Unit Testing without Assertions

Asked 16 years, 2 months ago Modified 1 year, 9 months ago

Viewed 27k times

42

Occasionally I come accross a unit test that doesn't Assert anything. The particular example I came across this morning was testing that a log file got written to when a condition was met. The assumption was that if no error was thrown the test passed.

I personally don't have a problem with this, however it seems to be a bit of a "code smell" to write a unit test that doesn't have any assertions associated with it.

Just wondering what people's views on this are?

unit-testing assert

Share

Improve this question

Follow

asked Sep 26, 2008 at 2:26

lomaxx

116k ● 58 ● 147 ● 180

16 Answers Sorted by: Highest score (default)

It's simply a very minimal test, and should be documented as such. It only verifies that it doesn't explode when run.

29

The worst part about tests like this is that they present a false sense of security. Your code coverage will go up, but it's illusory. Very bad odor.

Share Improve this answer

Follow

answered Sep 26, 2008 at 2:32

David M. Karr

15.2k ● 22 ● 109 ● 219

25

This would be the official way to do it:

```
// Act
Exception ex = Record.Exception(() => someCode());

// Assert
Assert.Null(ex);
```

Share Improve this answer

Follow

edited Jan 7, 2018 at 22:03

answered Sep 26, 2008 at 2:39

Brad Wilson

70.4k ● 9 ● 77 ● 85

2 Junit5: `Assertions.assertDoesNotThrow(() -> someCode());` – caduceus Mar 11, 2023 at 9:42

1 xUnit.net also has `Assert.DoesNotThrow()`, though it has the downside of being both the Act and Assert together in one line. For those who wish to more strictly observe AAA, `Record.Exception` is the preferred method. – Brad Wilson Mar 12, 2023 at 3:10

20

If there is no assertion, it isn't a test.

Quit being lazy -- it may take a little time to figure out how to get the assertion in there, but well worth it to know that it did what you expected it to do.

Share Improve this answer

Follow

answered Sep 26, 2008 at 3:32

ryw

9,635 ● 5 ● 29 ● 34

1 ha... I tend to feel this way, but I was wondering if there was a valid reason for the use of tests without assertions :)
– lomaxx Sep 26, 2008 at 6:26

For a counter-view to "If there is no assertion, it isn't a test" (at least with pytest), see Nick Chammas' comment on stackoverflow.com/a/20275035/805141 – Daniel Nov 22, 2017 at 16:55

11

These are known as **smoke tests** and are common. They're basic sanity checks. But they shouldn't be the only kinds of tests you have. You'd still need some kind of verification in another test.

Share Improve this answer

Follow

answered Sep 26, 2008 at 3:24

Mark Cidade

99.8k ● 33 ● 229 ● 237

2 These are not necessary smoke tests: softwaretestingfundamentals.com/smoke-testing – camposer Jul 5, 2016 at 7:52

2 It's a smoke test in the sense that if it runs without throwing an exception, it passes. – Mark Cidade Jul 5, 2016 at 13:42

11

Such a test smells. It should check that the file was written to, at least that the modified time was updated perhaps.

I've seen quite a few tests written this way that ended up not testing anything at all i.e. the code didn't work, but it didn't blow up either.

If you have some explicit requirement that the code under test doesn't throw an exception and you want to explicitly call out this fact (tests as requirements docs) then I would do something like this:

```
try
{
  unitUnderTest.DoWork()
}
catch
{
  Assert.Fail("code should never throw exceptions
but failed with ...")
}
```

... but this still smells a bit to me, probably because it's trying to prove a negative.

Share Improve this answer edited Sep 27, 2008 at 0:52

Follow

answered Sep 26, 2008 at 2:40

craigb

16.9k ● 7 ● 53 ● 63

I have to agree. I personally agree that you want to somehow verify that the write happened, not necessarily that the write was correct – lomaxx Sep 26, 2008 at 2:42

love this simple solution – Adam Weitzman Aug 13, 2021 at 17:40

4

In some sense, you are making an implicit assertion - that the code doesn't throw an exception. Of course it would be more valuable to actually grab the file and find the appropriate line, but I suppose something's better than nothing.

Share Improve this answer

Follow

answered Sep 26, 2008 at 2:29

chessguy

468 ● 5 ● 15

4

It can be a good pragmatic solution, especially if the alternative is no test at all.

The problem is that the test would pass if all the functions called were no-ops. But sometimes it just isn't feasible to verify the side effects are what you expected. In the ideal world there would be enough time to write the checks for every test ... but I don't live there.

The other place I've used this pattern is for embedding some performance tests in with unit tests because that was an easy way to get them run every build. The tests don't assert anything, but measure how long the test took and log that.

