

Unit Testing: Multiple Assertions And Lazy/Shallow Testing Are Evil

Unit Testing as a practice is like any other - there are good practices, and bad practices. Two of the worst practices are overloading tests with assertions, and writing lazy or shallow tests.


Before we recount the dire consequences of these practices, it's worth knowing why they are so attractive and not immediately perceived as being bad. In short, every test you write requires that you setup the test environment, create a scenario for possible failure, add an assertion, and then ensure the source code makes that assertion pass. This requires code - sometimes a lot of code. So adding multiple assertions to each test minimises the work needed to write tests, since using multiple assertions takes advantage of existing code to avoid writing new stuff to clutter your test classes. It can also help to tackle multiple but related results in the same test.

So long as you know the assertions will pass - this makes writing unit tests quite a bit faster at times. Unfortunately, a preoccupation with minimising test code also encourages developers to keep tests overly simple to the point that they do not dig deep enough into whether a test actually accomplishes its objective - often because that objective has never previously been documented.

These considerations lead to tests which may be similar to (using PHPUnit):

```
class GameTest extends PHPUnit_Framework_TestCase
```

```
{
    public function testScoresZeroWithNoScoring()
    {
        $game = new Game;
        $this->assertEquals(, $game->score);
        $this->assertEquals(, $game->scoreTotal);
        $game->score(1);
        $this->assertEquals(1, $game->score);
        $this->assertEquals(1, $game->scoreTotal);
    }
}
```

Here we have a simple test with four assertions. All four test \$score and \$scoreTotal to make certain they remain at zero after the object is initialised and no score (or zero score) is assigned. They then revisit the situation after a score has occurred. To the naked eye, the test is easy to understand. Here's a class which will pass the above test (if you spot the problem after seeing the class, you should give yourself a treat ).

```
class Game
```

```
{
    public $score = ;
    public $scoreTotal = ;
    public function score($score)
    {
        $this->score = $score;
        $this->scoreTotal += $score;
    }
}
```

Back to those dire consequences. Consider the output of test results showing a failure I will now introduce by initially setting \$score to 1 in the class.

```
PHPUnit 3.3.14 by Sebastian Bergmann.  
F  
Time: 0 seconds  
There was 1 failure:  
1) testScoreIsZeroWithNoScoring(GameTest)  
Failed asserting that <integer:1> matches expected value <integer:0>.  
D:\projects\tinker\GameTest.php:11  
FAILURES!  
Tests: 1, Assertions: 1, Failures: 1.
```

Multiple assertions do unfortunately have side effects. The most obvious one is that it only takes one assertion to fail, and the entire test will fail with it. It ignores whether other assertions in the same test would have actually passed (hence we see that the last line of the results show PHPUnit only executed the first assertion, and ignored all others), which leaves you blind as to whether the error is impacting other assertions. This creates a maintenance nightmare - for every test that fails, you're never certain what should have failed! You only get one part of the puzzle to work with.

Unit Tests should be specific. In fact, as a general rule, there should only be one assertion per test method. If a failed test doesn't immediately tell you where the problem is, and what assertions will fail, and offer at least some minimal description of the failed behaviour (typically the test title should be sufficiently descriptive) then its utility is severely reduced. You end up doing the same detective work needed in the absence of unit tests - which makes those unit tests less beneficial since you rob the maintainer of instantaneous specific feedback and force them to edit tests, often rewriting them to be more specific, and/or employ typical debugging approaches to locate the problem in the source code.

Another impact, is that multiple assertions are often a sign that the tests were written post development, or without attention to the behaviour of the class. This increases the risk that the tests are not only confusing when they fail, but that the tests are not even complete. Truly paying attention to the role of behaviour discourages multiple assertions and promotes specificity.

There's a side story here about the foolishness of believing that code coverage is an absolute measure of the effectiveness of a unit testing suite. It's not - it's only one metric to assist in that measurement, and not a very reliable one at that. It's entirely possible to gain 90% or even 100% code coverage without writing tests that cover even a quarter of the expected behaviour of the class. Code coverage measures how much of the source code lines are actually executed - it doesn't tell you if they were executed enough times, in the right order, or if the tests were even appropriate to start with.

