Unit Testing (.NET)

THE ART OF UNIT TESTING

Agenda

1. WHAT IS A UNIT TEST?
2. WE HAVE ALL WRITTEN UNIT TESTS (SORT OF)
3. UNIT TESTING FRAMEWORKS
4. NUNIT
5. EXTERNAL DEPENDENCY
6. HANDWRITTEN FAKES
7. ISOLATION (MOCKING) FRAMEWORKS
8. NSUBSTITUTE
9. CHARACTERISTIC OF GOOD UNIT TESTS
10. TDD
11. MORE TYPES OF AUTOMATED TESTS
12. RESOURCES

1

What Is a Unit Test?

Definition

Unit test is an automated piece of code that invokes the unit of work being tested and checks some assumptions about a single end result of that unit.

It's consistent in its results as long as the production code hasn't changed.

Why We Need It?

- It gives high-quality code at any time as the code is being tested continuously

- Helps in reducing software errors

- Speed up the delivery process

- Helps in increasing software development life cycle efficiency

2

We Have All Written Unit Tests (Sort Of)

Ways To Check if the Code Works

Have you ever met a developer who has *not* tested their code before handing it over?
Well, neither have I.

- You might have used a *console application* that called the various methods of a class or component.
- Or perhaps some specially created UI that checked the functionality of that class or component.
- Or maybe even manual tests run by performing various actions within the real application's UI.

3

Unit Testing Frameworks

Why We Need Them?

Frameworks supply the developer with a class library that contains
- Base classes or interfaces to inherit
- Attributes to place in your code to note which of your methods are tests
- Assertion classes that have special assertion methods you invoke to verify the code

Frameworks provide a test runner (a console or GUI tool) that
- Identifies tests in your code
- Runs tests automatically
- Indicates status while running
- Can be automated by the command line

The test runners usually provide information such as
- How many tests run
- How many tests didn’t run
- How many tests failed
- Which tests failed
- The reason tests failed
- The code location that failed

.NET Unit Testing Frameworks

MSTest, NUnit, xUnit

4

NUnit

What Is It?

NUnit is an open-source testing framework ported from JUnit.
The latest version of NUnit is NUnit3 that is rewritten with many new features and has support for a wide range of .NET platforms.

Has the best documentation (compared to the previously mentioned testing frameworks): https://docs.nunit.org/articles/nunit/intro.html

Creating NUnit Project (Installing NUnit)

1. Add a new project.

2. Search for “NUnit” template in the search box.

3. Select the first one from the results: “NUnit Test Project (.NET Core)”.

This shortcut is the same procedure as creating a “Class Library” project and manually installing these 3 NuGet packages within:
- NUnit – the core NUnit code.
- Microsoft.NET.Test.Sdk – running tests in Visual Studio.
- NUnit3TestAdapter – adapter for running NUnit tests in Visual Studio.

NUnit Test Class & Method

[TestFixture] - attribute that denotes a class that holds automated NUnit tests.

[Test] - attribute that can be put on a method to denote it as an automated test to be invoked.

Assert Class

The Assert class has static methods and is located in the NUnit.Framework namespace.

It's the bridge between your code and the NUnit framework, and its purpose is to declare that a specific assumption is supposed to exist.

Note:
One assert per test!

AAA (Arrange, Act, Assert)

Known also as Build – Operate – Check (Uncle Bob - Clean Code book)
Also similar to GWT (Given – When – Then)

Arrange objects, creating and setting them up as necessary.

Act on an object/SUT.

Assert that something is as expected.

Notes:
SUT – System Under Test
When you test something, you refer to the thing you're testing as the SUT.
Some people use CUT (Class Under Test or Code Under Test)

Namings

Project - [ProjectUnderTest].UnitTests

Class - [ClassName]Tests

Method - [UnitOfWorkName]_[ScenarioUnderTest]_[ExpectedBehavior]

UnitOfWorkName - The name of the method or group of methods or class you're testing (usually a method).

ScenarioUnderTest - The condition under which the unit is tested (such as "bad login", "invalid user"...). You could describe the parameters being sent to the method. Or the initial state of the system when the unit of work is invoked (such "no user exist", "user already exists"...)

ExpectedBehavior - What you expect the tested method to do under the specified conditions (ScenarioUnderTest). This could be one of three possibilities:
- return a value (a real value, or exception)
- change the state of the system
- call a third-party system.

The method name explains the test like:
When I call method X with Y, then it should do Z.

Parameterized Tests

[TestCase()] - parametrized test. This attribute sends a parameter to the method in the next line. If one of the [TestCase()] attributes fails, the other attributes are still executed by the test runner.

