


Mutation Testing Brain Dump

A few thoughts I'm pulling together regarding Mutation Testing for [PHPSpec](#) (that's the Behaviour-Driven Development framework I'm working on).

Mutation Testing is like testing, for tests. The idea is actually quite simple. Mutation testing adds small changes to your source code on the assumption that changing something, will most likely break it, which in turn means at least one test/spec should fail. If no tests fail, then it means our tests were incapable of detecting the deliberately introduced problem (i.e. we need to add more tests/specs to detect similar mutations). The reason why it's useful is that it gets around the problem of being over reliant on code coverage - just because some code executes when running tests, doesn't prove the tests will detect problems with it!

In PHP I've only ever seen one real implementation of Mutation Testing, which seems to be coming along for PHPUnit. Should be a lot of fun when that's finished and someone applies it to their PHPUnit test suites 


PHPUnit mutations rely on a few PECL extensions like `runkit` and `Parse_Tree` but I really wanted to try something even dumber so the `runkit/parse_tree` dependencies are not required. My way is not going to be relying on changing class definitions mid-process.

The main culprit in all of this will be the PHP Tokenizer. If you're not familiar with it, the [Tokenizer functions](#) let you access the Tokenizer in the Zend Engine so you can analyse PHP source code without fiddling about with byte-by-byte or PCRE based parsers you'd have to write otherwise. For example:

```
$source = '<?php $op=1; echo $op;';  
$tokens = token_get_all($source);  
var_dump($tokens);
```

This sample will output tokens as:


```
array(10) { [0]=> array(3) { [0]=> int(367) [1]=> string(6) " int(1) } [1]=>  
array(3) { [0]=> int(309) [1]=> string(3) "$op" [2]=> int(1) } [2]=> string(1)  
"=" [3]=> array(3) { [0]=> int(305) [1]=> string(1) "1" [2]=> int(1) } [4]=>  
string(1) ";" [5]=> array(3) { [0]=> int(370) [1]=> string(1) " " [2]=> int(1) }  
[6]=> array(3) { [0]=> int(316) [1]=> string(4) "echo" [2]=> int(1) } [7]=>  
array(3) { [0]=> int(370) [1]=> string(1) " " [2]=> int(1) } [8]=> array(3) {  
[0]=> int(309) [1]=> string(3) "$op" [2]=> int(1) } [9]=> string(1) ";" }
```

This looks like double dutch, but it's actually quite simple. Each of the 10 array elements is either a string or a sub-array. Each sub-array represents a token, and has three elements - the token number, the content, and the line number it was found on. The token number is equivalent to a token constant like `T_WHITESPACE` (the value 370) - handy if you recognise the `T_*` format from all your error messages 


Back to the topic, you can use this token analysis to locate places to apply mutations. Maybe I want to replace the `$op=1` piece of code with `$op=2` (a test relying on value being 1 would surely fail then!). This is easy using tokens - just search for all tokens of type `T_LNUMBER` (an integer) and replace them with some random value. I could replace variable names by searching for `T_VARIABLE` tokens, and so on.

Once a mutation (just one before rerun tests or specs to see if the mutation correctly generates a failure) is applied, you can reconstruct the source code string using a simple function like, and use the mutated code to replace the original clean stuff.

```
function getSource(array $tokens) {  
    $str = "";  
    foreach ($tokens as $token) {  
        if (is_string($token)) {  
            $str .= $token;  
        } else {  
            $str .= $token[1];  
        }  
    }  
    return $str;  
}
```

The way I'm thinking of playing with this mutation processing style is to do away with any external dependencies (no PECL to worry about phpizing) and just employ a working copy of the source code tree you want to apply mutation testing to - yep, just copy the code and tests somewhere where it's safe to generate mutations and run tests without effecting the original source. Pretty simple - I'll even let you define where to copy to 

Then it's a simple matter of creating a list of mutations to apply, apply them one by one (restore the original files after each mutation run) and call the local command line test/spec runner after each incremental mutation. As you can guess, mutation testing takes longer to run than normal testing (new test run per mutation in a new PHP process), but it will cease the second 1 failure/error/exception is detected. Also it would still be far far faster than compiled languages (which can be horribly slow).

