724 line test method…

… it's probably doing a little more than it should

Yes, this was a real thing.
There was one more test method in the same file – with >1500 lines of its own.



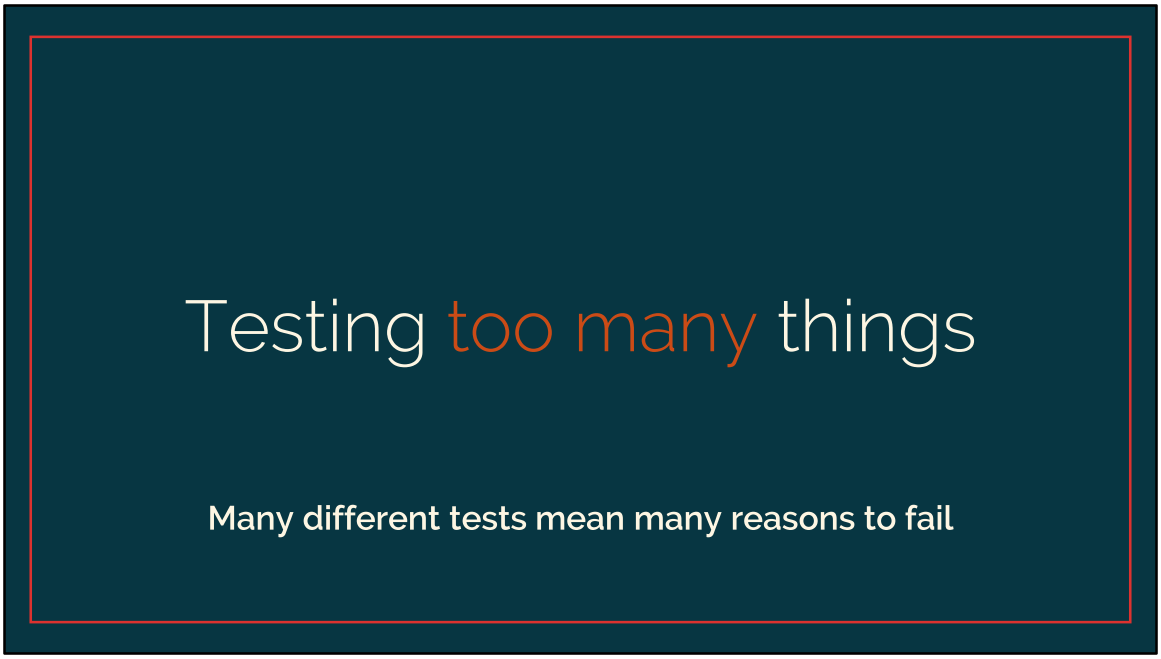
"Anything more than about ten lines is getting to be too much"

Credit: xUnit Test Patterns by Gerard Meszaros

There's a higher standard required:
Tests need to be easily readable by people who aren't familiar with them, not just by the author who wrote them
We can argue about ten … but I hope we can agree that 1724 is a few too many …

Tests that do too much

Test only one thing at a time

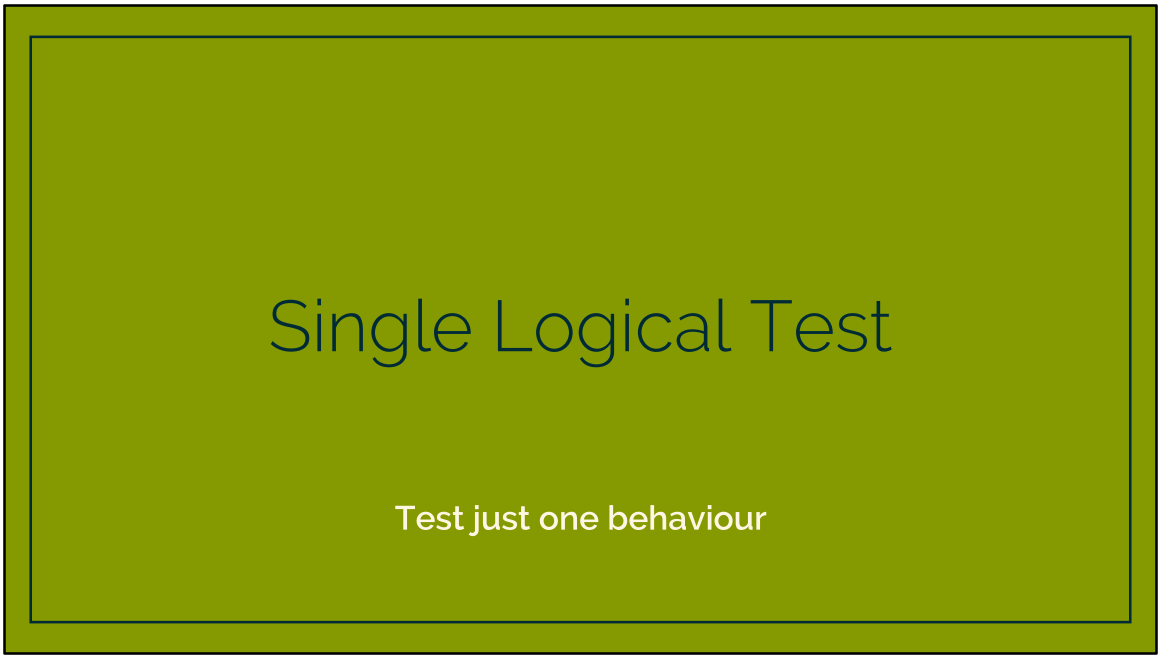


Testing too many things

Many different tests mean many reasons to fail

Worse, earlier failures mask the results of later tests

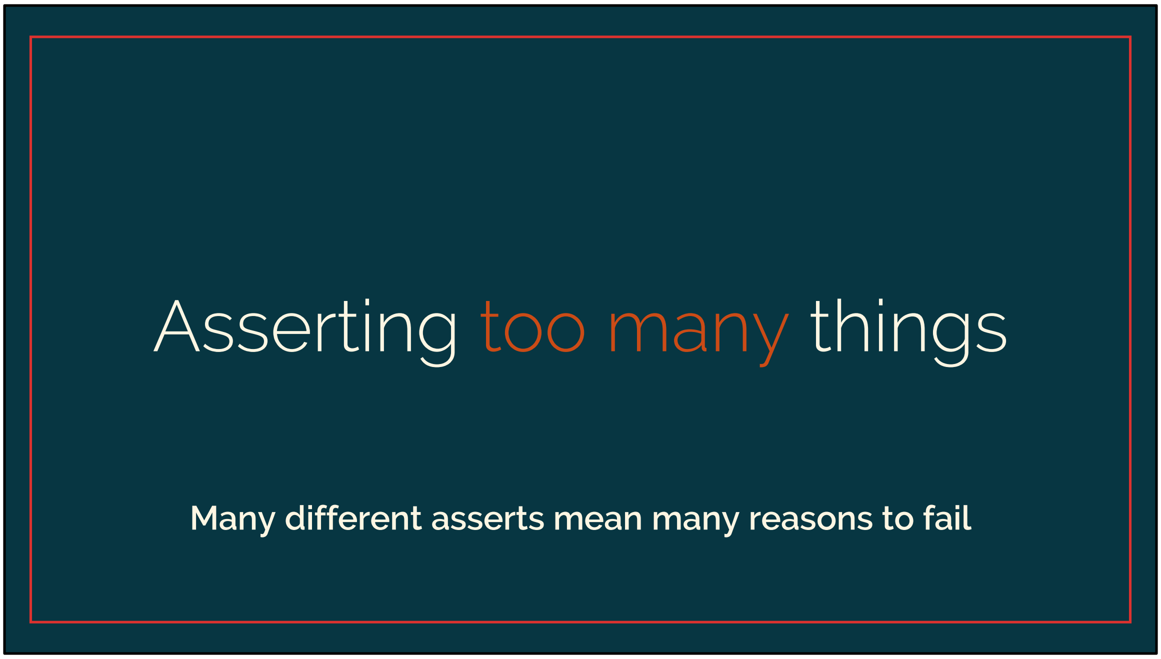
That 1724 line test method I mentioned earlier had Asserts every 6 or 8 lines – as soon as one of those fired, none of the other tests would have been tried