Note:
Be careful, the test name could become generic, and it will make the test less understandable. Don't put the expected result here, try to have meaningful name.

OneTimeSetUp, OneTimeTearDown, SetUp, TearDown

[OneTimeSetUp] - allow setting up state ONCE before all tests in a specific class run.

[OneTimeTearDown] - runs ONCE after all tests in a specific class run.

[SetUp] - this method runs before each test in that class.

[TearDown] - this method runs after each test in that class.

Notes:
- Try to use these less, they make the tests less readable.
- You may find that you're sharing state between tests if you're not careful.
- Those should be used in integration tests. For integration tests you can use also [SetUpFixture] to setup something for once for the whole namespace.
- Instead of these use a factory method (simple method that will be called from tests). In that way you won't have something that is created outside of the test. It'll be more readable.

Other Attributes

[Igonre] - doesn't run a test (ignores it). This should be rare cases! (possibly never, it confuses the future developer, it's similar to commented code)

[Category] - set up test to run under a specific category, such as slow tests and fast tests.

5

External Dependency

What Is It?

An external dependency is an object in your system that your code under test interact with and over which you have no control.

Fakes

Fakes are used to isolate the unit of work from the external dependencies.

Fake, Stub, Mock…

Fake denotes an object that looks like another object but can be used as a mock or stub! Used to describe mocks and stubs. Whether a fake is a stub or a mock depends on how it's used in the current test.

Stub

Stub is a controllable replacement for an existing dependency (or collaborator) in the system. A stub can never fail your test and is strictly there to simulate various situations.

Mock

Mock is a fake object in the system that DECIDES whether the unit test has passed or failed. It does so by verifying whether the object under test called the fake object as expected.

These are known also as *test doubles*. Test double is a generic term for any case where you replace a production object for testing purposes. Mentioned in Gerard Meszaros book - xUnit Test Patterns.
Test doubles according Gerard Meszaros: Dummy, Fake, Stubs, Spies, Mocks.

6

Handwritten Fakes

Faking an External Dependency

If we're following the SOLID principles the external dependency should be an interface (injected using the constructor, or in some cases using a method).

In that case it's easy to fake the external dependency, we should only create a class that implements this interface and use that class when interacting with the SUT. Also, you'll need to add some logic in the methods so they will behave as stubs or mocks.

The Problems With Handwritten Mocks and Stubs

There are several issues that crop up when using manual mocks and stubs:
- It takes time to write the mocks and stubs.
- It's difficult to write stubs and mocks for classes and interfaces that have many methods, properties, and events.
- To save state for multiple calls of a mock method, you need to write a lot of boilerplate code within the handwritten fakes.
- If you want to verify that all parameters on a method call were sent correctly by the caller, you'll need to write multiple asserts. That's a drag.
- It's hard to reuse mock and stub code for other tests. The basic stuff works, but once you get into more than two or three methods on the interface, everything starts getting tedious to maintain.
- Is there a place for fake that is both a mock and a stub? Very rarely. And I mean maybe once or twice in a project. I've only seen this a couple of times in the past couple of years myself.

7

Isolation (Mocking) Frameworks

Why We Need Them?

They allow you to isolate the unit of work from its dependencies (because of that isolation frameworks, but they are better known as mocking frameworks).
A set of programmable APIs that makes creating fake objects much simpler, faster, and shorter than hand-coding them.

Isolation frameworks matches nicely with the structure of arrange-act-assert.

Advantages of isolation frameworks vs manually creating fakes:
- easier parameter verification
- easier verification of multiple method calls
- easier fakes creation

.NET Isolation (Mocking) Frameworks

Constrained

Constrained because there are some things these frameworks aren't able to fake.
Unable to fake static methods, nonvirtual methods, nonpublic methods, and more.
NSubstitute, Moq, RhinoMocks, FakeItEasy...

Unconstrained

Use the profiling APIs. They don't generate and compile code at runtime that inherits from other code.
Telerik JustMock, Typemock Isolator, Microsoft Fakes...

8

NSubstitute

What Is It?

NSubstitute is designed as a friendly substitute for .NET mocking libraries.
It is an attempt to satisfy our craving for a mocking library with a succinct syntax that helps us keep the focus on the intention of our tests, rather than on the configuration of our test doubles.

You can call it NSub because is much easier to pronounce.

Great documentation: https://nsubstitute.github.io/help/getting-started/

Installing NSubstitute

1. Using NuGet package manager (choose “Manage NuGet packages…” in the project where you want to install it)
2. Search for “NSubstitute” in the packages search box (choose “nuget.org” as package source).
3. Install the first result “NSubstitute”. Now you’re ready to use this framework.

