



Unit Testing: One Test, One Assertion - Why It Works

In my last post I touched on the topic of multiple assertions in a unit test and linked it somewhat harshly to being a factor in "lazy" or shallow testing. I actually didn't intend on linking them at all, but the truth is that unit testing isn't a series of mutually exclusive practices - they are all linked to varying degrees. I'm sure I have at least a degree of laziness myself in testing 


The Basic Idea

As a recap, the idea is that every unit test should only have one single assertion. It's a fairly well known method to combat a range of problems evident in assertion loaded tests where the numerous assertions obscure the meaning of the test, and where a failed test does not tell you the specific assertions which failed (since tests traditionally fail after any assertion failure, the remaining assertions are never executed thus leaving you blind to whether they would have passed or failed). There are other problems with the approach as well - which is why I do not like seeing multiple assertions in a single test. I'm not the only one - I think. And just as there are problems, there are also exceptions when multiple assertions might prove necessary - personally I think this scenario is pretty rare however. After my years of unit testing, I certainly have not encountered all that many.

These effects by themselves make me fairly comfortable in enforcing Single Assertions in tests as a rule, not a guideline. Guidelines are broken easily, but with rules you can at least strike back and make certain you don't run aground on someone else's inexperience or ill fated experiments in making testing "easier". It's also extremely easy to detect 

. PHPUnit, for example, reports the number of both tests and assertions executed - so you can calculate a quick ratio of assertions to tests in mere seconds. I'm going to start calling it the Obscurity Ratio! Obviously, a ratio of 1:1 would be perfect - but I'm sure we can accept some multiple assertions within a tolerance level. A ratio of more than 1.1:1 would indicate creeping obscurity (a signal it's time for me to intervene, refactor, and do some mutation testing for extra assurance). A ratio of 2:1 indicates a lost cause. Anything higher and God help you all.

This Sounds Like A Bad Idea!

In watching the reactions to my original post (and not just the comments here), I've noticed that the excuses are generally similar. People have leaned towards justifying using multiple assertions for complex return values, or pointed out that my example was too simple (since that really really shows its a Bad Idea™ ), or

that all the mini-tests would take too long to write, and even longer to execute, or that the world will end since Peter Petrelli thinks this is all a Bad Idea™ and everyone knows Sylar invented it anyway.


Not to completely oppose all the opinions (it really was a simple example), but people are spending too much time looking at the end result, and not enough at the process behind it all. And it's by missing the process that

you are missing the point.

Now let me disclaim a little - I'm really really sure there are times multiple assertions are needed. So my main point is not that this is an absolute rule with no possible exceptions so help me God, rather that it should be a rule unless you are literally forced to tackle an exception to it.

Why It's A Good Idea...

Unit Testing has various interpretations. It's original genesis promoted the verification of source code, a means of making sure code worked. It was not long before the obvious flaws in that thinking emerged - how do you verify the verified? You already know the code works before you write the test, so what value does it really bring to the table? Does it really promote code quality and rapid development - or just create a horrible task some low ranking developer will be landed with?

This sparked a revolution - Test Driven Design (TDD). In TDD, tests stepped back from verifying, and entered a role as specifications. Before you wrote the source code, you first wrote a test which specified the behaviour you intended on adding. Once you had the behaviour specified, you then wrote just enough source code to make that test pass. The result was a process of simple steps - and here's where I diverge from others. A simple example works - because all half decent source code is nothing more than an amalgamation of simple examples - in fact, I dare anyone to prove otherwise. Then I can show you the PHP manual and you can explain how the hell you made PHP that complex 

In TDD we back away from testing complex highly involved code after the fact (too late then!), to preemptively writing tests to produce simpler straightforward code produced in steps. Simpler is better - always. The differences can be startling. Try writing two identical libraries both ways and you'll see how.

It's this notion of complexity that is often used to justify multiple assertions - if you break it down into the simple components that that complexity was built out of, multiple assertions don't have a leg to stand on (just the rare instances out of your control because they absolutely must be done). The most common (just to preempt you all) scenarios for complexity in testing assertions boil down to BIG stuff - deeply nested stuff. Arrays and XML are the most common ones. Mathematical equations for relativity are simpler than those!


But let's pass on through to the other benefits instead of obsessing over just one.

If your tests double as your specification to describe how a class should behave (and yes, I'm robbing the definition list in Behaviour-Driven Development, or BDD, blind) or documentation - you're doing nothing wrong. Tests should document behaviour. This is why test method names are important - they summarise the behaviour the test is verifying.

Now add another little rule: every test only verifies one single behaviour. When we make that connection we realise something even more startling...

1 Behaviour == 1 Assertion

Now we're into the meat of my madness, the gritty reality of the one assertion per test rule. Any behaviour is,

by its very nature, both specific and unique. It does not require multiple assertions. Either it does one single thing, or it doesn't do it at all. It's black and white. The only danger is that you get so smart, you think in terms of big behaviours which are really just lots of little behaviours bundled together. Enjoy applying TDD in that case...it will sink you faster than you can say "Peter Petrelli Is Depressing" and you'll end up back in the land of writing tests AFTER the code. Anyone going to admit to doing that? 

We all know who we are (yes, I do it too now and again).

That's why Test-Driven Design is so effective. It enforces a habit (once you actually understand the damn practice which is almost designed to hide the behaviour facet!) of specifying class behaviour by behaviour, assertion by assertion. It breaks down the complex overall purpose of any class into discrete simple steps, a series of tiny goals you can easily achieve.

Simple, tiny, discrete - like individual assertions. It's THAT simple.

