
Grumpy Programmer's
Minimum Viable
Tests
Mr. Grumpy
Assertions
Fixtures
Doubles
Monkey Patching
Written By:
Chris Hartjes

Minimum Viable Tests

Chris Hartjes

This book is for sale at http://leanpub.com/minimumviabletests

This version was published on 2017-10-11

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Testing Basics

What Is a Test?

There are several ways we can answer this question, to be honest. When I was first learning to write tests, I didn't really think much about it. I wrote some code, used the test runner, and looked to see if I got the expected results. If we back up a bit, I think we can get a much more useful answer.

From a very high level, a test is code you have written which confirms the code you are trying to test is behaving as expected. A very simple concept. Everything else used to describe what a test is, builds on this foundation.

If you look at it another way, there's a question you should ask yourself every time you write code: How can I prove this code works the way I am expecting it to?

Many people test using the old-school "code-save-refresh" method of testing. Some folks are lucky enough to use languages which allow you to try things out using a REPL[1] (Read-Eval-Print Loop). The problem with using this approach is you end up throwing away your tests. Using tools like PHPUnit can help you keep those tests around, and get the computer to run them for you. Computers are great at endlessly running tests for you. Far better to spend your mental energy solving problems, than manually running tests.

PHPUnit, being part of the `xUnit`[2] family of testing tools uses the idea of an assertion as a central concept.

> In computer programming, an assertion is a statement a predicate (Boolean-valued function, a true-false expression) is expected to always be true at that point in the code. If an assertion evaluates to false at run time, an assertion failure results, which typically causes the program to crash, or to throw an assertion exception.

With a few exceptions, tests you write will look like this:

- do some test setup
- run the code you're trying to test
- assert that you have gotten an expected result

I can't think of any better way to help you learn the basics of writing a test than actually writing some code using tests to guide us.

[1] https://en.wikipedia.org/wiki/Read-eval-print_loop
[2] https://en.wikipedia.org/wiki/XUnit

Testing FizzBuzz

For those who are not familiar with it, the FizzBuzz algorithm is a common 'programming interview' question. While I'm not generally a fan of asking people to solve problems during interviews, I feel like FizzBuzz is really good to use for testing examples. It covers iterating over collections, loops, and conditional statements.

Here's how FizzBuzz is supposed to work:

- you have a list of positive integers
- you iterate through the list, examining each item
- if the integer is divisible by 3, you return 'Fizz'
- if the integer is divisible by 5, you return 'Buzz'
- if the integer is divisible by both 3 and 5, you return 'FizzBuzz'
- otherwise just return the integer

Let's build this together using Test-Driven Development. I cannot recommend enough you following along with me and actually typing in all the code and watching the output of PHPUnit. Over the years I have found that to get good at testing requires a lot of actually writing tests and learning what works and doesn't. Every code base is different, and theory only takes you so far.

With that out of the way, let's start with writing a test which assumes the code is already working.

Handling Non-Integers

Assuming you have PHPUnit installed already (refer back to the chapter on PHPUnit basics for more details), let's write a test. How should we respond if the collection we're sent doesn't contain integers? Maybe throw an exception? This seems better than providing mixed result types.

There are two ways we can test exceptions. We can either use the `@expectedException` annotation, or we can use the built-in `setExpectedException()` method. Let's try the annotation way first.

```
1  require './vendor/autoload.php';
2
3  class FizzBuzzTest extends \PHPUnit_Framework_TestCase
4  {
5      /**
6       * @test
7       * @expectedException Exception
8       */
9      public function handlesNonIntegersCorrectly()
```