Creating a Fake (Substitute)

Use the static class Substitute and the static method For.
Generally this type will be an interface, but you can also substitute classes in cases of emergency. But be careful, substituting for classes can have some nasty side-effects!

Example:
var substitute = Substitute.For<ISomeInterface>();

Setting a Return (For Specific/Any Arguments)

To set a return value for a method call on a substitute, call the method as normal, then follow it with a call to NSubstitute's Returns() extension method.

Example:

```
var substitute = Substitute.For<ISomeInterface>();
substitute.MethodName(1, Arg.Any<int>(), Arg.Is<int>(x => x < 0)).Returns(3);
```

Throwing Exceptions

Callbacks can be used to throw exceptions when a member is called.

Example:

```
var substitute = Substitute.For<ISomeInterface>();

// For non-voids:
substitute.MethodName().Returns(x => { throw new Exception(); });

//For voids and non-voids:
substitute
        .When(x => x.MethodName())
        .Do(x => { throw new Exception(); });

// In both ways the calls will throw
Assert.Throws<Exception>(() => substitute.MethodName());
```

Checking Received Calls

This can be checked using the Received() extension method, followed by the call being checked.

Example:

```
var substitute = Substitute.For<ISomeInterface>();
var something = new SomethingThatNeedsACommand(substitute);

// Act
something.DoSomething();

// Assert
substitute.Received().MethodName();
```

Note:
Notice that in this case we're using the fake as a mock (previously we're using it as a stub).

Checking Call Order

Sometimes calls need to be made in a specific order. We can do that using the static class Received and the method InOrder.

Example:

```
var substitute1 = Substitute.For<ISomeInterface1>();
var substitute2 = Substitute.For<ISomeInterface2>();
var something = new SomethingThatNeedsACommand(substitute1, substitute2);

// Act
something.DoSomething();

// Assert
Received.InOrder(() => {
        substitute1.MethodName1();
        substitute2.MethodName();
        substitute1.MethodName2();
});
```

9

Characteristic of Good Unit Tests

The Pillars of Good Unit Tests

1. Trustworthiness - Developers will want to run trustworthy tests, and they'll accept the test results with confidence. Trustworthy tests don't have bugs, and they test the right things.

2. Maintainability - Unmaintainable tests are nightmares because they can ruin project schedules, or they may be sidelined when the project is put on a more aggressive schedule. Developers will simply stop maintaining and fixing tests that take too long to change or that need to change very often on very minor production code changes.

3. Readability - This means not just being able to read a test but also figuring out the problem if the test seems to be wrong. Without readability, the other two pillars fail pretty quickly: Maintaining tests becomes harder, and you can't trust them anymore because you don't understand them.

F.I.R.S.T.

- Fast - Tests should be fast! They should run quickly. When tests run slow, you won't want to run them frequently.

- Independent - Tests should not depend on each other! One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like. Isolated!

- Repeatable - Tests should be repeatable in any environment! You should be able to run them in the production environment, in the QA environment, on your laptop while riding home on the train without a network.

- Self-Validating - The tests should have a boolean output, either they pass or fail!

- Timely - The tests need to be written in a timely fashion. Unit tests should be written JUST BEFORE the production code that make them pass!

Only One...

One test class per tested class

- You want to be able to quickly locate all tests for a specific class (or one test class per unit of work under test).
- If we are compelled to split unit tests for a class across multiple files, that may be an indication that the class itself is poorly designed (the class doesn't follow the SOLID principles).

One/single concept per test

- We don't want long test functions that go testing one miscellaneous thing after another.

One assert per test

- The first time an assert fails, it actually throws a special type of exception that is caught by the test runner. That also means no other lines below the line that just failed will be executed.
- Testing only one concern - A concern is a single end result from a unit of work: a return value, a change to system state, or a call to a third-party object.

One mock per test

- In a test where you test only one thing, there should be no more than one mock object. All other fake object will act as stubs.
- Having more than one mock per test usually means you're testing more than one thing.

Avoid Logic in Tests

The chances of having bugs in your tests increase almost exponentially as you include more and more logic in them!

Focus on the end result, rather than implementation details.

If you have any of the following inside a unit test, your test contains logic that shouldn't be there:
- switch, if, or else statements
- foreach, for, or while loops

Private Methods

- Private methods are usually private for a good reason in the developer's mind.

- For testing purposes, the public contract (the overall functionality) is all that you need to care about.

- Think of it this way: no private method exists without a reason. Some public method ends up invoking this private method.

- Private method is usually part of a bigger unit of work.