Single Logical Test

Test just one behaviour

Testing one characteristic of the system under test

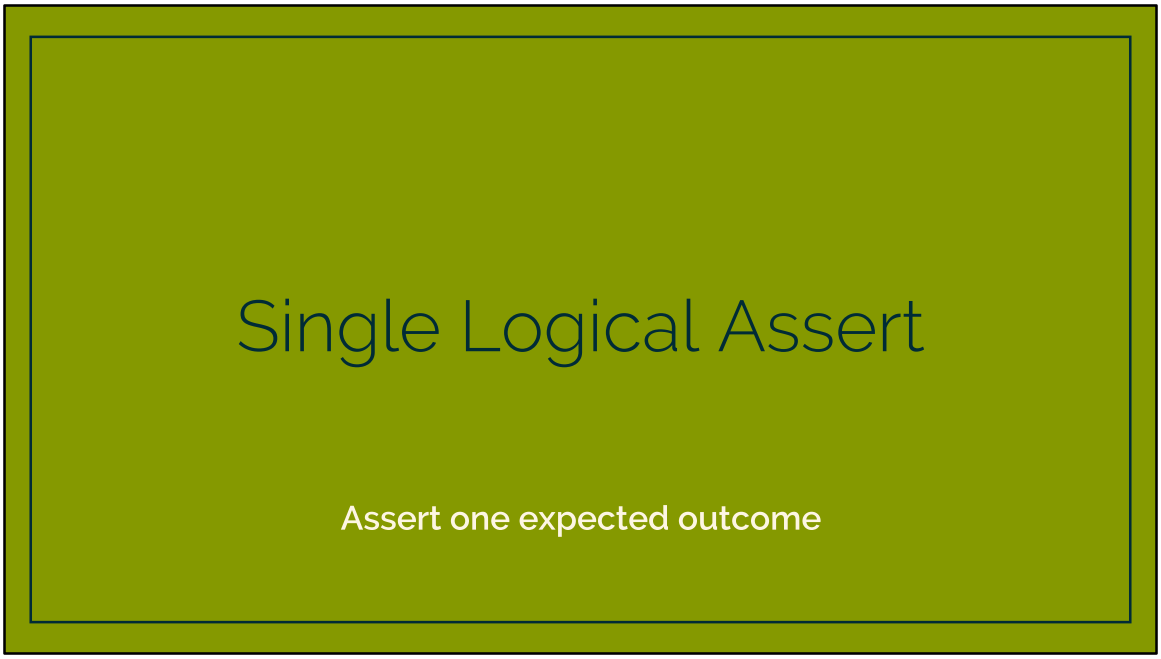


Asserting too many things

Many different asserts mean many reasons to fail

Worse, earlier failures mask the results of later tests

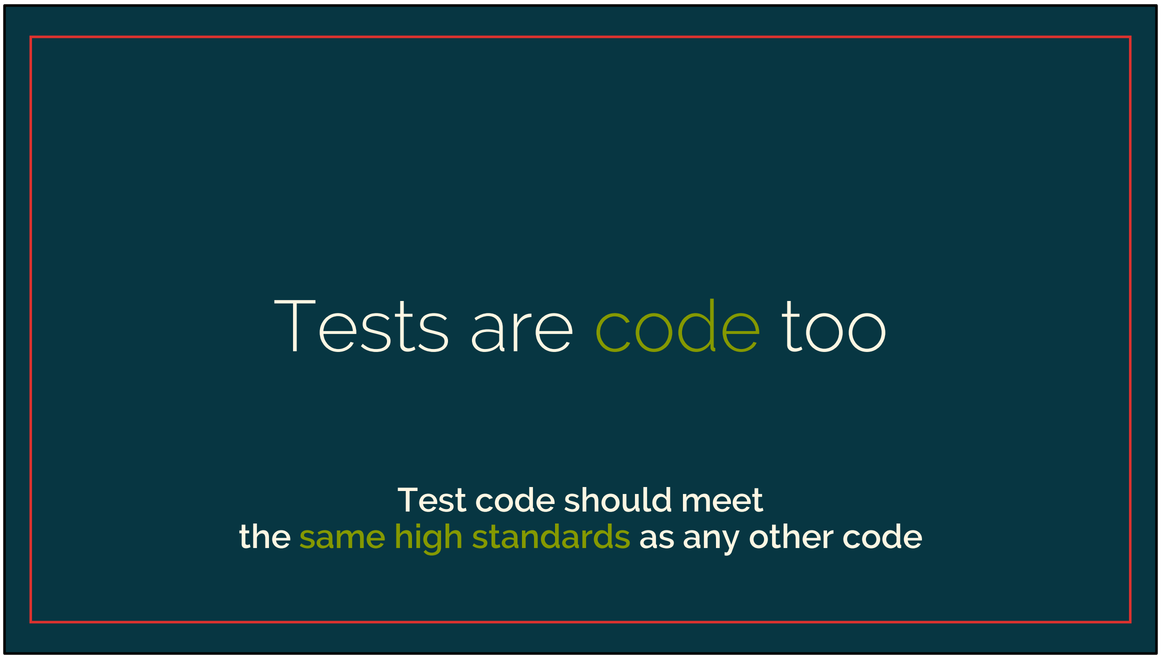
That 1724 line test method I mentioned earlier had Asserts every 6 or 8 lines – as soon as one of those fired, none of the other tests would have been tried