This is the problem some people will have noted from before (give yourself a treat!). The class obviously has more behaviour than the original passing test seemed aware of. Despite this, guess what the original test had as a code coverage metric? 100%



Here's how the test should have been written, code coverage and multiple assertions be damned. Your tests are only complete when you are absolutely certain they cover off on all class behaviour - and not a second

sooner.

```
class GameTest extends PHPUnit_Framework_TestCase
{
    public function testStartingLastScoreIsZero()
    {
        $game = new Game;
        $this->assertEquals(, $game->score);
    }
    public function testStartingTotalScoreIsZero()
    {
        $game = new Game;
        $this->assertEquals(, $game->scoreTotal);
    }
    public function testLastScoreOnlyStoresLastScoreRewarded()
    {
        $game = new Game;
        $game->score(2);
        $game->score(5);
        $this->assertEquals(5, $game->score);
    }
    public function testTotalScoreAccumulatesRewardedScores()
    {
        $game = new Game;
        $game->score(1);
        $game->score(2);
        $this->assertEquals(3, $game->scoreTotal);
    }
}
```

The first test class was very obviously smaller and simpler. Not only is it hampered by multiple confusing assertions, but its simplicity also indicates a lack of good test design - by reusing and ignoring specific test scenario setups (i.e. seeking a fail, before editing code to pass) it's a test suite that passes, but doesn't quite verify everything. Unfortunately, the two go hand in hand in my experience. To make the first test pass, and the second more specific tests fail, use the following version of the Game class.

```
class Game
{
    public $score = ;
    public $scoreTotal = ;
    public function score($score)
    {
        $this->score += $score;
        $this->scoreTotal = $score;
    }
}
```

Here we've simply assumed someone got confused, and mixed up the purpose of the two properties in the score() method, so the += sign has moved in error. Now how many times has that happened to you?



If you

run the original simpler and multiple assertion stuffed test - it will show everything is working as intended (as an aside, this is one example of where Mutation Testing could have picked up a problem which the original test couldn't detect).

```
PHPUnit 3.3.14 by Sebastian Bergmann.
```

```
.  
Time: 0 seconds  
OK (1 test, 4 assertions)
```

The later more detailed, specific tests show a different story:

```
PHPUnit 3.3.14 by Sebastian Bergmann.
```

```
..FF  
Time: 0 seconds  
There were 2 failures:  
1) testScoreOnlyStoresLastScoreRewarded(GameTest)  
Failed asserting that <integer:7> matches expected value <integer:5>.  
D:\projects\tinker\GameTest.php:38  
2) testTotalScoreAccumulatesRewardedScores(GameTest)  
Failed asserting that <integer:2> matches expected value <integer:3>.  
D:\projects\tinker\GameTest.php:46  
FAILURES!  
Tests: 4, Assertions: 4, Failures: 2.
```

Now imagine someone has written hundreds of tests in the manner of my first example. Can you imagine the world of hurt anyone attempting to refactor and maintain the underlying source code is facing? The countless tests they'll need to debug, rewrite, and expand? The tears of frustration? The hair pulling? The talking to your reflection because of a psychotic break?

Don't do that the next time you write unit tests



. Remember - be as specific as possible about what the class should do, and you will quickly realise that you only need one assertion per test. Sure, it means writing additional test code - but at least you're now writing tests that truly work!

Posted by Pádraic Brady in PHP General, PHP Security at 23:54

I think you've hinted at a lot of the problems with unit testing and their perceived effectiveness. Personally, I find the process in PHP to be very tedious compared to C# / Java, just out of a lack of good refactoring tools (extract method / create method / etc...). I also find that in the framework world, I find myself spending a lot of time doing integration testing (whether its JSON to Zend_Controller_Action's or ZF with Doctrine or passing variables to partial views...) and a lot less time coding complex business rules. Unit testing IMO seems to really shine when you're working with the business rules inside a domain.

I found your example clear and helpful. I often struggle with writing the tests in a TDD style in PHP. Writing the tests becomes as an after thought doesn't properly affect the kind of SoC you would see from a class written in a TDD fashion.


When your are testing on a whole, if there are methods you want to hide (make private), but also test, is Unit Testing an option?

When you move closer to production, do you enforce more data hiding?

Thanks for the post,

Jon Anonymous on Feb 12 2009, 23:36

You're absolutely right in a few ways. Unit testing business logic is usually not a bad process, but testing controllers/views unfortunately is only possible using functional (or integration) testing. It's one of my primary arguments why business logic never ever belongs in a Controller or View. It belongs in independent Model classes.