The other benefit of this is that the mutation system would be independent of the testing or BDD framework. You could conceivably use it for PHPT or SimpleTest, as well as PHPSpec (conceivably when framework adapters are implemented, that is 


). Even PHPUnit, though their built in mutation testing once finished will likely integrate better and run faster (won't require new PHP processes).

When I get around to playing with this in depth I'll post a little more with some actual code. Maybe to the usual suspects on the Devnetwork Forums to critique.

Posted by Pádraic Brady in PHP General, PHP Security at 01:30


Saturday, November 24. 2007


Ruby Testing Tools Missing From PHP

Warning: This entry uses the word "Ruby" in a context that explains how in "one aspect" it's "better than PHP". Try not to cry... 

. If you can't live with Ruby having a few tools PHP doesn't quite have yet, you shouldn't read any further.

To throw out a few keywords so some cross-language developers can sagely nod their heads: Autotest, Mutation Testing, Mocha, Rspec, Heckle...what's that other one? Hpricot.

Anyways, here's the pitch. I've been using Ruby for a year now and my pet peeves with PHP started getting a bit too much to comfortably endure. Luckily, for all my PHP readers who think I'm brilliant (or at least good enough to fake it 

Shhh...), I don't like developing web applications with Ruby, or that thing Rails. Whatever it is. It's a framework, right? So I'm on a splurge of writing PHP tools for the same things in cahoots with folk like Travis Swicegood (edit: as far as I know Travis is totally innocent of being a Ruby-Lover - I'm the one you want to flame for being "rubyesque" or whatever 

).

But Ruby has a few non language-specific testing ideas I love to bits. Behaviour-Driven Development is one of them (sorting that out with PHPSpec, and PHPT is showing some lighter weight advances which are worth noting). Another is ZenTest's Autotest.

Autotest rocks. Yabba dabba doo style.

If there was ever something I'd use Sara Goleman's introduction to the mystical art of PHP Extension writing for (great book), it's to interface to every notification daemon/app I have running on my OS's. You see, using any testing framework I open my IDE, edit some classes and their respective tests, flip to the command line, re-run my tests, wait a while, and get a result. It's all nice - and it sinks time once the tests reach the level of needing anything beyond a split second to run.

How much nicer then to have a tool checking all my project files for modifications, running only the subset of tests effected, and reporting results via a popup notice with green/red feedback colours - all in a second or two? What? No "phpspec -r" or "pear run-tests -rq"? No running all tests, or selecting a specific group/directory? Absolutely not. I don't even leave the IDE window... It's autotest afterall - human interaction not required.

Then there's Ruby's Heckle for Mutation Testing. Actually PHPUnit has something on this already (there's another Sara Goleman link to PECL's runkit which it uses). Not in the release branch yet, but hopefully it'll get in there at some point. For PHPSpec specifically I was thinking about planning mutation testing for a few months time, but couldn't really think of a good solution. PHPUnit's wiki has some posts from the Mutation Testing author (part of the Google Summer of Code) and some of the dependencies from PECL seem to carry some small problems making it more difficult than it should be.

I noted earlier today it may pay to remember PHP runs from the command line - so you can use multiple processes, and access multiple copies of class definitions. You could feasibly create 100 classes called Logger, mutate them on the fly on a per process basis, and throw them at tests all day long. You'd only need 100 sequential PHP processes (one per mutation - nothing odd there for a PHPT user) tied together to a Heckle-like PHP clone. The only concern would be using it only on code which is clever enough not to redeclare existing classes. Sounds like a simpler route compared to using PECL runkit et al. in the one process.

Just to finish up there's Ruby's Hpricot. Hpricot is plain annoying, though PHP isn't too far behind it. For many things DomDocument just works great. Hpricot however just feels better - how can you resist using a lean API like:

```
doc = Hpricot("That's my <b>spoon</b>, Tyler.")  
doc.at("b").swap("<i>fork</i>")  
doc.to_html  
# => "That's my <i>fork</i>, Tyler."
```

As an HTML parser and general toolkit it's addictive. It even parses XML




. You could probably just write a

clone right on top of DomDocument and who'd care?

Posted by Pádraic Brady in PHP General, PHP Security at 00:55

Wednesday, November 21. 2007

PHPUnit: Independent Mock Object/Stub Framework

Sometimes I think my readers must get terribly bored with my one track entries - I seem to roll through phases: a few weeks of OpenID, then a few of PHPSpec, then a detour for several months on X 

. That's what you get for reading an open source developer's weblog - we can only really work on one thing at a time so there's a lack of variety.

