## Principles of unit testing

There are two important reasons to use unit testing. Firstly, software developers should aspire to be professional and a full unit test suite shows a professional approach. Secondly, software written using unit tests is better software. Unit tests allow re-factoring and exercise the API. This in turn leads to decoupling and improved readability. They also provide living documentation of the code.

Developers using test doubles in the process of Test Drive Development will usually take one of two styles. The first style is known as Classical TDD and favours real objects, using doubles only where it is awkward to use the real thing. The second is known as Mockist TDD style and uses doubles for any object with “interesting” behaviour.

In the writer's opinion, test doubles have two major advantages. Firstly, they are faster than real objects and don't have dependencies. Secondly, they allow unit tests to be focussed on very particular parts of the code and can communicate the intent of the developer.

Examples of “uninteresting” objects are value objects or those with known behaviour such as iterators.

## Types of test double

The simplest type of test double is a dummy. These are used where a class only needs an object as a constructor argument. They therefore need only to extend a class or implement an interface; all methods returning null.

The next type of test double is a stub. These are probably the most commonly found and are used where classes retrieve data from other objects. The stub object is loaded with canned responses, which it returns as required.

A further type of test double is a spy and allows another way to test. Instead of checking the output from methods, spies are used to verify inputs to methods. Instead of returning data, spies return either null, or a reference to themselves, while recording how many times each method is called and with which arguments.

The final test double that we will look at is the mock. This has the characteristics of both the stub and the spy in that it both returns fully defined responses to method calls and also tracks the number of times each method is called.

Two types of test double will not be part of this analysis. Fakes are used instead of objects that are slow or rely on resources not available in the test environment. Temporary Test Stubs are used in place of code that has yet to be written.

## Dummies

Using dummies allows us to compare the basic creation syntax for test doubles.

PHPUnit dummies are created as follows:

*$dummy = $this->getMock('Class\_Or\_Interface\_Name');*

Prophecy dummies are created as follows:

*$dummy = $this->prophesize('Class\_Or\_Interface\_Name')->reveal();*

We note that the class or interface does not have to exist. Where a class or interface definition has been loaded, then method calls can be performed on the dummy, returning null by default.

We also note that object initialization differs between our two frameworks. Where a constructor method exists for a class, then PHPUnit will execute it, whereas Prophecy will ignore it.

Verdict: Dummies are easier to create using PHPUnit as they only need a single method call.

## Stubs

Using stubs allows to compare how behaviour is assigned to methods.

PHPUnit methods are configured as follows:

*$stub->method('getId')->willReturn(1);*

Prophecy methods are configured as follows:

*$stub->getId()->willReturn(1);*

The “arbitrary method call” used by Prophecy is more elegant, particularly where arguments are passed in method calls. It also allows the use of code-completion in the IDE and automatic renaming of unit test method calls where class are re-factored.

We note however two instances where PHPUnit might be preferred.

Firstly, PHPUnit does not impose restrictions on which methods are defined. In Prophecy, once a single method is defined, all methods used in the unit test must be defined. This restriction can make Prophecy less suited to modelling complex objects with many methods.

Secondly, Prophecy does not allow the definition of methods not listed in the class signature. An example of this is where \_\_call() is used to return responses to inaccessible methods. Behaviour such as this can only be reproduced using PHPUnit.

Verdict: Stubs are easier to create using Prophecy, particularly where method outputs depend on input arguments.

## Spies

Using spies changes our testing approach from verifying output to checking which methods are called, how many times they are called and with which arguments.

Spies are configured in PHPUnit as follows:

*$spy->expects($this->once())->method('getName')->with($this->equalTo('first'));*

and in Prophecy as follows:

*$spy->getName('first')->shouldHaveBeenCalled();*

Argument checks can be either specific or general. Examples of specific checks could be confirming the identity of an object or the exact match of a string. General checks, on the other hand, might check only certain properties of an object or a string using a regular expression.

Verdict: Basic spies are very easy to implement in Prophecy, however describing complex expectations feels more natural in PHPUnit.

A possible explanation for this may be found in this post by Konstantin Kudryashov, the creator of Prophecy:

<http://everzet.com/post/72910908762/conceptual-difference-between-mockery-and-prophecy>

For Mockery, read PHPUnit.

He describes in this post how “Prophecy puts messaging (aka **how** objects communicate) before structure (aka **when** objects communicate).”