Share Improve this answer

Follow

answered Sep 26, 2008 at 2:35

Rob Walker

47.4k ● 15 ● 100 ● 137

1 i disagree - it's not pragmatic, it's lazy ;) – ryw Sep 26, 2008 at 3:35

3

The name of the test should document this.

```
void TestLogDoesNotThrowException(void) {
    log("blah blah");
}
```

How does the test verify if the log is written without assertion ?

Share Improve this answer

Follow

answered Jul 1, 2009 at 19:20

philant

35.7k ● 11 ● 73 ● 113

2

In general, I see this occuring in integration testing, just the fact that something succeeded to completion is good enough. In this case Im cool with that.

I guess if I saw it over and over again in *unit tests* I would be curious as to how useful the tests really were.

EDIT: In the example given by the OP, there is some testable outcome (logfile result), so assuming that if no error was thrown that it worked is lazy.

Share Improve this answer Follow

answered Sep 26, 2008 at 2:30

Jim Burger
4,537 ● 1 ● 26 ● 27

1

We do this all the time. We mock our dependencies using JMock, so I guess in a sense the JMock framework is doing the assertion for us... but it goes something like this. We have a controller that we want to test:

```
Class Controller {
  private Validator validator;

  public void control(){
    validator.validate;
  }

  public setValidator(Validator validator){
this.validator = validator; }
}
```

Now, when we test Controller we dont' want to test Validator because it has it's own tests. so we have a test with JMock just to make sure we call validate:

```
public void testControlShouldCallValidate(){
```

```
mockValidator.expects(once()).method("validate");
  controller.control;
}
```

And that's all, there is no "assertion" to see but when you call control and the "validate" method is not called then the JMock framework throws you an exception (something like "expected method not invoked" or something).

We have those all over the place. It's a little backwards since you basically setup your assertion THEN make the call to the tested method.

Share Improve this answer Follow

answered Sep 26, 2008 at 2:48

community wiki
codeLes

1 Effectively, your expectation is an assertion by another name. This is good. But the equivalent "test" without expects would prove nothing. – Dale Apr 27, 2020 at 1:35

1

I've seen something like this before and I think this was done just to prop up code coverage numbers. It's probably not really testing code behaviour. In any case, I agree that it (the intention) should be documented in the test for clarity.

Share Improve this answer

Follow

answered Sep 26, 2008 at 3:22

cruizer

6,151 ● 2 ● 29 ● 34

1

I sometimes use my unit testing framework of choice (NUnit) to build methods that act as entry points into specific parts of my code. These methods are useful for profiling performance, memory consumption and resource consumption of a subset of the code.

These methods are definitely not unit tests (even though they're marked with the *[Test]* attribute) and are always flagged to be ignored and explicitly documented when they're checked into source control.

I also occasionally use these methods as entry points for the Visual Studio debugger. I use Resharper to step directly into the test and then into the code that I want to debug. These methods either don't make it as far as source control, or they acquire their very own asserts.

My "real" unit tests are built during normal TDD cycles, and they always assert something, although not always directly - sometimes the assertions are part of the mocking framework, and sometimes I'm able to refactor similar assertions into a single method. The names of those refactored methods always start with the prefix "Assert" to make it obvious to me.

Share Improve this answer

Follow

answered Oct 1, 2008 at 7:36

Lee
1,596 ● 2 ● 15 ● 21

0

I have to admit that I have never written a unit test that verified I was logging correctly. But I did think about it and came across this discussion of how it could be done with JUnit and Log4J. Its not too pretty but it looks like it would work.

Share Improve this answer

Follow

answered Sep 26, 2008 at 2:35

Brian Matthews
8,596 ● 7 ● 47 ● 68

When using a 3rd party framework for logging, I tend to test that the act of logging worked correctly, rather than directly testing the log file output. I would interface out the logging framework so I could fake it under test and just ensure the right methods were being called. – Jim Burger Sep 26, 2008 at 2:39

0

Tests should always assert something, otherwise what are you proving and how can you consistently reproduce evidence that your code works?

Share Improve this answer

Follow

answered Sep 26, 2008 at 2:46

David Robbins
10k ● 7 ● 54 ● 83

1 I think the statement "should always assert something" is misleading. Tests should always "prove" something, but that does not require an "assert" statement. – codeLes Sep 26, 2008 at 3:02

How do you prove that you didn't just forget to write an assertion? I've seen tests accidentally pass exactly because no assertions were made -- adding a NOP `Assert.Pass();` makes your intent that little bit clearer. – Chris Oldwood Sep 11, 2020 at 8:47

0

I would say that a test with no assertions indicates one of two things:

1. a test that isn't testing the code's important behavior, or
2. code without any important behaviors, that might be removed.

