Unit testing

17 Januari 2011, voor Codequest.nl

Je kent het vast wel. Je bent uren bezig geweest met een hippe nieuwe applicatie, en ineens begint deze onverwacht gedrag te vertonen. Is hier nou echt niets tegen te doen?

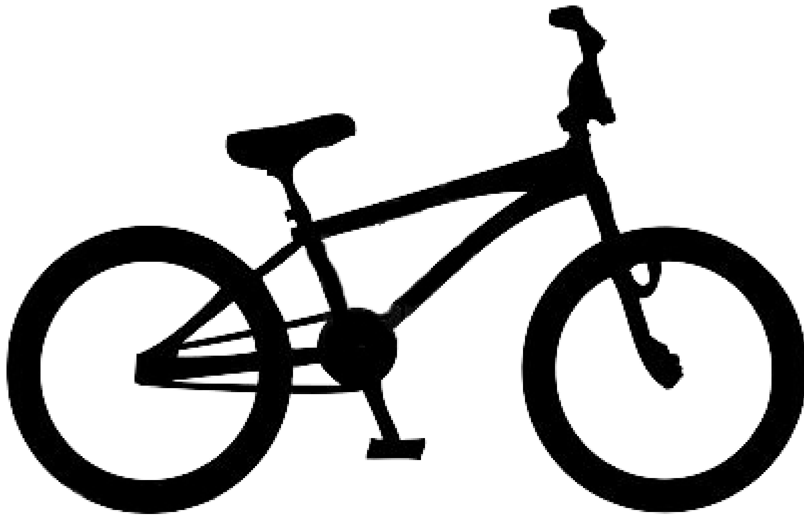
Wat houdt testen in

Iedereen heeft het wel eens meegemaakt, of gaat dit nog meemaken. Je levert een applicatie op en na langdurig 'testen' blijken er toch rare dingen te gebeuren. Het rekenwerk komt niet goed uit, een mailtje wordt niet verstuurd of - nog erger - naar de verkeerde persoon, de bestanden worden niet goed weggeschreven, etc. etc.. Je zou toch zweren dat je alles volledig getest had, en alles ook echt op de juiste manier zou moeten werken. Het is toch altijd weer irritant dat dit niet altijd het geval blijkt te zijn. Hier valt wat aan te doen, maar sta je eigenlijk wel stil bij de bepaling van of een applicatie nu werkt of niet?

Trial and error

Veel applicaties worden geschreven via een methode die 'trial and error' genoemd wordt. Het gebeurd vaak dat iemand begint met de ontwikkeling van een functionaliteit, en deze blijft uitbreiden tot dat deze naar wens van zichzelf of de opdrachtgever is. Zodra deze situatie bereikt is, wordt deze vervolgens opgeleverd en wordt er verder geprogrammeerd aan de volgende functionaliteit.

Hoewel deze methode een goedkope manier is om tot een goed inzetbare applicatie te komen, kleven hier absoluut nadelen aan. Terwijl jij zelf een duidelijk beeld hebt van hoe je applicatie moet gaan werken, zullen de eindgebruikers hiervan dit minder hebben. Gebruiken zij de functionaliteit echt wel op de juiste manier? Als dit niet goed afgevangen wordt, kan dit leiden tot beveiligingslekken, maar net zo snel tot onverwachte bugs binnen jouw eindresultaat. Als we dit in de realiteit brengen, ziet dit er als volgt uit:



- Kan de fiets fietsen? ✔
- Kan de fiets sturen? ✔

Alles lijkt goed te werken, toch?

Wat is unit testing?

Hoewel applicaties soms lijken te doen wat er van ze verwacht wordt. In het vorige hoofdstuk liet ik een voorbeeld zien, waarin dit ook het geval was. Toch zijn er wellicht situaties waarin dit niet persee het geval hoeft te zijn. Om bij het voorbeeld te blijven; wat zou er gebeuren als er achteruit getrapt wordt op de fiets? Als je naar links stuurt, beweegt te fiets zich dan ook naar links?