Anyway, after that minor detour, this one's a brief notice that we've commenced work (as of a while ago really) on PHPUnit.


If you remember, I had one of those long winded blog posts about Mock Objects and Stubs in SimpleTest/PHPUnit and PHPT. Now that I've really launched into PHPSpec development, I'm finding a few spots here and there where my attention levels drop and I need a diversion. Which means PHPUnit is seeing activity - there's a whole branch written over a few days to get an API discussion going between developers.

The purpose of PHPUnit is to write an independent Mock Object/Stub framework - something you can use in any scenario or xDD approach - from a PHPUnit Unit Test, to a PHPSpec Spec, to a PHPT test file. SimpleTest isn't left out either - PHPUnit is self contained so it should travel well.

Once we get over the Mock vs Stub approach question to such a framework (Are Mocks complicated Stubs, or Stubs simplified Mocks?) we'll have a firm design and an API to develop with. I'm currently favouring an API in a simple readable form, e.g.

```
// setup
$mockPerson = PHPUnit::mock('Person');
$mockPerson->shouldReceive('hasName')->withNoArgs()->once()->andReturn(false);
// implemented use case
$mockPerson->hasName(); //FALSE
// verification of Mock expectations
$mockPerson->verify(); //TRUE
```

Not sure if it will stay that way, have some alternate API appended (less verbose), or end up as something different. I'll let you know.

A good feeling about this small project is that we have peers checking in. Sebastian Bergmann for PHPUnit, me for PHPSpec, and Travis Swicegood for PHPT and maybe SimpleTest. So at least it's not just me rumbling around enforcing my divine will 

If you want to look around the experiemental branch code, we're hosted over at <http://code.google.com/p/phpmock/>.

Posted by Pádraic Brady in PHP General, PHP Security at 01:19

Sunday, November 18, 2007

PHP Devnetwork Forums: Now cooler than ever!

The coolest PHP forum ever spawned, [PHP Developers Network Forum](#) has had Christmas early. With the move to a new server following some troublesome hosting problems recently which led to issues accessing the forum for my daily (well, hourly at times) dose of everything an excellent PHP community offers, it's back.

To quote [Kieran Huggins](#) :

Wicked! Blazing fast and twice as sexy.

A hearty thanks to all involved. Here's how damn addictive it is:

Our users have posted a total of 420710 articles

We have 34110 registered users

Add to that how many libraries members have written - like Swiftmailer and HTMLPurifier and PHPSpec and SHA256 Native PHP and AP3 and ... well you get the point. Cool developers, decent folk, countless PHP questions from the mundane to the theoretical heights answered every day. Even I barely get a post in edgeways!

Posted by Pádraic Brady in PHP General, PHP Security at 10:12

PHPSpec Reporting Gets A Needed Boost

PHPSpec is closing in on its first stable release, so the time had finally come to spruce up its output! No more the simple reporting of failed specs - now you get a few more details in a readable format, exceptions and errors even come with traces. In addition, I've implemented specdoc output as an option (using "-s") so you can get a list of specs in their plain text form.

I've pasted in a copy of the output from a real PHPSpec run and added a few issues to show how they are output. I've been writing some experimental source code for a possible Mock Object framework in PHP (see last blog entry), so I used PHPSpec to apply Behaviour-Driven Development in its development. You can read the actual spec files over at <http://phpmock.googlecode.com/svn/branches/padraic/specs/>.

PHPSpec development snapshots have been updated on <http://dev.phpspec.org>, which also hosts the manual being written (HTML).

```
$ phpspec -rs
```

```
EE.....P..
```

```
Finished in 0.334152936935 seconds
```

```
p h p mock method expectations
-should expect each method exactly once unless otherwise specified (ERROR)
-should throw default exception if method call count is unexpected (EXCEPTION)
-should expect number of methods in times term
-should return self from times term invocation
-should set return value in terms and default to returning value for all calls
-should return values in order of setting but return last value remaining on
other calls
-should return self from return term invocation
-should set expected args using with term
-should return self from with term invocation
-should allow no calls for zeroormoretimes term
-should allow one call with once term
-should throw exception if less than once using once tern
-should throw exception if greater than once using once term
-should return self instance from once term
-should allow two calls with twice term
-should throw exception if less than twice using twice term
-should throw exception if greater than twice using twice term
-should return self instance from twice term
-should allow no calls with never term
-should throw exception if any call after never term
-should return self instance from never term
-should allow any calls for zeroormoretimes term
-should allow zero calls for zeroormoretimes term
-should return self instance from zeroormoretimes term
-should set times minimum with once using atleast term
-should set times minimum with times using atleast term
```