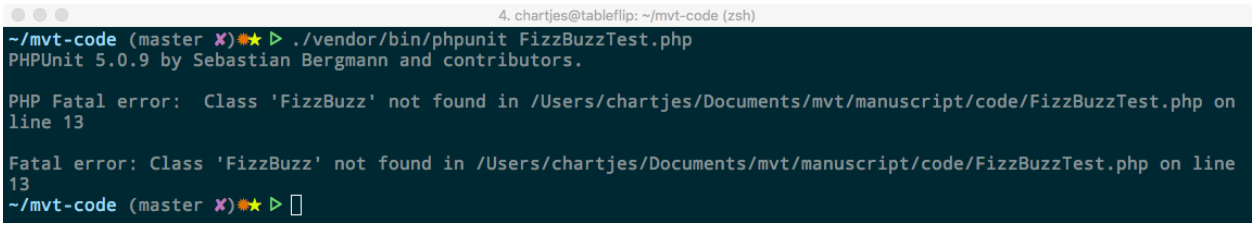
```
10      {
11          $fizz_buzz = new FizzBuzz();
12          $collection = ['A', 1, 'B', 'C', 1.234];
13          $fizz_buzz->process($collection);
14      }
15  }
```

Save this file as `FizzBuzzTest.php`.

Now run the test:

```
./vendor/bin/phpunit FizzBuzzTest.php
```

It should fail because we haven't written any of our FizzBuzz code yet.



Test Fails Because No FizzBuzz Exists

I know what some of you are thinking–how can you write a test before you've actually written the code? If you've never done things the TDD way, it is definitely intimidating and likely very different from how you typically do things.

In this case, I am using TDD to design the interfaces we're going to use. Often, I change what the final product looks like, but the tests guide me towards a working implementation of code. Okay, so what is our process method going to look like?

How about we take the collection being sent in and filter out anything which isn't an integer. Then we can compare this filtered list to the original, throwing an exception if they don't match.

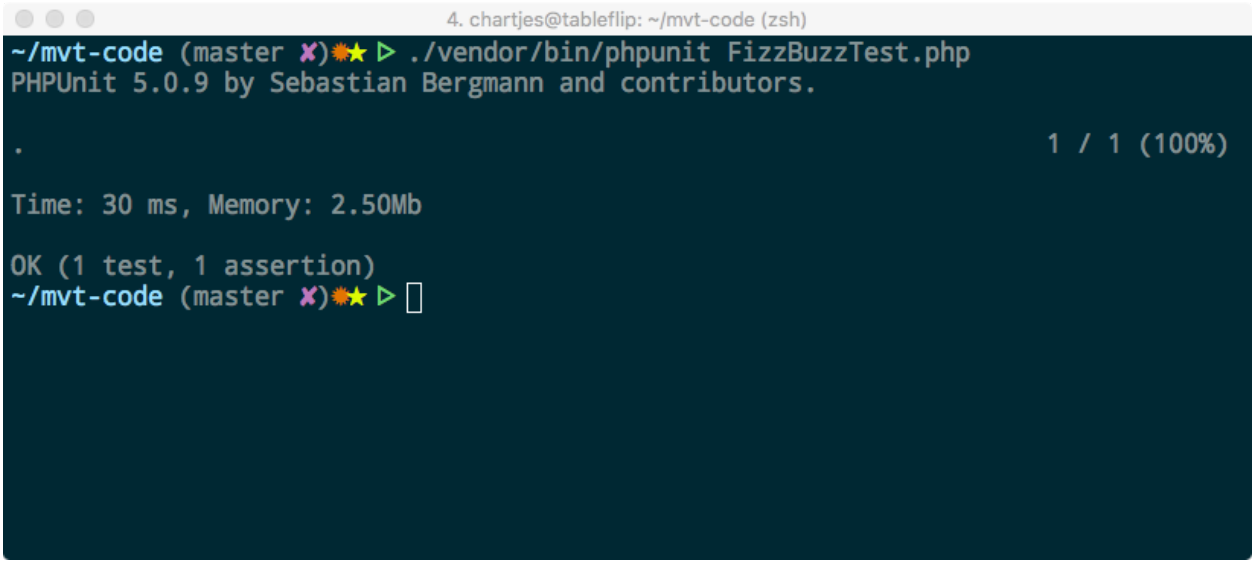
```
1   class FizzBuzz
2   {
3       public function process($collection)
4       {
5           $filtered_collection = array_filter($collection, function($item) {
6               if (is_int($item)) {
7                   return true ;
8               }
9
10              return false;
11          });
12
```

```
13          if ($filtered_collection !== $collection) {
14              throw new \Exception("Did not receive collection of integers");
15          }
16
17          return $filtered_collection;
18      }
19  }
```