Unit testing is een methode waarmee losse stukken code (units) getest worden op de correcte werking. Het maakt binnen een unit test dus ook niet uit wat het eindresultaat moet doen, zolang het stuk code doet wat er van hem verwacht wordt, is de test gelukt en kan er verder gegaan worden naar de volgende test. Indirect zal dit natuurlijk ook betrekking hebben op het eindresultaat. Als wij het maken van een fiets nog eens nalopen, maar nu door middel van unit testing, zal ongeveer hetvolgende bereikt worden:

- Als mijn voorste tandwiel rechtsom draait, draait het achterste randwiel dan ook rechtsom?
- Als ik achteruit trap, zal het voorste tandwiel dan stoppen met draaien?
- Als het voorste voorwiel stopt met draaien, zal de fiets dan vaart minderen?
- Als ik mijn stuur naar links beweeg, zal het voorwiel dan ook naar links bewegen?
- Etc., etc.

Conclusie: alle bovenstaande tests zijn gelukt, dus de fiets zal op een correcte manier kunnen fietsen en sturen. Niet alleen weten we dat het eindresultaat kan wat er van verwacht wordt, maar ook dat de functionaliteit op de correcte manier wordt uitgevoerd.

Testbare code

Unit testing leidt over het algemeen dus tot een applicatie waarin meer functionaliteit getest wordt, en hierdoor minder onverwachts gedrag ontstaat. Toch kunnen niet alle projecten direct onderworpen worden aan een set unit tests. De code moet testbaar zijn.

Laten we bijvoorbeeld eens kijken naar het volgende voorbeeld:

```
1.  <?php
2.
3.  function doe_iets_en_daarna_iets_anders($input) {
4.      // Hier een hoop code om de eerste
5.      // functionaliteit uit te kunnen voeren
6.      if(! $succes)
7.          return false;
8.
9.      // Hier een hoop code om de tweede
10.     // functionaliteit uit te kunnen voeren
11.     if(! $succes)
12.         return false;
13.
14.     return true;
15. }
16.
17. ?>
```

In bovenstaande voorbeeld wordt er false teruggeven als de functie mislukt is, en true als deze gelukt is. Hieruit kan het resultaat van de functie toch eenvoudig afgelezen worden. Het probleem ligt echter in de bepaling van wat er precies fout is gegaan als de functie mislukt. Is dat het eerste stuk code, of is dat het tweede stuk code. Doordat dit uit een unit test niet direct duidelijk is, is deze test eigenlijk niet bruikbaar.

Conclusie: het resultaat van een unit test moet duidelijk aangeven waar een fout zich precies voordoet.

De vraag is nu dus, hoe kan bovenstaande stuk code wel goed testbaar gemaakt worden? Eigenlijk is dit simpel door één van de basisprincipes van programmeren goed te hanteren: *Een functie of methode is verantwoordelijk voor slechts één functionaliteit.*

Nogmaals ons bovenstaande voorbeeld, welke nu testbaar zal zijn:

```
1.  <?php
2.
3.  function doe_iets_en_daarna_iets_anders($input) {
4.      if(! doe_iets($input))
5.          return false;
6.
7.      if(! doe_iets_anders($input))
8.          return false;
9.      return true;
10. }
11.
12. function doe_iets($input) {
13.     // Hier de code om 'iets' te doen
14.     return $succes;
15. }
16.
17. function doe_iets_anders($input) {
18.     // Hier de code om 'iets anders' te doen
19.     return $succes;
20. }
21.
22. ?>
```

Wij hebben nog steeds dezelfde functionaliteit als dat we voorheen hadden. Toch is de code hierin opgesplitst in meerdere losse functies, die beiden goed ondergebracht kunnen worden in aparte unit tests. Zodra we deze 3 functies onderbrengen in een testcase, weten we nog steeds niet wat er nou

mislukt is zodra de functie doe_iets_en_daarna_iets_anders() mislukt. Echter hebben wij 2 andere tests die zullen wel aanduiden waar het probleem precies zit.

Conclusie: om tot testbare code te komen, is het verstandig losse functies en methodes zo kort mogelijk te houden. Zodra een functionaliteit uit meerdere stappen bestaat, maak hier dan losse functies van.

PHPUnit

In de vorige hoofdstukken heb ik laten zien hoe unit testing werkt, en waar de code aan moet voldoen om daadwerkelijk testbaar te zijn. In dit hoofdstuk zal ik één van de tools beschrijven die je kunt gebruiken voor het opzetten en uitvoeren van unit tests: PHPUnit.

