Testing para Aplicaciones Symfony2

```php
<?php
// src/Acme/DemoBundle/Tests/Controller/DemoControllerTest.php
namespace Acme\DemoBundle\Tests\Controller;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class DemoControllerTest extends WebTestCase
 {
    public function testIndex()
    {
       $client = static::createClient();

       $crawler = $client->request('GET', '/demo/hello/Fabien');

       $this->assertGreaterThan(
         0,
         $crawler->filter('html:contains("Hello Fabien")')->count()
     );
    }
}
```

Testing para Aplicaciones Symfony2

Fernando Arconada

Este libro está a la venta en http://leanpub.com/testingsymfony2

Esta versión se publicó en 2015-02-23

¡Twitea sobre el libro!

Por favor ayuda a Fernando Arconada hablando sobre el libro en Twitter!

El hashtag sugerido para este libro es #testingsf2.

Descubre lo que otra gente está diciendo sobre el libro haciendo click en este enlace para buscar el hashtag en Twitter:

https://twitter.com/search?q=#testingsf2

Índice general

Mocks, Dummies, Stubs y otros

En los tests normalmente nos centramos en el comportamiento de un objeto, de ahí el nombre de test unitario, pero en el mundo real un objeto necesita de otros para realizar su función. Cuando estamos haciendo testing, rápidamente acabamos incorporando a nuestro vocabulario la palabra Mocks.

Cuando estamos haciendo un test unitario y la clase que queremos testear (el SUT -Subject Under Test-) tiene objetos colaboradores (objetos que usa para desarrollar su funcionalidad) debemos sustituirlos por otros objetos que actúan como los dobles de los actores en las películas para asegurarnos el aislamiento del SUT. A estos dobles les llamamos comúnmente Mocks, pero luego acabamos escuchando palabras como Dummy y Stub y terminamos con dudas sobre lo que estamos empleando. Voy a intentar aclarar la diferencia entre los diferentes tipos. Pero ¿por qué reemplazar los colaboradores por otros objetos falsos o dobles? - porque pueden ser costosos de crear - hacen los tests lentos - pueden tener comportamientos que no controlamos (esto es lo mas importante) - pueden generar otros objetos que interfieran con otros tests - necesitamos controlar el valor concreto que devuelven para emplearlo en SUT

Doubles

Todos los colaboradores que reemplazamos por objetos “de mentira” que insertamos para controlar el SUT son “dobles” o Doubles o Test Doubles. Un Test Double es el nombre que damos a los objetos que reemplazan a los colaboradores. Sirve tanto para un Mock que para un Stub, así que si tenemos duda en una conversación y no queremos hacer el ridículo, empleando la expresión Test Double acertaremos seguro. De los diferentes tipo de Test Doubles que vamos a ver a continuación, sólo los Mocks verifican el comportamiento. El resto de Doubles sólo verifican el estado.

Dummy

Un objeto de tipo dummy es un objeto tonto, un objeto que no vamos a usar para nada pero que necesitamos por ejemplo para poder satisfacer las necesidades de un constructor y que luego no vamos a utilizar.

Ejemplo de un Dummy con Prophecy para ser insertado en la clase `Markdown` y cumplir con los requerimientos del constructor:

```php
<?php
$eventDispatcher = $this->prophesize('MarkdownEventEventDispatcher');
$markdown = new Markdown($eventDispatcher->reveal());
```

Fakes

Un objeto double de tipo Fake añade un poco más de funcionalidad a los Dummy. Si intentamos llamar a un método de un objeto Dummy tendremos un error. Hay veces que tenemos que poder llamar a un método de un Dummy simplemente para que nuestro test continue y no tener un error. Para esto están los Fake que son Dummies con métodos, pero ojo, estos métodos no hacen ni devuelven nada.

