PHPUNIT RICETTARIO DEL PROGRAMMATORE SCONTROSO

CHRIS HARTJES
PIETRO ALBERTO ROSSI

PHPUnit Ricettario Del Programmatore Scontroso

Chris Hartjes and Pietro Alberto Rossi

This book is for sale at http://leanpub.com/phpunitricettariodelprogrammatorescontroso

This version was published on 2014-03-24

Indice

Introduzione

Quando ho scritto il libro “The Grumpy Programmer’s Guide To Building Testable Applications In PHP” il mio obiettivo era di insegnare alle persone come scrivere codice che può essere facilmente testato. Ci sono molte informazioni su come usare i vari tools per il testing ma non ero molto d’accordo con questi.

Utilizzando il vostro motore di ricerca, trovate come risolvere determinati casi, ma è difficile trovare un posto che vi mostri varie soluzioni se non di basso livello.

Ho fatto molte ricerche. Mi sono iscritto ad un fantastico corso, product development, cominciai a fare ancora più ricerche, e ho iniziato a creare una soluzione che ero sicuro avrebbe aiutato le persone a risolvere la sofferenza nello scrivere test per il loro codice PHP utilizzando PHPUnit.

Il risultato in questo libro. Ho provato a realizzare vari esempi di codice compresivi di spiegazioni sulle decisioni prese per scriverli.

Non penso che questo sia un libro che leggerete fino alla fine. E’ molto più probabile che finirete per utilizzarlo un capitolo per volta, imparando le competenze di solo una parte del processo di testing.

Come sempre, sono benvenuti i vostri feedback via Twitter e App.net (@grmpyprogrammer) o via email all’indirizzo chartjes@littlehart.net.

Per la versione italiana @sprik89 o mail a pietroalberto.rossi@gmail.com.

Data Provider

Perchè dovreste usare un Data Provider

Uno dei principali obiettivi che ci dovrebbe sempre essere è scrivere il minor codice possibile per risolvere un determinato problema. Questo non è differente quando si tratta di test, che sono veramente nulla di più che codice.

Una delle prime lezioni che ho imparato è che quando ho iniziato a scrivere quello che leggevo sulle test suite è la ricerca di duplicazioni nei test. Ecco un esempio di una situazione in cui questo può accadere.

Molti programmatori avranno familiarità con il problema di FizzBuzz[1], se non altro perché è comunemente presentato come un problema da risolvere come parte di un colloquio. In mia opinione, è un buon problema da presentare perchè tocca vari aspetti reali della programmazione di base.

Quando scrivete test per FizzBuzz, quello che dovete fare è passare un set di valori e verificare che loro sono "FizzBuzzed" correttamente. Il risultato è avere test multipli identici ma che differiscono per il valore con cui li testate. I data provider vi permettono di semplificare questo processo.

Un data provider è un modo per creare set multipli di dati da testare che possono essere passati ad un test method come parametro. Voi create un metodo che è disponibile nella classe ed il tuo test ritornerà un array di valori che coincideranno con i parametri che state passando al test.

Lo so, suona molto complicato di quanto realmente lo è. Guardiamo un esempio.

Uno sguardo a tutti i questi test

Se non sapete cosa è un data provider, i FizzBuzz test a cosa possono assomigliare!?

```
1  <?php
2  class FizzBuzzTest extends PHPUnit_Framework_Testcase
3  {
4      public function setup()
5      {
6          $this->fb = new FizzBuzz();
7      }
8
9      public function testGetFizz()
10     {
```

[1] http://en.wikipedia.org/wiki/FizzBuzz

```
11          $expected = 'Fizz';
12          $input = 3;
13          $response = $this->fb->check($input);
14          $this->assertEquals($expected, $response);
15      }
16
17      public function testGetBuzz()
18      {
19          $expected = 'Buzz';
20          $input = 5;
21          $response = $this->fb->check($input);
22          $this->assertEquals($expected, $response);
23      }
24
25      public function testGetFizzBuzz()
26      {
27          $expected = 'FizzBuzz';
28          $input = 15;
29          $response = $this->fb->check($input);
30          $this->assertEquals($expected, $response);
31      }
32
33      function testPassThru()
34      {
35          $expected = '1';
36          $input = 1;
37          $response = $this->fb->check($input);
38          $this->assertEquals($expected, $response);
39      }
40  }
```

Sono sicuro che vedete il pattern:

- multipli valori di input
- test estremamente simili in configurazione ed esecuzione
- stesse asserzioni

Creare Data Provider

Un data provider è un altro metodo nella vostra classe test che ritorna un array di risultati, dove ogni risultato è un array stesso. PHPUnit converte il risultato di ritorno in parametri di input che il test accetta.

```php
<?php
public function fizzBuzzProvider()
{
    return array(
        array(1, '1'),
        array(3, 'Fizz'),
        array(5, 'Buzz'),
        array(15, 'FizzBuzz')
    );
}
```

Il nome della funzione per il provider non importa, ma prende di significato quando il test fallisce e verrà stampato il nome del data provider.

Per utilizzare un data provider, dobbiamo aggiungere l'annotazione alla docblock che precede il test in modo da dire a PHPUnit di utilizzarlo.

```php
<?php
/**
 * Test for our FizzBuzz object
 *
 * @dataProvider fizzBuzzProvider
 */
public function testFizzBuzz($input, $expected)
{
    $response = $this->fb->check($input);
    $this->assertEquals($expected, $response);
}
```

Adesso abbiamo solamente un test (meno codice da mantenere) e possiamo aggiungere scenari via data provider. Dobbiamo anche imparare a capire, mentre analizziamo quello che abbiamo da testare, quali test abbiamo realmente necessità di scrivere.

Quando utilizziamo un data provider, PHPUnit eseguirà il test method ogni volta per ogni set di dati presenti nel provider. Se il test fallisce vi indicherà quale indice nell'array associato è stato usato per eseguire quel test.

Esempi più complessi

Non pensate che potreste avere solo semplici data provider. Quello che vi serve fare è ritornare un array di array, dove per ogni risultato è presente un parametro che il tuo test method si aspetta.

Qui un esempio più complesso:

```php
<?php
public function complexProvider()
{
    // Read in some data from a CSV file
    $fp = fopen("./fixtures/data.csv");
    $response = array();

    while ($data = fgetcsv($fp, 1000, ",")) {
        $response[] = array($data[0], $data[1], $data[2]);
    }

    fclose($fp);

    return $response;
}
```

Quindi non pensate che avete bisogno di limitare quello che i vostri data provider possono fare. L'obiettivo è di creare un sufficiente set di dati da testare scrupolosamente.

Data Provider Trick

Siccome un data provider ritorna un array associativo, potete assegnargli chiavi descrittive aggiuntive per aiutarvi nel debugging. Per esempio, potete riscrivere il data provider per il nostro FizzBuzz test cosi:

```php
<?php
return array(
    'one'      => array(1, '1'),
    'fizz'     => array(3, 'Fizz'),
    'buzz'     => array(5, 'Buzz'),
    'fizzbuzz' => array(15, 'FizzBuzz')
);
```

Inoltre, un data provider non ha bisogno di essere un metodo della stessa classe. Potete usare metodi di altre classi, ricordandosi di definirle di tipo `public`. Potete utilizzare anche i namespace. Ecco un esempio:

- `@dataProvider Foo::dataProvider`
- `@dataProvider Grumpy\Helpers\Foo::dataProvider`

Questo permette di creare classi di aiuto che sono soltato data provider e ridurre l'ammontare di codice duplicato nel test stesso.