Now let's add a test to look for the exception.

```
1   require './vendor/autoload.php';
2   require './FizzBuzz.php';
3
4   class FizzBuzzTest extends \PHPUnit_Framework_TestCase
5   {
6       /**
7        * @test
8        * @expectedException \Exception
9        */
10      public function handlesNonIntegersCorrectly()
11      {
12          $fizz_buzz = new FizzBuzz();
13          $collection = ['A', 1, 'B', 'C', 1.234];
14          $fizz_buzz->process($collection);
15      }
16  }
```

Run the test the same way as before and it should pass.



Exception is correctly thrown

If we wanted to use `setExpectedException()`, it's a simple change: you remove the annotation, then specify the type of exception and the message the exception gives.

```
1  require './vendor/autoload.php';
2  require './FizzBuzz.php';
3
4  class FizzBuzzTest extends \PHPUnit_Framework_TestCase
5  {
6      /**
7       * @test
8       */
9      public function handlesNonIntegersCorrectly()
10     {
11         $this->setExpectedException(
12             "Exception",
13             "Did not receive collection of integers"
14         );
15         $fizz_buzz = new FizzBuzz();
16         $collection = ['A', 1, 'B', 'C', 1.234];
17         $fizz_buzz->process($collection);
18     }
19 }
```

Run the test again and you should see it has passed.

Testing for Fizz

Okay, now time to create a test verifying we've handled the Fizz case properly. Add the following test method to your class:

```
    /**
     * @test
     */
    public function handlesFizzCorrectly()
    {
        $fizz_buzz = new FizzBuzz();
        $collection = [1, 2, 3, 4, 6, 7, 9];
        $expected_result = [1, 2, 'Fizz', 4, 'Fizz', 7, 'Fizz'];
        $result = $fizz_buzz->process($collection);
        $this->assertEquals(
            $expected_result,
            $result,
            "FizzBuzz did correctly find Fizz values"
        );
    }
```