Ejemplo de un objeto Fake que como he dicho es como un Dummy, pero con algo más de funcionalidad, en este caso se va a declarar un método para que cuando se llame internamente no de error. El método puede ser llamado con un argumento de tipo `'Markdown\Event\EndOfLineListener'`:

```php
<?php
$eventDispatcher = $this->prophesize('Markdown\Event\EventDispatcher');
    $eventDispatcher->addListener(Argument::type('Markdown\Event\EndOfLineListen\
er'));
    $markdown = new Markdown($eventDispatcher->reveal());
```

Stubs

En los Stubs lo importante es lo que devuelve la llamada a sus métodos, son una forma de garantizar la salida de los objetos colaboradores. Por ejemplo "cuando llamo a `$em->find(1)` devolverá un objeto entidad". La finalidad de los Stubs es reemplazar una funcionalidad concreta del colaborador para garantizar el funcionamiento del colaborador.

Con Mockery se crearía de la siguiente forma:

```php
<?php
$miMock = m::mock('MiClase');
$miMock->shouldReceive('readTemp')->andReturn(11);
```

Con PHPUnit:

```
<?php
$miMock  = $this->getMock('MiClase');
    $miMock->expects($this->any())->method('readTemp')->will($this->returnValue(\
11));
```

Con PHPSpec y Prophecy

```
    <?php
    $miMock = $this->prophesize('MiClase');
    $miMock->readTemp()->willReturn(11);
```

Mocks

Es un tipo de objeto Double sobre los que establecemos unas expectativas de uso y del que no nos preocupa controlar lo que devuelve su llamada.

Por ejemplo: "espero que el objeto colaborador `$em->flush()` sea llamado una sola vez dentro del SUT", o que no sea llamado nunca, o al menos una vez, o que `persist()` sea llamada con un objeto de tipo `MiEntidad`.

Con Mockery se crearía de la siguiente forma:

```
<?php
$miMock = m::mock('MiClase');
$miMock->shouldReceive('readTemp')->times(3);
```

Con PHPUnit:

```
<?php
$miMock  = $this->getMock('MiClase', array('readTemp', '', false);
    $miMock->expects($this->exactly(3))
```

Con PHPSpec y Prophecy:

```
    <?php
    $miMock = $this->prophesize('MiClase');
    $miMock->readTemp()->shouldBeCalledTimes(3);
```

Escuelas o estilos de Mocking

Cuando hablamos de reemplazar los colaboradores de la clase a testear he dado a entender que hay que reemplazar todos los colaboradores por algún tipo de Test Double, pero hay dos corrientes de Mocking: **la escuela clásica (escuela de Chicago) y los Mockists o Mockistas (escuela de Londres)**.

La corriente clásica de mocking establece que hay que usar colaboradores que sean objetos reales siempre que sea posible y sólo en los casos que no se pueda o la relación con el colaborador sea complicada usar Test Doubles. Por el contrario, los Mockistas dicen que cuando estamos escribiendo un test todos los colaboradores deben ser algún tipo de Test Doubles.

Ahora bien, ¿qué corriente seguir? cada uno debe sentirse cómodo y seguro con lo que hace así que no se puede imponer un estilo. Es posible que a uno le ofrezca más confianza emplear colaboradores reales en vez de Test Doubles.

Cuando estamos escribiendo el test y nos encontramos que la clase del SUT tiene un colaborador debemos tomar una decisión. Los Mockistas, crearán un Test Double. Los de la escuela clásica deben decidir si el colaborador es suficientemente sencillo como para usar un objeto real o si es complicado de emplear o tiene un comportamiento difícil de reproducir en cuyo caso emplearán un Test Double.

Si no estamos escribiendo tests para un código heredado e intentamos seguir un flujo de trabajo de TDD en el que los tests se escriben antes que el código es muy posible que los objetos colaboradores todavía no existan y por lo tanto no nos quede más remedio que emplear Test Doubles. La diferencia es que los seguidores de la escuela clásica deberán reemplazar estos dobles por los objetos reales cuando los hayan creado. Para evitar este trabajo la gente de la escuela clásica suele centrarse primero en crear los objetos de dominio.