Single Logical Assert

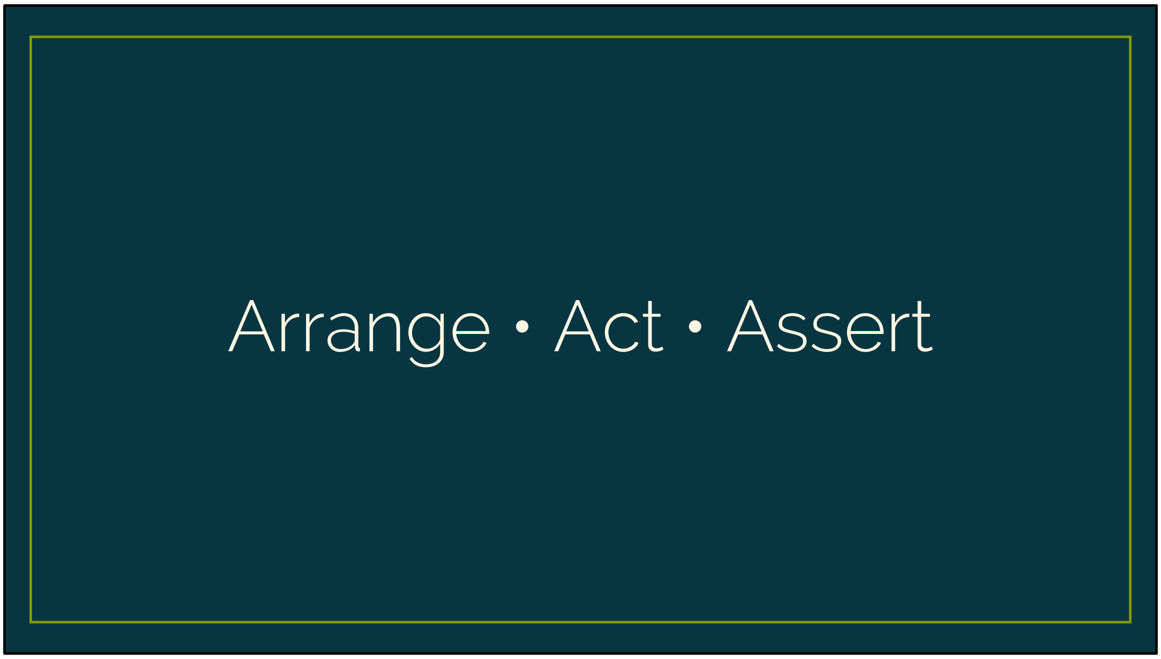
Assert one expected outcome

Mentioned earlier but worth calling out again
May require multiple actual assert statements
If the same set of related asserts crops up more than once, consider creating a dedicated assert method that wraps them in a consistent manner



Tests are code too

Test code should meet
the same high standards as any other code

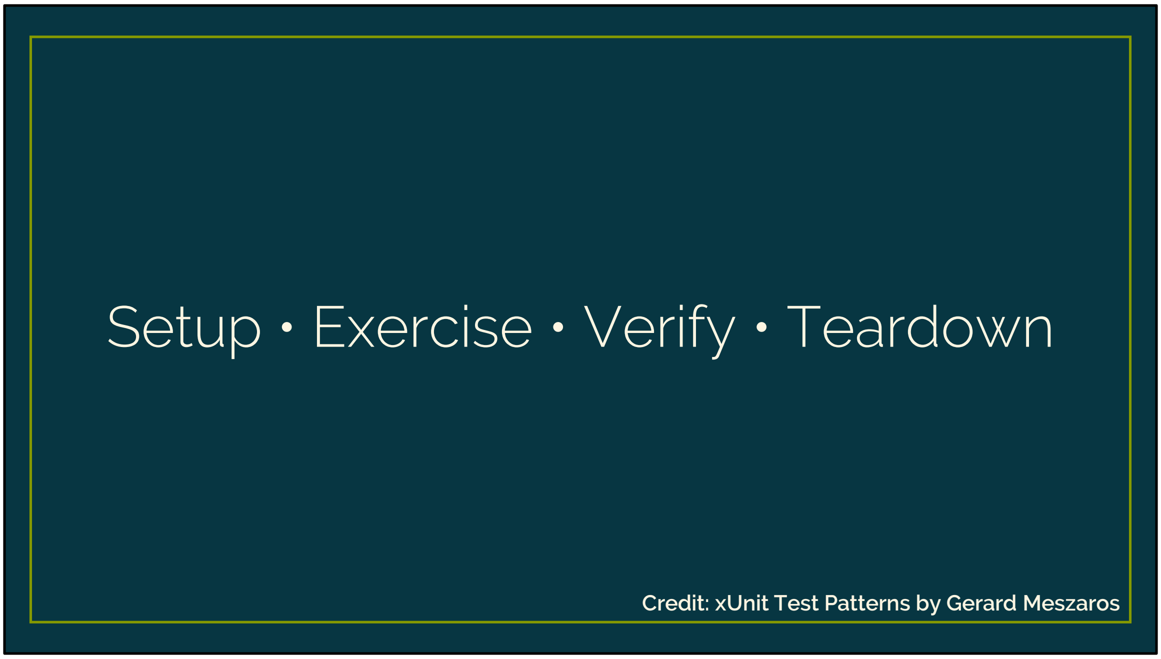


Arrange • Act • Assert

This is the classic approach, but one that works really well

Arrange – set everything up for the test you want to do
Act – do the thing
Assert – check to see if it worked

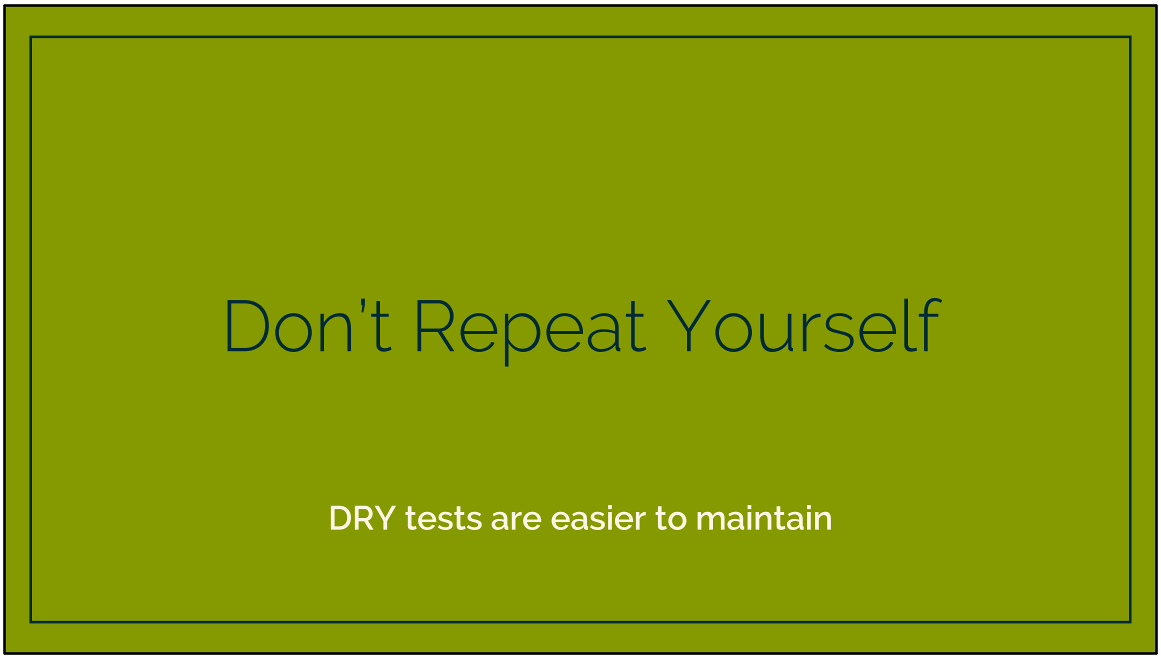
When you Assert, what do you check?