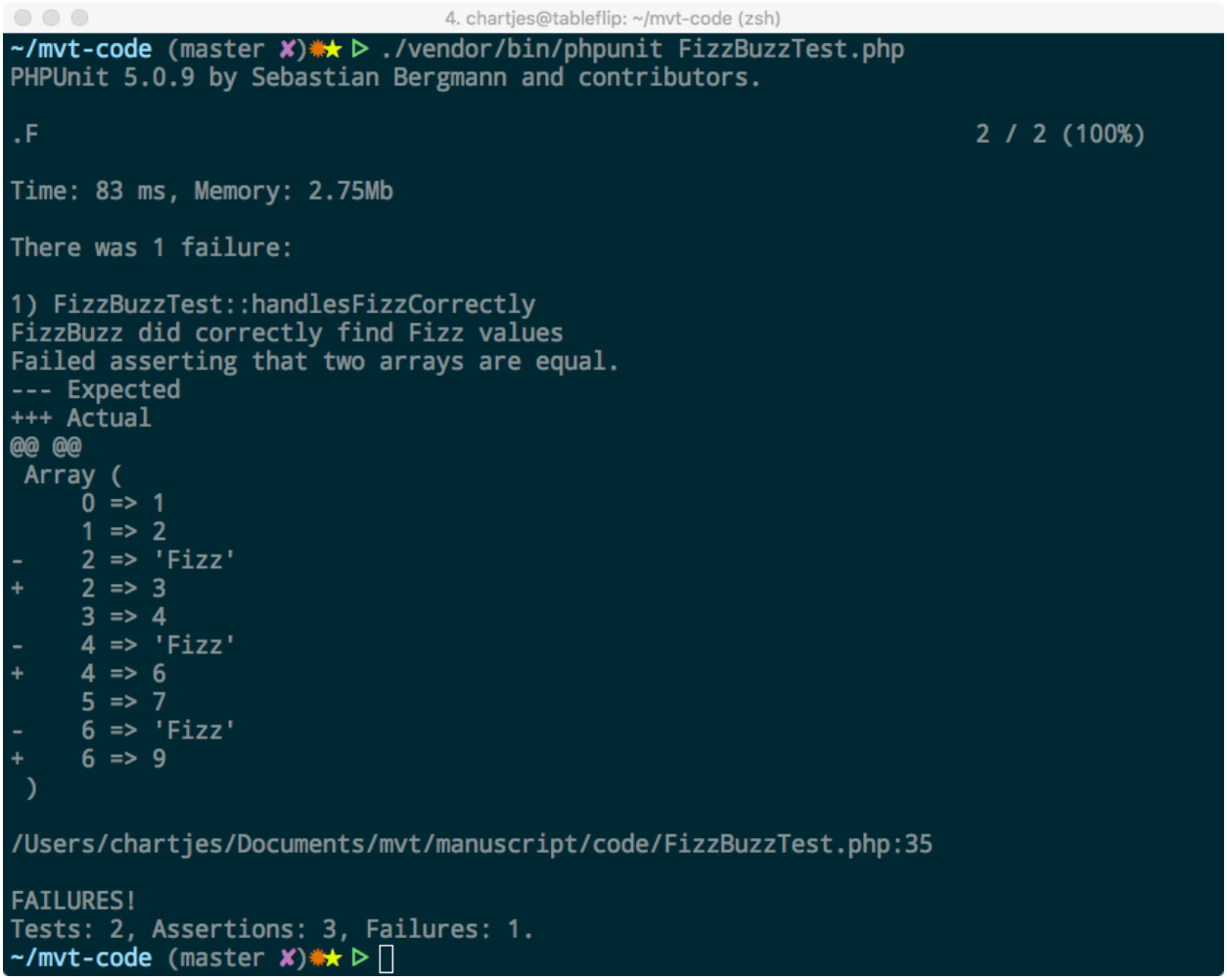
In this test, we are using an assertion to verify our results. In plain English we are asking PHPUnit "We want the result coming from our code to match our expected result".

There are a very large number of assertions built into PHPUnit (39 last I counted) for you to use. I have found in most cases you will be using a very small selection of them:

- `$this->assertTrue(...)`
- `$this->assertFalse(...)`
- `$this->assertEquals(...)`

Each assertion has its own required number of parameters, but every single one takes an optional final parameter which is a message you want to be displayed by the test runner if your test happens to fail. Having human-readable messages when things go wrong is always good.

Run your test again and it should fail; there is a lot more information here for you to see.



Fizz not correctly found

Out of the box, PHPUnit provides us with some good information to go on. Let's break it down a little bit.

First, it tells us we ran two out of two tests and one of them failed. That's what the 'F' means.

Next, it tells us how long the test took to run and how much memory was used. This is useful when you are following the path of "make it work, then make it work better."

Finally, it highlights what test actually failed and tries to show us the differences between what we expected to happen and what really happened. If you've done some work with version control systems such as Git, the way PHPUnit shows you is familiar.

You can clearly see it is showing us in the places where we expected 'Fizz' we got the number back. Let's fix it.

```
require './vendor/autoload.php';
require './FizzBuzz.php';

class FizzBuzzTest extends \PHPUnit_Framework_TestCase
{

    /**
     * @test
     */
    public function handlesNonIntegersCorrectly()
    {
        $this->setExpectedException(
            "Exception",
            "Did not receive collection of integers"
        );
        $fizzBuzz = new FizzBuzz();
        $collection = ['A', 1, 'B', 'C', 1.234];
        $fizzBuzz->process($collection);
    }

    /**
     * @test
     */
    public function handlesFizzCorrectly()
    {
        $collection = [1, 2, 3, 4, 6, 7, 9];
        $expected_result = [1, 2, 'Fizz', 4, 'Fizz', 7, 'Fizz'];
        $fizz_buzz = new FizzBuzz();
        $result = $fizz_buzz->process($collection);
        $this->assertEquals(
            $expected_result,
            $result,
            "FizzBuzz did not correctly find Fizz values"
        );
    }
}
```

When you run the test, it should be failing in the same place.