En realidad, si seguimos la corriente clásica de testing no hacemos tests unitarios sino tests de integración en mayor o menor medida. Esto tiene la ventaja que podemos detectar algunos errores que se nos pueden escapar si por ejemplo no hemos escrito adecuadamente las expectativas de los colaboradores. De todas formas aunque en los tests unitarios de los Mockistas se nos pueden escapar algunos posible errores se puede compensar fácilmente si junto a los tests unitarios ejecutamos tests de aceptación que atraviesan el sistema y no sólo una clase.

De qué no hacer Test Doubles

En la línea de la escuela Mockista o de Londres está la idea de hacer Tests Doubles de todos los objetos colaboradores de nuestro SUT. Esto no es exactamente cierto siempre, o mejor dicho, para todo tipo de objetos.

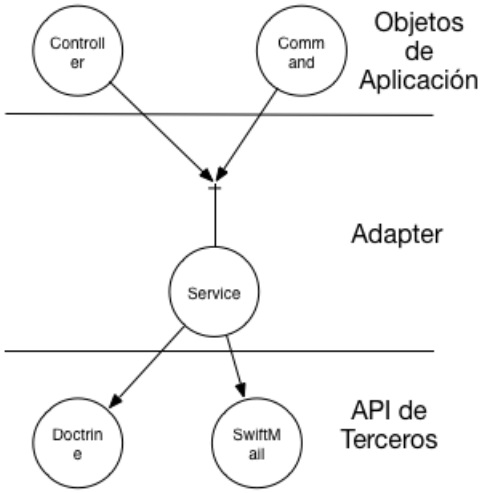
Los Value Objects son objetos simples que no se identifican por una propiedad de identidad $id sino por el valor de los atributos. Los podemos considerar como si fuesen tipos primitivos del lenguaje pero adaptado a nuestro dominio.

Un ejemplo clásico son las clases que representan colores o direcciones postales.

```php
<?php
$color1 = new Color('Rojo'); //Color es un Value Object
$color2 = new Color('Rojo');
$color1 == $color2; // true
$color1 === $color2; // false, igualdad no significa identidad
```

De estos Value Object no necesitamos hacer Doubles puesto que los podemos crear sin ningún tipo de complicación ni sobrecoste y debemos tratarlos como primitivas del lenguaje.

Otro tipo de objetos colaboradores para los que no debemos hacer Test Doubles son los objetos que no podemos modificar. Concretamente las librerías de terceros [Growing Object-Oriented Software Guided by Tests, capítulo 8]. El feedback que obtenemos de los tests no lo podemos trasladar a un refactoring de la librería. Aunque dispongamos del código fuente no es conveniente modificar objetos de terceros. Además no podemos tener total seguridad de que el Double que creemos imite perfectamente el comportamiento de la librería incluso tras una actualización. Para integrar librerías de terceros en nuestro código debemos hacerlos mediante una capa de adaptación e interfaces que representen la relación de nuestra lógica de negocio con el mundo exterior.



Integración de librerías de terceros

Probaremos el comportamiento de estas librerías de terceros mediante tests de integración en los que sustituiremos nuestros propios objetos de aplicación por Doubles para mantener la filosofía de aislamiento y repetibilidad.

Hay una excepción y es que cuando el comportamiento de la librería sea complicado de reproducir en los tests de integración, como por ejemplo puede ser el caso de una cache, es preferible emplear un Double en su lugar.

Librerías de Mocks

Ahora que más o menos tenemos claro qué son los Mocks, Stubs y allegados hay que ver como los creamos en nuestros tests. Para ello tenemos diversas posibilidades.

PHPUnit

La primera opción es no usar ninguna librería, porque casi seguro que ya estaremos usando PHPUnit[1] para nuestros tests. A fin de cuentas es la referencia para el testing con PHP. Un Mock o un Stub con PHPUnit se crea de la siguiente forma:

```
1  <?php
2  $emMock  = $this->getMock('\Doctrine\ORM\EntityManager',
3      array('getRepository', 'getClassMetadata', 'persist', 'flush'), array(), '',\
4   false);
5      $emMock->expects($this->any())
6      ->method('getRepository')
7      ->will($this->returnValue(new FakeRepository()));
8      $emMock->expects($this->any())
9      ->method('getClassMetadata')
10     ->will($this->returnValue((object)array('name' => 'aClass')));
11     $emMock->expects($this->any())
12     ->method('persist')
13     ->will($this->returnValue(null));
14     $emMock->expects($this->any())
15     ->method('flush')
16     ->will($this->returnValue(null));
```

Usar PHPUnit para esta labor tiene los siguientes inconvenientes:

- Hay que escribir demasiado código.
- No queda clara cual es la diferencia entre Mocks y Stubs.

