Flutter test session by Ezzabuzaid

Agenda

1. Introduction
2. Test suite, Test case and Test report
3. Naming Convention
4. Test against interface
5. What is not unit test
6. Structure your tests
7. Test configuration
8. Test hooks
9. Parameterized test
10. Test **double**
11. test() vs group()
12. Exception and Assertion
13. Documentation session

Introduction

Software testing is important aspect in the vast majority of the applications nowadays, with test we can refactor without fair and augment without breaking.

We all have been in situation were we look at function that is not understandable due to the complex integration or bad flow and a lot of other cases were there's something wrong!

Test can help us to avoid all of this shit and drive the code to be intuitive, sensible, strict-full and coupled less!

This guide will wake you through the basics of what is unit tests and how to write, understand, improve them, then we will start by writing what a code does in variety of cases, exploring the important notes and demonstrating how to test against interface!

Test Case, Suite and Report

- **Test Case**
  - is a set of actions
  - executed to verify a particular feature or functionality
  - of your software application working as expected

- **Test Suite**
  - is a collection of test cases
  - each test case represent a unit of work under test

- **Test report**
  - A test report should tell whether the current application revision satisfies the requirements for the people who are not necessarily familiar with the code.
  - the tester, the DevOps engineer who is deploying the code and the **Future you** two years from now.

This can be achieved best if the tests speak at the requirements level and include 3 parts:

1. What is being tested? For example, the ProductsService.addNewProduct method.
2. Under what circumstances and scenario? For example, no price is passed to the method.
3. What is the expected result? For example, the new product is not approved.

Let's jump to naming convention to make it clear,

 # Naming Convention

The basic naming of a test comprises of three main parts:

[UnitOfWorkName]_[ScenarioUnderTest]_[ExpectedBehavior].

- Unit Of Work Name
  - is The name of the method or group of methods or classes you're testing.

- Scenario
  - is The conditions under which the unit is tested,
  - such as "bad login" or "invalid user" or "good password."
  - You could describe the parameters being sent to the public method or the initial state of the system when the unit of work is invoked such as "system out of memory" or "no users exist" or "user already exists."

- Expected Behavior
  - is What you expect the tested method to do under the specified conditions.
  - This could be one of three possibilities:
    - return a value as a result (a real value, or an exception).
    - change the state of the system as a result (like adding a new user to the system, so the system behaves differently on the next login).
    - or call a third-party system as a result (like an external web service).

C# Examples

1. AddNewProductTest_NoPriceWasPassed_ShouldNotAddTheProduct
2. AddNewProductTest_PriceWasPassed_ShouldAddThePrice
3. AddNewProductTest_ErrorThrown_ShouldExitProgram

Dart Example

```dart
group('AddNewProductTest', () {
      test('NoPriceWasPassed_ShouldNotAddTheProduct', (){
      });

      test('PriceWasPassed_ShouldAddThePrice', (){
      });

      test('ErrorThrown_ShouldExitProgram', (){
      });
})
```

- The **UnitOfWork** presents the group name and **StateUnderTest** only if there's one state
- The **State(Scenario)UnderTest** presents the group name only if there's multiple state
- The **ExpectedBehavior** presents several test cases

Using *Story teller approach*

Something should/not happen when a condition is evaluated.

```
group('',  () {
      test(should not proceed if the form is not valid , ()=>{
            ui.login();
            expect(bloc.login, calls.zero);
      });

      test(should not change the price if no price was passed, ()=>{
            ProductsService.updateProduct();
            expect(price).not.toBeChanged();
      });

      test(should change the price if a price was passed, ()=>{
            ProductsService.updateProduct(PRICE);
            expect(price).toBeChanged();
      });
});
```

Regardless the name that you stick to, make sure that the

- Test name should express a specific requirement
  - Your unit test name should express a specific requirement.
  - This requirement should be somehow derived from either a business requirement or a technical requirement.
  - In any case, that requirement has been broken down into small enough pieces, each of which represents a test case.
  - If your test is not representing a requirement, why are you writing it? Why is that code even there
  - unit tests are a design specification of how a certain *behaviours* should work, don’t over assert everything

- Tests are not commented
  - Don’t do it!
  - Tests have a reason to be or not.
  - Don't comment them out because they are too slow, too complex or produce false negatives.
  - Instead, make them fast, simple and trustworthy.