De naam PHPUnit is eigenlijk logisch. Het is een tool uit de xUnit familie, die bedoeld is voor PHP. Voor andere talen zijn er veelal tools beschikbaar die een soortgelijke naam hebben, bijvoorbeeld JUnit, CUnit, ASPUnit, etc..

De installatie van PHPUnit kan eenvoudig via PEAR. Er is een eigen PEAR channel beschikbaar, waar alle PHPUnit verwante pakketten te vinden zijn. Deze is pear.phpunit.de. Hierin kun je de package phpunit/PHPUnit terugvinden, waarmee de nieuwste versie van het complete PHPUnit framework geïnstalleerd wordt.

```
1.  # pear channel-discover pear.phpunit.de
2.  # pear install phpunit/PHPUnit
```

Nu PHPUnit geïnstalleerd is, kunnen wij eens kijken naar hoe unit tests voor PHP in elkaar zitten. Ik zal in mijn voorbeelden klassen gebruiken. Heb jij nog geen ervaring met object georiënteerd programmeren? Wellicht is het dan raadzaam om mijn eerdere artikel hierover door te lezen. Stel dat wij de volgende klasse hebben:

```
1.  <?php
2.      class Example {
3.          private $string;
4.
5.          public function __construct($string = 'Hello World!') {
6.              $this->string = $string;
7.          }
8.
9.          public function getString() {
10.             return $this->string;
11.         }
12.     }
13. ?>
```

We hebben een klasse Example, die onthoudt wat wordt meegegeven in de constructor en deze weer terug kan geven door middel van de getString() methode. Omdat we zeker willen weten dat de methode getString() wel de juiste string weergeeft als deze aangeroepen wordt, zullen we hiervoor een unit test schrijven.

```
<?php

1.      require 'Example.php';
2.      require 'PHPUnit/Framework.php';
3.
4.      class ExampleTest extends PHPUnit_Framework_TestCase {
5.          public function test_construct() {
6.              $object = new Example();
7.              $this->assertType('Example', $object);
8.          }
9.
10.         public function test_getString() {
11.             $object = new Example();
12.             $this->assertEquals('Hello World!', $object->getString());
13.
14.             $object = new Example('Test 123');
15.             $this->assertEquals('Test 123', $object->getString());
16.         }
17.     }
18. ?>
```

In bovenstaande voorbeeld staat een correcte unit test beschreven, waarin zowel de constructor als de methode getString() getest worden. Er wordt begonnen met het opnemen van de te testen klasse in de code, om vervolgens het PHPUnit framework in te laden. Let op: de locatie naar PHPUnit hoeft niet te betekenen dat PHPUnit opgenomen is in jouw project. Standaard komen de klassen uit PEAR in een map opgenomen die opgenomen is in de paden waar PHP zal zoeken naar in te laden bestanden.

Nu alles ingeladen is, kan de daadwerkelijke test geschreven worden. Wij maken opnieuw een klasse, welke een uitbreiding is op één van de klassen binnen het PHPUnit framework. Vervolgens nemen wij hier 2 methoden in op, waarvan we de naam laten beginnen met 'test_'. Dit stellen de tests voor, waarvan wij er in dit geval dus 2 opnemen:

1. In de eerste test wordt de constructor van de klasse 'Example' aangeroepen, en zullen wij testen of de contructor wel een object van het juiste type, namelijk het type 'Example', teruggeeft.
2. Vervolgens wordt dit nogmaals gedaan, en zal getest worden of het resultaat van de getString() methode wel overeen komt met wat wij verwachten. Omdat de constructor van 'Example' zelf al een standaard string zal opslaan mits er iets wordt meegegeven, zal deze test uit 2 delen bestaan. In het eerste deel zal vergeleken worden of de getString() methode teruggeeft wat we verwachten als er niets wordt meegegeven aan de constructor. Het tweede deel test dezelfde functionaliteit in het geval dat we wel iets meegeven aan de constructor.

Benieuwd naar het resultaat?