- should throw exception if less than minimum using atleast term
- should return self instance from atleast term
- should set times maximum with once using atmost term
- should set times maximum with times using atmost term
- should throw exception if more than twice using twice term
- should return self instance from atmost term
- should allow any args using withanyargs term
- should return self instance from withanyargs term
- should disallow args using withnoargs term
- should return self instance from withnoargs term
- should obey ordering via sequence of ordered term calls
- should disallow method calling if method has specified order
- should allow method calling of unordered expectations in any order
- should return self instance from ordered term
- should throw specified exception using andthrow term
- should throw specified exception with message using andthrow term
- should throw exception if class passed to andthrow term not an exception
- should return self instance from andthrow term

p h p mock

- should create mock inheriting class type from original
- should create mock inheriting type if original an interface
- should create mock with specified name
- should create stub from an array of methods and return values
- should throw exception if class is final
- should throw exception if class does not exist
- should not throw class not exists exception if an interface
- should mock all allowed public methods which do nothing since mocked
- should mock public static methods which do nothing since mocked (PENDING)
- should retain method parameter definitions in mocks
- should return true on validation by default if no expectations set

55 examples, 0 failures, 1 error, 1 exception, 1 pending

Errors:

1)

```
'p h p mock method expectations should expect each method exactly once unless otherwise specified' ERROR
exception 'PHPSpec_Runner_ErrorException' with message 'more bad stuff happenin'' in
/user/opt/projects/phpmock/specs/PHPMockMethodExpectationsSpec.php:19
Stack trace:
#0 [internal function]: PHPSpec_ErrorHandler(1024, 'more bad stuff ...', '/user/opt/...', 19, Array)
#1 /user/opt/projects/phpmock/specs/PHPMockMethodExpectationsSpec.php(19): trigger_error('more bad stuff ...')
#2 /user/opt/projects/phpspec/trunk/src/PHPSpec/Runner/Example.php(81): DescribePHPMockMethodExpectations->itShouldExpectEachMethodExactlyOnceUnlessOtherwiseSpecified()
#3 /user/opt/projects/phpspec/trunk/src/PHPSpec/Runner/Collection.php(73):
```

```
PHPSpec_Runner_Example->execute()  
#4 /user/opt/projects/phpspec/trunk/src/PHPSpec/Runner/Base.php(51):  
PHPSpec_Runner_Collection->execute(Object(PHPSpec_Runner_Result))  
#5 /user/opt/projects/phpspec/trunk/src/PHPSpec/Runner/Base.php(44):  
PHPSpec_Runner_Base->executeExamples()  
#6 /user/opt/projects/phpspec/trunk/src/PHPSpec/Console/Command.php(82):  
PHPSpec_Runner_Base::execute(Object(PHPSpec_Runner_Collection),  
Object(PHPSpec_Ru  
nner_Result))  
#7 /user/opt/projects/phpspec/trunk/src/PHPSpec/Console/Command.php(125):  
PHPSpec_Console_Command::main()  
#8 {main}
```

Exceptions:

```
1)  
'p h p mock method expectations should throw default exception if method call  
count is unexpected' EXCEPTION  
exception 'PHPSpec_Exception' with message 'no specification object created yet'  
in /user/opt/projects/phpspec/trunk/src/PHPSpec/Context.php:92  
Stack trace:  
#0 /user/opt/projects/phpspec/trunk/src/PHPSpec/Runner/Example.php(89):  
PHPSpec_Context->getCurrentSpecification()  
#1 /user/opt/projects/phpspec/trunk/src/PHPSpec/Runner/Collection.php(73):  
PHPSpec_Runner_Example->execute()  
#2 /user/opt/projects/phpspec/trunk/src/PHPSpec/Runner/Base.php(51): PHPSp  
ec_Runner_Collection->execute(Object(PHPSpec_Runner_Result))  
#3 /user/opt/projects/phpspec/trunk/src/PHPSpec/Runner/Base.php(44):  
PHPSpec_Runner_Base->executeExamples()  
#4 /user/opt/projects/phpspec/trunk/src/PHPSpec/Console/Command.php(82):  
PHPSpec_Runner_Base::execute(Object(PHPSpec_Runner_Collection),  
Object(PHPSpec_Runner_Result))  
#5 /user/opt/projects/phpspec/trunk/src/PHPSpec/Console/Command.php(125):  
PHPSpec_Console_Command::main()  
#6 {main}
```