Now, let's add code to our `FizzBuzz` class to correctly handle turning integers which are a multiple of 3 into `Fizz`

Let's use an array map to iterate through the collection. Add the following code to replace the statement returning our filtered collection.

```
        $result = array_map(function($item) {
            if ($item % 3 === 0) {
                return 'Fizz';
            }

            return $item;
        }, $filtered_collection);

        return $result;
```

Functional programming is the future! All kidding aside, I do like to use the built-in `array_*` methods when I have to manipulate a collection in an array. It looks nicer than a conditional statement inside a `foreach` loop, which should make Rafael Dohms proud[3].

Run the test and it should pass:

Fizz correctly found

Make Buzz Work

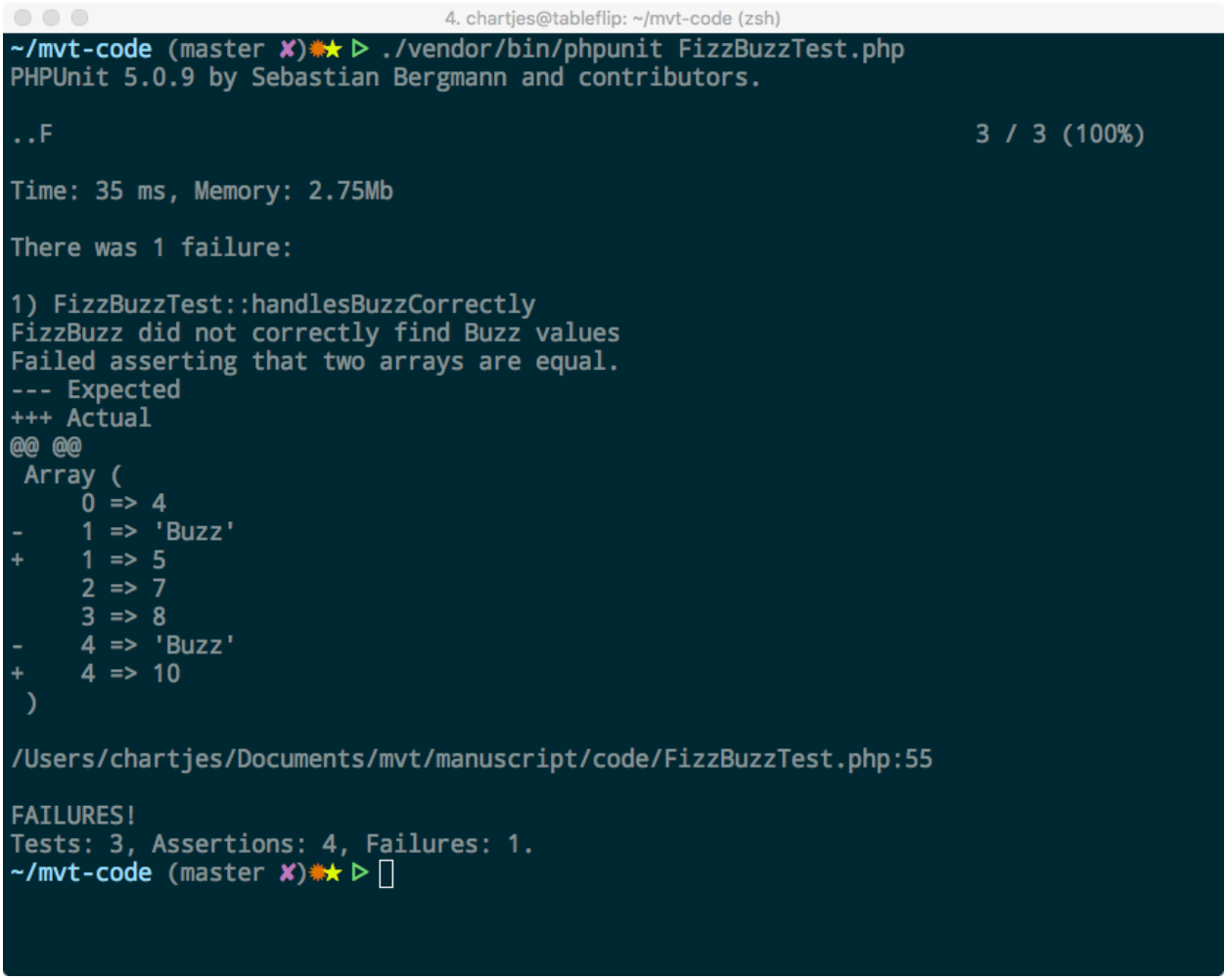
Feeling great about having some tests pass? Let's add one which checks to make sure we handle the `Buzz` case correctly.