Setup • Exercise • Verify • Teardown

Credit: xUnit Test Patterns by Gerard Meszaros

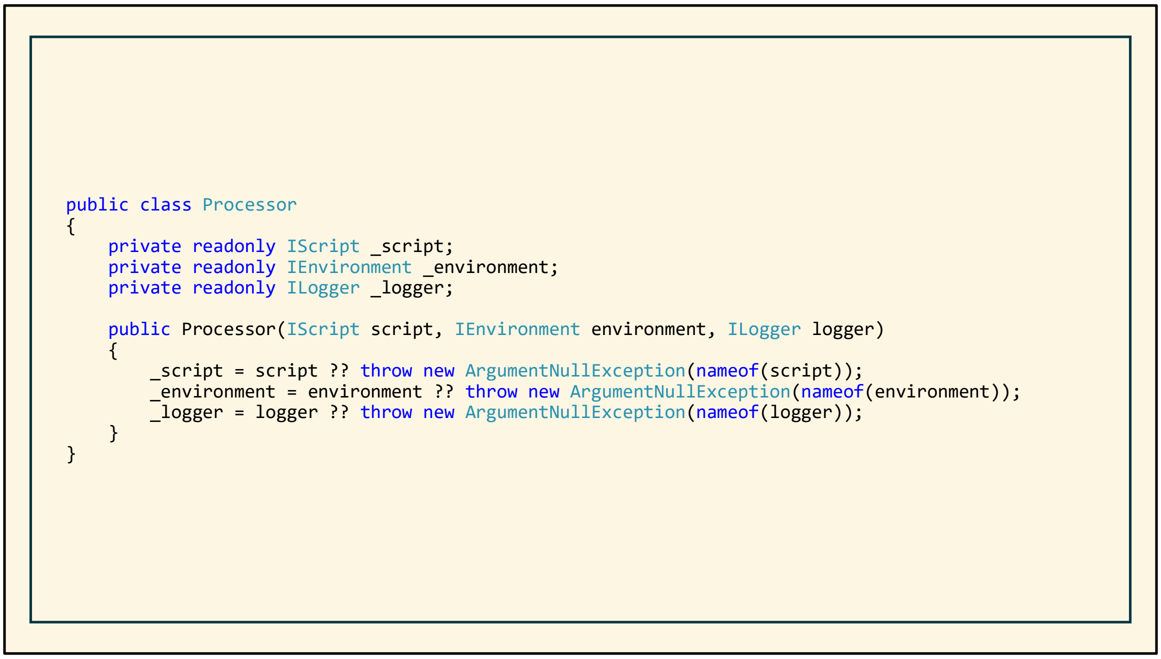
An alternative that serves the same purpose



Don’t Repeat Yourself

DRY tests are easier to maintain

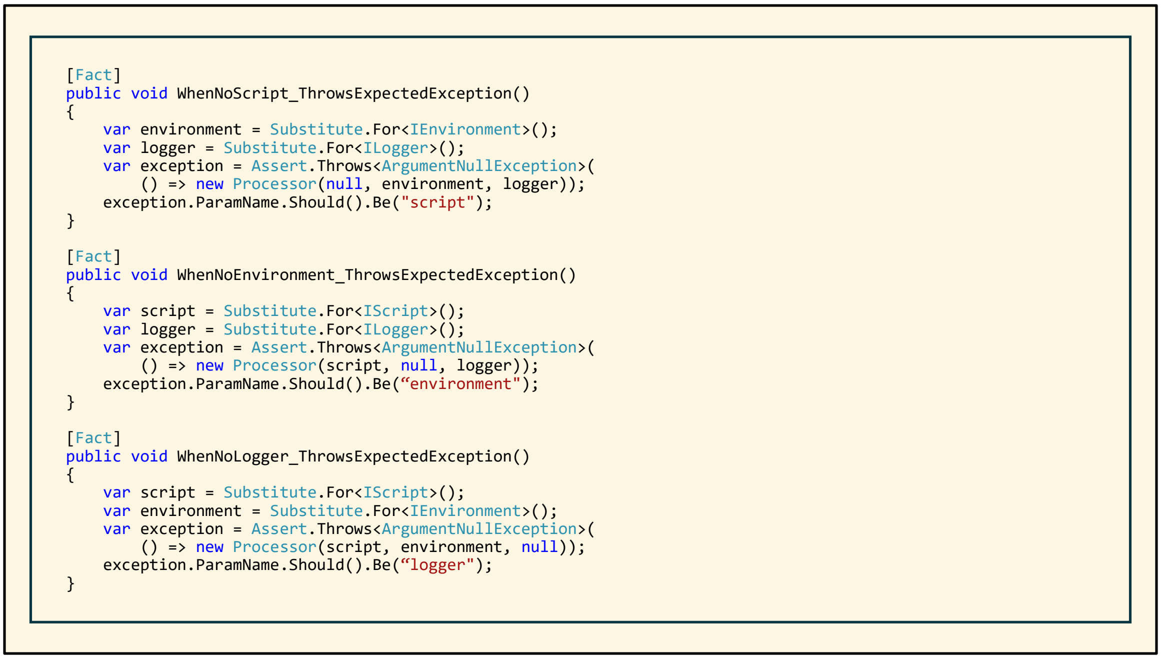
Refactor tests to eliminate boilerplate and repetitive setup



```
public class Processor
{
    private readonly IScript _script;
    private readonly IEnvironment _environment;
    private readonly ILogger _logger;

    public Processor(IScript script, IEnvironment environment, ILogger logger)
    {
        _script = script ?? throw new ArgumentNullException(nameof(script));
        _environment = environment ?? throw new ArgumentNullException(nameof(environment));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }
}
```

Consider this class; we want to test that the constructor properly throws exceptions if the parameters are null

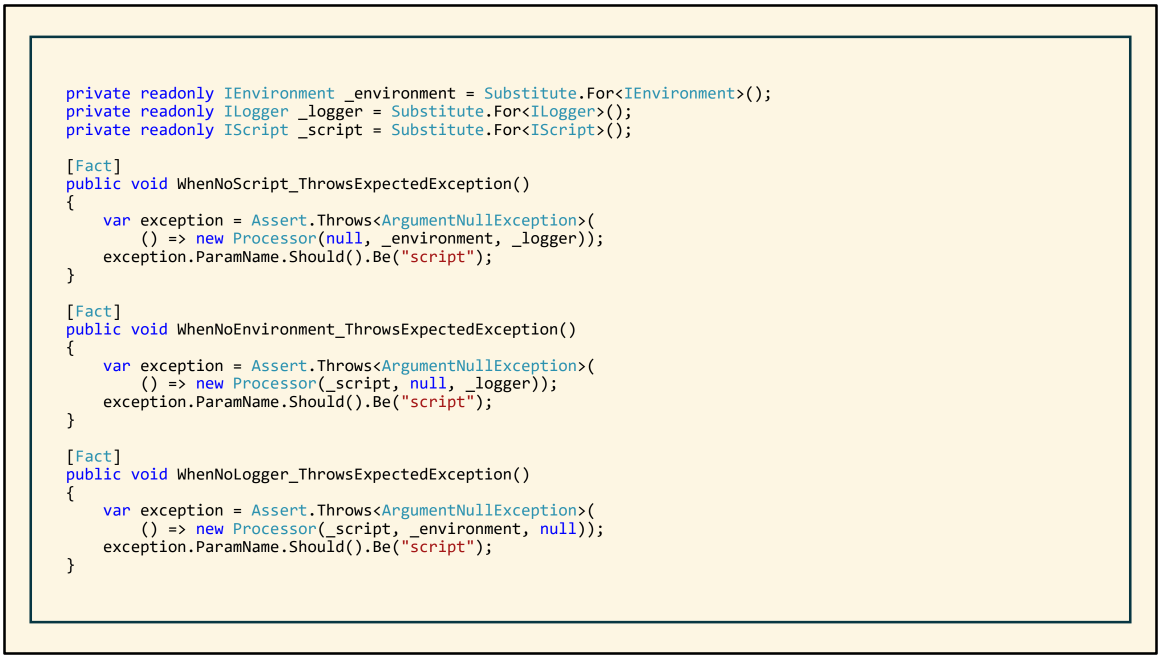
The **?? throw** style is new in C#7



```
[Fact]
public void WhenNoScript_ThrowsExpectedException()
{
    var environment = Substitute.For<IEnvironment>();
    var logger = Substitute.For<ILogger>();
    var exception = Assert.Throws<ArgumentNullException>(
        () => new Processor(null, environment, logger));
    exception.ParamName.Should().Be("script");
}

[Fact]
public void WhenNoEnvironment_ThrowsExpectedException()
{
    var script = Substitute.For<IScript>();
    var logger = Substitute.For<ILogger>();
    var exception = Assert.Throws<ArgumentNullException>(
        () => new Processor(script, null, logger));
    exception.ParamName.Should().Be(“environment");
}

[Fact]
public void WhenNoLogger_ThrowsExpectedException()
{
    var script = Substitute.For<IScript>();
    var environment = Substitute.For<IEnvironment>();
    var exception = Assert.Throws<ArgumentNullException>(
        () => new Processor(script, environment, null));
    exception.ParamName.Should().Be(“logger");
}
```

