


## 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

Looks great 

Anonymous on Dec 2 2007, 02:56