De resultaten van PHPUnit

In het vorige hoofdstuk hebben wij onze unit test opgezet. Nu willen wij natuurlijk het resultaat hiervan zien. Dit is een goed moment om eens te kijken naar hoe PHPUnit precies uitgevoerd wordt. Eigenlijk heel simpel:

```
1. # phpunit ExampleTest
```

Let op: de tests voor Example in klasse ExampleTest, moeten worden ondergebracht in het bestand ExampleTest.php. In het bovenstaande voorbeeld ga ik er even vanuit dat uw working directory de map is waarin dit bestand zich bevind.

Het moment van de waarheid. Zijn de tests succesvol, ja of nee?

```
1. # phpunit ExampleTest
2. PHPUnit 3.2.16 by Sebastian Bergmann.
3.
4. ..
5.
6. Time: 0 seconds
7.
8.
9. OK (2 tests)
```

Conclusie: de tests zijn succesvol dus de klasse werkt zoals wij verwachten. Bij unit testing is het belangrijk dat de tests altijd de correctheid bewijzen van de code. Het aanpassen van tests is dus alleen toegestaan indien deze incorrect zijn, niet alleen omdat de test niet lukken terwijl jij van mening bent dat alles wel werkt.

De tests zijn dus gelukt. Wil dit zeggen dat onze tests nu gebruiksklaar zijn? Wellicht kunnen wij nog wat toevoegen. De tests uit het eerdere voorbeeld gingen direct goed. Wat zou er echter gebeuren als de tests niet lukken? De output hiervan ziet er als volgt uit:

```
1. # phpunit ExampleTest
2. PHPUnit 3.2.16 by Sebastian Bergmann.
3.
4. .F
5.
6. Time: 0 seconds
7.
8. There was 1 failure:
9.
10. 1) test_getString(ExampleTest)
11. Failed asserting that two strings are equal.
12. expected string <Test 123>
13. difference      <        ?>
14. got string      <Test 1234>
15. /home/development/tristan/ExampleTest.php:16
16.
17. FAILURES!
18. Tests: 2, Failures: 1.
```

Met de unit test vers in het geheugen, is op dit moment misschien direct duidelijk wat er mis gaat. Maar weet je dit volgende maand nog steeds? En over een jaar dan?

Om de tests duidelijk te houden, is het aan te raden om hierbinnen daadwerkelijk te beschrijven wat er precies fout is gegaan. Laten we de test uit het vorige hoofdstuk iets aanpassen:

```
1.  public function test_getString() {
2.          $object = new Example();
3.          $this->assertEquals('Hello World!', $object->getString(), 'De test
    mislukt als er niets meegegeven wordt aan de constructor');
4.          $object = new Example('Test 123');
5.          $this->assertEquals('Test 123', $object->getString(), 'De test mislukt
    als er iets wordt meegeven aan de constructor.');
6.  }
```

Je ziet dat er aan de assert functies een extra argument is meegegeven. Hieruit kan opgemaakt worden wat er fout is gegaan tijdens het testen van de klasse. Indien de test nu mislukt, zul je het volgende resultaat terugkrijgen:

```
1.  # phpunit ExampleTest
2.  PHPUnit 3.2.16 by Sebastian Bergmann.
3.
4.  .F
5.
6.  Time: 0 seconds
7.
8.  There was 1 failure:
9.
10. 1) test_getString(ExampleTest)
11. De test mislukt als er iets wordt meegeven aan de constructor.
12. Failed asserting that two strings are equal.
13. expected string <Test 123>
14. difference      <        ?>
15. got string      <Test 1234>
16. /home/development/tristan/ExampleTest.php:16
17.
18. FAILURES!
19. Tests: 2, Failures: 1.
```

Er is nu een extra regel toegevoegd aan de output, waarin wij onze berichtgeving omtrent het resultaat van een test terug kunnen vinden. Ook als je deze test over een paar maanden uitvoert, weet je dus precies wat er mis is gegaan, namelijk: 'De test mislukt als er iets wordt meegeven aan de constructor.'. Dit is een goede manier om je tests duidelijk te houden, ook voor in de toekomst maar ook voor anderen die te maken krijgen met de test en deze niet geschreven hebben.

Nu we weten hoe unit tests werken in PHP, is het toch eigenlijk jammer dat we alleen code zonder afhankelijkheden op andere klassen, of eventueel een database kunnen testen. In werkelijkheid kom je het juist vaak tegen dat er dit soort afhankelijkheden bestaan. Is het testen hiervan echt niet mogelijk?