Three tests; these aren’t hard to read – but each includes a fair amount of noisy setup

Worse, the setup is repeated across multiple tests

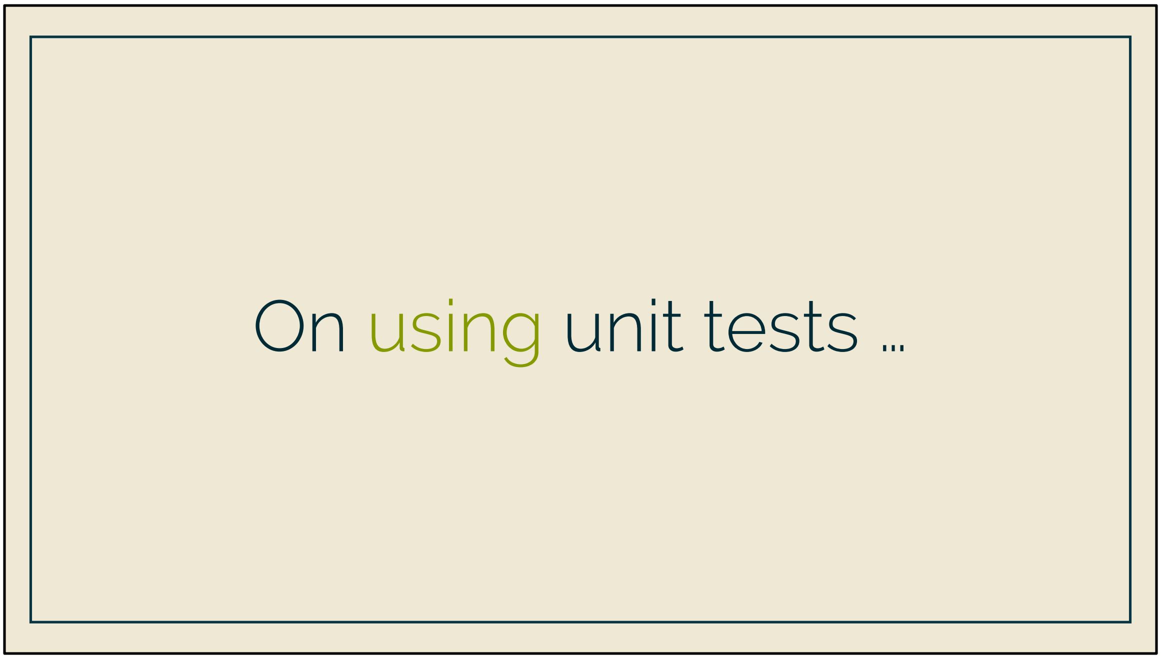


```
private readonly IEnvironment _environment = Substitute.For<IEnvironment>();
private readonly ILogger _logger = Substitute.For<ILogger>();
private readonly IScript _script = Substitute.For<IScript>();

[Fact]
public void WhenNoScript_ThrowsExpectedException()
{
    var exception = Assert.Throws<ArgumentNullException>(
        () => new Processor(null, _environment, _logger));
    exception.ParamName.Should().Be("script");
}

[Fact]
public void WhenNoEnvironment_ThrowsExpectedException()
{
    var exception = Assert.Throws<ArgumentNullException>(
        () => new Processor(_script, null, _logger));
    exception.ParamName.Should().Be("script");
}

[Fact]
public void WhenNoLogger_ThrowsExpectedException()
{
    var exception = Assert.Throws<ArgumentNullException>(
        () => new Processor(_script, _environment, null));
    exception.ParamName.Should().Be("script");
}
```

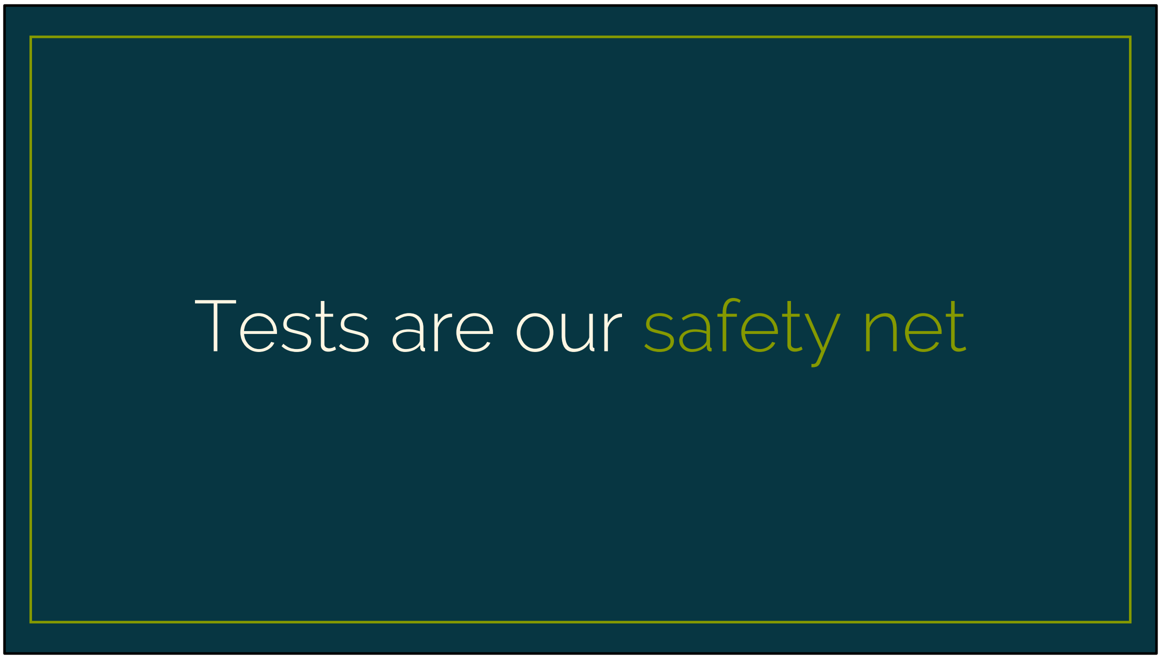
By moving the noise into member variables we make each test easier to read

Guideline for what to move: don’t move things out of the test if they’re being tested