I would suggest that testing private methods is essentially pointless effort. If it's hidden, it shouldn't be directly tested since a hidden property or method is a prime target for refactoring - if refactoring breaks tests it should only be because it broke some behaviour - not the name of a private method 

That said, if it was not originally hidden then tests should exist for it - if you hide more methods or data afterwards, consider segregating the related tests into a different suite of tests that you can mark as deprecated (since they should be allowed to break during any refactoring). Anonymous on Feb 13 2009, 00:38

I'm going to disagree to an extent. While I agree with the idea behind the assertion (no pun intended) that each test method should contain only one assertion, there are some compelling reasons to break this rule.

For instance, when a method returns a particular data structure, and you need to test for the presence of a datum inside it, As an example, let's say a method were expected to return the following structure:

```
array(  
  'foo' => array(  
    'bar' => 'baz',  
  ),  
);
```

and you want to test the value of the \$test['foo']['bar'] key. I'd suggest that you want to bail early if you detect you haven't got an array, or if the array does not contain the 'foo' key, or the 'bar' subkey:

```
$test = $this->object->getData();  
$this->assertTrue(is_array($test), &quot;did not receive array?&quot;);  
$this->assertTrue(array_key_exists('foo', $test), &quot;did not find 'foo' key?&quot;);  
$this->assertTrue(is_array($test['foo']), &quot;'foo' item is not an array?&quot;);  
$this->assertTrue(array_key_exists('bar', $test['foo']), &quot;did not find 'bar' key?&quot;);  
$this->assertContains('b', $test['foo']['bar']);
```

Sure, PHP will throw an error if you simply do this:

```
$test = $this->object->getData();  
$this->assertContains('b', $test['foo']['bar']);
```

PHPUnit will report the error, but I'd suggest that having the additional assertions can make debugging later breakages easier -- you may realize that you're now returning a different data type entirely, or that the internal structure has changed, etc. Without the additional assertions, all you know is that you got an error.

I'd argue that this approach meets the same goals you're evangelizing here -- the assertions are specific, and build on one another. If one fails, because of the order in which they are performed, it can be assumed the others should fail as well.

Granted, you could write a test case for each assertion -- and there may be good reasons to do that (the example above is highly contrived). But there are definitely places where this approach is a more maintainable one.

Regardless, nice post! Anonymous on Feb 13 2009, 01:32

metthew your usecase is obsolete since the invention of assertEquals()



Anonymous on Feb 13 2009, 03:59

Sorry for the poor code highlighting in comments by the way - could never get it works with how escaping work in S9Y



I do admit there is no one absolute rule though (frankly I don't believe there is such a thing in any practice) - I'm sure there are cases where multiple assertions may make sense but they would have to be extraordinary in nature.

In your example, however, why not simply do as @elias suggested? If the test is being setup correctly, the output Array should be identical on every test run (another of those testing practices to be highlighted another day). So just recreate the expected array and compare both for equality.

Granted - your simple array makes that easy and it may not scale to a far more complex Array but I can't remember the last time I had to use multiple assertions to handle a case.

Also, in your case using multiple tests may not be appropriate (so you're right to avoid that option for sure) - if it's one single behaviour, then it should be represented by one single test. Otherwise the failures would just cascade and confuse the issue of what went wrong.

How I measure test grains is through behaviour and by specifying a bit beforehand. If in explaining the array structure, you find yourself describing it as a sequence of behaviours rather than a single fixed format, then it's a fair guess that separate tests might be appropriate. Depends on the actual situation. For example, I could say an Array should be a structure holding URLs as the keys, and tag strings as values at all times. Irrespective of the values it should hold, would that make the structure itself worthy of a test? Would any other test miss that point and see a mixed up array (tags as keys, and URLs as values) as valid simply because it addresses the value, and not the structure? Anonymous on Feb 13 2009, 04:25

I think one of the biggest problems with writing unit tests is that people aren't always sure what a GOOD unit test looks like. Obviously if you've got a bunch of them in your codebase that's a good start but most teams are finding their own way with unit testing. It's only with experience do these improve in quality and structure. Anonymous on Feb 13 2009, 16:49

Thanks for the blog post, excellent points. But I don't understand why you (given your depth of understanding) would say that "testing controllers/views unfortunately is only possible using functional (or integration) testing". There seems to be a widespread myth that this is the case with the Zend Framework, but it isn't. See my blog post at <http://is.gd/jp9S>. Anonymous on Feb 13 2009, 19:13

The example I had was admittedly contrived, but I was definitely trying to get across the idea that sometimes, when testing a discrete behavior, you may need to do multiple assertions against a value to ensure the data structure is as expected.

assertEquals() is nice, but even for the given example it would have failed -- I was checking that a particular value within the structure contained a particular string, not that it equalled a given value (for instance, I may be setting a number of values, and performing the assertions within a loop to ensure that all values propagate to the data structure). But first, I had to test that the structure was as expected -- if it was not, as a developer, I'd like to know as soon as possible so I can figure out what is actually being returned. In that

particular case, multiple assertions make sense.