Add this test after the one for `Fizz`:

[3] http://www.slideshare.net/rdohms/bettercode-phpbenelux212alternate

```
    /**
     * @test
     */
    public function handlesBuzzCorrectly()
    {
        $collection = [4, 5, 7, 8, 10];
        $expected_result = [4, 'Buzz', 7, 8, 'Buzz'];
        $fizz_buzz = new FizzBuzz();
        $result = $fizz_buzz->process($collection);
        $this->assertEquals(
            $expected_result,
            $result,
            "FizzBuzz did not correctly find Buzz values"
        );
    }
```

When you run the test, you should see it fail similarly to this:



Buzz fails

We could expand our anonymous function inside the `array_map` to look for the `Buzz` case.

```
        $result = array_map(function($item) {
            if ($item % 3 === 0) {
                return 'Fizz';
            }

            if ($item % 5 === 0) {
                return 'Buzz';
            }

            return $item;
        }, $filtered_collection);
```

That should pass.

Buzz passes

Check That FizzBuzz is Correctly Detected

We have the final case we need to worry about. Integers which are multiples of three and five need to return FizzBuzz. First, as in this entire chapter, we write a test assuming it all works.

Add the following test method after our `Buzz` one:

```
1     /**
2      * @test
3      */
4     public function handlesFizzAndBuzzCorrectly()
5     {
6         $collection = [14, 15, 30, 31, 45];
7         $expected_result = [14, 'FizzBuzz', 'FizzBuzz', 31, 'FizzBuzz'];
8         $fizz_buzz = new FizzBuzz();
9         $result = $fizz_buzz->process($collection);
10        $this->assertEquals(
11            $expected_result,
12            $result,
13            "FizzBuzz did not correctly find FizzBuzz values"
14        );
15    }
```

Make sure the test fails.