Thing 1

Most of the comments in this thread are about thing 1, and I would agree that if code under test has any important behavior, then it should be possible to write tests that make assertions about that behavior, either by

1. asserting on a function/method return value,
2. asserting on calls to 'test double' dependencies, or
3. asserting on changes to visible state.

If the code under test has important behavior, but there aren't assertions on the correctness of that behavior, then the test is deficient.

Your question appears to belong in this category. The code under test is supposed to log when a condition is met. So there are at least two tests:

- Given that the condition is met, when we call the method, then does the logging occur?
- Given that the condition is not met, when we call the method, then does the logging not occur?

The test would need a way to arrange the state of the code so that the condition was or was not met, and it would need a way to confirm that the logging either did or did not occur, probably with some logging 'test double' that just recorded the logging calls (people often use mocking frameworks for this.)

Thing 2

So how about those other tests, that lack assertions, but it's because the code under test doesn't do anything important? I would say that a judgment call is required. In large code bases with high code velocity (many commits per day) and with many simultaneous contributors, it is necessary to deliver code incrementally in small commits. This is so that:

1. your code reviewers are not overwhelmed by large complicated commits
2. you avoid merge conflicts
3. it is easy to revert your commit if it causes a fault.

In these situations, I have added 'placeholder' classes, which don't do anything interesting, but which provide the structure for the implementation that will follow. Adding this class now, and even using it from other classes, can help show reviewers how the pieces will fit together even if the important behavior of the new class is not yet implemented.

So, if we assume that such placeholders are appropriate to add, should we test them? It depends. At the least, you will want to confirm that the class is syntactically valid, and perhaps that none of its incidental behaviors cause uncaught exceptions.

For examples:

- Python is an interpreted language, and so your continuous build may not have a way to confirm that your placeholder class is syntactically valid unless it executes the code as part of a test.
- Your placeholder may have incidental behavior, such as logging statements. These behaviors are not important enough to assert on because they are not an essential part of the class's behavior, but they are potential sources of exceptions. Most test frameworks treat uncaught exceptions as errors, and

> so by executing this code in a test, you are confirming that the incidental behavior does not cause uncaught exceptions.

Personally I believe that this reasoning supports the temporary inclusion of assertion-free tests in a code base. That said, the situation should be temporary, and the placeholder class should soon receive a more complete implementation, or it should be removed.

As a final note, I don't think it's a good idea to include asserts on incidental behavior just to satisfy a formalism that 'all tests must have assertions'. You or another author may forget to remove these formalistic assertions, and then they will clutter the tests with assertions of non-essential behavior, distracting focus from the important assertions. Many of us are probably familiar with the situation where you come upon a test, and you see something that looks like it doesn't belong, and we say, "I'd really like to remove this...but it makes no sense why it's there. So it might be there for some potentially obscure and important reason that the original author forgot to document. I should probably just leave it so that I 1) respect the intentions of the original author, and 2) don't end up breaking anything and making my life more difficult." (See Chesterton's fence.)

answered Sep 8, 2021 at 17:57

Carl G

18.2k ● 14 ● 95 ● 118

0

Nothing beats a good old SO thread:) I think everyone missed the psychological/brain-muscle aspect of the issue.

TL;DR: if that is in the team's agreed guidelines, so be it.

Every time I encounter unexpectedly formatted, styled, ordered, etc., code, even if I understand what it does, it takes me extra brain cells and time to process. Brain cells and time are limited and being surprised by exotic coding patterns should be a rare occurrence in a productive work environment. **So if your team does this consistently, keep doing it. If not, take time to decide and follow the it.**

I would argue that even if such a test is pointless, it's better to have it in a pattern. Chaos is not chaos if it's recognizable, so if the code base has a bunch of tests with no assertions, it's easier to find and fix them.

Long answer

I'm very late to the party and started writing tests with jest for a React TS web app 2 months ago and have found myself in an identical discussion about tests with no assert (expect() in jest) statements.

Testing if the DOM has an element with an in-build getByText method will either return the element or throw an error, so one could save a few lines of code by just getting elems like this and being sure that if the test passed, the elems were for sure rendered to the DOM.

While the tech is different, the principle is the same. Having read all the answers and comments here + the few python answers from a link, I noticed the sole focus on the tech side of the matter, but the more important issue I would say is wasting human time and energy to answer the inevitable questions that will come with time, like "why are there no assertions here?..", "is this test complete?..", "what does it mean if it passed?", "or if it fails?..", "who wrote this?.", "why?..", "why me?.."

Share Improve this answer

Follow

answered Mar 17, 2023 at 8:58

Aidas Baublys
29 • 3