Mockery

Mockery[2] es posiblemente la librería más usada en PHP para crear Mocks y Stubs. Realmente simplifica mucho la tarea de escribir código y lo hace mas legible. El mismo ejemplo de antes en PHPUnit escrito con Mockery sería así:

[1] http://phpunit.de/
[2] https://github.com/padraic/mockery

```php
<?php
$emMock = \Mockery::mock('\Doctrine\ORM\EntityManager',
        array(
            'getRepository' => new FakeRepository(),
            'getClassMetadata' => (object)array('name' => 'aClass'),
            'persist' => null,
            'flush' => null,
     ));
```

Además de ser muchísimo más claro y cómodo, Mockery ofrece muchas otras facilidades.

Integrar Mockery en PHPUnit es muy sencillo:

```php
<?php
use \Mockery as m;

class TemperatureTest extends PHPUnit_Framework_TestCase
{

    public function tearDown()
    {
        m::close();
    }

    public function testGetsAverageTemperatureFromThreeServiceReadings()
    {
        $service = m::mock('service');
        $service->shouldReceive('readTemp')->times(3)->andReturn(10, 12, 14);
        $temperature = new Temperature($service);
        $this->assertEquals(12, $temperature->average());
    }

}
```

La parte importante de la integración es el método `tearDown()`. La llamada estática a `m::close()` limpia el contenedor de Mockery para ese test y ejecuta las tareas de verificación necesarias para revisar las expectativas.

Prophecy

Es una librería de mocking bastante nueva creada expresamente para satisfacer las necesidades de PHPSpec2. Respecto a Mockery notaremos que cambia la sintaxis que a mi entender es muy clara y

legible. Hay otras diferencias más importantes respecto a Mockery, pero voy a reservar un apartado entero para esto a continuación.

Lo que nos debe quedar claro es que si nos decantamos por PHPSpec2, aunque podemos usar Mockery u otra librería, Prophecy[3] es la librería en la que se van a centrar los desarrolladores y por lo tanto donde encontraremos más soporte y garantía de continuidad.

AspectMock

Esta librería se ha creado dentro del framework de testing de Codeception. Se basa en las librería de AOP GoAOP[4] y podemos hacer cosas como crear doubles de métodos estáticos.

Un ejemplo de test con AspectMock usado dentro de PHPUnit:

```php
<?php
use AspectMock\Test as test;

class UserTest extends \PHPUnit_Framework_TestCase
{
    protected function tearDown()
    {
        test::clean(); // remove all registered test doubles
    }

    public function testDoubleClass()
    {
        $user = test::double('demo\UserModel', ['save' => null]);
        \demo\UserModel::tableName();
        \demo\UserModel::tableName();
        $user->verifyInvokedMultipleTimes('tableName',2);
    }
}
```

La principal ventaja de AspectMock[5] es que podemos emplearlo para hacer mocks de casi cualquier código PHP. No siempre nos enfrentamos con software bien estructurado y que haya sido creado teniendo en cuenta el testing. Es posible que queremos hacer tests del algo que nunca escuchó hablar de inyección de dependencias.