Pending:

```
1)  
'p h p mock should mock public static methods which do nothing since mocked'  
PENDING  
Static methods will need static handling
```

Posted by Pádraic Brady in PHP General, PHP Security at 21:27

Sunday, November 11. 2007

Mocks, Stubs, And SimpleTest Wins

The word is that SimpleTest is moving towards PHP5 in the near future which is great news for all Mockists in PHP. When I moved to PHP, SimpleTest became the main ingredient in many a coding session up to 2 in the morning but once PHP5 gained traction and I was seduced into leaving PHP4 behind I found myself relying more heavily on PHPUnit. Not that SimpleTest is anti-PHP5 in any way (only some small things and obviously E_STRICT giving it a heart attack), but more projects I didn't control leaned heavily towards PHPUnit once PHP5 took off.


Why the noting of SimpleTest? Because it's a Mockist library. Between PHPUnit and SimpleTest, ST under Marcus Baker's leadership stomps all over PHPUnit for Mock Object support, and has done for years. PHPUnit is a great Unit Testing library, but I just don't like it when applying TDD (or even some limited BDD before recently). The SimpleTest documentation (which is itself deceptively simple) is basically a TDD tutorial which runs you through basic Unit Testing, how test-first approaches lead to better design decisions, and how Mocks and Stubs make designing even better by exploring the interactions between objects (i.e. SimpleTest's introduction to TDD is behaviour driven) and server resources.

PHPUnit has always seemed to attract people who prefer to test state, or use Stubs predominantly (hand coded objects with fixed return values), or at a minimum those who can endure the lack of Mock Objects in their lives. I prefer none of these - being behaviour focused. I find it really hard to work with a Unit Testing library that first spent years ignoring Mock Object support, and which then bundles one which is a) a bit foggy, and b) rarely used or highlighted anywhere, and c) useful mainly on a hit or miss basis (some nagging missing features).

Is this short sightedness on my part? Maybe it's just a matter of development styles? I am afterall relatively new to PHPUnit - SimpleTest has served all my needs previously since time began. What does the atypical PHPUnit user do though when they need a Mock Object or Stub? Hand code it???

I've become more interested in Mock Object frameworks recently, mainly to see the broad range available (count them for Java or Ruby and you get quite a collection) before dabbling in one for PHPSpec on the premise a loosely coupled Mock Object framework has value beyond PHPSpec (which I'm sure it does - phpt + mocks would be interesting, but other uses suggest themselves). PHPSpec + PHPMock = BDD At Last.

My vision is still pretty SimpleTest tinted. Noting this is only one part of PHPUnit I'm bashing, but the PHPUnit MO API leaves too much to be desired. It looks quite cute - PHP5, fluent interface, etc., all fluffy and friendly - but the fluent interface doesn't flow naturally, and it seems fickle to a fault. Surely my syntax opinions are questionable, but I just like to have one that doesn't need to be memorised so precisely and flows more intuitively. Handling of indexed method calls with parameters would also be nice - nearly everything I do with Mocks/Stubs needs it at some point. Dropping the fluent interface entirely might actually be an improvement until (if) it's changed.

Is syntax really that important? I know Rubyists would have fun debating the benefits of Mocha vs Flexmock vs rspec mocks. Nearly the whole argument ends up being a syntax choice coloured by your approach to TDD or BDD. Some like to use real objects, others stubs, others mocks, and others even partial-objects that are more real than stubs but not production quality. Martin Fowler even has a wonderously clarifying post on Mock Objects and Stubs, and the kind of people who prefer each. Going by it, SimpleTesters who refuse to get with the popular program of PHPUnit (lots of them about still 


- many live over on Devnetwork) seem to more likely be inclined towards the BDD-related style of TDD - focusing on behaviour and object interactions over state and simple verification.