More Notes

- Don't skip trivial tests
- Test boundary conditions
- Use a coverage tool
- Test should be easy to implement
- Test should be relevant tomorrow
- Test should be consistent in its results! (should always returns the same result if you don't change anything between runs)

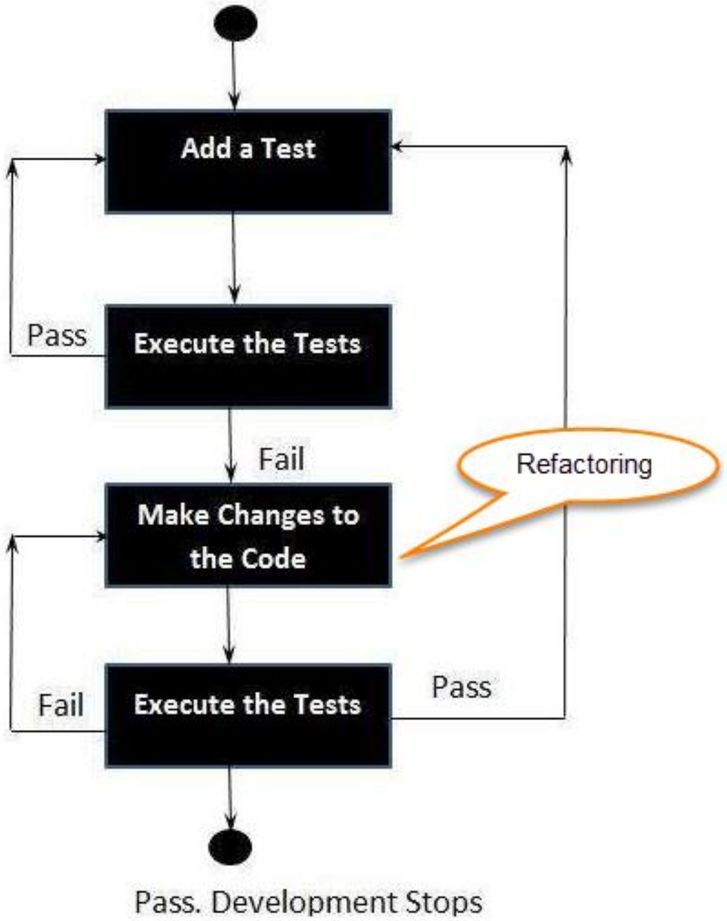
10

TDD

Definition

TDD - Test Driven Development

The three laws (rules) of TDD:
1. Don't write any production code until you have written a failing unit test.
2. Don't write more of a unit test than is sufficient to fail or fail to COMPILE (if that is a new method/class, if refactoring something, etc).
3. Don't write any more production code than is sufficient to pass the failing test.



Why It Is So Important?

In real programming you had to spend lots of time writing code, and then lots more time getting it to compile. And in this whole process you'll know if the code works in the end.

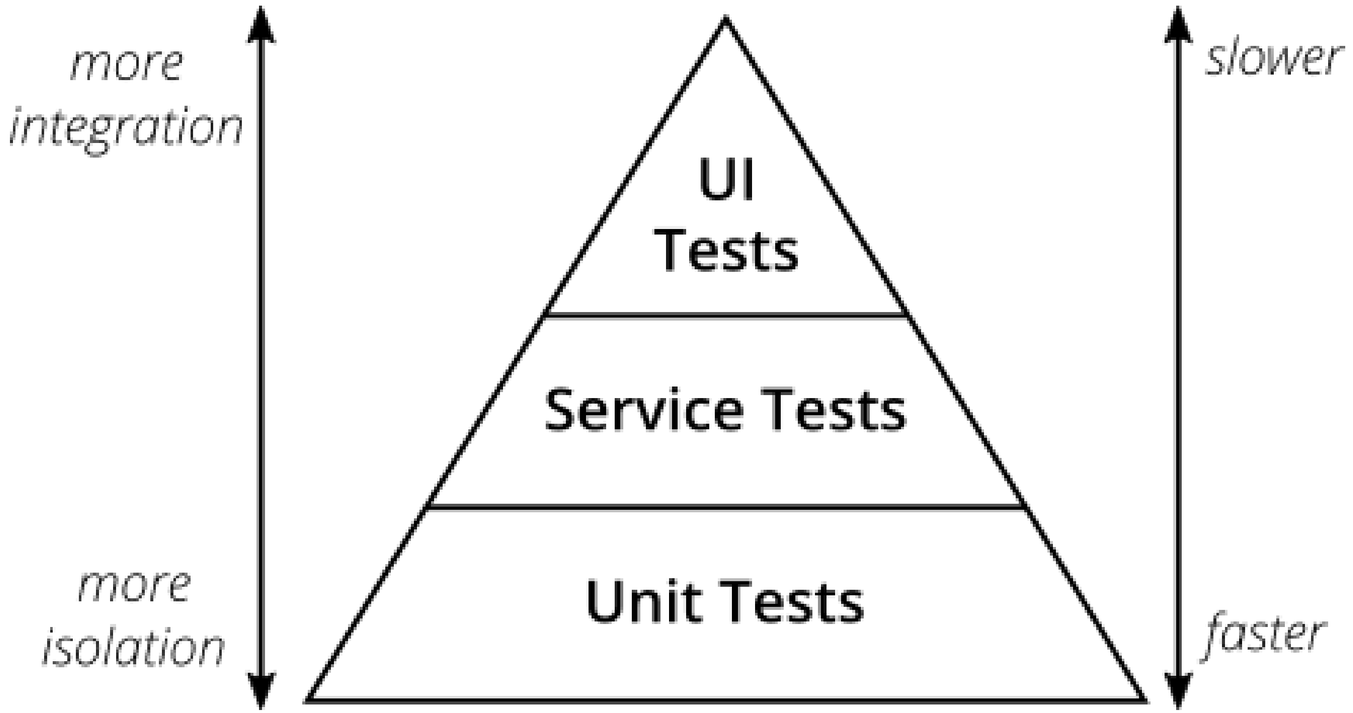
But using TDD every part of the code will be tested in the same moment it's written! No need to wait to compile and run the whole codebase.