Het testen van ontestbare code

Nu we weten hoe de unit tests in elkaar zitten, zouden we toch eens moeten kijken naar ontestbare code. Tot op zeker hoogte is het namelijk mogelijk om dit te testen. Dit kan op twee manieren gedaan worden: een fixture - waar ik in dit artikel niet te diep op in zal gaan - en mock objects.

De fixture van een test, is de staat waarin de applicatie gebracht zal worden, alvorens de test uitgevoerd wordt. Dit kan op de volgende manier bereikt worden:

```
1.  <?php
2.      ...
3.      class ExampleTest extends PHPUnit_Framework_TestCase {
4.          public function setUp() {
5.              // Breng de applicatie naar de gewenste staat
6.              // Hier uw code om tot de gewenste staat te komen
7.          }
8.  
9.          public function tearDown() {
10.             // Breng de applicatie terug naar de originele staat
11.             // Bijvoorbeeld database terugzetten naar origineel, etc.
12.         }
13.     }
14. ?>
```

In bovenstaande test zijn een setUp() en tearDown() methode opgenomen. Deze beginnen niet met test_ en zijn daarom ook **geen** tests. Deze methodes zijn speciale methodes om een applicatie naar de gewenste staat te brengen, en deze na de tests weer terug te brengen in de originele staat. Ik zal hier in dit artikel verder niet gaan, omdat het manipuleren van bestaande dat zoals bijvoorbeeld databases wat mij betreft beter niet in een test ondergebracht kunnen worden. Mits dit mogelijk is, zou ikzelf dus kiezen voor het gebruik van de volgende methode.

Mock objects

Een andere methode voor het consolideren van de te testen code, is het gebruik van mock objects. Met een voorbeeld wordt dit een stuk duidelijker. Stel, we hebben de volgende code:

```
1.  <?php
2.      class Counter {
3.          private $count = 0;
4.  
5.          public function increment($databaseTable) {
6.              $this->count++;
7.              if(! $databaseTable->setCount($this->count))
8.                  return false;
9.  
10.             return true;
11.         }
12.     }
13. ?>
```

De klasse counter houdt een tellertje bij, die initieel op 0 staat. Zodra deze wordt opgehoogd door middel van de increment(), wordt er in de database een waarde geüpdatet. Bij het schrijven van een test hiervoor, willen we eigenlijk niet dat de database aangepast wordt. Het gaat immers om een test, niet om productie data.

Een mock object is een methode, waarin je een object kan beschrijven. Je kunt dus beschrijven welke methodes deze heeft, hoe vaak deze aangeroepen moeten worden en met welke argumenten, maar ook wat deze teruggeeft als alles goed werkt. Bij het testen neem je dus de aanname dat de afhankelijkheden allemaal doen wat er van ze verwacht wordt (en neemt deze waar mogelijk uiteraard op in aparte tests). De unit test voor bovenstaande klasse komt er dan als volgt uit te zien:

```php
<?php

1.      require 'Counter.php';
2.      require 'PHPUnit/Framework.php';
3.
4.      class CounterTest extends PHPUnit_Framework_TestCase {
5.          public function test_increment() {
6.              $object = new Counter();
7.
8.              // Wij maken een database object om later te gebruiken.
9.              // Deze heeft de methode setCount()
10.             $database = $this->getMock('Database', array('setCount'));
11.
12.             // Hieronder wordt gedefinieerd wat er verwacht wordt qua
13.             // aanroepen op het zojuist aangemaakte mock object. Hij wordt
14.             // 1 keer aangeroepen ($this->once()) met een waarde van groter
15.             // dan 0 ($this->greatherThan(0)) en geeft vervolgens true terug
16.             // ($this->returnValue(true)).
17.             $database->expects($this->once())
18.                 ->method('setCount')
19.                 ->with($this->greaterThan(0))
20.                 ->will($this->returnValue(true));
21.
22.             // Nu kunnen we de test uitvoeren, maar in plaats van een object
23.             // welke de database daadwerkelijk aanpast, geven we nu het mock
24.             // object mee. De database wordt dus niet gemanipuleerd
25.             $this->assertTrue($object->increment($database));
26.         }
27.     }
28. ?>
```

In bovenstaande test nemen we de aanname dat het database object doet wat er van deze verwacht wordt. Nu kunnen we de code die hiermee werkt eens goed gaan testen. Zodra we bovenstaande test uitvoeren, zien we dan ook het volgende:

```
1.  # phpunit CounterTest.php
2.  PHPUnit 3.2.16 by Sebastian Bergmann.
3.
4.  .
5.
6.  Time: 0 seconds
7.
8.
9.  OK (1 test)
```

De test lukt!