In this case, we moved bystanders, not core cast



On using unit tests …

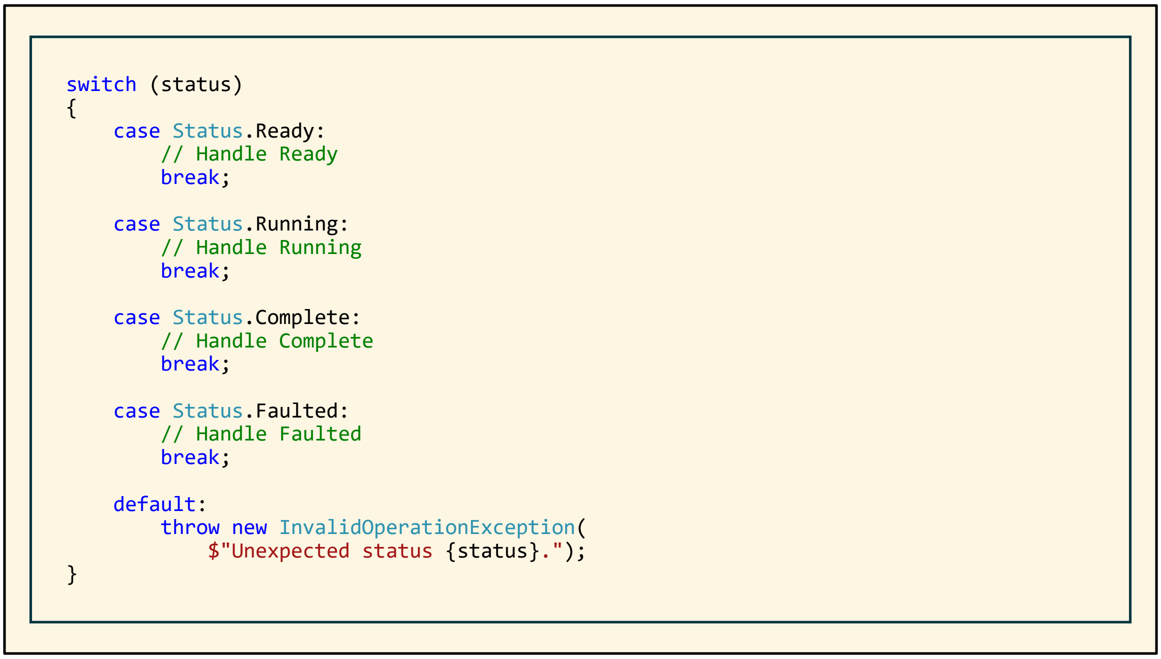


Tests are our safety net

Code Coverage

More coverage is better

More coverage is better – code that isn’t covered by tests is completely untested
Though a simple number doesn’t tell us much about the quality of the tests
-----
There are limits



```
switch (status)
{
    case Status.Ready:
        // Handle Ready
        break;

    case Status.Running:
        // Handle Running
        break;

    case Status.Complete:
        // Handle Complete
        break;

    case Status.Faulted:
        // Handle Faulted
        break;

    default:
        throw new InvalidOperationException(
            $"Unexpected status {status}.");
}
```

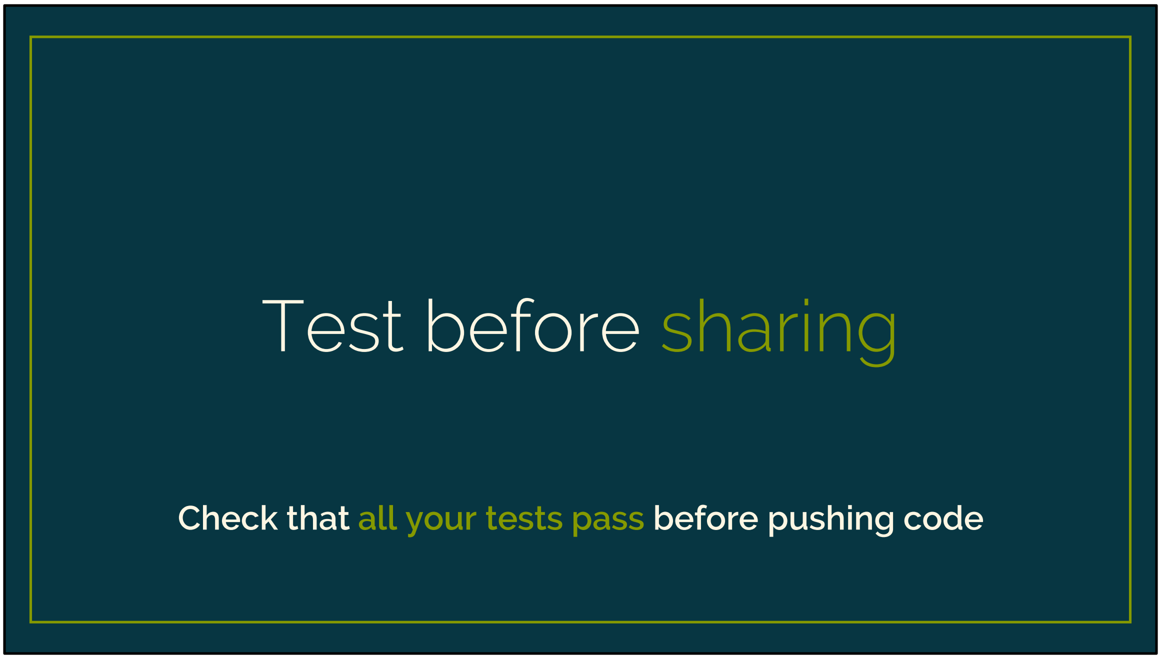
Enum with four values
Switch that handles all values AND has a default that throws, to catch future maintenance errors

The **default** branch is there to ensure it fails informatively if a new status value is introduced.
This is **good** defensive programming – but that branch cannot be tested.