I mean here's a quick thought. PHPUnit's standard Mock Object example:

```
$observer = $this->getMock('Observer', array('update'));
$observer->expects($this->once())
    ->method('update')
    ->with($this->equalTo('something'));
```

Seriously - this is just horrible! What about:

```
$observer = $this->getMock('Observer', array('update'));
$observer->shouldReceive('update')->once()->with('something');
```

Anyone see a difference? Obviously, with all my moaning above I prefer the second. It's simpler, less typing, reads like plain English (I seem to note that a lot recently 

), does the same thing, and just feels cleaner. I used Stub since it's slightly more accurate for what this is doing - a canned response from an otherwise real object. SimpleTest refers to these as Partial Mocks; respects to Marcus but I still consider them Stubs with partial Mock behaviour - just to be vague, difficult and convoluted 

Personally though, this is all enough to convince me SimpleTest remains the Mockist's hero. PHPUnit is all shiny and bespeckled, but it's just not working for me since I rely heavily on Mock Objects and really really don't like writing Stubs (except for server/slow resources) if I can help it.

Posted by Pádraic Brady in PHP General, PHP Security at 06:32

Wednesday, November 7, 2007

PHPSpec Manual (Work In Progress)

Just a quick update. I started writing a draft manual recently, so I'm publishing it online at:

<http://dev.phpspec.org/manual>

I'll keep adding to it over the days, weeks and months ahead. The current form is going to be mainly a draft. When it's done, I'll go through it and do some serious editing. Next priority is getting in the material on actually using and implementing PHPSpec...

Posted by Pádraic Brady in PHP General, PHP Security at 05:35

The PHPSpec 0.2.0dev API

I had a request to make a very quick summary of the [PHPSpec](http://dev.phpspec.org) API for the current development snapshots over on <http://dev.phpspec.org>. We're lacking documentation for right now, so here's a brief overview.

First of all, since this is a relatively short-life development period, there are a few basic assumptions in place for now.

1. One spec class per file
2. The file name reflects the classname
3. Spec classnames start with "Describe"

A current valid spec class would be similar to

```
class DescribeNewLoggerUsingFileStorage extends PHPSpec_Context
{
    public function itShouldCreateCreateNewFileIfNoneExists()
    {
        $this->pending();
    }
    public function itShouldUseAnExistingFileIfOneExistsWithoutTruncatingIt()
    {
        $this->pending();
    }
}
```

This based on a plain text spec such as:

New logger using file storage:

- should create new file if none exists
- should use an existing file if one exists without truncating it

So the filename in this case would be DescribeNewLoggerUsingFileStorage.php. In reality there should be a less restrictive naming scheme but for now it narrows the possible variations while development is progressing.

The PHPSpec_Context class is a bit like your typical xUnit TestCase class. It's the ultimate parent for all specs. It's called a Context because each class should describe a class or system of classes for a "Given" or context, i.e. the condition or environment of the class being tested. In the case above, the context is that we've just instantiated a Logger which writes messages to a file.

PHPSpec_Context::pending() simple marks an example as awaiting completion.

We can flesh out the spec's examples (note: a spec is a collection of executable examples) as follows.

```
class DescribeANewLoggerUsingFileStorage
{
    public function itShouldCreateNewFileIfNoneExists()
```

```

{
    $file = $this->getTmpFileName();
    $logger = new Logger( $file );
    $this->spec(file_exists($file))->should->beTrue();
}
public function itShouldUseAnExistingFileIfOneExistsWithoutTruncatingIt()
{
    $file = $this->getTmpFileName();
    file_put_contents($file, 'Hello' . "\n");
    $logger = new Logger( $file );
    $this->spec(file_get_contents($file))->should->be('Hello' . "\n");
}
public function after()
{
    unlink($this->getTmpFileName());
}
public function getTmpFileName()
{
    return sys_get_temp_dir() . DIRECTORY_SEPARATOR . 'logger_tmp_file';
}
}

```

You can execute a spec by using the command line script "phpspec" available also for Windows as a batch file.

```
phpspec DescribeANewLoggerUsingFileStorage
```

You can also simple execute all specs within a directory tree using.

```
phpspec -r
```

Specs can also be continually re-run using the -a flag.