In de eerste instantie ziet dit er een beetje magisch uit. We geven een object mee dat eigenlijk op een andere manier is opgebouwd dan het origineel. Wordt dit echt wel meegenomen in de test? Laten we het mock object eens vertellen dat er een waarde, groter dan 1 meegegeven moet worden. Dit zal uiteraard niet kloppen omdat de initiële waarde van de teller 0 is, en wij hierbij maar 1 optellen. Omdat 1 niet groter dan 1 is, is de verwachting dan ook dat de test mislukt. Eens kijken wat PHPUnit zegt:

```
# phpunit CounterTest.php

1.  PHPUnit 3.2.16 by Sebastian Bergmann.
2.
3.  F
4.
5.  Time: 0 seconds
6.
7.  There was 1 failure:
8.
9.  1) test_increment(CounterTest)
10. Expectation failed for method name is equal to <string:setCount> when invoked 1
    time(s)
11. Parameter  0  for  invocation  Database::setCount(<integer:1>)  does  not  match
    expected value.
12. Failed asserting that <integer:1> is greater than <integer:1>.
13. /usr/share/php/PHPUnit/Framework/MockObject/Mock.php(193) : eval()'d code:28
14. /home/development/tristan/Counter.php:7
15. /home/development/tristan/CounterTest.php:20
16.
17. FAILURES!
18. Tests: 1, Failures: 1.
19. <="" span="" style="color: rgb(255, 0, 0); ">
```

Conclusie: PHPUnit gaat goed om met mock objects. Zodra deze op een manier gebruikt wordt welke er niet verwacht wordt, mislukt de test. De code wordt dus getest of werking, maar ook op het aanspreken van afhankelijkheden. Dit is een goede manier om dit soort ontestbare code te testen.

Conclusies

Unit testing is een goede manier om code te testen. Het kan niet beredeneren, dus wanneer iets goed lijkt te werken, zal unit testing hier alsnog doorheen prikken. Tevens gaat het bij unit testing niet perse om het eindresultaat, maar meer om de stappen die genomen worden om tot het eindresultaat te komen: het test units van code en niet het totaalplaatje. Toch kan bij een succesvol afgewerkte set van tests wel aangenomen worden dat het eindresultaat ook klopt.

Unit testing test op verwachte resultaten. Zodra ik iets meegeef in een methode of functie, verwacht ik dat hier een bepaalde waarde uitkomt. Zodra dit niet gebeurd, zal de functionaliteit waarschijnlijk niet goed werken. Tests zijn in deze ook leading. Een test mag theoretisch nooit aangepast worden, omdat deze precies beschrijft hoe een bepaalde klasse opgebouwd moet worden, en wat deze precies moet kunnen met bepaalde input.

Testbare code is code die geen afhankelijkheden heeft naar andere systemen of een database. In de praktijk zal het echter vaak voorkomen dat dit niet het geval is. Tot op zekere hoogte kan code geïsoleerd worden, om op deze manier alsnog testbaar te worden. Dit kan op twee manieren, waarbij mock objects wat mij betreft de voorkeur heeft. In sommige gevallen is dit echter niet voldoende, en moet er gewerkt worden met fixtures.

Hoe gaan we nu verder

In dit artikel heb ik slechts de basis van unit testing beschreven. Er valt hierover nog veel meer te vertellen, maar zeker ook zelf uit te zoeken. Er is genoeg naslagwerk te vinden over dit onderwerp en eventueel specifiek PHPUnit. Een goed begin is om de manual van het PHPUnit framework even door te nemen. Deze documentatie gaat dieper in op de volledige functionaliteit en mogelijkheden van unit testing in PHP. Wellicht is dit een goede bron van inspiratie voor jouw toekomstige projecten. Een link naar de PHPUnit manual vind je hier.

Succes met het testen!