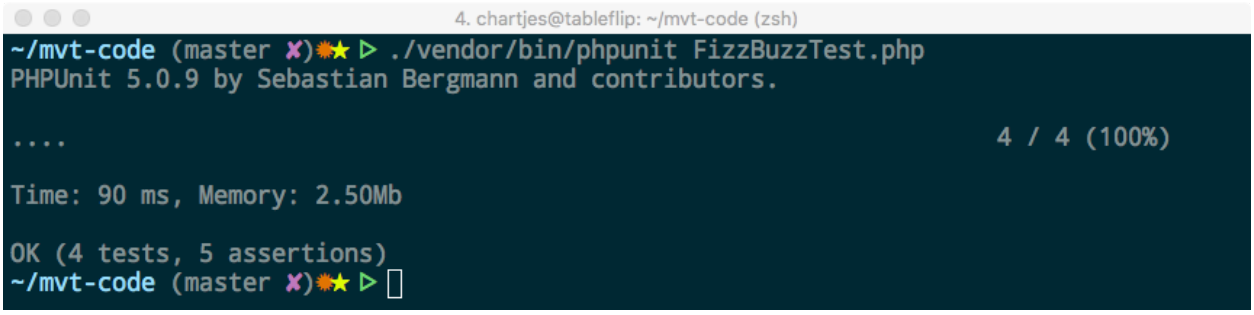
FizzBuzz failes

Now, isn't this output interesting? Given our collection, it did correctly figure out 15, 30, and 45 are multiples of Fizz. I think a quick change will make it work. Refactor the code inside our anonymous function which is part of our map to look like this:

```
1         $result = array_map(function($item) {
2             $response = '';
3
4             if ($item % 3 === 0) {
5                 $response .= 'Fizz';
6             }
7
8             if ($item % 5 === 0) {
9                 $response .= 'Buzz';
10            }
11
```

```
12              if ($response === '') {
13                  $response = $item;
14              }
15
16              return $response;
17          }, $filtered_collection);
```

The test passes.



FizzBuzz failes

So now we have a completely working implementation `FizzBuzz` and we built it using TDD. Remember, we do the bare minimum to make our tests pass. But we now have a slightly different problem–there is a lot of repetition in our tests. Let's get rid of some of it.

Data Providers to Reduce Repetition

PHPUnit offers a very handy feature allowing you to create a test which accepts multiple sets of data for testing purposes. It's perfect for those situations when you find yourself writing a series of tests only differing in minor ways.

Data providers are public functions which return arrays of arrays. To make one for our `FizzBuzz` transform tests, add this class function just before our first test method.

```
1       public function collectionDataProvider()
2       {
3           return [
4               [
5                   [1, 2, 3, 4, 6, 7, 9],
6                   [1, 2, 'Fizz', 4, 'Fizz', 7, 'Fizz']
7               ],
8               [
9                   [4, 5, 7, 8, 10],
10                  [4, 'Buzz', 7, 8, 'Buzz']
```

```
            ],
            [
                [14, 15, 30, 31, 45],
                [14,'FizzBuzz','FizzBuzz',31,'FizzBuzz']
            ]
        ];
    }
```

We have an array of arrays inside the data provider method. Each 'root' array contains one array full of numbers we want transformed and one array containing what the transformed results should look like.

We can just add more items to the data provider if we want even more assurance our transforms are correct. It also saves us from duplicating code which sets what our test collections and expected results are.

Now, we have to create a new, more generic test using the data provider. Add this code to replace `handlesFizzCorrectly()`, `handlesBuzzCorrectly()` and `handlesFizzAndBuzzCorrectly()` respectively. There are a few new things going on here, and I will explain them.

```
    /**
     * @test
     * @dataProvider collectionDataProvider
     */
    public function transformsCollectionsCorrectly($collection, $expected)
    {
        $fizz_buzz = new FizzBuzz();
        $result = $fizz_buzz->process($collection);
        $this->assertEquals(
            $expected,
            $result,
            "FizzBuzz did not correctly transform values"
        );
    }
```

First, you'll see we added an annotation to the doc block to let PHPUnit know which data provider we want. The annotation is case sensitive (I remember spending 30 minutes debugging a problem caused by this), and the associated value is simply the name we gave our data provider method.

Next, we add two parameters to the test method matching what the data provider will be giving us as inputs.

Run the test and you should see something similar to the following:

Tests with a data provider pass

I think we've pretty much covered the basics of writing a test. It's a very straight-forward process and definitely not something you should be intimidated by. After all, if you can write tests you can write code.

Things to Look out For

One of the advantages of writing code using TDD is you are forced to determine dependencies up front. Will I need a database connection? How will I get a value from global registry into this method? Often these are important decisions, but most of the time they are not.

Once you do have a large number of tests for your application, you can go back and correct some earlier decisions which have turned out to not quite be right. Make a change, run the tests, see what broke, and cheer when it all works!

As with our initial steps through implementing `FizzBuzz`, don't be worried to go back and change things as you use tests to guide your design. Use the tests to explore different ways of solving the same problem.

Pay close attention to what your tests are telling you about the code you are writing. Watch for duplication in your tests and consider if you can use a data provider and a refactored, more generic test case to cover way more potential outputs.

We've covered the first steps in your path to writing Minimum Viable Tests. Next, we're going to spend some time talking about a powerful tool which has generated some very strong opinions regarding its use.