Miniproject in Test and Verification - Part 2

Jacob Karstensensen Wortmann
Sam Sepstrup Olesen
Nicklas Andersen
sw805f14

May 28, 2014

2a - Results of Test Cases

Environment setup

To avoid corrupting the data that is running on production, we made a script that makes an exact copy of the database to perform all of our tests on. An example of how the testing could corrupt the database is testing creation and deletion of a recipe. When the test is run, a recipe will be created to ensure that this functionality works. If for any reason this recipe could not be deleted it would be possible for the users to see this recipe. The copying of the database could prove to be a bottleneck if the database were to get very big.

PHPUnit

The basic approach of using PHPUnit is to create a new file that includes the functions which should be tested. The new file must have a new class that extends `PHPUnit_Framework_TestCase`. All public methods in this new class, which are named with a prefix of `test`, will be included in the test. Inside each of these methods all kinds of assertions can be made, an example could be `$this->assertNotNull($var)`.

To demonstrate how we have performed the unit tests, the tests `testSetRecipe()` and `testGetRecipe()` have been included. These tests ensure that after a new recipe is written to the database, the exact same recipe can be retrieved again with no missing data.

```php
/**
 * @depends testSetUnit
 * @depends testSetIngredient
 * @covers DB::setRecipe
 */
public function testSetRecipe($unit, $ingredient) {
    $url = 'http://localhost/food/lib/recipebackend.php';
p
    $fields = array(
        'test'=>"true",
        'recipeName'=>$this->recipeName,
12          'recipeDesc'=>$this->recipeDesc,
13          'group1'=>$this->groupName,
14          'group1_exchange1'=>"",
15          'group1_exchange1_mandatory'=>$this->firstExchangeMandatory,
16          'group1_exchange1_ingredient1'=>$ingredient,
17          'group1_exchange1_ingredient1_amount'=>$this->firstIngAmount,
18          'group1_exchange1_ingredient1_unit'=>$unit,
19          'step1'=>$this->firstStepDesc,
20          'step2'=>$this->secondStepDesc
21          );
22
23      $result = $this->sendCurl($url, $fields);
24      $this->assertNotNull($result);
25
26      $resultRecipeId = (int)strip_tags($result);
27      $this->assertGreaterThan(0, $resultRecipeId);
28
29      return $resultRecipeId;
30  }
```

Lines 2-3 The test depends on both `testSetUnit()` and `testSetIngredient()`, we need both ingredients and a unit of measurement to create a recipe.

Line 4 We specify that this test covers the `setRecipe()` method in the `DB` class. This is because the code coverage analysis tool does not properly detect coverage when using cURL.

Line 6 `testSetRecipe()` takes two parameters: the ID of the inserted unit of measurement from `testSetUnit()` and the ID from `testSetIngredient()` respectively.

Line 7 The `$url` variable is initialised to the value address of the file that handles creation of recipes. *recipebackend.php* is used by the recipe creator tool.

Lines 9-21 An array is initialised with key value pairs that represent the POST variables that is send to the given url.

Line 23 The request is send to the server with the given POST variables and the result is stored in `$result`.

Line 24 It is checked that the result is not null. If the result was null it would mean that something went wrong while inserting the data.

Lines 26-27 The ID of the newly inserted recipe must be greater than 0.

Line 29 Return the ID of the insert recipe. This ID is used by the `testGetRecipe()` method to check that we can extract the recipe correctly again.

```
 1 /**
 2  * @depends testSetRecipe
 3  */
 4 public function testGetRecipe($recipeId) {
 5     $jsonResult = self::$DB->getRecipe($recipeId, $this->language, $this->
           metric);
 6     $recipe = json_decode($jsonResult);
 7
 8     //Check that the recipe is equal to the one we inserted.
 9     $this->assertEquals($recipeId, $recipe->id);
10     $this->assertEquals($this->recipeName, $recipe->name);
11     $this->assertEquals($this->recipeDesc, $recipe->desc);
12     $this->assertEquals($this->recipeImage, $recipe->image);
13     $this->assertEquals($this->groupName, $recipe->groups[0]->name);
14     $this->assertNotNull($recipe->groups[0]->order);
15     $this->assertEquals($this->ingredientName, $recipe->groups[0]->
           exchanges[0]->ingredients[0]->name);
16     $this->assertEquals($this->firstIngAmount, $recipe->groups[0]->exchanges
           [0]->ingredients[0]->amount);
17     $this->assertEquals($this->metricName, $recipe->groups[0]->exchanges
           [0]->ingredients[0]->unit);
18
19     $this->assertEquals(((bool)$this->firstExchangeMandatory), $recipe->
           groups[0]->exchanges[0]->mandatory);
20 }
```

Lines 2 The test depends on testSetRecipe().