[3] https://github.com/phpspec/prophecy
[4] https://github.com/lisachenko/go-aop-php
[5] https://github.com/Codeception/AspectMock

Diferencia conceptual entre Mockery y Prophecy

Traducción/adaptación del post de Konstantin Kudryashov @everzet http://everzet.com/post/72910908762/conceptual-difference-between-mockery-and-prophecy

La idea del artículo es explicar la diferencia sintáctica y de implementación entre ambas librerías. Conceptualmente y en contraposición con Mockery, Prophecy prioriza los mensajes de los objetos (cómo se comunican los objetos) sobre la estructura (cuándo se comunican los objetos)

Un ejemplo:

```php
<?php
interface User
{
   public function getRating();
   public function setRating($rating);
}

class UserRatingCalculator
{
   public function increaseUserRating(User $user, $add = 1)
   {
       $user->setRating($user->getRating() + $add);
   }
}
```

Es más o menos una variación del ejemplo de la calculadora. El test con Mockery sería algo así:

```php
<?php
$user = Mockery::mock('User');
$user->shouldReceive('getRating')->andReturn(2);
$user->shouldReceive('setRating')->with(4)->once();

$calc = new UserRatingCalculator();
$calc->increaseUserRating($user->mock(), 2);
```

El test con Prophecy seria este:

```
    <?php
    $user = $prophet->prophesize('User');
    $user->getRating()->willReturn(2);
    $user->setRating(4)->shouldBeCalled();

    $calc = new UserRatingCalculator();
    $calc->increaseUserRating($user->reveal(), 2);
```

Excepto por pequeñas diferencias ambos tests parecen iguales. Compliquemos un poco más las cosas. Digamos que ahora queremos disparar un evento antes y después de un cambio en el rating. La nueva calculadora sería esta:

```
<?php
class UserRatingCalculator
{
    private $dispatcher;

    public function __construct(EventDispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }

    public function increaseUserRating(User $user, $add = 1)
    {
        $this->dispatcher->userRatingIncreasing($user->getRating());
        $user->setRating($user->getRating() + $add);
        $this->dispatcher->userRatingIncreased($user->getRating());
    }
}
```

Ahora necesitamos verificar qué evento es disparado con un argumento específico. En Mockery el test sería este:

```php
<?php
$user = Mockery::mock('User');
$user->shouldReceive('getRating')->andReturn(2, 2, 4);
$user->shouldReceive('setRating')->with(4)->once();

$disp = Mockery::mock('EventDispatcher');
$disp->shouldReceive('userRatingIncreasing')->with(2)->once();
$disp->shouldReceive('userRatingIncreased')->with(4)->once();

$calc = new UserRatingCalculator($disp->mock());
$calc->increaseUserRating($user->mock(), 2);
```

La clave está en fijarse cómo se ha hecho el stub del método `getRating()` con tres valores de retorno consecutivos. A esto de le llama **structure binding**, fijación a la estructura del código, significa que los tests son dependientes de como el código está escrito (estructurado), hay tres llamadas consecutivas en ese orden para devolver esos valores.

Con Prophecy la solución es diferente:

```php
    <?php
    $user = $prophet->prophesize('User');
    $user->getRating()->willReturn(2);
    $user->setRating(Argument::type('integer'))->will(function($rating) {
    $this->getRating()->willReturn($rating);
})->shouldBeCalled();

    $disp = $prophet->prophesize('EventDispatcher');
$disp->userRatingIncreasing(2)->shouldBeCalled();
$disp->userRatingIncreased(4)->shouldBeCalled();

$calc = new UserRatingCalculator($disp->reveal());
$calc->increaseUserRating($user->reveal(), 2);
```

Prophecy usa un enfoque tipo **message binding** (orientado el mensaje), que significa que el comportamiento del método no cambia en el tiempo, sino es cambiado por el otro método.

¿Cuál es la diferencia real entre el enfoque de ambas librerías? Consideremos un cambio en la calculadora:

```php
<?php
public function increaseUserRating(User $user, $add = 1)
{
    $initialRating = $user->getRating();
    $this->dispatcher->userRatingIncreasing($initialRating);

    $user->setRating($initialRating + $add);

    $resultingRating = $user->getRating();
    $this->dispatcher->userRatingIncreased($resultingRating);
}
```