Back to the full spec example, PHPSpec_Context::spec() method returns an implementation of a Domain Specific Language (DSL) which basically means it's a specific language designed for Behaviour-Driven Development. This contrasts to an xUnit API based on assertions. The DSL is intended as being more readable, and more intuitive to write examples with.

A very basic call above uses the "be" Matcher, and the "should" Expectation. In PHPSpec, you can state whether any Matcher "should" or "shouldNot" pass. The Matchers presently available (by no means a complete list sufficient for general usage quite yet):

```

be()
equal()
beEqualTo()
beAnInstanceOf()
beGreaterThan()
beLessThan()
beGreaterThanOrEqualTo()
beLessThanOrEqualTo()
beEmpty()

```

beTrue()
beFalse()


Scheduled for the next day or two:

beBetween()
beNull()
beOfType()
beIdenticalTo()
beSet()
match() // PCRE Regular Expressions
raise() // Exceptions
trigger() // Errors

In addition, I recently implemented "Predicate" matching. Predicates are basically methods starting with "is" or "has", e.g. hasMessages(), isPrepared(), hasPlayer(), present in your implementation classes. Using PHPSpec's predicate matcher you can write an example such as:

```
$this->spec($someObject)->should->haveMessages();
```


This calls \$someObject->hasMessages(), and compares the result to the boolean TRUE to ascertain a match. Of course the should/shouldNot expectations work here also to interpret whether a matcher result should pass or fail.

Output from [PHPSpec](#) is still a bit unintelligent. I have updated it to follow the traditional xUnit output (same style as PHPUnit for example) but data such as error line numbers, and backtraces, and string comparison results are not yet possible. Obviously a big priority before any public release could be finalised 

We're homing in on an initial public release but I'll hold off until minimal documentation is in place. In the meantime, you can download regularly updated snapshots from <http://dev.phpspec.org>. Note that the tar.gz file is a typical archive, and the .tgz file is a PEAR installable package. The second is much preferred since it will install the necessary script files.

Posted by Pádraic Brady in PHP General, PHP Security at 18:47

PHPSpec hits Subversion Revision 100

What is it? [PHPSpec](#) is a Behaviour-Driven Development framework for PHP currently entering it's fourth week of consuming my free time 

What is Behaviour-Driven Development? BDD is Test-Driven Development without all the crap that nobody really understands and everyone tends to get wrong anyway, including something called a domain specific language (DSL) that means specs are incredibly readable and clear.

Why is this better? Because it's easy to get into, requires less experience building, and dumps the traditional Unit Testing API in favour of a domain specific language inspired by Plain English (TM).

Everyone's favourite question is going to be how BDD differs from TDD, which is entirely valid but a question which usually leads to the short insufficient answer or the long inevitably boring answer. So I'll try to put it in perspective insufficiently and refer to a much better explanation later in the near future.

Essentially, TDD is performed using a Unit Testing framework. Whether it's SimpleTest, PHPUnit, Test::Unit or JUnit, it doesn't really matter since they all share the same common approach. You call everything a Test, and within your tests you assert things. In TDD we advocate the idea of testing before coding - since it leads to better code. But TDD tends to be obsessed with Testing unless explained very very clearly. And because it's so closely linked to testing, programmers understandably get TDD confused with Testing and think they are facets of the same thing. In fact it's common to assume TDD is a testing practice, when in reality tests are just the byproduct of a design practice.

BDD takes a slightly different tack. It cheats. Instead of teaching you all about Unit Testing, it just jettisons all that baggage and drops you into the practice of "describing behaviour with specifications". You do not start with a test for BDD, you start by specifying the behaviour of the system (classes/methods) you intend writing. If that sounds horribly familiar (it should to any experienced hardened practitioner of TDD) it's because this really is what TDD is supposed to be about - if you can find it in the small print on the last page of all those TDD tutorials or the artfully concealed manual page of your UT documentation.

So with BDD, you identify a system, write specifications for it, and then implement code to the required specifications. If your code coverage sucks, well, what the hell, you're not testing so unless you have a code coverage of

Posted by Pádraic Brady in PHP General, PHP Security at 10:18