One of the biggest benefits of TDD is:
You're testing the test itself! First you're seeing the test fails and later you're seeing it passes. If you expect it to fail and it passes, you might have a bug in your test or you're testing the wrong thing.

11

More Types of Automated Tests

Testing Pyramid



Integration Tests

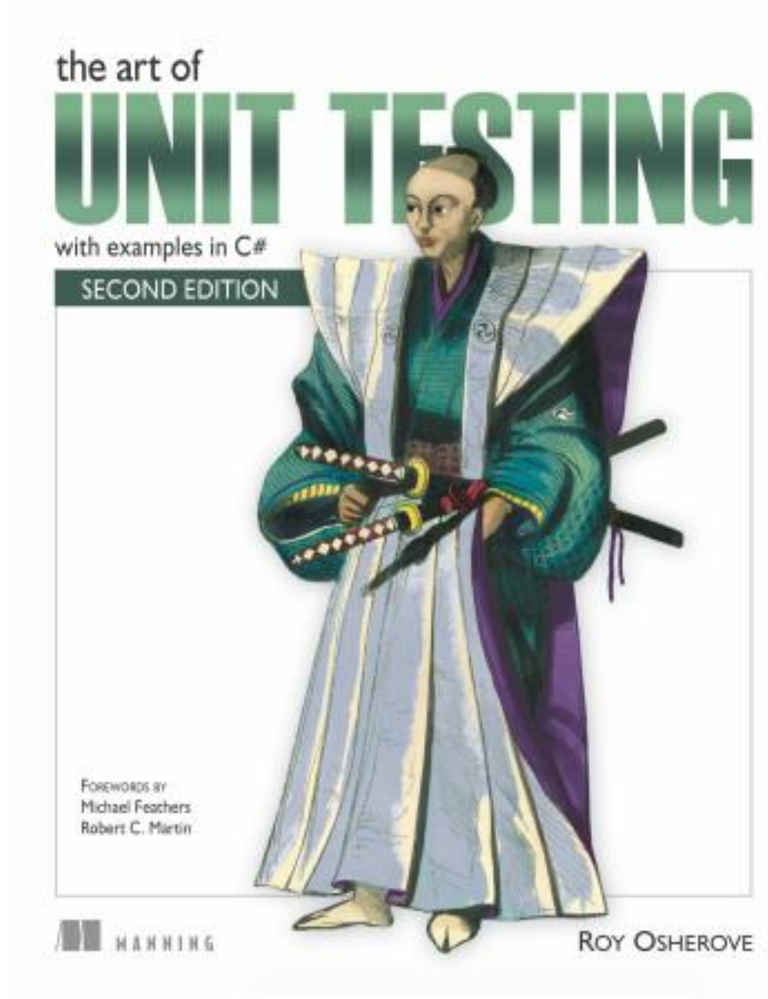
Any tests that aren't FAST and consistent and that use one or more REAL DEPENDENCIES of the units under test.

Integration testing is testing a unit of work without having full control over all of it and using one or more of its real dependencies, such as time, network, database, threads, random number generators, and so on.

12

Resources

The Art of Unit Testing (The Unit Testing Bible)




endava