100% Code Coverage

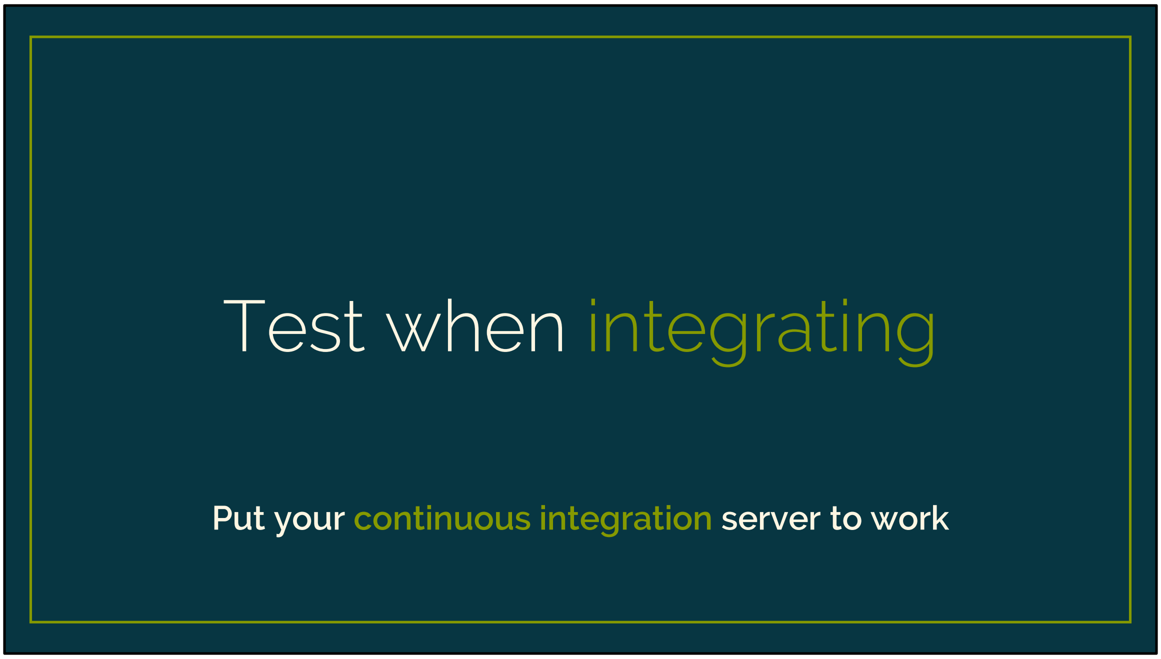
Don’t waste your time

Getting to 100% can involve a lot of hard work that may not be worth it



Test before sharing

Check that all your tests pass before pushing code



Test when integrating

Put your continuous integration server to work

If you don’t have one, try TeamCity – the Professional edition is free forever

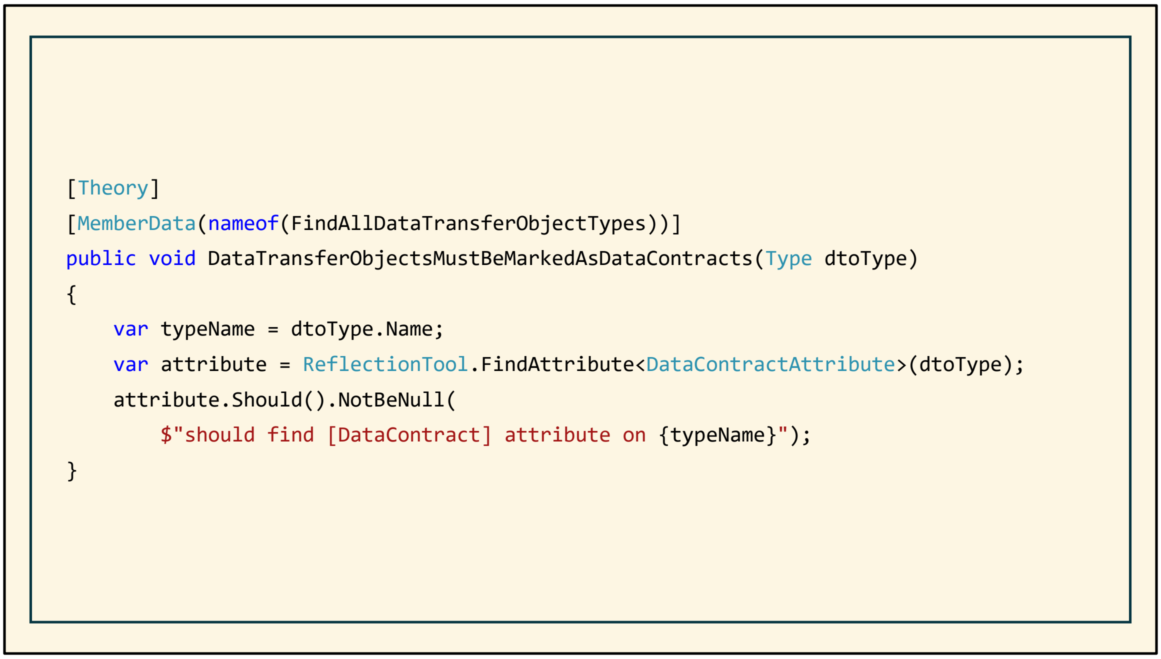
Test in production

Consider including some tests when you deploy

App with integrated test suite for troubleshooting

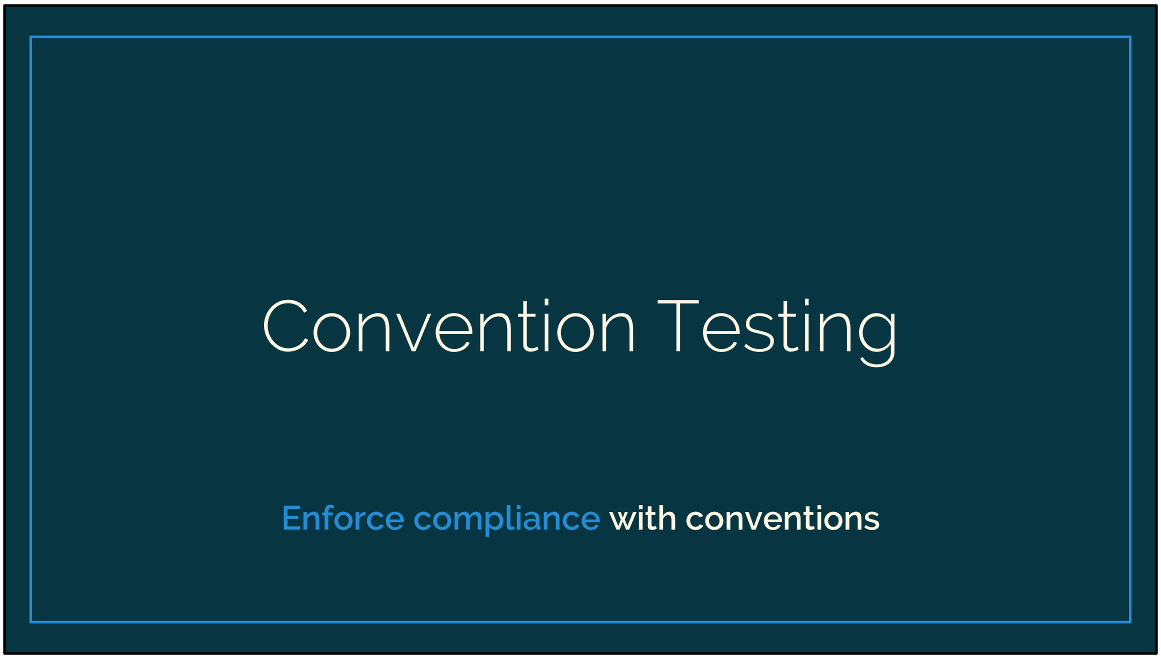
Here’s an idea …

If every data transfer object
must have a [DataContract] attribute …



```
[Theory]
[MemberData(nameof(FindAllDataTransferObjectTypes))]
public void DataTransferObjectsMustBeMarkedAsDataContracts(Type dtoType)
{
    var typeName = dtoType.Name;
    var attribute = ReflectionTool.FindAttribute<DataContractAttribute>(dtoType);
    attribute.Should().NotBeNull(
        $"should find [DataContract] attribute on {typeName}");
}
```

Memberdata – find all the data transfer object types we want to check
ReflectionTool – find the [DataContract] attribute method for a given type, returning null if not found
“Because” message included to explain why the test failed