- Test is stateless
  - And mustn’t change the outer state.
  - Unit tests are isolated and independent of each other.
  - Any given behaviour should be specified in **one and only one test.**
  - The execution/order of execution of one test **cannot affect the others.**

- Test cover the general case and the edge cases

- Test doesn’t use for loops or conditions inside a tests to evaluate assertions,
  - only valid case for **for loop** is to evaluate test cases in order to mimic the **parameterized** test way

- Test case is black box so test only the behaviours, not the internal implementation (private properties and methods)
  - Changing the internal implementation of a class/object will not necessarily force you to refactor the tests
  - If a test is failing, we might have to debug to know which part of the code needs to be fixed (***we won’t write tests for tests***)

- Each test following the ***F.I.R.S.T*** principle
  - Fast: They do not do much, so obviously they are quick to run.
  - Independent: Each test sets up a new person and passes in all the parameters that are required for the test.
  - Repeatable: Tests should be repeatable in any environment without varying results.
    If they do not depend on a network or database, then it removes the

possible reasons for tests failing,
as the only thing they depend on is the code in the class or method that is being tested.
If the test fails, then the method is not working correctly or the test is set up wrong — and nothing else.

- Self-validating: Each test has a single assert or two at most, which will determine whether the test passes or fails. (Look at the picture below)
- Timely: Unit tests should be written just before the production code that makes the test pass. This is something that you would follow if you were doing TDD (Test Driven Development), but otherwise it might not apply.

- Tests are the live documentation of the code.

This what single assertion implies

:)

```
it('should properly sanitize strings', () => {
  expect(sanitizeString('Avi'+String.fromCharCode(243)+'n')).toBe('Avion');
  expect(sanitizeString('The space')).toBe('The-space');
  expect(sanitizeString('Weird chars!!')).toBe('Weird-chars-');
  expect(sanitizeString('file name.zip')).toBe('file-name.zip');
  expect(sanitizeString('my.name.zip')).toBe('my-name.zip');
});
```

Better: write a test for each type of sanitization. It will give a nice output of all possible cases, improving maintainability.

:) :)

```
it('should sanitize a string containing non-ASCII chars', () => {
  expect(sanitizeString('Avi'+String.fromCharCode(243)+'n')).toBe('Avion');
});

it('should sanitize a string containing spaces', () => {
  expect(sanitizeString('The space')).toBe('The-space');
});

it('should sanitize a string containing exclamation signs', () => {
  expect(sanitizeString('Weird chars!!')).toBe('Weird-chars-');
});

it('should sanitize a filename containing spaces', () => {
  expect(sanitizeString('file name.zip')).toBe('file-name.zip');
});

it('should sanitize a filename containing more than one dot', () => {
  expect(sanitizeString('my.name.zip')).toBe('my-name.zip');
});
```

Test against interface

your class shouldn’t know or care which implementation of the dependency it’s working with hence we need to start thinking of making an abstraction layer that helps to get the right implementation despite it’s owner
there’s a handy patterns that help to solve this problem, dependency injection and service locator

What is not unit test

1. It talks to the database

2. It communicates across the network
3. It touches the file system

Structure your tests

- AAA
  - Structure your tests with 3 well-separated actions
  - Arrange, Act & Assert
  - Following this structure guarantees that the reader spends no brain-CPU on understanding the test plan
  - You start by arranging a fake object, you act on the thing you're testing, and then you assert on something at the end of the test.
    - A - Arrange: All the setup code to bring the system to the scenario the test aims to simulate. This might include instantiating the unit under test constructor, adding DB records, mocking/stubbing on objects and any other preparation code.
    - A - Act: Execute the unit under test. Usually 1 line of code
    - A - Assert: Ensure that the received value satisfies the expectation. Usually 1 line of code.

- GWT
  - Mainly used with BDD frameworks
  - Given, When & Then
  - **Given** Failed login page **When** user clicks on submit **Then** an error should show.

- Everything should be black-box tested
- Don't "foo", use realistic input data
- Don't catch errors, expect them
- Write a test to verify the work and not just to write them
- No test better more than a dummy test because "There's no point in writing a bad unit test, unless you're learning how to write a good one and these are your first steps in this field."

Test double

When it comes to test a function that invoke http, extensive calculation or even calls my dad, we need to be careful that it will not say (HI)!
All what we need to know is that the function tried to do and after doing it did a predetermined logic, that's where the **Test Double**

Test Double is a generic term for any case where you replace a production object for testing purposes.