That's why the one assertion per test is a good idea - because it's obvious in TDD. It's what works, and it ensures your tests do exactly what they're supposed to be doing - verifying discrete behaviour, and documenting that behaviour to make other developer's lives a bit easier (and saner).

The Spontaneous Test Population Explosion Myth

This is the one myth everyone knows targets the one assertion per test rule. The question is whether it's a myth, or whether something else is. The idea is that by requiring one assertion per test, you end up with more tests, more code, higher execution times, etc. Which is to say...more behaviours. Which is to say - where the hell did the new behaviours come from?


Unwittingly, the excuse reveals itself to be the myth. If you have tests, and they can survive as multiple tests - you pretty much admitted your original tests were collections of behaviours. Welcome to the land of Obscure Tests - they test many things, and nobody can figure out just what.

Tests are documentation. Like any form of writing, you can block together ideas into a monologue or use properly formatted paragraphs and bullet points to break things down into a more digestible form for reading. It comes with a cost, but discrete tests are more helpful as documentation.

As for execution times, I really don't understand why people find this a problem. PHPUnit let's you run any test class in isolation, and you can group test suites in any shape or form. If your tests are running slow, treat them like any other code - use XDebug to locate the bottlenecks and slow tests and do something about them. Use more efficient code, refactor them to hell and back, isolate the slow tests - we can all do performance optimisation.

The other alternative is to let tests run in the background and use a notification system to report failures while you go away and write another test or two. I do this myself - see what notification apps you can hook into from PHP or some other test environment tool. I had working Snarl code for a PHP extension working under Windows somewhere if you can use a compiler and really need it.

Attitude Counts

Another compelling reason to adopt one assertion per test is your ego. Look at me - I have an ego, I write stuff because I enjoy doing it and am motivated by other people reading it and thinking it's the best thing ever written 


. We all have egos. My ego has me writing free books (though the prospect of a new Macbook Pro is another motive...hehehe).

Unfortunately our ego is also a major enemy in unit testing. If you find yourself saying you prefer multiple assertions because it works for you, and you can understand them, and they make perfect sense to you, and hey, your code works with 200% code coverage - then count how many "you" words you just used (or "I" if writing first person!).

It's not about you!

Other developers have to read your tests eventually, unless it's a top secret project only you will ever work on. Don't saddle the rest of the world with the product of your ego. Everyone finds simpler tests easier to work with. The general rule of thumb is pretty simple - if you give someone a copy of your tests, and nothing else, could they ever write the code to fit those tests in a reasonable time from scratch without begging you for assistance?

Conclusion


More food for thought 

. No pretty code to look at for this one so if you haven't read the last post yet, go read it now.

Until next time...when I find something else to moan about.

Posted by Pádraic Brady in Irishisms, PHP General, PHP Security at 18:37

This ofcourse demands that you have thought youre solutions through enough to be able to write tests prior to code... not always so im afraid... Anonymous on Feb 18 2009, 12:01

First: Padraic, I enjoyed your first post on this subject and I have to admit I did write mutiple assetions per test, but no longer 

This

second post is great, thank you for your details reasonings.

Now all I have to do is refactor my existing test... Joy! I think I'll wait till I have to do work on them.

Second: Micke - You dont't need to think your solution through!

This is the process I tend to take and (unless I'm mistaken) the TDD way:


1. Work out what I want the class to do. - Usually here I have only have a very rough Idea of what the class should do and how it should do it.
2. Write the tests to discover how you want to interact with the object. - This is where I discover exactly how I want the class to work (what feels most naturally and effient when using the class) and how I think the class is going to work internally.
3. Create Class Structure and ensure all test pass with minimal code. - This stage allows me to work out how the internals are going to work and I usually put a few empty protected and private methods in place.
4. Write the object - By this point I have a very good understanding of what the class should do and how it should do it. All my tests

are in place and working so I'll have the class done in a very short period of time. Anonymous on Feb 18 2009, 14:46

Not exactly true, though of course it can help to have at least researched what you're doing and perhaps written a mockup library/app to explore what problems you may meet.

This isn't exclusive to TDD, BDD or anything else - it effects any form of testing universally.

The problem is that TDD focuses on small behaviours and let's them build up gradually, whereas in non-TDD situation you are often struggling with the entire problem head on. By avoiding an all out collision, the step by step approach is far more focused and I find it's easier to get things done that way since it's basically enforcing the old adage of breaking down your problems into much smaller ones. Anonymous on Feb 18 2009, 15:58

Works for me 

. You get finely grained tests tracking smaller behaviours that in aggregate make something complex look really simple. It hasn't failed me yet. Anonymous on Feb 18 2009, 16:00

Your "1 behaviour == 1 assertion" statement has been bothering me for a week now, and yet I couldn't find myself completely disagreeing with it. Until tonight, when I found a perfect example to the contrary.

In this particular case, I have a method that accepts three different types of input -- so three types of behavior. In two cases, calling the method will trigger multiple observers (in one case a subset, in the other case all of them). In order to properly test the behavior, I need to make assertions against each observer to determine state. In this case, the behavior can be summed up as, "when calling the method with an array of arguments, the observers referenced should each be triggered." This begs for multiple assertions.

Another case is when you need to test the state of a value object. While you can certainly create an equivalent value object and do an assertion on equality, this can in many cases be more work and more contrived than simply running multiple assertions.

I think that your statement is definitely an **ideal** to work towards -- but I think it should be tempered with a good dose of practicality and examination of the behavior being tested. Anonymous on Feb 22 2009, 02:30