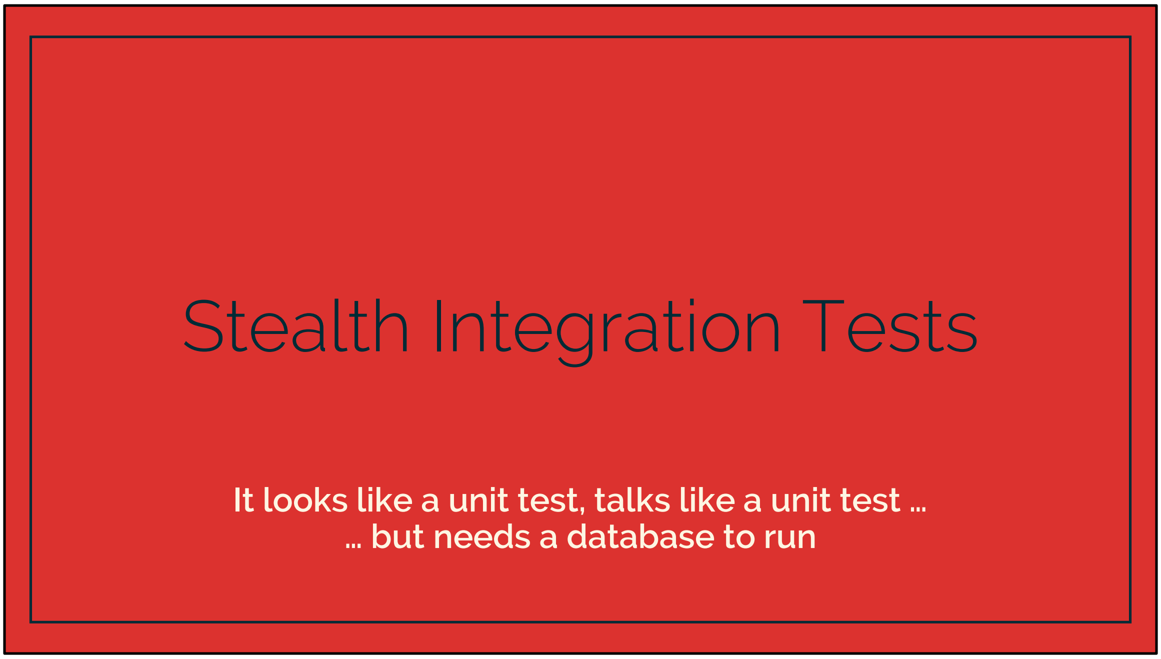


Convention Testing

Enforce compliance with conventions

With a suite of well chosen convention tests, you can walk future maintainers through specific extension scenarios

Other antipatterns to avoid…



Stealth Integration Tests

It looks like a unit test, talks like a unit test …
… but needs a database to run

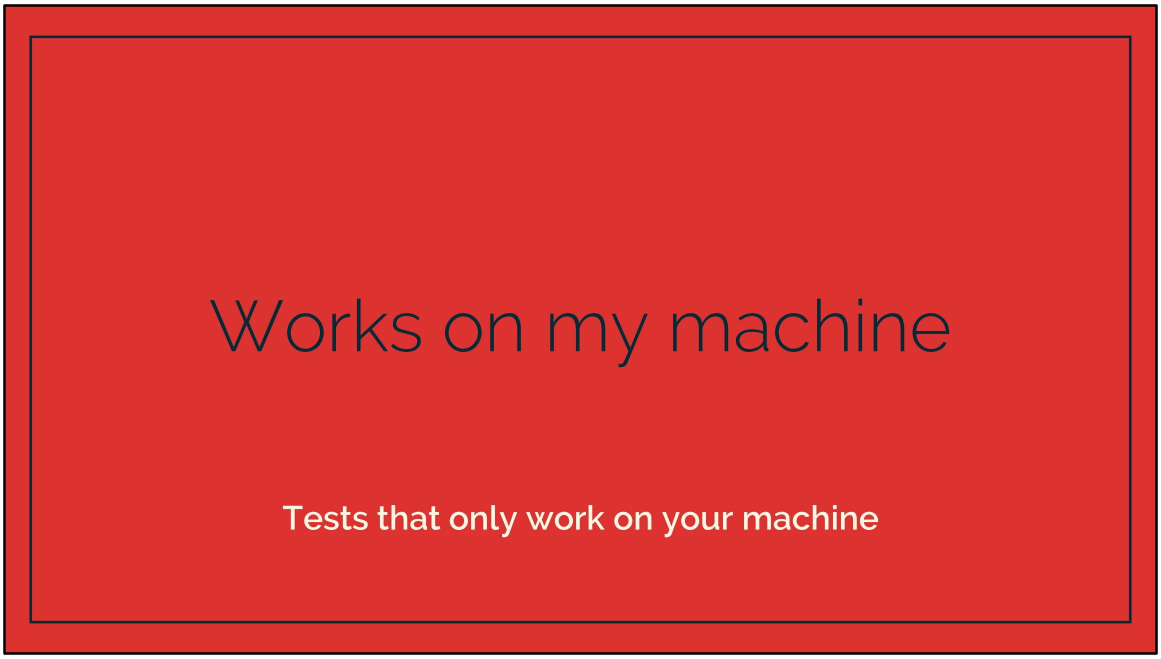
Or particular files on disk, or IIS to be configured or a valid current printer or …

Testing the Framework

You don’t need to test that List<T> works

Rowdy Tests

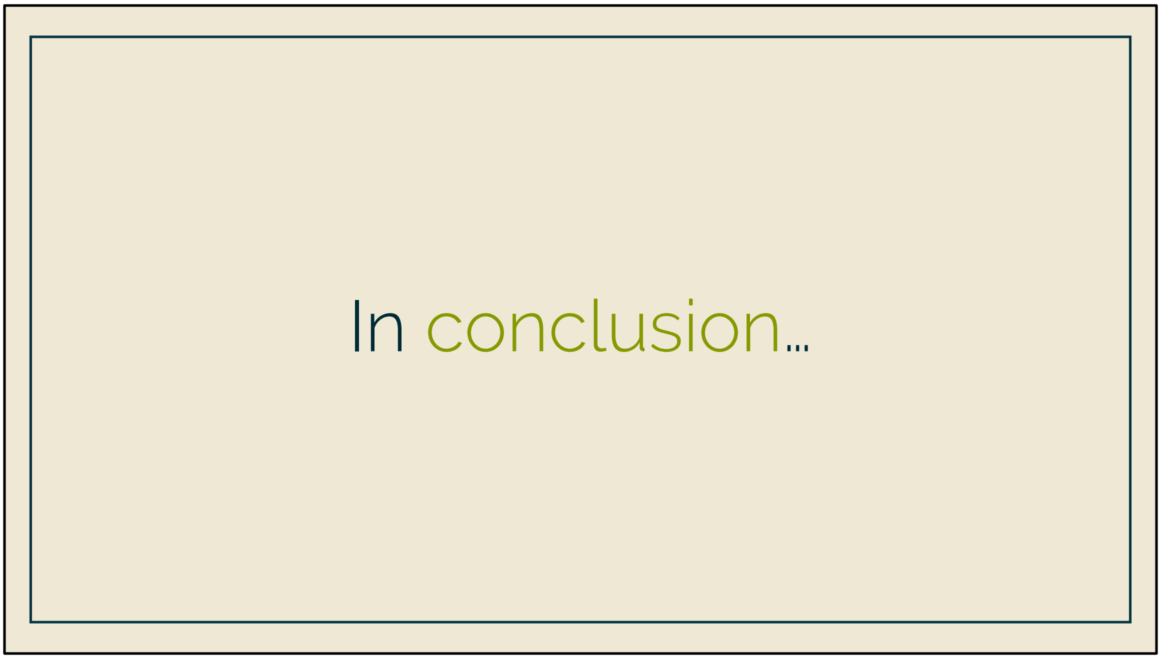
Good tests are quiet – unless they fail



Works on my machine

Tests that only work on your machine

…



In conclusion…

Unit tests are FIRST

Declarative test names

Easy to read test code & results

Single logical test & assert

Don't repeat yourself

Obtuse naming

Tests that do too much

Stealth integration tests

Testing the framework

Rowdy tests

Works on my machine

Thanks

@unrepentantgeek
bevan@nichesoftware.co.nz
http://www.nichesoftware.co.nz