- [Stub]: is a controllable replacement for an existing dependency (or collaborator) in the system. By using a stub, you can test your code without

dealing with the dependency directly.
    1. provide a predetermined response from a collaborator
    2. take a predetermined action from a collaborator, like throwing an exception
    3. used to charge the state,

- [Mock]: used to verify the behavior
    1. verify the contract between the code under test and a collaborator
    2. verify the collaborator's method is called the correct number of times
    3. verify the collaborator's method is called with the correct parameters
    4. throw an exception if they receive a call they don't expect

- [Spy]: partial mock.
    1. Mock a specific method or properties.

- The purpose of using **Mock** or **Stub** is to eliminate testing all the dependencies of a class or function so your tests are more focused and simpler in what they are trying to prove.
- To stop thinking about the difference between **STUB** and **MOCK**
    - **Stub** is like making sure a method returns the correct value
    - **Mock** is like actually **stepping into the method** and making sure everything inside is correct before returning the correct value.

Isolation framework is a set of programmable APIs that makes creating fake objects much simpler, faster, and shorter than hand-coding them.
Dynamic fake object is any stub or mock that's created at runtime without needing to use a handwritten (hardcoded) implementation of that object.
Using dynamic fakes removes the need to hand-code classes that implement interfaces or derive from other classes, because the needed classes can be generated for the developer at runtime, in memory, and with a few simple lines of code.

Test hooks

A test hook is function that run before or after a test case.
Often, while writing tests, you have some setup work that needs to happen before tests run, and you have some finishing work that needs to happen after tests run.
1. setUp
2. setUpAll
3. tearDown
4. tearDownAll

there's two approaches

Test configuration

You can configure the test by putting an annotation on top of the test suite or by passing it as argument to test functions (test(), group())

Test configuration can be applied globally be setup a test configuration file For more information: https://github.com/dart-lang/test/blob/master/pkgs/test/doc/configuration.md#tags
Or by annotate the test suite by @ConfigName

- Tagging tests
  Imagine that you have a feature that need more special work on specific platform only and you don't want to run it on the others, think of them as they are test filter

- Test Timeout
  The default time for the test to pass is 30s and this can be changed by
  1. @Timeout(const Duration(seconds: 45))
  2. @Timeout.factor(1.5)
  the value we got from changing the timeout is to test a specific case where the time is a factor, e.g testing a stream the does some heavy mapping before emitting the final result

- Skip Test
  If a test, group, or entire suite isn't working yet and you just want it to stop complaining, you can mark it as "skipped". The test or tests won't be run, and, if you supply a reason why, that reason will be printed. In general, skipping tests indicates that they should run but is temporarily not working.
  if you don't have time or for some reasons writing test lack for the being time then write a description and just skip to further time

Parameterized test

Sometimes you'll need to test a function with different parameter from file or hard coded, unfortunately this is not offered out of the box by dart so we need to work around it in order to mimic the behavior

Java example: https://www.guru99.com/junit-parameterized-test.html
C# example in code
Dart example: in code

```
int add (a, b){ return a + b; }
test() {
    var result1 = add(1, 2);
    var result2 = add(3, 4);
    var result3 = add(5, 6);
    expect(result1, 3);
    expect(result2, 7);
```

```
    expect(result3, 11);
}
```

Instead of, we can do like this

```
group('should add') {
    [
        [1, 2, 3],
        [3, 4, 7],
        [5, 6, 11]
    ].forEach((numbers){
        test('numbers[0] + numbers[1] equal to numbers[2]', (){
            expect(add(numbers[0], numbers[1] , numbers[2]))
        });
    })
}
```

This approach will help you to identify where the problem is

Properties of a test

The tests that you write should have three properties that together make them good:

- **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.

- **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.

- **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 fall pretty quickly. Maintaining tests becomes harder, and you can't trust them anymore because you don't understand them.

Together, the three pillars ensure your time is well used. Drop one of them, and you run the risk of wasting everyone's time

- **Writing trustworthy tests**
    - There are several indications that a test is trustworthy.
        - If it passes, you don't say, "I'll step through the code in the debugger to make sure."
        - You trust that it passes and that the code it tests works for that

specific scenario.
        - If the test fails, you don't tell yourself, "Oh, it's supposed to fail," or "That doesn't mean the code isn't working."
        - You believe that there's a problem in your code and not in your test.
        - In short, a trustworthy test is one that makes you feel you know what's going on and that you can do something about it.