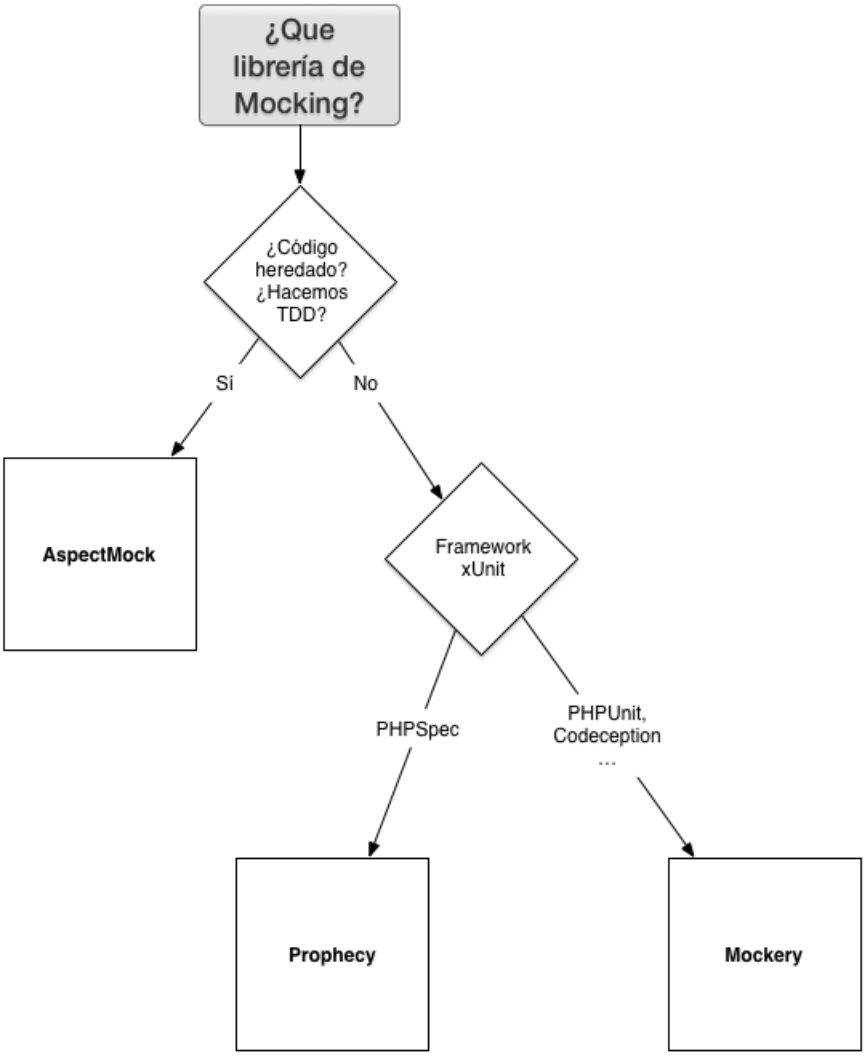
Simplemente hemos puesto el rating inicial en una variable local `$initialRating` por clarificar el código. El problema es que este pequeño cambio hace que se rompa el test hecho con Mockery puesto que ahora hay sólo dos llamadas a `getRating()` en vez de tres. Si los test están asociados a la estructura y esta cambia entonces el test falla.

En el caso de Prophecy el test inicial sigue pasando porque el test se ha creado centrado en los mensajes que se pasan entre los objetos y esto no ha cambiado.

La diferencia conceptual no esta en cómo escribir los tests, sino cuándo hay que corregirlos. Mockery te puede poner en la situación en la que tengas que rehacer los tests simplemente porque has hecho un refactoring que afecta a la estructura. Prophecy postula que en este caso el test no debería fallar porque el comportamiento de los objetos sigue siendo el mismo.

Qué librería de Mocking elegir

Un pequeño árbol de decisión nos puede asistir a la hora de elegir una librería de Mocking



Árbol de decisión de librería de mocking

Este árbol refleja lo que lo elegiría, es posible que haya gente que no emplearía Prophecy por ser muy nueva y porque Mockery tiene mucha comunidad y es muy sencilla de integrar en casi cualquier contexto de testing. Lo que tengo claro es que no emplearía el sistema de Mocking de PHPUnit porque hay que escribir demasiado código y es difícil de leer.