Line 4 testGetRecipe() takes the ID that was returned from testSetRecipe() as a parameter.

Lines 5-6 The method to get a recipe is called from the DB class, and the JSON result is deserialised. This enables us to handle the result as a PHP object.

Line 7 The $url variable is initialised to the value address of the file that handles creation of recipes. *recipebackend.php* is used by the recipe creator tool.

Lines 9-19 Numerous assertions are made to ensure that the data in the fetched object is equal to the data we used when writing the recipe to the database.

Result

```
Time: 1 second, Memory: 3.75Mb
OK (16 tests, 45 assertions)
```

Listing 1: The result of the PHPUnit test.

The goal was to achieve 100% statement coverage of the DB class and as seen in Figure 0.1 this was succeeded. Since all of the tests pass, we can be more confident that all of the functionality on the server works as intended. As long as the mobile application sends the correct data of the correct format, a valid response should be sent back.

```
Score: 78.12%
(14 escaped on 64 mutants)
```

Listing 2: The result of the Mutagenesis test.

The tests have been run trough a mutation testing tool, judgedim/mutagenesis, to ensure that the tests work as intended. The mutation test will change certain aspects of the source code before running the tests. This includes changing strings to random strings, changing integers, negating boolean statements, etc. If a test still passes after the source code has been altered, the test is considered defective. The mutation testing tool gives a success rate of 78.12% which means that 78.12% of the mutants made the tests fail. It is a satisfactory success rate, but could be improved if we had more time.

```
$exchange = new Exchange();
// Original
$exchange->mandatory = ($exchangerow->mandatory == 1) ? true : false;
// Mutant
$exchange->mandatory = ($exchangerow->mandatory == 1) ? true : true;
```

Listing 3: Example of an escaped mutant.

Listing 3 shows an example of a mutant that has escaped. When a recipe is made with the recipe creation tool the mandatory value is set to either 1 or 0. We want to represent this value as a boolean in the database, thus making the simple if statement seen in Listing 3. The way to fix this mutant is to check that the value of mandatory still matches the value set in the recipe creation tool after it has been inserted in the database.

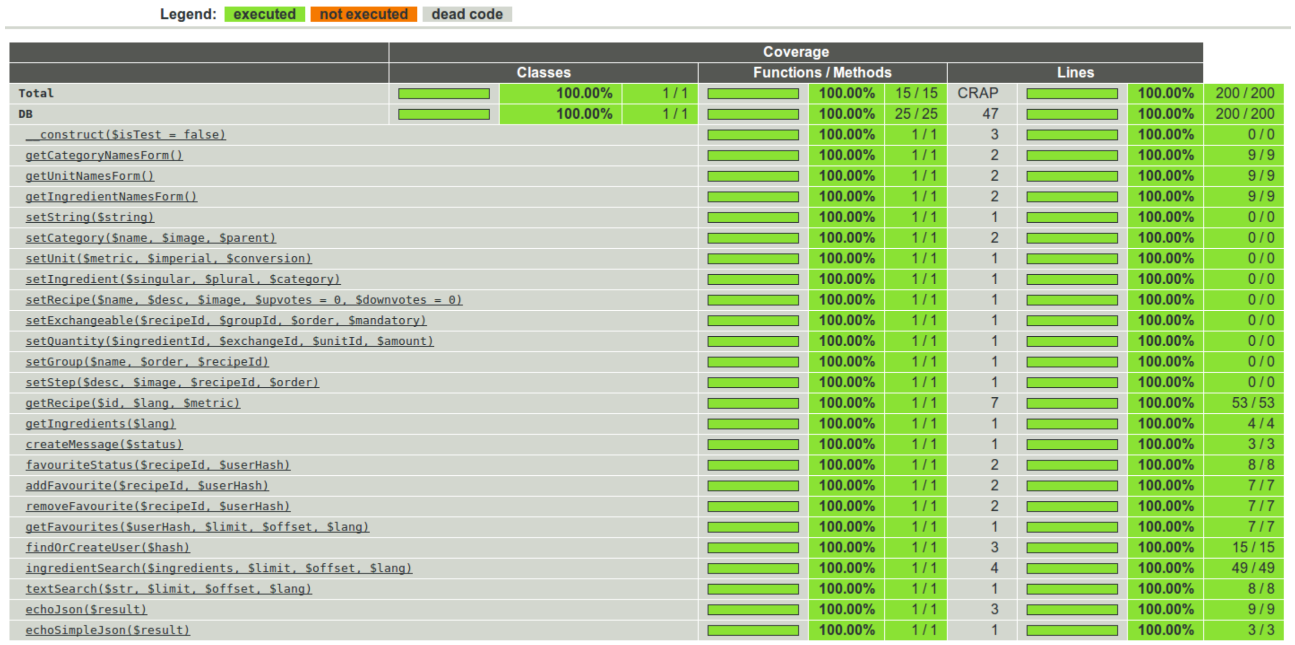
2b - Reflections

We quickly realised that having 100% code covering does not necessarily mean much. A better metric for determining test quality could be:

$$CodeCoverage \times MutationScore = TestQuality$$

This forces you to write tests of high quality since it would be easy to achieve a high code coverage if the assertions always passed regardless of input/output. Likewise, if you only have a code coverage of 50% the test quality will never surpass 50%

Our experience with mutation testing is generally positive since it quickly enables us to find mistakes in the unit tests. A good example is the snippet seen in Listing 3. A test for this case would not be difficult to write, but we did not spot the error ourselves. It might vary from tool to tool, but the tools that we tried for mutating PHP unit tests were all difficult to setup.



Legend: executed | not executed | dead code

	Coverage									
	Classes			Functions / Methods				Lines		
Total	🟩	100.00%	1 / 1	🟩	100.00%	15 / 15	CRAP	🟩	100.00%	200 / 200
DB	🟩	100.00%	1 / 1	🟩	100.00%	25 / 25	47	🟩	100.00%	200 / 200
__construct($isTest = false)				🟩	100.00%	1 / 1	3	🟩	100.00%	0 / 0
getCategoryNamesForm()				🟩	100.00%	1 / 1	2	🟩	100.00%	9 / 9
getUnitNamesForm()				🟩	100.00%	1 / 1	2	🟩	100.00%	9 / 9
getIngredientNamesForm()				🟩	100.00%	1 / 1	2	🟩	100.00%	9 / 9
setString($string)				🟩	100.00%	1 / 1	1	🟩	100.00%	0 / 0
setCategory($name, $image, $parent)				🟩	100.00%	1 / 1	2	🟩	100.00%	0 / 0
setUnit($metric, $imperial, $conversion)				🟩	100.00%	1 / 1	1	🟩	100.00%	0 / 0
setIngredient($singular, $plural, $category)				🟩	100.00%	1 / 1	1	🟩	100.00%	0 / 0
setRecipe($name, $desc, $image, $upvotes = 0, $downvotes = 0)				🟩	100.00%	1 / 1	1	🟩	100.00%	0 / 0
setExchangeable($recipeId, $groupId, $order, $mandatory)				🟩	100.00%	1 / 1	1	🟩	100.00%	0 / 0
setQuantity($ingredientId, $exchangeId, $unitId, $amount)				🟩	100.00%	1 / 1	1	🟩	100.00%	0 / 0
setGroup($name, $order, $recipeId)				🟩	100.00%	1 / 1	1	🟩	100.00%	0 / 0
setStep($desc, $image, $recipeId, $order)				🟩	100.00%	1 / 1	1	🟩	100.00%	0 / 0
getRecipe($id, $lang, $metric)				🟩	100.00%	1 / 1	7	🟩	100.00%	53 / 53
getIngredients($lang)				🟩	100.00%	1 / 1	1	🟩	100.00%	4 / 4
createMessage($status)				🟩	100.00%	1 / 1	1	🟩	100.00%	3 / 3
favouriteStatus($recipeId, $userHash)				🟩	100.00%	1 / 1	2	🟩	100.00%	8 / 8
addFavourite($recipeId, $userHash)				🟩	100.00%	1 / 1	2	🟩	100.00%	7 / 7
removeFavourite($recipeId, $userHash)				🟩	100.00%	1 / 1	2	🟩	100.00%	7 / 7
getFavourites($userHash, $limit, $offset, $lang)				🟩	100.00%	1 / 1	1	🟩	100.00%	7 / 7
findOrCreateUser($hash)				🟩	100.00%	1 / 1	3	🟩	100.00%	15 / 15
ingredientSearch($ingredients, $limit, $offset, $lang)				🟩	100.00%	1 / 1	4	🟩	100.00%	49 / 49
textSearch($str, $limit, $offset, $lang)				🟩	100.00%	1 / 1	1	🟩	100.00%	8 / 8
echoJson($result)				🟩	100.00%	1 / 1	3	🟩	100.00%	9 / 9
echoSimpleJson($result)				🟩	100.00%	1 / 1	1	🟩	100.00%	3 / 3

Figure 0.1: Code coverage analysis.