I'll also note that most cases I've needed to do this have been during refactoring, testing web services, or testing existing code -- when doing TDD or BDD, it's largely unnecessary. Anonymous on Feb 14 2009, 02:56

The problem with this post, and every post and every manual example I have ever seen regarding unit testing is that none of them ever come close to representing a real-world situation.

The test suite for the PEAR Installer takes approximately 35 minutes to run on my old machine, about 10 minutes on my new one.

As always, however, I enjoy the food for thought in your posts, it's a welcome addition to planet php.

If I were to extract every assertion into its own test, the same suite would run in approximately 55 minutes, or twice as slowly. This discourages actually running the tests on changes, for the obvious reason that you never get anything done if every code change requires an hour to wait for the tests to complete.

In addition, the phpt test program does not terminate on failed assertions, which is a far more useful setup in my experience.

Before using phpt, I often found myself performing extraordinary acrobatics just to mung the returned data to a state in which an assertion could be used to test its accuracy, which introduces far more potential for bad tests.


I think your example more closely highlights the problems with brain-dead testing. Testing without thought for code execution, or simply testing without regard for program logic is a guaranteed failure, regardless of how few assertions you plop into a test.

The same is true of brain-dead coding, of course, and there is no formula to fix that either. Anonymous on Feb 14 2009, 08:58

The problem as always, is that creating a real example isn't as easy as it looks. Also, most short posts are always out to illustrate a point in a simple easy to grasp situation so the code is always simplified.

Not sure what other people do, but more unit testing libraries offer features for isolating test runs - you can run specific test cases relevant only to the code you are testing so runtime is minimised. I wish I had finished it in PHP, but you can also write autotest code which constantly monitors which files/class, and which tests for those, is currently being modified to maintain a watch on only the relevant test suites.

Maybe I should take that up in a future topic and finish the PHP version I was tinkering with. Anonymous on Feb 15 2009, 04:18

I suppose I apply fairly narrow definitions. In my mind (which is not to say it's fully right 

) there is nearly always an element of integration testing with actions which generate Views. You're essentially testing the action, the view, and all the classes used in the process. Your flashmessenger example is something that works much better since you added a Setter, and avoided Zend_View, so you could more easily mock objects.

Just as an aside, we both probably know that isn't likely to improve with PHPUnit given its lack of decent Mock Object support. I think that's another reason I turned off calling it unit testing in this respect - most ZF users are using PHPUnit and skipping object doubles is all too common.

There's nothing wrong with that form of testing though (far from it!) but is it really unit testing? On the flipside, a heavy Model is far more likely to see unit testing since it's not constrained by the VC automation and acts like any other group of easy mockable classes. Anonymous on Feb 15 2009, 04:55

I can see that my comment was a bit misleadin. The myth is not that ZF controllers can't be unit tested. The problem is that what's currently being passed off as unit testing is not unit testing, or at least that it unnecessarily depends on too much luggage such as the

ZF FrontController.

But in fact, you can just instantiate the controller and mock the dependencies. It's just harder than it should be. What we really need is dependency injection. Anonymous on Feb 15 2009, 16:04

Sorry, with the clarification that made perfect sense. Agree wholeheartedly! We badly need dependency injection in controllers. A while back I was using a simple DI wrapper so I could intercept and replace objects used by Controllers. Very simple stuff, but we need a formal component for standardisation of practices and I believe there's at least one replacement for the now dead Zend_Di doing the proposal rounds. Anonymous on Feb 15 2009, 20:36

In principle I agree, in the Zend Framework i had to refactor several tests, because when they failed it could be any number of 20 assertions that caused the failure, which is a pain in the ass to debug.

Bu what about objects? matthew had the example of a complex array datastructure, which can be tested using assertEquals() in some cases, but objects for example don't allow this feature.

How would you test with one assertion a derived value object coming from a complex calculation and having more than one member variable?

If i have a domain model, that does intensive calculations and returns me an array of derived value objects, i can't see no other way than writing more than one assertion to test for the correctness of the data structure. Anonymous on Feb 22 2009, 07:05