**Unit test**

Code that invokes a unit of work within the confines of a project layer while faking external dependencies and validates an assumption about one specific scenario.

**Unit of Work**

Everything that happens from invoking a public method to it returning the results after it is finished; It is the work done along the path you see the debugger take through your code.

**Integration test**

Code that invokes a unit of work that crosses project boundaries, uses actual external dependencies (e.g. database, system clock), an/or validates many different aspects (e.g. authorisation end points) about the code under test.

**Stub**

A substitute for a dependency in the code under test that allows the code to compile and the dependency to return data as specified by the test but importantly cannot itself directly make a test fail.

**Mock**

A substitute for a dependency in the code under test that knows how many times each of its methods were called and in what order so that it can validate an assumption about how the dependency was used and therefore make a test fail.

**Fake**

A generic term for a replacement of a real dependency with something the test specifies, which includes both stubs and mocks.

**Seam**

A seam is place where you can alter a behaviour in your program without editing in that place.

**Three-part naming**

Unit of Work – Initial condition - Expected result

Examples

UserLogsIn\_WithValidCredentials\_ExpectedResult

UserLogsIn\_WithBadPassword\_RedirectsToHome

UserLogsIn\_FailsThreeTimes\_LocksOutAccount

Scan test names quickly

Same Unit of work sorts together

Read like business rules

Use builder Pattern to create service injections with return test data as methods inside the builder

**Tips on Poorly tested code**

A test that checks a constructor works is unnecessary as the subsequent test do check this implicitly

Mocking a constructor & asserts the construction of the mock is no value - test real constructor

Checking Type of a return value is very low value as the test would break on its own

A test that checks the wording of a message is very low value

Check exception types not exception messages

Checking only that a property in class exists is low value – others will test this implicitly

Asserting a static value is an unreliable test- other tests may change the static value -racing condition

**Tips on testable code**

Avoid the “new” keyword – ok for lists, arrays, primitives

Avoid mutable global state

Add seams- nullify difficult to work calls/methods by overriding them in the test instance

Composition over inheritance – use Dependency Injection

Example: Base class for logging – ensure the dependencies in this base class are also injectable. When running unit tests mock these dependencies hence avoid side effects

All sub classes must dependably implement the interface - No not implemented - **L**iskov substitution

Use interfaces – **I**nterface Segregation – allows use of fakes for real ones when testing

Constructors takes interfaces – Dependency injection - abstraction

**Risk Driven Testing**

Different inputs and scenarios

Test **values** – returned

Example: returns…

Test **state** of the system changed as expected/not

Example: updated (assert change of date time variable – using a DateTime Service)

Test **Interaction** – test a specific method was called during the unit of work execution

Example: assert SaveChanges() is called only once when updating

Using a DateTime service - helps **reliably** test the system time – using the System date time is technically integration testing

Gut check on interaction tests – don’t do if testers would be doing it

**Brittle Test Types**

Change detector tests

If you change code without affecting the interface signature or business logic and the test break => brittle test

Private inspector

Tests directed at a private/protected method – inhibit free refactoring, they can be accessed from public method

Business rule binder

If you change a business rule and tests don’t break => brittle, adding to shouldn’t break tests

**Tips for flexible tests**

Maximum of one mock per test

Write fewer than 10% of tests with mocks - signal the code is harder to refactor into services/libraries if more mocks used in a unit test

Rarely directly test a private method - should be accessible from a public method & tested there

Test by real world scenario not by method

**Unit Testing Checklist**

1. Test name describers the scenario; Unit of Work - Initial Condition - Expected Result
2. Contains arrange, act, assert
3. Stays within one project layer (API, Web, Service, Data)
4. It is a state, value or interaction test
5. Fakes all dependencies
6. Does not fake more than it has to
7. Tests for behaviour over one-test-to-one-method design
8. Mocks at most one dependency – Avoid brittle tests
9. Favours the public API – tests know less about the implementation details- flexibility to update
10. Asserts against one object – resist temptation to asset more – if more should be on same object/type (4)
11. Does not repeat code that could be refactored in to builder/Factory methods – DRY