Expect vs verify
Another bad habit is trying to replace every assert with Mockito's verify(). It's important to clearly understand what is being tested: interactions between collaborators can be checked with verify(), while confirming the observable results of an executed action is done with asserts.

- Wrapping up!
    - Any test case should be
        - Fast
        - Readable
        - Maintainable
        - Trustworthy
        - Consistant
        - To be a code!
    - Whenever a bug is found, create a test that replicates the problem **before touching any code**. (QC always raise a bugs)
    - Don't defer writing tests!, **Skip** them, create blank test only with name, so you can go back to it!

Excerpts from *Art of unit Testing book* by *Roy Osherove*.

- **A unit test**
    - is an automated piece of code that invokes the unit of work being tested, and then checks some assumptions about a single end result of that unit.
    - A unit test is almost always written using a unit testing framework.
    - It can be written easily and runs quickly.
    - It's trustworthy, readable, and maintainable.
    - It's consistent in its results as long as production code hasn't changed.

Refactoring means changing a piece of code without changing its functionality. If you've ever renamed a method, you've done refactoring. If you've ever split a large method into multiple smaller
method calls, you've refactored your code. The code still does the same thing, but it becomes easier to maintain, read, debug, and change.

Over-specification is the act of specifying too many things that should happen that your test shouldn't care about; for example, that stubs were called.

Test driven development can make your code quality soar, decrease the number of bugs, raise your confidence in the code, shorten the time it takes to find bugs, improve your code's design, and keep your manager happier.
BUT It's important to realize that TDD doesn't ensure project success or tests that are robust or maintainable. It's quite easy to get caught up in the technique of TDD and not pay attention to the way unit tests are written: their naming, how maintainable or readable they are, and whether they test the right things or might have bugs.

Having many mocks, or many expectations, in a single test can ruin the readability of the test so it's hard to maintain or even to understand what's being tested. If you find that your test becomes unreadable or hard to follow, consider removing some mocks or some mock expectations or separating the test into several smaller tests that are more readable.

It's considered good practice to test only one concern per test. Testing more than one concern can lead to confusion and problems maintaining the test. Having two mocks in a test is the same as testing several end results of the same unit of work. If you can't name your test because it does too many things, it's time to separate it into more than one test.

The term continuous integration is literally about making the automated build and integration process run continuously. You could have a certain build script run every time someone checks in source code to the system, or every 45 minutes, or when another build script has finished running, for example. A CI server's main jobs are these:

- Trigger a build script based on specific events
- Provide build script context and data such as version, source code, and artifacts from other builds, build script parameters, and so on www.it-ebooks.info Automated builds running automated tests
- Provide an overview of build history and metrics
- Provide the current status of all the active and inactive builds

One of the biggest failed projects I worked on had unit tests. Or so I thought. I was leading a group of programmers creating a billing application, and we were doing it in a fully test-driven manner—writing the test, then writing the code, seeing the test fail, making the test pass, refactoring, and starting all over again. The first few months of the project were great. Things were going well, and we had tests that proved that our code worked. But as time went by, requirements changed. We were forced to change our code to fit those new

requirements, and when we did, tests broke and had to be fixed. The code still worked, but the tests we wrote were so brittle that any little change in our code broke them, even though the code was working fine. It became a daunting task to change code in a class or method because we also had to fix all the related unit tests. Worse yet, some tests became unusable because the people who wrote them left the project and no one knew how to maintain the tests, or what they were testing. The names we gave our unit-testing methods were not clear enough, and we had tests relying on other tests. We ended

Leave a codebase better than what you found it. Never settle for just enough, doing small cleanups pay off in the long run, even if they serve only for readability purposes by making things easier for the next developer.

The one-test-class-per-class pattern is the simplest and most common pattern for organizing tests. You put all the tests for all methods of the class under test in one big test class. When you're using this pattern, some methods in the class under test may have so many tests that the test class becomes difficult to read or browse. Sometimes the tests for one method drown out the other tests for other methods. That in itself could indicate that maybe the method test is doing too much. TIP Test readability is important. You're writing tests as much for the person who will read them as for the computer that will run them. I cover readability aspects in the next chapter. If the person reading the test has to spend more time browsing the test code than understanding it, the test will cause maintenance headaches as the code gets bigger and bigger. That's why you might think about doing it differently.